Characterization theorem for primitive recursive algorithms

Pierre Valarcher, LACL (Paris East, France)

Phiippe 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.

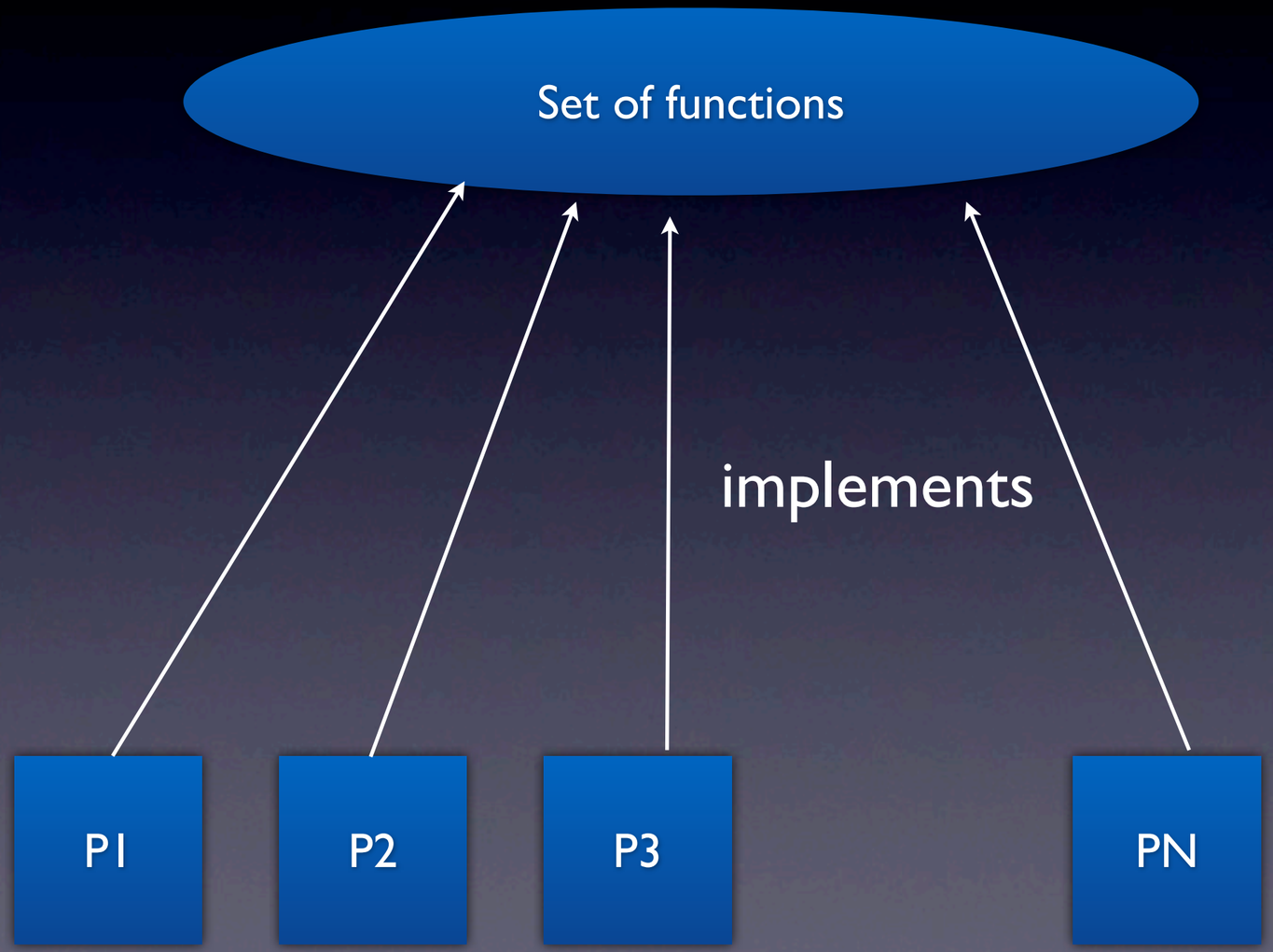
Set of functions

- There are programming languages that compute a set of functions but they are limited from an algorithmic point of view.

