A characterization of Polynomial Space with Forks

Romain Péchoux
joint work with Emmanuel Hainry and Jean-Yves Marion

Lorraine University
Inria project Carte, Loria

30 mai 2012

Outline

ICC and polynomial Space

ICC related works on polynomial space :

- Function algebra with parameter substitution (Leivant-Marion 94)
- Function algebra with ramified recurrence (Leivant-Marion 97)
- Quasi-interpretation with LPO (Bonfante-Marion-Moyen 07)
- Lambda calculus with LLL based type system (Gaboardi-Ronchi Della Rocca-Marion 07)
- Higher-order types or life without cons (Jones 01)
- Matrix calculus (Niggl-Wunderlich, Jones-Kristiansen, Moyen)

Our approach

Take the methodology of the type system presented in LICS 2011 (Marion) that combines :

- data ramification principle (or tiering)
- with non-interference based type system
- on a simple imperative language

in order to characterize polynomial space on an imperative language with wait/fork mechanism.

Advantages of the presented methodology :

- a good expressivity
- very close to C-fork processes

Simple While Language with fork/wait

$$
\begin{array}{lcl}
E, E_1, \ldots, E_n \in \mathrm{Exp} & ::= & X \mid \mathbf{op}(E_1, \ldots, E_n) \\
I \in \mathrm{Inst} & ::= & \mathtt{fork}() \mid \mathtt{wait}(E) \\
C, C' \in \mathrm{Cmd} & ::= & X{:=}E \mid C;\, C' \mid \mathtt{skip} \\
 & & \mid \mathtt{while}(E)\mathtt{do}\{C\} \\
 & & X{:=}I \mid \mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ C' \\
\mathrm{P} \in \mathrm{Proc} & ::= & \mathtt{return}\ X \mid C;\, \mathrm{P}
\end{array}
$$

$X \in \mathcal{X}$ and $\mathbf{op} \in \mathbb{O}$

Main results

A type system for imperative programs such that :

- Typable programs are computable in polynomial space under some restrictions :
  - termination
  - confluence
  - lock-freedom
  - the return type
- Each polynomial space problem can be computed by a typed program
- In a terminating program, all processes compute in polynomial time (polynomial number of steps)

A process being either the main program process or a subprocess created by a fork instruction.

Informal semantics : store, configuration and environment

Let $\mathbb{W}$ be the set of words over Σ.
Sequential commands are evaluated as usual
A process P is evaluated inside a configuration $c = (\mathrm{P}, \mu)_\rho$:

- P is the program counter
- A store $\mu : \mathcal{X} \to \mathbb{W}$ mapping each variable of P to a value
- A set of ids ρ, the sons of c
- each configuration has an id (an integer). The main process id is 1.

All configurations are stored in an environment $\mathscr{E}$, a partial function, mapping an id $\in \mathbb{N}$ to a configuration c.

Informal semantics : fork

At the beginning, there is only one configuration (the main process) of id 1 and with $\rho = \emptyset$.
A fork instruction creates a new child :

- with a new id (set to the next available integer)
- that runs concurrently of its father
- with its own duplicated memory (the store and the program counter are duplicated)
- the child id is stored in the father ρ

Informal semantics : wait

The wait instruction provides a *one-way* communication and is the only way for a father to communicate with its child.

A $wait(E)$ instruction :

- evaluates the expression E to a binary number $\underline{n}$ encoding id n
- if $n \in \rho$ and the child is *returning* then :
  - the child return value is passed to the father
  - the child is erased
- otherwise the father waits for its children

Note that children of a killed father may still be alive

Example

```
P:   X := fork();
Q:   if X > 0 then {
R:       Y := wait(X);
         Y := Y + 1
     } else {
         Y := 17
     }
S:   return Y
```

