Characterization theorem for primitive recursive algorithms

Pierre Valarcher, LACL (Paris East, France)

Philippe Andary, Bruno Patrou (Univ. Rouen, FR)
• Some negative results in programming
• Abstract state machine and primitive recursive algorithms (PRA)
• A programming language for PRA
Some results
Some results

- Colson

- The PR language (computes primitive recursive functions) does not implement the good algorithm for the minimum function
Some results

- **Colson**
  - The PR language (computes primitive recursive functions) does not implement the good algorithm for the minimum function

- **Moschovakis, Van Dries**
  - The PR language with some primitives (<, /2, ...) does not implement the Stein’s gcd algorithm
Some results

- **Colson**
  
  The PR language (computes primitive recursive functions) does not implement the good algorithm for the minimum function

- **Moschovakis, Van Dries**
  
  The PR language with some primitives (<, /2, ...) does not implement the Stein’s gcd algorithm

- **Crolard, Lacas, Valarcher**
  
  The LOOP language (Meyer, Ritchie) does not implement the good algorithm for the minimum function
There are programming languages that compute a set of functions but they are limited from an algorithmic point of view.
There are programming languages that compute a set of functions but they are limited from an algorithmic point of view.
There are programming languages that compute a set of functions but they are limited from an algorithmic point of view.
There are programming languages that compute a set of functions but they are limited from an algorithmic point of view.
We are interested in TOTAL programming

This means that we can compute only total function

For instance, we choose a programming language that computes only the set of primitive recursive functions
We are interested in TOTAL programming

This means that we can compute only total function

For instance, we choose a programming language that computes only the set of primitive recursive functions
We want to compare programming language: we want to say something like «P is better than Q»
We want to compare programming language: we want to say something like «P is better than Q»

In fact a language is composed of two parts: Types and primitives    Control structure
We want to compare programming language: we want to say something like «P is better than Q»

In fact a language is composed of two parts: Types and primitives Control structure
We want to compare programming language: we want to say something like «P is better than Q»

In fact a language is composed of two parts: Types and primitives Control structure

\[
\begin{align*}
0, +1, -1, \pi_i & \quad PR \\
\circ & \\
f(0, Y) & \rightsquigarrow g(Y) \\
f(x + 1, Y) & \rightsquigarrow h(x, f(x, Y), Y)
\end{align*}
\]
We want to compare programming language: we want to say something like «P is better than Q»

In fact a language is composed of two parts: Types and primitives Control structure

\[
\begin{align*}
\text{PR:} & \quad 0, +1, -1, \pi_i \\
& \quad f(0, Y) \rightsquigarrow g(Y) \\
& \quad f(x + 1, Y) \rightsquigarrow h(x, f(x, Y), Y)
\end{align*}
\]

\[
\begin{align*}
\text{PRV:} & \quad 0, +1, -1, \pi_i \\
& \quad f(0, Y) \rightsquigarrow g(Y) \\
& \quad f(x + 1, Y) \rightsquigarrow h(x, f(x, j(x, Y)), Y)
\end{align*}
\]
• As there are algorithms that are not programmable
• As we want to compare languages from an algorithmic point of view

We need a formal definition for the notion of algorithms
Abstract State Machine (very quickly)
Abstract State Machine (very quickly)

- Evolving algebra (first name)
Abstract State Machine (very quickly)

- Evolving algebra (first name)

- A state is a first-order logic algebra (the API)
Abstract State Machine (very quickly)

- Evolving algebra (first name)

- A state is a first-order logic algebra (the API)

- A transition between 2 states is done by 4 rules:
Abstract State Machine (very quickly)

- Evolving algebra (first name)

- A state is a first-order logic algebra (the API)

- A transition between 2 states is done by 4 rules:
  - $R_1 ::= \text{if } C \text{ then } R$ (conditional)
Abstract State Machine (very quickly)

- Evolving algebra (first name)

- A state is a first-order logic algebra (the API)

- A transition between 2 states is done by 4 rules:
  - $R_1 ::= \text{if } C \text{ then } R$ (conditional)
  - $R_2 ::= f(t_1, ..., t_N) := t_0$ (update)
Abstract State Machine (very quickly)