Set of functions

P1 P2 P3 PN

- 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



dimanche 21 juin 2009

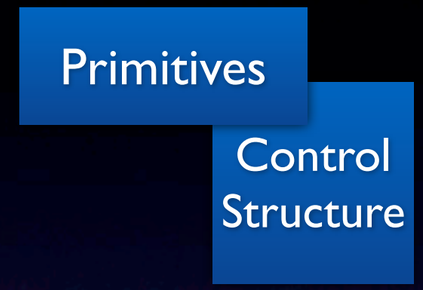
We want to compare programming language : we want to say something like
«P is better than Q»

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

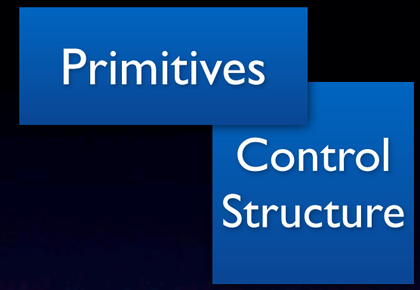
We want to compare programming language : we want to say something like
«P is better than Q»

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



PR

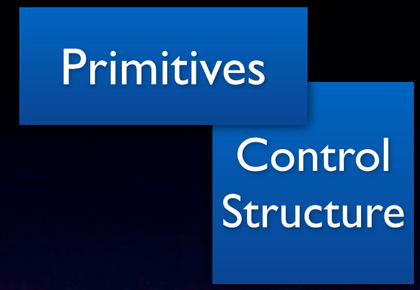
$0, +1, -1, \pi_i$

$\circ$

$$
\begin{array}{lcl}
f(0, Y) & \rightsquigarrow & g(Y) \\
f(x+1, Y) & \rightsquigarrow & h(x, f(x, Y), Y)
\end{array}
$$

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



PR

$0, +1, -1, \pi_i$

$\circ$

$$f(0, Y) \rightsquigarrow g(Y)$$
$$f(x+1, Y) \rightsquigarrow h(x, f(x, Y), Y)$$

PRV

$0, +1, -1, \pi_i$

$\circ$

$$f(0, Y) \rightsquigarrow g(Y)$$
$$f(x+1, Y) \rightsquigarrow h(x, f(x, j(x, Y)), Y)$$

- 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 :
  - R1 ::= if C 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 :
  - R1 ::= if C then R (conditional)
  - R2 ::= f(t1, ..., tN) := t0 (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 ::= if C then R (conditional)
  - R2 ::= f(t1, ..., tN) := t0 (update)
  - R3 ::= Ri || ... || Rk (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 ::= if C then R (conditional)
  - R2 ::= f(t1, ..., tN) := t0 (update)
  - R3 ::= Ri || ... || Rk (simultaneous rules)
  - R4 ::= 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 ::= 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
- We stop when two consecutive states are unchanged