Initial environment :
$\mathscr{E}(1) = (P, \mu)_{\emptyset}$
Fork evaluation :
$\mathscr{E}(1) = (Q, \mu\{X := \underline{2}\})_{\{2\}}$ $\mathscr{E}(2) = (Q, \mu\{X := \underline{0}\})_{\emptyset}$
After some steps :
$\mathscr{E}(1) = (R, \mu\{X := \underline{2}\})_{\{2\}}$ $\mathscr{E}(2) = (S, \mu\{X := \underline{0}, Y := \underline{17}\})_{\emptyset}$
Wait evaluation :
$\mathscr{E}(1) = (S, \mu\{X := \underline{2}, Y := \underline{18}\})_{\{2\}}$ $\mathscr{E}(2) = \perp$

Semantics of expressions and configurations

$$(X, \mu) \overset{e}{\to} \mu(X)$$

$$(\mathbf{op}(E_1, \ldots, E_n), \mu) \overset{e}{\to} [\![\mathbf{op}]\!](d_1, \ldots, d_n)$$
$$\text{if } \forall i, (E_i, \mu) \overset{e}{\to} d_i$$

$$(\texttt{skip}; \mathrm{P}, \mu)_\rho \overset{c}{\to} (\mathrm{P}, \mu)_\rho$$

$$(X{:=}E; \mathrm{P}, \mu)_\rho \overset{c}{\to} (\mathrm{P}, \mu\{X \leftarrow d\})_\rho$$
$$\text{if } (E, \mu) \overset{e}{\to} d$$

$$(\texttt{if } E \texttt{ then } C_{\texttt{tt}} \texttt{ else } C_{\texttt{ff}}; \mathrm{P}, \mu)_\rho \overset{c}{\to} (C_w; \mathrm{P}, \mu)_\rho$$
$$\text{if } (E, \mu) \overset{e}{\to} w \in \{\texttt{tt}, \texttt{ff}\}$$

$$(\texttt{while}(E)\texttt{do}\{C\}; \mathrm{P}, \mu)_\rho \overset{c}{\to} (\mathrm{P}, \mu)_\rho$$
$$\text{if } (E, \mu) \overset{e}{\to} \texttt{ff}$$

$$(\texttt{while}(E)\texttt{do}\{C\}; \mathrm{P}, \mu)_\rho \overset{c}{\to} (C; \texttt{while}(E)\texttt{do}\{C\}; \mathrm{P}, \mu)_\rho$$
$$\text{if } (E, \mu) \overset{e}{\to} \texttt{tt}$$

Semantics of environments

Let $\mathscr{E}' = \mathscr{E}[i := c]$ be defined by :

- $\mathscr{E}'(j) = \mathscr{E}(j), \forall j \in dom(\mathscr{E}) - \{i\}$,
- $\mathscr{E}'(i) = c$

The transition $\rightarrow$ for process evaluation is defined by :

$$\mathscr{E}[i := c] \rightarrow \mathscr{E}[i := c']$$
$$\text{if } c \xrightarrow{\mathrm{c}} c'$$

$$\mathscr{E}[i := (X{:=}\mathtt{fork}();\, \mathrm{P}, \mu)_\rho]$$
$$\rightarrow \mathscr{E}[i := (\mathrm{P}, \mu\{X \leftarrow \underline{n}\})_{\rho \cup \{n\}}, n := (\mathrm{P}, \mu\{X \leftarrow \underline{0}\})_\emptyset]$$
$$\text{with } n = \sharp\mathscr{E} + 1$$

$$\mathscr{E}[i := (X{:=}\mathtt{wait}(E);\, \mathrm{P}, \mu)_\rho]$$
$$\rightarrow \mathscr{E}[i := (\mathrm{P}, \mu\{X \leftarrow \mu'(Y)\})_\rho, n := \bot]$$
$$\text{if } (E, \mu) \xrightarrow{\mathrm{e}} \underline{n},\ n \in \rho \text{ and } \mathscr{E}_n = (\mathtt{return}\ Y, \mu')$$

Strong normalization, lock-freedom and confluence

Strong normalization

A process P is strongly normalizing if $\forall \mu$ there is no infinite reduction starting from $(\mathrm{P}, \mu)_\emptyset$ through the relation $\rightarrow$.