- Evolving algebra (first name)
- A state is a first-order logic algebra (the API)
- A transition between 2 states is done by 4 rules:
  - \( R1 ::= \text{if } C \text{ then } R \) (conditional)
  - \( R2 ::= f(t_1, ..., t_N) := t_0 \) (update)
  - \( R3 ::= R_i \parallel ... \parallel R_k \) (simultaneous rules)
Abstract State Machine (very quickly)

- Evolving algebra (first name)
- A state is a first-order logic algebra (the API)
- A transition between 2 states is done by 4 rules:
  - $R1 ::= \text{if } C \text{ then } R$ (conditional)
  - $R2 ::= f(t_1, ..., t_N) := t_0$ (update)
  - $R3 ::= R_i || ... || R_k$ (simultaneous rules)
  - $R4 ::= \text{skip}$ (nothing)
Abstract State Machine (very quickly)

- Evolving algebra (first name)
- A state is a first-order logic algebra (the API)
- A transition between 2 states is done by 4 rules:
  - **R1 ::=** if C then R (conditional)
  - **R2 ::=** f(t1, ..., tN) := t0 (update)
  - **R3 ::=** Ri || ... || Rk (simultaneous rules)
  - **R4 ::=** skip (nothing)
- A loop outside this,
Abstract State Machine (very quickly)

- Evolving algebra (first name)

- A state is a first-order logic algebra (the API)

- A transition between 2 states is done by 4 rules:
  - R1 ::= if C then R (conditional)
  - R2 ::= f(t1, ..., tN) := t0 (update)
  - R3 ::= Ri || ... || Rk (simultaneous rules)
  - R4 ::= skip (nothing)

- A loop outside this,
  - a finite set of data are changed between states
Abstract State Machine (very quickly)

- Evolving algebra (first name)

- A state is a first-order logic algebra (the API)

- A transition between 2 states is done by 4 rules:
  - \[ R1 ::= \text{if } C \text{ then } R \] (conditional)
  - \[ R2 ::= f(t1, ..., tN) := t0 \] (update)
  - \[ R3 ::= R_i \parallel ... \parallel R_k \] (simultaneous rules)
  - \[ R4 ::= \text{skip} \] (nothing)

- A loop outside this,
- a finite set of data are changed between states
- We stop when two consecutive states are unchanged

dimanche 21 juin 2009
$\pi = \text{A set of Rules}$
First Order State

\[ \Pi = \text{A set of Rules} \]

\[ F = \text{1st order structure} \]

CS = Rules of ASM

An asm

A set of ASM of level of abstraction \( F \)
$X = n, \ Y = m, X_1, Y_1, \ stop = \ false, \ init = \ true, +1, -1, 0, =$

IF init then
  $X_1 := X$
  $Y_1 := Y$
  init := false

IF $X_1 = 0$ and $\neg stop$ and $\neg init$ then
  stop := true
  $r := X$

IF $Y_1 = 0$ and $\neg stop$ and $\neg init$ then
  stop := true
  $r := Y$

IF $X_1 > 0$ and $\neg stop$ and $\neg init$ then
  $X_1 := X_1 - 1$

IF $Y_1 > 0$ and $\neg stop$ and $\neg init$ then
  $Y_1 := Y_1 - 1$

IF stop then
  skip
Gurevich theorem
Gurevich theorem

- All algorithms are representable by ASM with the good level of abstraction
Gurevich theorem

- All algorithms are representable by ASM with the good level of abstraction
- the level of abstraction is in the first order structure
Gurevich theorem

- All algorithms are representable by ASM with the good level of abstraction
  - the level of abstraction is in the first order structure
- Then ASM are algorithms (formally)
Primitive recursive algorithms (ASM-PR)

- 0-ary variables
- Natural values
- $+1$, $-1$, $=$ (equality)

Arithmetic ASM
Primitive recursive algorithms (ASM-PR)

- 0-ary variables
- Natural values
- $+1, -1, = \text{(equality)}$

we take arithmetic ASM
Primitive recursive algorithms (ASM-PR)

- 0-ary variables
- Natural values
- $+1, -1, = \text{(equality)}$

Arithmetic ASM

- we take arithmetic ASM
- we restrict the length of run to be primitive recursive
Primitive recursive algorithms (ASM-PR)