dimanche 21 juin 2009

First Order State

π = A set of Rules

An asm

First Order State

π = A set of Rules

An asm

F = 1st order structure

CS = Rules of ASM

A set of ASM of level of abstraction F

X = n, Y = m, X1, Y1,
stop = false, init = true,
+1, -1, 0, =

```
IF init then
  X1 := X
  Y1 := Y
  init := false
IF X1 = 0 and ¬stop and ¬init then
  stop := true
  r := X
IF Y1 = 0 and ¬stop and ¬init then
  stop := true
  r := Y
IF X1 > 0 and ¬stop and ¬init then
  X1 := X1 - 1
IF Y1 > 0 and ¬stop and ¬init then
  Y1 := Y1 - 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, = (equality)

Arithmetic ASM

- we take arithmetic ASM

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, = (equality)

Arithmetic ASM

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

ASM-PR

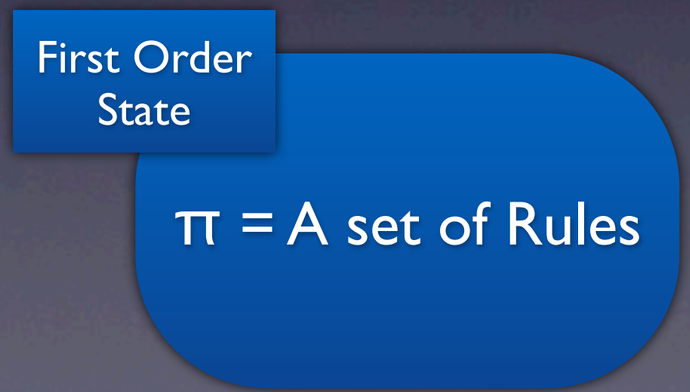
Primitive recursive algorithms (ASM-PR)

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

Arithmetic ASM

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

ASM-PR



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**

ASM-PR

(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

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

Universes: var, natural
val: var → **N** ∪ {val(undef)}
natural → **N**
succ: natural → natural
pred: natural → natural
val(**succ**(e)) ≡ val(e) + 1
val(**pred**(e)) ≡ val(e) – 1 if val(e) > 0
≡ 0 else

$\text{Loop}_{\text{exit}}$ programming language

expr_{int} ::= var | natural | **succ**(expr_{int}) | **pred**(expr_{int}) | **undef**

$\text{expr}_{\text{bool}}$ ::= var = expr_{int} | $\neg$ $\text{expr}_{\text{bool}}$ | $\text{expr}_{\text{bool}} \wedge \text{expr}_{\text{bool}}$ | $\text{expr}_{\text{bool}} \vee \text{expr}_{\text{bool}}$ | **true** | **false**

$\text{val}(v = e) \equiv \text{val}(v) = \text{val}(e)$

$\text{val}(\neg e) \equiv \neg \text{val}(e)$

$\text{val}(e_1 \wedge e_2) \equiv \text{val}(e_1) \wedge \text{val}(e_2)$

$\text{val}(e_1 \vee e_2) \equiv \text{val}(e_1) \vee \text{val}(e_2)$

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

$expr_{bool}$ ::= var = $expr_{int}$ | ¬ $expr_{bool}$ | $expr_{bool}$ ∧ $expr_{bool}$ | $expr_{bool}$ ∨ $expr_{bool}$ | **true** | **false**

prog ::= com_{list}

com_{list} ::= com com_{list} | ε

Universes : com_{list}, com

head : com_{list} → com ∪ {undef}

tail : com_{list} → com_{list} ∪ {undef}

append : com x com_{list} → com_{list}

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

$expr_{bool}$ ::= var **=** $expr_{int}$ | ¬ $expr_{bool}$ | $expr_{bool}$ ∧ $expr_{bool}$ | $expr_{bool}$ ∨ $expr_{bool}$ | **true** | **false**

prog ::= com_{list}

com_{list} ::= com com_{list} | ε

com ::= var **:=** $expr_{int}$;

```
if (head(p) = (v := e ;)) then
    val(v) := val(e)
    p := tail(p)
endif
```

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

$expr_{bool}$ ::= var **=** $expr_{int}$ | ¬ $expr_{bool}$ | $expr_{bool}$ ∧ $expr_{bool}$ | $expr_{bool}$ ∨ $expr_{bool}$ | **true** | **false**

prog ::= com_{list}

com_{list} ::= com com_{list} | ε

com ::= var **:=** $expr_{int}$;
::= **if** $expr_{bool}$ **then** com_{list} **else** com_{list} **endif** ;

```
if (head(p) = (if e then com_then else com_else endif ;)) then
    if (val(e) = true) then
        p := append(com_then, tail(p))
    endif
    if (val(e) = false) then
        p := append(com_else, tail(p))
    endif
endif
```

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

$expr_{bool}$ ::= var **=** $expr_{int}$ | ¬ $expr_{bool}$ | $expr_{bool}$ ∧ $expr_{bool}$ | $expr_{bool}$ ∨ $expr_{bool}$ | **true** | **false**

prog ::= com_{list}

com_{list} ::= com com_{list} | ε

com ::= var **:=** $expr_{int}$;
::= **if** $expr_{bool}$ **then** com_{list} **else** com_{list} **endif** ;
::= **loop** $expr_{int}$ **do** com_{list} **endloop** ;