Lock-freedom

If $\mathscr{E} \not\rightarrow$ and $\mathscr{E}_1 = (X{:=}\mathtt{wait}(E);\, \mathrm{P}, \mu)_\rho$ then $\mathscr{E}$ is locked.
A process P is *lock-free* if $\forall \mu$, there is no locked environment $\mathscr{E}'$ s.t. $(\mathrm{P}, \mu)_\emptyset \xrightarrow{\star} \mathscr{E}'$.

Confluence

A process P is *confluent* if $\forall \mu$, $(\mathrm{P}, \mu)_\emptyset \xrightarrow{\star} \mathscr{E}'$ and $(\mathrm{P}, \mu)_\emptyset \xrightarrow{\star} \mathscr{E}''$, $\exists \mathscr{E}^3$ s.t. $\mathscr{E}' \xrightarrow{\star} \mathscr{E}^3$ and $\mathscr{E}'' \xrightarrow{\star} \mathscr{E}^3$.

Example of non-confluent process


```
P: X := fork();
   Y := fork();
   return Y
```

The main process will return the process identifier of its second son.
Depending on the order in which execution of the subprocesses occurs, this identifier can be either 3 or 4.

Tiers and typing environments

- *Tiers* are two elements $\mathbf{0}, \mathbf{1}$ underlying a boolean lattice $(\{\mathbf{0}, \mathbf{1}\}, \preceq, \mathbf{0}, \vee, \wedge)$ such that $\mathbf{0} \preceq \mathbf{1}$
- Operator types τ are defined by

$$\tau ::= \alpha \mid \alpha \longrightarrow \tau, \ \alpha \in \{\mathbf{0}, \mathbf{1}\}$$

- A *variable typing environment* Γ maps each variable in $\mathcal{V}$ to a tier in $\{\mathbf{0}, \mathbf{1}\}$
- An *operator typing environment* Δ maps each operator **op** of arity n to a set $\Delta(\mathbf{op})$ of operator types of the shape $\tau = \alpha_1 \longrightarrow \ldots \alpha_n \longrightarrow \alpha$

$$\frac{\Gamma(X) = \alpha}{\Gamma, \Delta \vdash X : \alpha} \text{ (EV)}$$

$$\frac{\Gamma, \Delta \vdash E_i : \alpha_i \qquad \alpha_1 \longrightarrow \ldots \longrightarrow \alpha_n \longrightarrow \alpha \in \Delta(\mathbf{op})}{\Gamma, \Delta \vdash \mathbf{op}(E_1, \ldots, E_n) : \alpha} \text{ (EO)}$$

$$\frac{\Gamma, \Delta \vdash X : \mathbf{0}}{\Gamma, \Delta \vdash_\beta X{:=}\texttt{fork}() : \mathbf{0}} \text{ (F)}$$

$$\frac{\Gamma, \Delta \vdash E : \mathbf{0} \quad \Gamma, \Delta \vdash X : \alpha}{\Gamma, \Delta \vdash_\beta X{:=}\texttt{wait}(E) : \alpha} \; \alpha \preceq \beta \quad \text{(W)}$$

$$\frac{\Gamma, \Delta \vdash X : \alpha \qquad \Gamma, \Delta \vdash E : \alpha' \qquad E \in \mathrm{Exp}}{\Gamma, \Delta \vdash_\beta X{:=}E : \alpha} \; \alpha \preceq \alpha' \quad \text{(CA)}$$

$$\frac{\Gamma, \Delta \vdash_\beta C : \alpha \qquad \Gamma, \Delta \vdash_\beta C' : \alpha'}{\Gamma, \Delta \vdash_\beta C; C' : \alpha \vee \alpha'} \text{ (CC)}$$