- 0-ary variables
- Natural values
- +1, -1, = (equality)

Arithmetic ASM

- we take arithmetic ASM
- we restrict the length of run to be primitive recursive
Primitive recursive algorithms (ASM-PR)

- 0-ary variables
- Natural values
- $+1, -1, = \text{(equality)}$

Arithmetic ASM

- we take arithmetic ASM
- we restrict the length of run to be primitive recursive

$\pi = \text{A set of Rules}$

First Order State

with $f$
Primitive recursive algorithms (ASM-PR)

- 0-ary variables
- Natural values
- +1, -1, = (equality)

Arithmetic ASM

- we take arithmetic ASM
- we restrict the length of run to be primitive recursive

$(A, f)$ with $A$ an arithmetic ASM and $f$ a primitive recursive function
Remarks

• the algorithm of minimum function which decrements alternatively its data is definable with an Arithmetic ASM

• the Stein’s gcd algorithm is definable with an Arithmetic ASM
Remarks

• the algorithm of minimum function which decrements alternatively its data is definable with an Arithmetic ASM

• the Stein’s gcd algorithm is definable with an Arithmetic ASM

As we have a set of algorithms for PR functions, we are looking for a programming language that implements all those algorithms
Loop\_exit programming language
Universes: var, natural
val: var → \( \mathbb{N} \cup \{\text{val(undef)}\} \)

\[
\text{natural} → \mathbb{N}
\]

\textbf{succ}: natural → natural
\textbf{pred}: natural → natural

\[
\text{val(succ(e))} \equiv \text{val(e)} + 1
\]
\[
\text{val(pred(e))} \equiv \begin{cases} 
\text{val(e)} - 1 & \text{if val(e) > 0} \\
0 & \text{else}
\end{cases}
\]
Loop_{exit} programming language

\begin{align*}
\text{\textbf{expr}_{int}} &::= \text{var} \mid \text{natural} \mid \textbf{succ}(\text{expr}_{int}) \mid \textbf{pred}(\text{expr}_{int}) \mid \textbf{undef} \\
\text{\textbf{expr}_{bool}} &::= \text{var} = \text{expr}_{int} \mid \neg \text{expr}_{bool} \mid \text{expr}_{bool} \land \text{expr}_{bool} \mid \text{expr}_{bool} \lor \text{expr}_{bool} \mid \text{true} \mid \text{false}
\end{align*}

\begin{align*}
\text{val}(\nu = e) &\equiv \text{val}(\nu) = \text{val}(e) \\
\text{val}(\neg e) &\equiv \neg \text{val}(e) \\
\text{val}(e_1 \land e_2) &\equiv \text{val}(e_1) \land \text{val}(e_2) \\
\text{val}(e_1 \lor e_2) &\equiv \text{val}(e_1) \lor \text{val}(e_2)
\end{align*}
Loop exit programming language

\[\begin{align*}
\text{expr}_{\text{int}} & ::= \text{var} \mid \text{natural} \mid \text{succ}(\text{expr}_{\text{int}}) \mid \text{pred}(\text{expr}_{\text{int}}) \mid \text{undef} \\
\text{expr}_{\text{bool}} & ::= \text{var} = \text{expr}_{\text{int}} \mid \neg \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \land \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \lor \text{expr}_{\text{bool}} \mid \text{true} \mid \text{false} \\
\text{prog} & ::= \text{com}_{\text{list}} \\
\text{com}_{\text{list}} & ::= \text{com} \text{com}_{\text{list}} \mid \varepsilon
\end{align*}\]

Universes : \text{com}_{\text{list}} \to \text{com} \\
head : \text{com}_{\text{list}} \to \text{com} \cup \{\text{undef}\} \\
tail : \text{com}_{\text{list}} \to \text{com}_{\text{list}} \cup \{\text{undef}\} \\
append : \text{com} \times \text{com}_{\text{list}} \to \text{com}_{\text{list}}
Loop\textsubscript{exit} programming language