```
if (head(p) = loop e do com endloop ;) then
    if (val(e) = 0) then
        p := tail(p)
    endif
    if (not(val(e) = 0)) then
        p := append(com, append((loop pred(e) do com endloop ;), tail(p))
   endif
endif
```

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

$expr_{bool}$ ::= var **=** $expr_{int}$ | ¬ $expr_{bool}$ | $expr_{bool}$ ∧ $expr_{bool}$ | $expr_{bool}$ ∨ $expr_{bool}$ | **true** | **false**

prog ::= com_{list}

com_{list} ::= com com_{list} | ε

com ::= var **:=** $expr_{int}$;
::= **if** $expr_{bool}$ **then** com_{list} **else** com_{list} **endif** ;
::= **loop** $expr_{int}$ **do** com_{list} **endloop** ;
::= **exit** ;

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

$Loop_{exit}$ programming language

$expr_{int}$ ::= var | natural | **succ**($expr_{int}$) | **pred**($expr_{int}$) | **undef**

$expr_{bool}$ ::= var **=** $expr_{int}$ | ¬ $expr_{bool}$ | $expr_{bool}$ ∧ $expr_{bool}$ | $expr_{bool}$ ∨ $expr_{bool}$ | **true** | **false**

prog ::= com_{list}

com_{list} ::= com com_{list} | ε

com ::= var **:=** $expr_{int}$;
::= **if** $expr_{bool}$ **then** com_{list} **else** com_{list} **endif** ;
::= **loop** $expr_{int}$ **do** com_{list} **endloop** ;
::= **exit** ;

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

endFlag then need to appears in each previous conditions

- Loop and $Loop_{exit}$ 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

$X_1 \leftarrow m$ $X_2 \leftarrow n$

$\mathcal{P}$:

```
Z1 := X1 ;
Z2 := X2 ;
loop X1 do
   Z1 := pred(Z1) ;
   Z2 := pred(Z2) ;
   if Z1 = 0 then
      Y := X1 ;
      exit ;
   endif ;
   if Z2=0 then
      Y := X2 ;
      exit ;
   endif ;
endloop ;
```

- Loop and $Loop_{exit}$ compute the same functions (primitive recursive functions)
- But min(m, n) is computed by $Loop_{exit}$ with the « good » algorithm

$X_1 \leftarrow m$ $X_2 \leftarrow n$

$\mathcal{P}$:

```
Z1 := X1 ;
Z2 := X2 ;
loop X1 do
   Z1 := pred(Z1) ;
   Z2 := pred(Z2) ;
   if Z1 = 0 then
      Y := X1 ;
      exit ;
   endif ;
   if Z2=0 then
      Y := X2 ;
      exit ;
   endif ;
endloop ;
```

The complexity $T_{\mathcal{P}}$ of a $\text{Loop}_{\text{exit}}$ program (or Loop) $\mathcal{P}$ is the complexity of the ASM describing semantics when it is restricted to $\mathcal{P}$

The complexity $T_{\mathcal{P}}$ of a $Loop_{exit}$ program (or Loop) $\mathcal{P}$ is the complexity of the ASM describing semantics when it is restricted to $\mathcal{P}$

Remark

For any PR function f there exists a Loop program $\mathcal{P}$ computing f with :

$$T_{\mathcal{P}} \geq f$$

dimanche 21 juin 2009

Theorem

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

Sketch of proof

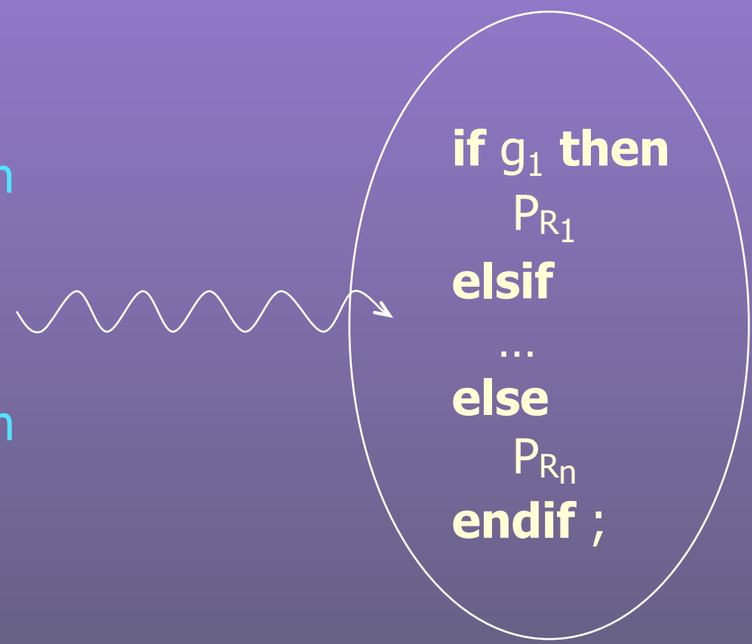
```
par
    if g1 then
        R1
    endif
    ...
    if gn then
        Rn
    endif
endpar
```