$$\frac{\Gamma, \Delta \vdash E : \mathbf{1} \qquad \Gamma, \Delta \vdash_\beta C : \alpha}{\Gamma, \Delta \vdash_\beta \mathtt{while}(E)\mathtt{do}\{C\} : \mathbf{1}} \; \textit{(CW)}$$

$$\frac{\Gamma, \Delta \vdash E : \alpha \qquad \Gamma, \Delta \vdash_\beta C : \alpha \qquad \Gamma, \Delta \vdash_\beta C' : \alpha}{\Gamma, \Delta \vdash_\beta \mathtt{if}\ E\ \mathtt{then}\ C\ \mathtt{else}\ C' : \alpha} \; \textit{(CB)}$$

$$\frac{}{\Gamma, \Delta \vdash_\beta \mathtt{skip} : \alpha} \; \textit{(CS)}$$

$$\frac{\Gamma, \Delta \vdash_\beta C : \mathbf{0}}{\Gamma, \Delta \vdash_\beta C : \mathbf{1}} \; \textit{(CSub)}$$

$$\frac{\Gamma, \Delta \vdash_\beta C : \alpha \quad \Gamma, \Delta \vdash X : \beta}{\Gamma, \Delta \vdash C; \mathtt{return}\ X : \beta} \; \textit{(P)}$$

Some more intuitions

- The type discipline precludes values from flowing from tier **0** to **1** (but not command because of CS)
- Consequently, while loop guards are enforced to be of tier **1** (CW)
- In a (CB) rule the guard tier is equal to tier of both branches (could be weakened)
- However information may flow in the opposite direction (CA)
- The annotation β keeps tract of the return type and is used by wait instructions (W)

Example :

```
found^0 := ff^0 : 0 ;
n^1 := length(str)^1 : 1;
l^1 := n/2^1 : 1;
x^0 := fork()^0 : 0;
while l>0^1 do {
      if x>0^0 then
         c^0 := getchar(str^1, l^1)^1 : 0;
      else
         c^0 := getchar(str^1, n−l^1)^1 : 0;
      if c=='*'^0 then
        found^0 := tt^0 : 0
      else skip : 0 ;
     l^1 := l−1^1 : 1
 } : 1
if x>0^0 then {
     sonf^0 := wait(x)^0 : 0;
     found^0 := or(found, sonf)^0 : 0
} else skip : 0 ;
return found^0
```

Neutral, positive and polynomial operators

An operator **op** is :

1. *neutral* (**op** $\in Ntr$) if :
   1.1 either $\forall n \in \mathbb{N}^*, \ [\![op]\!](\vec{\mathbb{W}}_n) \subseteq \mathbb{W}_{n-1}$.
   1.2 or there is a polynomial $P_{\mathbf{op}}$ s.t. $\forall n \in \mathbb{N}^*, \ \exists \mathbb{V}_n^{\mathbf{op}} \subseteq \mathbb{W}_n$,

   $$[\![\mathbf{op}]\!](\vec{\mathbb{W}}_n) \subseteq \mathbb{V}_n^{\mathbf{op}} \text{ and } \sharp \mathbb{V}_n^{\mathbf{op}} \leq P_{\mathbf{op}}(n)$$

2. *positive* (**op** $\in Pos$) if **op** $\notin Ntr$ and there is $c_{op} \in \mathbb{N}$ s.t. :

   $$\forall \vec{d} \in \mathbb{W}^m, \ |[\![\mathbf{op}]\!](\vec{d})| \leq \max_{i \in [1,m]} |d_i| + c_{op}$$

3. *polynomial* (**op** $\in Pol$) if **op** $\notin Ntr \cup Pos$ and there is a polynomial Q_{op} s.t. :

   $$\forall \vec{d} \in \mathbb{W}^m, \ |[\![\mathbf{op}]\!](\vec{d})| \leq Q_{\mathbf{op}}(\max_{i \in [1,m]} |d_i|)$$