\begin{align*}
\text{expr}_{\text{int}} &::= \text{var} \mid \text{natural} \mid \text{succ}(\text{expr}_{\text{int}}) \mid \text{pred}(\text{expr}_{\text{int}}) \mid \text{undef} \\
\text{expr}_{\text{bool}} &::= \text{var} = \text{expr}_{\text{int}} \mid \neg \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \land \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \lor \text{expr}_{\text{bool}} \mid \text{true} \mid \text{false} \\
\text{prog} &::= \text{com}_{\text{list}} \\
\text{com}_{\text{list}} &::= \text{com} \text{com}_{\text{list}} \mid \varepsilon \\
\text{com} &::= \text{var} := \text{expr}_{\text{int}} ;
\end{align*}

if (head(p) = (v := e ;)) then
\begin{align*}
\text{val}(v) &:= \text{val}(e) \\
p &:= \text{tail}(p)
\end{align*}
endif
Loop exit programming language

\[ \text{expr}_{\text{int}} ::= \text{var} \mid \text{natural} \mid \text{succ} (\text{expr}_{\text{int}}) \mid \text{pred} (\text{expr}_{\text{int}}) \mid \text{undef} \]

\[ \text{expr}_{\text{bool}} ::= \text{var} = \text{expr}_{\text{int}} \mid \neg \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \land \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \lor \text{expr}_{\text{bool}} \mid \text{true} \mid \text{false} \]

\[ \text{prog} ::= \text{com}_{\text{list}} \]

\[ \text{com}_{\text{list}} ::= \text{com} \text{ com}_{\text{list}} \mid \epsilon \]

\[ \text{com} ::= \text{var} ::= \text{expr}_{\text{int}} \; ; \]

\[ ::= \text{if} \; \text{expr}_{\text{bool}} \; \text{then} \; \text{com}_{\text{list}} \; \text{else} \; \text{com}_{\text{list}} \; \text{endif} \; ; \]

\[
\text{if} \ (\text{head}(p) = (\text{if} \ e \ \text{then} \ \text{com}_{\text{then}} \ \text{else} \ \text{com}_{\text{else}} \ \text{endif} \ ;)) \ \text{then}
\]

\[
\text{if} \ (\text{val}(e) = \text{true}) \ \text{then}
\]

\[
\quad p := \text{append}(\text{com}_{\text{then}}, \text{tail}(p))
\]

\[
\quad \text{endif}
\]

\[
\text{if} \ (\text{val}(e) = \text{false}) \ \text{then}
\]

\[
\quad p := \text{append}(\text{com}_{\text{else}}, \text{tail}(p))
\]

\[
\quad \text{endif}
\]

\[
\text{endif}
\]
Loop exit programming language

\[
\begin{align*}
\text{expr}_{\text{int}} & ::= \text{var} \mid \text{natural} \mid \text{succ}(\text{expr}_{\text{int}}) \mid \text{pred}(\text{expr}_{\text{int}}) \mid \text{undef} \\
\text{expr}_{\text{bool}} & ::= \text{var} = \text{expr}_{\text{int}} \mid \neg \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \land \text{expr}_{\text{bool}} \mid \text{expr}_{\text{bool}} \lor \text{expr}_{\text{bool}} \mid \text{true} \mid \text{false} \\
\text{prog} & ::= \text{com}\_\text{list} \\
\text{com}\_\text{list} & ::= \text{com} \text{com}\_\text{list} \mid \varepsilon \\
\text{com} & ::= \text{var} ::= \text{expr}_{\text{int}} ; \\
& \quad ::= \text{if} \text{expr}_{\text{bool}} \text{then} \text{com}\_\text{list} \text{else} \text{com}\_\text{list} \text{endif} ; \\
& \quad ::= \text{loop} \text{expr}_{\text{int}} \text{do} \text{com}\_\text{list} \text{endloop} ;
\end{align*}
\]

if (head(p) = \text{loop} e \text{do} \text{com} \text{endloop} ;) then
\[
\begin{align*}
\text{if} (\text{val}(e) = 0) \text{then} \\
& \quad \text{p} ::= \text{tail}(p) \\
\text{endif} \\
\text{if} (\text{not} (\text{val}(e) = 0)) \text{then} \\
& \quad \text{p} ::= \text{append}(\text{com}, \text{append}((\text{loop} \text{pred}(e) \text{do} \text{com} \text{endloop} ;) , \text{tail}(p))) \\
\text{endif} \\
\text{endif}
\end{align*}
\]
if (head(p) = exit ;) then
   endFlag := true
endif
Loopexit programming language