```
if g1 then
    PR1
elsif
    ...
else
    PRn
endif ;
```

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 ;

core-program $\mathcal{P}_{\mathcal{A}}$
with $T_{\mathcal{P}_{\mathcal{A}}} \leq$ cst

Sketch of proof

par
  if g_1 then
    R_1
  endif
  …
  if g_n then
    R_n
  endif



if g_1 **then**
  P_{R_1}
elsif
  …
else
  P_{R_n}
endif ;
endpar

With:

R_i: par
  $x_1 := e_1$
  …
  $x_k := e_k$
endpar



P_{R_i}: x'_1 **:=** x_1 ;
  x_1 **:=** e_1 ;
  x'_2 **:=** $x_2[x'_1/x_1]$;
  x_2 **:=** $e_2[x'_1/x_1]$;
  …
  x'_k **:=** $x_k[x'_1/x_1]…[x'_{k-1}/x_{k-1}]$;
  x_k **:=** $e_k[x'_1/x_1]…[x'_{k-1}/x_{k-1}]$;

R_i: skip



P_{R_i}: **exit** ;

« min » example

The π of the ASM

The core-program

« min » example

```
if (Y = undef) and (Z1 ≠ 0) and (Z2 ≠ 0) then
    Z1 := pred(Z1)
    Z2 := pred(Z2)
endif
if (Y = undef) and (Z1 ≠ 0) and (Z2 = 0) then
    Y:= X2
endif
if (Y = undef) and (Z1 = 0) then
    Y := X1
endif
if (Y ≠ undef) then
    skip
endif
```

The π of the ASM