Safe operator typing environment

Definition

Δ is *safe* if $\forall \mathbf{op} \in dom(\Delta)$ and $\forall \alpha_1 \to \ldots \to \alpha_n \to \alpha \in \Delta(\mathbf{op})$ we have :

- if $\mathbf{op} \in Ntr$ then $\alpha \preceq \wedge_{i=1,n}\alpha_i$,
- if $\mathbf{op} \in Pos$ then $\alpha = \mathbf{0}$,
- if $\mathbf{op} \in Pol$ then $\forall i \in [1, n],\ \alpha_i = \mathbf{1}$ and $\alpha = \mathbf{0}$

Intuitively :

- Neutral operators are iterable ($\mathbf{1} \longrightarrow \mathbf{1}$ in a while loop guard).
- Positive operators are not iterable but composable ($\mathbf{0} \longrightarrow \mathbf{0}$ in a while-loop command).
- Polynomial operators are neither iterable nor composable ($\mathbf{1} \longrightarrow \mathbf{0}$).

Examples

$$
\begin{array}{llll}
\textit{(Ntr 1.1)} & [\![pred]\!](u) & = \varepsilon & \text{if } u = \varepsilon \\
 & & = w & \text{if } u = a.w \\
\textit{(Ntr 1.2)} & [\![==]\!](u, w) & = \mathtt{tt} & \text{if } u = w \\
 & & = \mathtt{ff} & \text{otherwise.} \\
\textit{(Pos)} & [\![suc_d]\!](b) & = d.b & \\
\textit{(Pol)} & [\![calloc]\!](u, w) & = \underbrace{w.\cdots.w}_{|u| \text{ times}} &
\end{array}
$$

If Δ is a safe operator typing environment then :
$\Delta(pred) \subseteq \{\mathbf{0} \longrightarrow \mathbf{0}, \mathbf{1} \longrightarrow \mathbf{1}, \mathbf{1} \longrightarrow \mathbf{0}\}$,
$\Delta(==) \subseteq \{\mathbf{1} \longrightarrow \mathbf{1} \longrightarrow \mathbf{1}, \alpha \longrightarrow \beta \longrightarrow \mathbf{0}, \alpha, \beta \in \{\mathbf{0}, \mathbf{1}\}\}$,
$\Delta(suc_d) \subseteq \{\mathbf{1} \longrightarrow \mathbf{0}, \mathbf{0} \to \mathbf{0}\}$,
$\Delta(calloc) = \{\mathbf{1} \longrightarrow \mathbf{1} \longrightarrow \mathbf{0}\}$.

Safe process

Definition (Safe process)

Given Γ a variable typing environment and Δ a operator typing environment, a process P is a *safe process* if :

- P is well-typed wrt Γ and Δ, i.e. $\Gamma, \Delta \vdash \mathrm{P} : \beta$
- and Δ is safe

The search(str) program is safe wrt Γ, Δ provided.

Polynomial space

Theorem

The set of Pspace decision problems is exactly the set of problems decided by :

- *safe,*
- *confluent,*
- *strongly normalizing,*
- *and lock-free processes* P

Corollary

If P *is a safe, confluent, strongly normalizing and lock-free processes* P *such that* $\Delta, \Gamma \vdash \mathrm{P} : \mathbf{1}$ *then the function computed by* P *is in FPspace.*

Intermediate Lemmata on tier **1** values

Lemma (Simple security)

Given a safe process P *wrt* Γ *and* Δ*, if* $\Gamma, \Delta \vdash E : \mathbf{1}$ *then* $\forall X \in \mathcal{V}(E), \Gamma(X) = \mathbf{1}$ *and all operators in* E *are neutral.*

Lemma (Bounded size)

