Extraction of complexity bounds from first-order functional programs

7 Dec. 2002

Roberto Amadio

Ludwig-Maximilian Universität and *Université de Provence*

Plan

- Part 1: Mobile/embedded code motivations and approaches.
- Part 2: Some classical results on *functional algebras*.
- Part 3: Restrictions enforcing *space* bounds.
- Part 4: *Max-Plus* quasi-interpretations.

Part 1: Mobile/embedded code motivations and approaches

- Scenarios for resource guarantees
- Proof carrying code approach
- Mobile Resource Guarantees project

Scenarios for Resource Guarantees

- Programmable switches (Penn PLAN project): requires termination.
- Applications threads in a smart card (Gemplus): needs to predict memory consumption.
- In combination with synchronous programming (Pareto).

Proof carrying code approach (Lee-Necula)

- Define *security policy* (e.g. no memory faults).
- Code comes with *evidence* (a proof) of its conformity to the security policy.
- Receiver can (easily) *check evidence* before running the code.
- To increase *efficiency* and *trust-in-compiler*, code is low level (assembler).

PCC (continued)

- Burden is on the code producer: it has to *generate the proof*.
- The proof is formalized in some suitable (Hoare) *program logic* and represented as a λ term in some *logical framework* (*e.g.* LF).
- Proof *compression* and *quick proof check* is an issue.

A couple of frequent questions. . .

- How do you generate the evidence?
- Why don't you just *monitor* the execution?

. . . **and some remarks**

- Can easily rewrite a program so that it respects a certain resource bound: just insert a time out/a memory counter/. . .
- *I.e.*, producer can insert *dynamic checks* whenever it is unable to prove *statically* that the program guarantees certain resource bounds.
- Of course, the more dynamic checks the less efficient (and useful) the program. Still, having dynamic checks performed by the program rather than by the monitor is usually more efficient.

IST Global Computing project *Mobile resource guarantee*

- Concentrates on *resource bounds* security policy: given an input of size n the program will run in at most time $T(n)$ and space $S(n)$.
- To be useful, bounds have to be *precise* and they have to be *valid* for the implemented abstract machine.
- *High-level language* (CAMELOT): a restricted functional language (no functions as results) with a type system to guarantee certain bounds on heap space consumption.
- *Low-level language* (GRAIL): an imperative language with some notion of class and object which is sufficient to implement the abstract machine.

MRG (continued)

- *Defined:* implementation of CAMELOT in GRAIL and cost model for GRAIL.
- *Under development:* Hoare logic with *heap* management for GRAIL implemented in ISABELLE (cf. Abadi-Leino logic, Reynolds' separation logic).
- *Expected:* automatic generation of proofs for the GRAIL code resulting from the compilation of *well-typed* CAMELOT code.

Part 2: Some classical results on *functional algebras*

- A first-order functional language.
- Bounded recursion on notation.
- Ramification.
- Limits of programming with ramification.

A first-order functional language

- *Inductive types*

$$\mu t.(\ldots \mathsf{c} : \tau_1, \cdots, \tau_n \to t, \ldots)$$

- Values, patterns, expressions:

$$v ::= \mathsf{c}(v, \ldots, v)$$
$$p ::= x \parallel \mathsf{c}(p, \ldots, p)$$
$$e ::= x \parallel \mathsf{c}(e, \ldots, e) \parallel f(e, \ldots, e)$$

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

$$f(x_1, \ldots, x_n) =$$
$$\ldots$$
$$x_1 = p_1, \ldots, x_n = p_n \quad \Rightarrow e$$
$$\ldots$$

Bounded recursion on notation (Cobham)

$$\mu t.(\epsilon : t, \mathsf{0} : t \to t, \mathsf{1} : t \to t) \quad \text{(binary words)}$$

$$\begin{array}{ll} f(x, \vec{y}) & = \\ x = \epsilon & \Rightarrow g(\vec{y}) \\ x = \mathsf{i}x' & \Rightarrow h_{\mathsf{i}}(f(x', \vec{y}), x', \vec{y}) \\ \mathsf{with} & |f(x, \vec{y})| \leq P(|x|, |\vec{y}|), P \text{ polynomial.} \end{array}$$

BRN (continued)

- Without bound can still define exponential:

$$\begin{array}{ll} d(x) = & e(x) = \\ x = \epsilon \Rightarrow \epsilon & x = \epsilon \Rightarrow \mathsf{0}(\epsilon) \\ x = \mathsf{i}(x') \Rightarrow \mathsf{i}(\mathsf{i}(x')) & x = \mathsf{i}(x') \Rightarrow d(e(x')) \end{array}$$

- With bound can evaluate in PTIME.
- Vice versa, BRN can simulate polynomially many steps of TM.