```
if Y = undef ∧ ¬(Z1 = 0) ∧ ¬(Z2 = 0) then
    Z'1 := Z1 ;
    Z1 := pred(Z1) ;
    Z'2 := Z2 ;
    Z2 := pred(Z2) ;
elsif Y = undef ∧ ¬(Z1 = 0) ∧ Z2 = 0 then
    Y' := Y ;
    Y := X2 ;
elsif Y = undef ∧ Z1 = 0 then
    Y' := Y;
    Y := X1 ;
elsif ¬(Y = undef) then
    exit ;
endif ;
```

The core-program

Sketch of proof (continued)

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

Sketch of proof (continued)

↳ insert $\mathcal{P}_{\mathcal{A}}$ (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 ?**

Sketch of proof (continued)

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

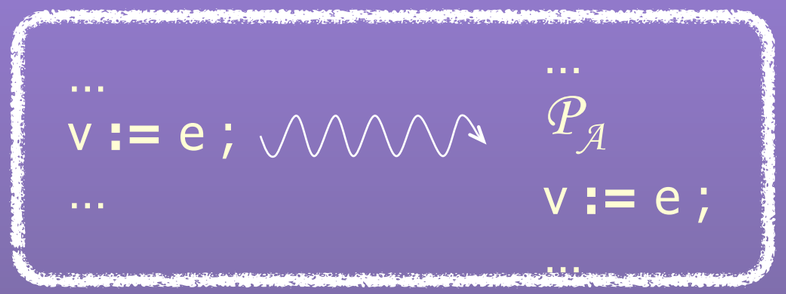
How ? **In which Loop program ?**

How ?

↳ insert $\mathcal{P}_{\mathcal{A}}$ (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 ?**

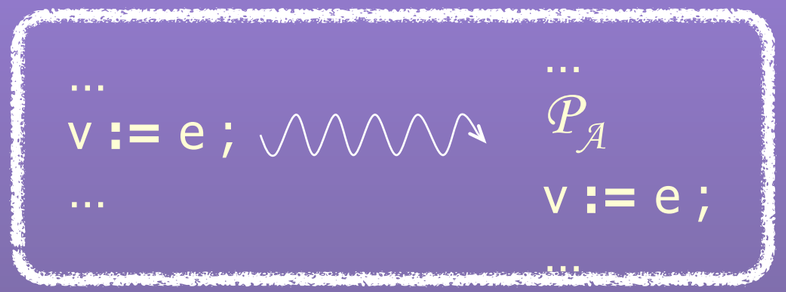
How ?

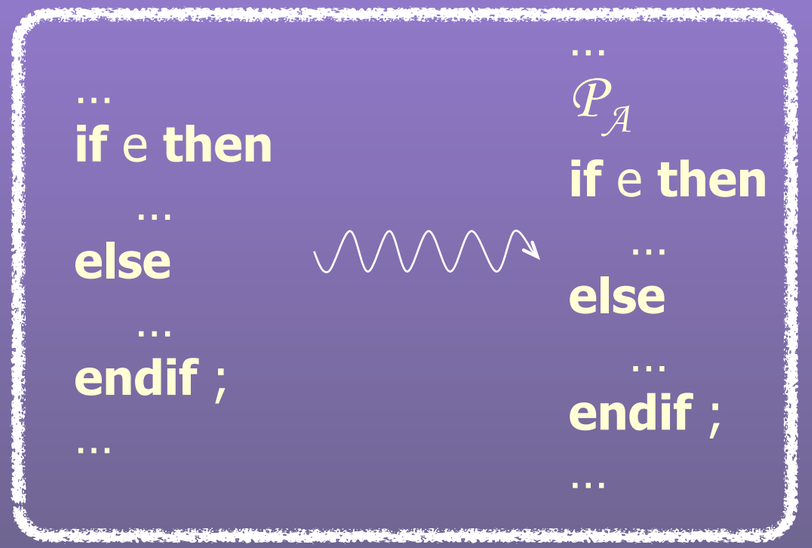


↳ insert $\mathcal{P}_{\mathcal{A}}$ (the core-program) in a Loop program Q to execute it as many times as long are the runs in $\mathcal{A}$.