\[expr_{\text{int}} ::= \text{var} | \text{natural} | \text{succ}(\text{expr}_{\text{int}}) | \text{pred}(\text{expr}_{\text{int}}) | \text{undef}\]

\[expr_{\text{bool}} ::= \text{var} = \text{expr}_{\text{int}} | \lnot \text{expr}_{\text{bool}} | \text{expr}_{\text{bool}} \land \text{expr}_{\text{bool}} | \text{expr}_{\text{bool}} \lor \text{expr}_{\text{bool}} | \text{true} | \text{false}\]

\[\text{prog} ::= \text{com}_{\text{list}}\]

\[\text{com}_{\text{list}} ::= \text{com} \text{com}_{\text{list}} | \epsilon\]

\[\text{com} ::= \text{var} ::= \text{expr}_{\text{int}} ;
\text{com} ::= \text{if} \text{expr}_{\text{bool}} \text{then} \text{com}_{\text{list}} \text{else} \text{com}_{\text{list}} \text{endif};
\text{com} ::= \text{loop} \text{expr}_{\text{int}} \text{do} \text{com}_{\text{list}} \text{endloop} ;
\text{com} ::= \text{exit} ;
\]

if (head(p) = exit ;) then
  endFlag := true
endif

endFlag then need to appears in each previous conditions
Loop and Loop\textsubscript{exit} compute the same functions (primitive recursive functions)
Loop and Loop_{exit} compute the same functions (primitive recursive functions)

But min(m, n) is computed by Loop_{exit} with the « good » algorithm

\[
\begin{align*}
X_1 & \leftarrow m & X_2 & \leftarrow n \\
\mathcal{P} : & \quad Z_1 := X_1 ; \\
 & \quad Z_2 := X_2 ; \\
 & \quad \text{loop } X_1 \text{ do} \\
 & \quad \quad Z_1 := \text{pred}(Z_1) ; \\
 & \quad \quad Z_2 := \text{pred}(Z_2) ; \\
 & \quad \quad \text{if } Z_1 = 0 \text{ then} \\
 & \quad \quad \quad Y := X_1 ; \\
 & \quad \quad \quad \text{exit} ; \\
 & \quad \quad \quad \text{endif} ; \\
 & \quad \quad \text{if } Z_2 = 0 \text{ then} \\
 & \quad \quad \quad Y := X_2 ; \\
 & \quad \quad \quad \text{exit} ; \\
 & \quad \quad \quad \text{endif} ; \\
& \quad \text{endloop} ;
\end{align*}
\]
Loop and Loop\textsubscript{exit} compute the same functions (primitive recursive functions)

But min(m, n) is computed by Loop\textsubscript{exit} with the « good » algorithm

\begin{align*}
X_1 & \leftarrow m & X_2 & \leftarrow n \\
\mathcal{P} : & \quad \begin{array}{l}
Z_1 := X_1 ; \\
Z_2 := X_2 ; \\
\text{loop } X_1 \text{ do} \\
\quad Z_1 := \text{pred}(Z_1) ; \\
\quad Z_2 := \text{pred}(Z_2) ; \\
\quad \text{if } Z_1 = 0 \text{ then} \\
\qquad Y := X_1 ; \\
\qquad \text{exit} ; \\
\quad \text{endif} ; \\
\quad \text{if } Z_2 = 0 \text{ then} \\
\qquad Y := X_2 ; \\
\qquad \text{exit} ; \\
\quad \text{endif} ; \\
\text{endloop} ;
\end{array}
\end{align*}
The complexity $T_P$ of a Loop$_{exit}$ program (or Loop) $P$ is the complexity of the ASM describing semantics when it is restricted to $P$. 
The complexity $T_P$ of a Loop$_{exit}$ program (or Loop) $P$ is the complexity of the ASM describing semantics when it is restricted to $P$.