Ramification (Bellantoni-Cook and Leivant)

- $f(\vec{x};\vec{y})$: split arguments in *Normal* ($\vec{x}$) and *Safe* ($\vec{y}$).
- $N \leq S$: *Normal* can be regarded as a *subtype* of *Safe*.
- $f(\mathsf{i}x \ldots; \ldots) \Rightarrow h(\ldots; f(x, \ldots; \ldots), \ldots)$.
  Recurrence parameters are *Normal*, *Result of a recurrence* is *Safe* ($\Rightarrow$ typing of *exponential* fails).
- *Constructors* are overloaded, sending safe to safe and normal to normal.
- *Composition*: $g(h_1(\vec{x}; \_); h_2(\vec{x}; \vec{y}))$.

Ramified –size– addition and multiplication

$$a(x; y) =$$

$$x = \epsilon \Rightarrow y$$

$$x = \mathsf{i}x' \Rightarrow \mathsf{i}(a(x; y))$$

$$m(x, y;) =$$

$$x = \epsilon \Rightarrow \epsilon$$

$$m(\mathsf{i}x', y;) \Rightarrow a(y; m(x', y;))$$

Limits of ramification

$$sort(l;) =$$

$$l = \epsilon \Rightarrow \epsilon$$

$$l = \mathsf{i}(x) \Rightarrow insert_{\mathsf{i}}(sort(x;);) \qquad (*)$$

$$insert_{\mathsf{0}}(x;) = \mathsf{0}(x)$$

$$insert_{\mathsf{1}}(x;) =$$

$$x = \epsilon \Rightarrow \mathsf{1}(\epsilon)$$

$$x = \mathsf{1}(x') \Rightarrow \mathsf{1}(\mathsf{1}(x'))$$

$$x = \mathsf{0}(x') \Rightarrow \mathsf{0}(insert_{\mathsf{1}}(x';))$$

(*) $insert_1$ waits for normal but gets safe (*cf.* exponential).

Part 3: Restrictions enforcing space bounds

- Consider *general recursive programs* but find (implicit) way to bound the size of results.
- We analyse two cases:
  - Jones' no-cons condition.
  - Hofmann's type system for *in-place update*.

Jones' no cons condition

- No constructors of positive arity on the right-hand side of the rule.
- Enough to characterize PTIME *problems*.
- Simple *functions* such as reverse cannot be represented.

Hofmann's type system for *in-place update*

- Relies on an –empty– resource type ρ and *affine* typing.
- An element of resource type is understood as a memory cell.
- Constructors take an extra-argument of type ρ. Also functions may get extra-arguments of type ρ.
- In a rule $x_1 = p_1, \ldots, x_n = p_n \Rightarrow e$, resources have to be balanced:

$$\Gamma \vdash p_i, i = 1, \ldots, n \quad \Rightarrow \quad \Gamma \vdash_{\mathit{aff}} e$$

- Data transformations are *non-size increasing* and language can be compiled so that no dynamic heap memory allocation is required.

Part 4: Max-Plus quasi-interpretations

We look for an *automatic* method for inferring bounds on the size of computed values for *general* recursive programs, *without* annotations.

- *Quasi-interpretations* as a tool to bound size of values.
- *Max-Plus* polynomials.
- *Synthesis* problem.

Assign functions over non-negative rationals

$$q_{\mathsf{c}} = \begin{cases} 0 & \mathsf{c} \text{ constant} \\ d + \Sigma_{i=1,\ldots,n} x_i & \text{otherwise, with } d \geq 1 \end{cases}$$

$$q_f : (\mathbf{Q}^+)^k \rightarrow \mathbf{Q}^+ \text{ monotonic and } q_f \geq \pi_i$$

Quasi-interpretation (Marion *et al.*)

Extension of assignment to expressions:

$$q_x = x$$
$$q_{\mathsf{c}(e_1,\ldots,e_n)} = q_{\mathsf{c}}(q_{e_1},\ldots,q_{e_n})$$
$$q_{f(e_1,\ldots,e_n)} = q_f(q_{e_1},\ldots,q_{e_n})$$

Condition an assignment must satisfy to be a quasi-interpretation:

$$q_f(q_{p_{i,1}},\ldots,q_{p_{i,n}}) \geq q_{e_i}$$

NB Quasi-interpretations are inspired by polynomial interpretations for termination proofs.