How ? **In which Loop program ?**

How ?

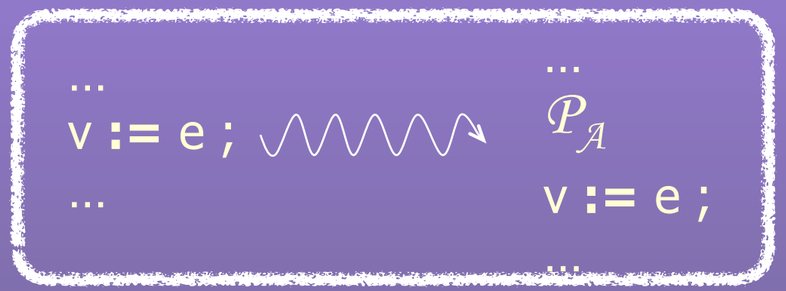


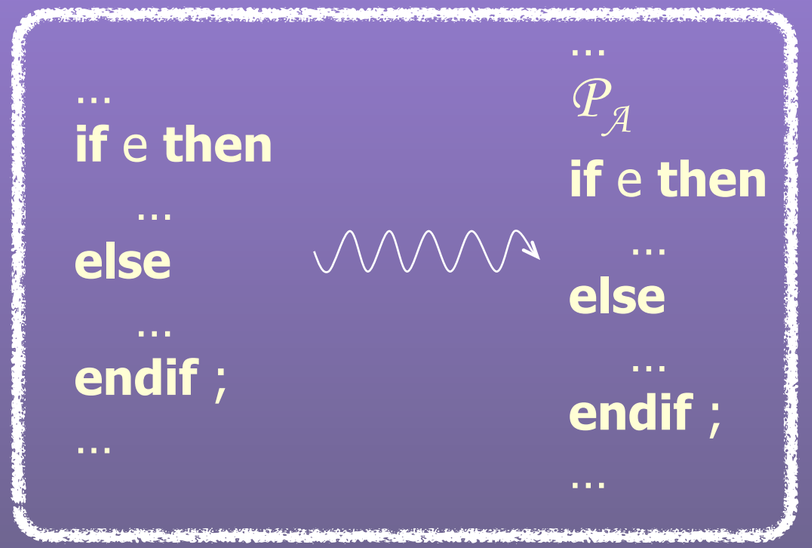


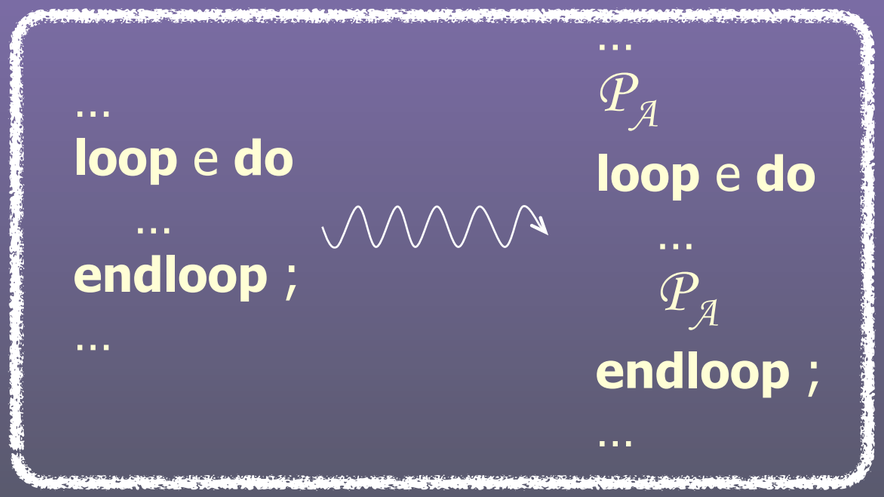
↳ insert $\mathcal{P}_{\mathcal{A}}$ (the core-program) in a Loop program Q to execute it as many times as long are the runs in $\mathcal{A}$.

How ? **In which Loop program ?**

How ?



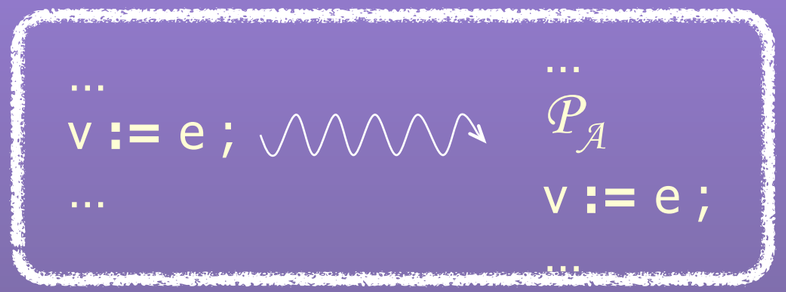


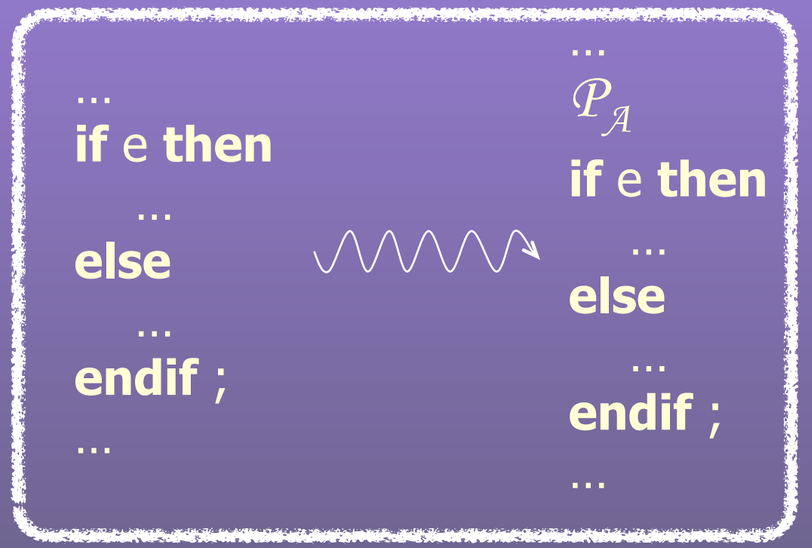


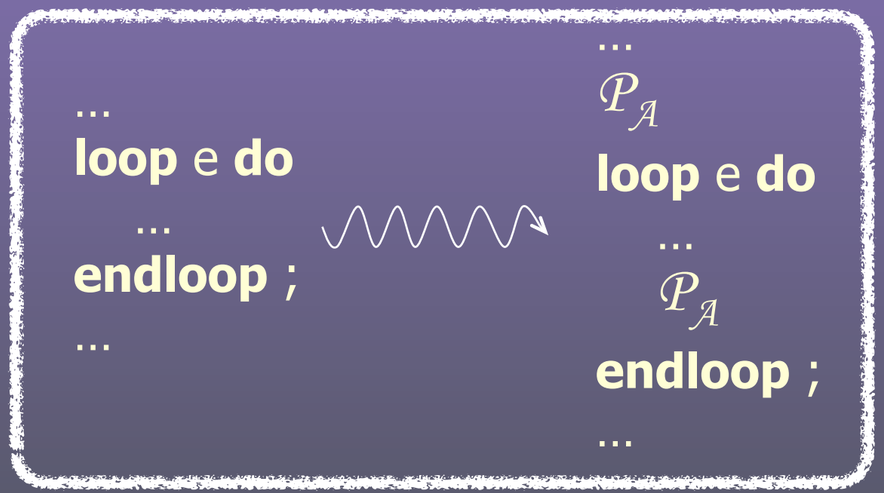
↳ insert $\mathcal{P}_{\mathcal{A}}$ (the core-program) in a Loop program Q to execute it as many times as long are the runs in $\mathcal{A}$.

How ? **In which Loop program ?**

How ?







$$T_{Q[\mathcal{P}]} \leq \text{cst} \times T_Q$$

Sketch of proof (continued)

In which Loop program ?

Sketch of proof (continued)

In which Loop program ?

$\mathcal{Q}$: a Loop program which computes $\mathcal{C}$ (which is PR since $\mathcal{A}$ computes a PR)

and such that $T_{\mathcal{Q}} \geq c_{\mathcal{A}}$ (from previous lemma)

Sketch of proof (continued)