Remark

For any PR function $f$ there exists a Loop program $P$ computing $f$ with:

$$T_P \geq f$$
Theorem

Let $f$ be a PR function. To any ASM-PR $(\mathcal{A}, c)$ computing $f$ can be associated a Loop$_{\text{exit}}$ program $P$ also computing $f$ such that: $T_P \in O(c_{\mathcal{A}})$
Sketch of proof

par
    if \( g_1 \) then
        \( R_1 \)
    endif
    ...
    if \( g_n \) then
        \( R_n \)
    endif
endpar

if \( g_1 \) then
    \( P_{R_1} \)
elsif ...
else
    \( P_{R_n} \)
endif ;
Sketch of proof

par
  if \( g_1 \) then
    \( R_1 \)
  endif
...
  if \( g_n \) then
    \( R_n \)
  endif
endpar

\[
\text{if } g_1 \text{ then } \quad P_{R_1} \\
\text{elsif} \\
\text{else} \quad P_{R_n} \\
\text{endif ;}
\]

core-program \( P_A \)
with \( T_{P_A} \leq \text{cst} \)

dimanche 21 juin 2009
Sketch of proof

With:

\[ \text{R}_i: \text{par} \]
\[ x_1 := e_1 \]
\[ \ldots \]
\[ x_k := e_k \]
\[ \text{endpar} \]

\[ \text{P}_{R_i}: x'_1 := x_1 ; \]
\[ x_1 := e_1 ; \]
\[ x'_2 := x_2[x'_1/x_1] ; \]
\[ x_2 := e_2[x'_1/x_1] ; \]
\[ \ldots \]
\[ x'_k := x_k[x'_1/x_1][x'_2/x_2] \ldots [x'_{k-1}/x_{k-1}] ; \]
\[ x_k := e_k[x'_1/x_1][x'_2/x_2] \ldots [x'_{k-1}/x_{k-1}] ; \]

\[ \text{R}_i: \text{skip} \]

\[ \text{P}_{R_i}: \text{exit} ; \]
« min » example

The π of the ASM

The core-program
if \((Y = \text{undef})\) and \((Z_1 \neq 0)\) and \((Z_2 \neq 0)\) then
\[Z_1 := \text{pred}(Z_1)\]
\[Z_2 := \text{pred}(Z_2)\]
endif
if \((Y = \text{undef})\) and \((Z_1 \neq 0)\) and \((Z_2 = 0)\) then
\[Y := X_2\]
endif
if \((Y = \text{undef})\) and \((Z_1 = 0)\) then
\[Y := X_1\]
endif
if \((Y \neq \text{undef})\) then
skip
endif

/* min */ example
The π of the ASM

if \(Y = \text{undef} \land \neg (Z_1 = 0) \land \neg (Z_2 = 0)\) then
\[Z'_1 := Z_1;\]
\[Z_1 := \text{pred}(Z_1);\]
\[Z'_2 := Z_2;\]
\[Z_2 := \text{pred}(Z_2);\]
elsif \(Y = \text{undef} \land \neg (Z_1 = 0) \land Z_2 = 0\) then
\[Y' := Y;\]
\[Y := X_2;\]
elsif \(Y = \text{undef} \land Z_1 = 0\) then
\[Y' := Y;\]
\[Y := X_1;\]
elsif \(Y = \text{undef}\) then
exit;
endif;

The core-program
Sketch of proof (continued)

- insert $P_{\mathcal{A}}$ (the core-program) in a Loop program $\mathcal{Q}$ to execute it as many times as long are the runs in $\mathcal{A}$. 
Sketch of proof (continued)

- insert $P_A$ (the core-program) in a Loop program $Q$ to execute it as many times as long are the runs in $A$.

  How?    In which Loop program?
Sketch of proof (continued)

\[ \text{insert } P_{\mathcal{A}} (\text{the core-program}) \text{ in a Loop program } Q \text{ to execute it as many times as long are the runs in } \mathcal{A}. \]

**How ?**

**In which Loop program ?**

---

How ?
Sketch of proof (continued)

\[ \text{How ?} \quad \text{In which Loop program ?} \]

How ?

\[
\begin{array}{c}
\text{...} \\
v := e ; \quad P_A \\
\text{...}
\end{array}
\]

How ?

\[
\begin{array}{c}
\text{...} \\
v := e ; \\
\text{...}
\end{array}
\]
Sketch of proof (continued)

insert $P_A$ (the core-program) in a Loop program $Q$ to execute it as many times as long are the runs in $A$.

How ?

In which Loop program ?