Basic properties

- $|v| \leq q_v \leq d|v|$, for v value, d constant.
- $e \mapsto v$ then $q_e \geq q_v \geq |v|$.
- Can evaluate $f(v_1, \ldots, v_n)$ in $2^{O(q_{f(v_1,\ldots,v_n)})}$.

A simple evaluator

$$
\begin{aligned}
&\mathit{Eval}(e) = \textsf{case}\\
&e \text{ value} \quad : \; e\\
&e \equiv E[f(v_1, \ldots, v_n)] \text{ and}\\
&\exists\, \sigma \;\; (\sigma(p_j) = v_j, j = 1, \ldots, n) :\\
&\qquad\quad \textsf{let } v' = \mathit{Eval}(\sigma(e)) \textsf{ in}\\
&\qquad\quad \mathit{Eval}(E[v'])\\
&\textsf{else} \;\; : \;\; \mathit{Return} \perp
\end{aligned}
$$

NB This program can be run on a linearly bounded APDA and, by Cook's theorem, it can be transformed to run in EXPTIME.

An evaluator with *memoization*

$Eval_m(e) = \textsf{case}$

e value : e

$e \equiv E[f(v_1, \ldots, v_n)]$ and $\exists\, \sigma, i \;\; (\sigma(p_{i,j}) = v_j, j = 1, \ldots, n)$:

$(new, v'') := Insert(f(v_1, ..., v_n));$ $\quad \Leftarrow$

case

new : let $v' = Eval_m(\sigma(e_i))$ in $\quad (1)$

$Update(f(v_1, ..., v_n), v');$ $\quad \Leftarrow$

$Eval_m(E[v'])$

$\neg new, v'' \neq \bot$: $Eval_m(E[v''])$ $\quad (2) \quad \Leftarrow$

else : $Return\ \bot$ $\quad \Leftarrow$

else : $Return\ \bot$

Insertion sort revisited

The program admits the following quasi-interpretation:

$$q_{\mathsf{i}} = x + 1, \quad q_{sort} = x, \quad q_{insert_{\mathsf{i}}} = x + 1.$$

No cons **revisited**

A program conforming to Jones' restriction admits the following multi-linear quasi-interpretation

$$q_{\mathsf{c}} \quad = 1 + \Sigma_{i=1,\ldots,n} x_i \qquad q_f \quad = max(x_1, \ldots, x_n)\ .$$

In-place update revisited

If a program has an *affine typing* then its *erasure* of resource arguments admits the following multi-linear quasi-interpretation:

$$q_{\mathsf{c}} = 1 + \Sigma_{i=1,\ldots,n} x_i \qquad q_f = r(f) + \Sigma_{i=1,\ldots,n} x_i$$

where $r(f)$ is the number of resource arguments of f.

Lower bounds on expressivity: Qbf

$$qbf(\phi) = \quad check(\phi, nil)$$

$$check(\phi, l) =$$

$$\phi = \mathsf{v}(x) \quad \Rightarrow mem(x, l)$$

$$\phi = \mathsf{o}(\phi', \phi'') \quad \Rightarrow or(check(\phi', l), check(\phi'', l))$$

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

Quasi-interpretation

$$q_{\mathsf{v}} = x + 1, \quad q_{\mathsf{o}} = q_{\mathsf{all}} = x + y + 1, \quad q_{qbf} = x,$$

$$q_{or} = q_{mem} = max(x, y), \quad q_{check} = \phi + l$$

Lower bound on expressivity: exponential time TM

- Can also simulate TM running in $2^{O(n)}$.
- Define

$$T : \mathit{Input} \times \mathit{Step} \times \mathit{Position} \rightarrow \mathit{State} \times \mathit{Letter}$$

- $T(x, s, p) = (q, a)$ iff the machine with input x after s steps arrives in state q with character a at position p.
- s, p can be stored in space $O(|x|)$ and we can do basic arithmetic modulo $2^{O(|x|)}$.
- $T(x + 1, s, p)$ can be defined recursively in terms of $T(x, s, p - 1)$, $T(x, s, p)$, $T(x, s, p + 1)$.