In which Loop program ?

$\mathcal{Q}$: a Loop program which computes $\mathcal{C}$ (which is PR since $\mathcal{A}$ computes a PR)

and such that $T_{\mathcal{Q}} \geq c_{\mathcal{A}}$ (from previous lemma)

min example (insertion of $\mathcal{P}_{min}$ in a loop program for min)

_Y **:=** $_X_1$;

_Z **:=** $_X_2$;

loop $_X_1$ **do**
    _Z := pred(_Z) ;

endloop ;

if _Z = 0 then

    _Y := $_X_2$;
endif;

Sketch of proof (continued)

In which Loop program ?

$\mathcal{Q}$: a Loop program which computes $\mathcal{C}$ (which is PR since $\mathcal{A}$ computes a PR)

and such that $T_{\mathcal{Q}} \geq c_{\mathcal{A}}$ (from previous lemma)

min example (insertion of $\mathcal{P}_{min}$ in a loop program for min)

$\mathcal{P}_{min}$
_Y **:=** $_X_1$;

$\mathcal{P}_{min}$
_Z **:=** $_X_2$;

$\mathcal{P}_{min}$
loop $_X_1$ **do**
    _Z := pred(_Z) ;
    $\mathcal{P}_{min}$
endloop ;

$\mathcal{P}_{min}$
if _Z = 0 then
    $\mathcal{P}_{min}$
    _Y := $_X_2$;
endif;

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 ?