How ?

How ?
Sketch of proof (continued)

$n$ insert $P_A$ (the core-program) in a Loop program $Q$ to execute it as many times as long are the runs in $A$.

**How?**

**In which Loop program?**

How?

```
\ldots 
\begin{align*}
v &:= e; \\
\cdots &\cdots \\
&\begin{cases}
\text{if } e \text{ then } \\
\text{else } \\
\text{endif }
\end{cases} \\
\cdots &\cdots \\
\end{align*}
\begin{cases}
\text{if } e \text{ then } \\
\text{else } \\
\text{endif }
\end{cases}
\ldots \\
\end{align*}
\begin{align*}
\cdots &\cdots \\
&\begin{cases}
\text{if } e \text{ then } \\
\text{else } \\
\text{endif }
\end{cases} \\
\cdots &\cdots \\
\\ldots &\cdots \\
endloop \\
\ldots \\
\end{align*}
\begin{align*}
\cdots &\cdots \\
&\begin{cases}
\text{if } e \text{ then } \\
\text{else } \\
\text{endif }
\end{cases} \\
\cdots &\cdots \\
\\ldots &\cdots \\
endloop \\
\ldots \\
\end{align*}
\begin{align*}
\cdots &\cdots \\
&\begin{cases}
\text{if } e \text{ then } \\
\text{else } \\
\text{endif }
\end{cases} \\
\cdots &\cdots \\
\\ldots &\cdots \\
endloop \\
\ldots \\
\end{align*}
\begin{align*}
\cdots &\cdots \\
&\begin{cases}
\text{if } e \text{ then } \\
\text{else } \\
\text{endif }
\end{cases} \\
\cdots &\cdots \\
\\ldots &\cdots \\
endloop \\
\ldots \\
\end{align*}
```

(dimanche 21 juin 2009)
Sketch of proof (continued)

- insert $P_A$ (the core-program) in a Loop program $Q$ to execute it as many times as long are the runs in $\mathcal{A}$.

**How?**

**In which Loop program?**

How?

```
... v := e ;
... P_A
... v := e ;
...
```

```
... if e then
... v := e ;
... else
... endif ;
...
```

```
... loop e do
... P_A
... loop e do
... v := e ;
... endloop ;
...
```

```
... if e then
... v := e ;
... else
... endif ;
...
```

```
... endloop ;
...
```

$T_{Q[P]} \leq \text{cst } T_Q$
Sketch of proof (continued)

In which Loop program?
In which Loop program?

\( Q \): a Loop program which computes \( c \) (which is PR since \( A \) computes a PR)
and such that \( T_Q \geq c_A \) (from previous lemma)
In which Loop program?

Q: a Loop program which computes $c$ (which is PR since $A$ computes a PR) and such that $T_Q \geq c_A$ (from previous lemma)

min example (insertion of $P_{min}$ in a loop program for min)

\[
_Y := _X_1 ;
\]
\[
_Z := _X_2 ;
\]

\textbf{loop} \ _X_1 \ \textbf{do}
\[
_Z := \text{pred}(\_Z) ;
\]
\textbf{endloop} ;

if $\_Z = 0$ then

\[
_Y := _X_2 ;
\]
endif;

Sketch of proof (continued)
In which Loop program?

$Q$: a Loop program which computes $c$ (which is PR since $A$ computes a PR) and such that $T_Q \geq c_A$ (from previous lemma)

min example (insertion of $P_{\text{min}}$ in a loop program for min)

\begin{verbatim}
P_{\text{min}}
_Y := _X_1 ;
P_{\text{min}}
_Z := _X_2 ;
P_{\text{min}}
loop _X_1 do
    _Z := pred(_Z) ;
P_{\text{min}}
endloop ;
P_{\text{min}}
if _Z = 0 then
    _Y := _X_2 ;
endif;
\end{verbatim}
Conclusion

- We use ASM to represent algorithms
- We define a class of algorithms for all primitive recursive functions
- We show that all those algorithms are writable in a total programming language: LoopExit
- CLAIM: LoopExit is «the best programming language» for primitive recursive functions

Some questions

- Can we do this with other classes of functions?
- Does there exist another language (functional) equivalent to LoopExit?
- Study class of algorithms rather class of functions?