NB Again, this is a rephrasing of Cook's theorem (from EXPTIME to APDA).

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: αx.
- Polynomial of degree d with n indeterminates:

$$max_{I:\{1,\ldots,n\}\to\{0,\ldots,d\}}(I(1)x_1 + \cdots + I(n)x_n + a_I)$$

- For a given degree *synthesis problem* can be expressed as validity of $\exists\forall$ Presburger formula. Look for something more efficient…

Lower bound on complexity of synthesis

Prop The synthesis problem is NP-hard.

- Reduction from SAT.
- Devise rules that force $q_f = max(x_1, \ldots, x_n)$. E.g.

  $$f(\mathsf{c}(x)) \Rightarrow f(f(\mathsf{c}(x)))$$

  forces $q_f = max(a, x)$.
- Simulate boolean variables with constructors' coefficients.

NB This lower bound does *not* depend on bounding the degree of the polynomials or the size of the rules.

Multi-linear polynomials

- Multi-linear = Degree of every variable is at most 1:

$$max_{I\subseteq\{1,\ldots,n\}}(\Sigma_{i\in I}x_i + a_I)$$

- Multi-linear polynomials have a *normal form*...

$$J \subseteq K \subseteq \{1,\ldots,n\} \quad \Rightarrow a'_J \geq a'_K$$

- ... and then they are easy to *compare*:

$P_1 \geq P_2, P_1$ multi-linear $\;\Rightarrow\;$ P_2 multi-linear.

Suppose P_1, P_2 multi-linear. $P_1 \geq P_2$ iff $a^1_I \geq a^2_I, I \subseteq \{1,\ldots,n\}$

Upper bound on complexity of synthesis

Prop For programs with rules of bounded size the synthesis problem for multi-linear polynomials is NP-complete.

- Compute the interpretations of $q_{f(p_1,\ldots,p_n)}$ and q_e and reduce to the satisfaction of a system of inequalities over $\mathbf{Q}^+_{max}$.
- Use *non-determinism* to eliminate *max* from $q_{f(p_1,\ldots,p_n)}$ on the right-hand side of the inequality.

  $$max(A, B) \geq C \text{ becomes } (A \geq C \wedge A \geq B) \vee (B \geq C \wedge B \geq A)$$

- Eliminate *max* in q_e in polynomial time. Idea:

  $$A \geq max(B, C) \text{ becomes } A \geq z, z \geq B, z \geq C$$

Upper bound (continued)

- Get a system with constraints of the shape:

$$x = -\infty \qquad\qquad y \geq 1$$

$$x + \Sigma_{j=1,\dots,l}\alpha_j y_j \geq \Sigma_{j=1,\dots,n}\beta_j x_j + \Sigma_{j=1,\dots,l}\gamma_j y_j$$

- Send to $-\infty$ all the variables for which no $x \geq 0$ constraint can be inferred. Idea on *boolean* variables: satisfaction of formulae $\bigvee_{j\in J} x_j$ or $x \Rightarrow \bigvee_{j\in J} x_j$ can be decided efficiently.
- Hence reduce to a *linear programming* problem over $\mathbf{Q}^+$ (it is possible to look for *optimal* solutions).

NB If the size of the rules is not bound then the method requires exponential space just to write the solution.

Work in progress/problems

- Look for synthesis subproblems with polynomial complexity.
- Determine complexity of the synthesis problem for higher degrees.
- Consider quasi-interpretations in more complicated type theories (co-inductive types, higher-order types).

Related work

- Pareto *et al.* *sized types.*
  - Functions definitions are annotated with Presburger's functions, *i.e.* type-checking rather than type-inference.
  - Type checking uses OMEGA library to validate $\forall\exists$ Presburger's formulae.
- Hofmann-Jost *heap analysis.*
  - Annotate judgments $x : \tau, f \vdash e : \tau', g$ with the interpretation: evaluation of $[v/x]e$ requires $f(|v|)$ heap, and if $[v/x]e \mapsto v'$ then it releases $g(|v'|)$ heap.
  - Goal: lower bound on heap size needed to complete evaluation.
  - Synthesis method over linear affine functions (no *max*).