Given a safe process P *wrt* Γ *and* Δ *s.t.* $\Gamma, \Delta \vdash E : \mathbf{1}$*, for each store* μ*, if* $\forall X \in \mathcal{V}(E),\ \mu(X) \in \mathbb{W}_n$ *and* $(E, \mu) \xrightarrow{e} d$ *then* $d \in \mathbb{W}_n$*.*

Lemma (Bounded cardinality)

Given a safe process P *wrt* Γ *and* Δ *and* $\Gamma, \Delta \vdash E : \mathbf{1}$*, the number of distinct values taken by* E *during the evaluation of* $(\mathrm{P}, \mu)_\emptyset$ *is bounded polynomially in* $|\mu|$*.*

Process tree

Definition (Process tree)

The process tree $T(\mathscr{E})$ of an environment $\mathscr{E}$ is defined by :

- the nodes are the configurations $\{\mathscr{E}_1, \ldots, \mathscr{E}_{\sharp\mathscr{E}}\}$
- the root is $\mathscr{E}_1$;
- for each $l \in [1, \sharp\mathscr{E}]$, there is an edge from $\mathscr{E}_l = (\mathrm{P}, \mu)_\rho$ to $\mathscr{E}_k$, if $k \in \rho$.

Given a process tree T, its degree is denoted $d(T)$ and height $h(T)$.

$\rightsquigarrow$ pstree

Intermediate Lemmata on the process tree

Lemma (Bounded degree)

Given a strongly normalizing and safe process P*, there exists a polynomial* Q *s.t.,* $\forall \mu$*, if* $(\mathrm{P}, \mu)_{\emptyset} \to^{*} \mathscr{E}$ *then* $d(T(\mathscr{E})) \leq Q(|\mu|)$*.*

Lemma (Bouded height)

Given a strongly normalizing and safe process P*, there exists a polynomial* Q *s.t.,* $\forall \mu$*, if* $(\mathrm{P}, \mu)_{\emptyset} \to^{*} \mathscr{E}$ *then* $h(T(\mathscr{E})) \leq Q(|\mu|)$*.*

Lemma (Subprocesses in polynomial time)

Given a strongly normalizing and safe process P*, there is a polynomial* Q *s.t.,* $\forall \mu$ *and* $\forall i \in \mathbb{N}$*, if* $(\mathrm{P}, \mu)_{\emptyset} \Rightarrow_{i}^{k} \mathscr{E}$ *then* $k \leq Q(|\mu|)$*.*

where $\Rightarrow_{i}^{k}$ means that the i-th configuration has been evaluated k times.

Pspace abiding strategy

Lemma (Bounded stores)

Given a strongly normalizing and safe decision process P, *there exists a polynomial* Q *such that,* $\forall \mu$, *if* $(\mathrm{P}, \mu)_\emptyset \to^* \mathscr{E}$ *then* $\forall i \leq \sharp\mathscr{E}$, *if* $\mathscr{E}_i = (\mathrm{P}_i, \mu_i)_{\rho_i}$ *then* $|\mu_i| \leq Q(|\mu|)$.

Soundness.

We define a lazy Pspace abiding evaluation strategy that :

Init defines the current process to be the main process configuration

- executes the current process as long as possible
- on a wait instruction updates the current process to the waited process
- on a return instruction updates the current process to the father

$\square$

Completeness

We write a safe process computing QBF :

- when an $\exists x$ ($\forall$) is encoutered a fork instruction is called :
  - the son evaluates the remaining formula with x set to `tt`
  - whereas the father evaluates the remaining formula with x set to `ff`
  - at the end the father gets his son's result and computes the disjunction with its own result (conjunction)
- A calloc operator is used in order to store the processses id.

Conclusions

- We have a characterization of Pspace combining non-interference and tiering methodologies
- The system is expressive (very close to C fork programs or Unix processes)
- It allows the programmer to simulate malloc/calloc operators (Polynomial operators)
- Possible extension on threads with creation