Mathematics for Informatics

Carlos Areces and Patrick Blackburn

areces@loria.fr blackbur@loria.fr

http://www.loria.fr/~areces http://www.loria.fr/~blacbur

INRIA Lorraine
Nancy, France

2007/2008

Composition

Let $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$. $h : \mathbb{N}^n \to \mathbb{N}$ is obtained from f and $g_1, \ldots, g_k$ by composition if

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

Composition

Let $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$. $h : \mathbb{N}^n \to \mathbb{N}$ is obtained from f and $g_1, \ldots, g_k$ by composition if

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

Theorem

If h is obtained from the (partially) computables functions $f, g_1, \ldots, g_k$ by composition then h is (partially) computable.

Composition

Let $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$. $h : \mathbb{N}^n \to \mathbb{N}$ is obtained from f and $g_1, \ldots, g_k$ by composition if

$$h(x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_k(x_1, \ldots, x_n))$$

Theorem

If h is obtained from the (partially) computables functions $f, g_1, \ldots, g_k$ by composition then h is (partially) computable.

Proof.

The following program computes h:

$$\begin{aligned} Z_1 &\leftarrow g_1(X_1, \ldots, X_n) \\ &\vdots \\ Z_k &\leftarrow g_k(X_1, \ldots, X_n) \\ Y &\leftarrow f(Z_1, \ldots, Z_k) \end{aligned}$$

If $f, g_1, \ldots, g_k$ are total then h is total. $\square$

Recursion for $h : \mathbb{N} \to \mathbb{N}$

$h : \mathbb{N} \to \mathbb{N}$ is obtained from $g : \mathbb{N}^2 \to \mathbb{N}$ by primitive recursion if

$$\begin{aligned} h(0) &= k \\ h(t+1) &= g(t, h(t)) \end{aligned}$$

Recursion for $h : \mathbb{N} \to \mathbb{N}$

$h : \mathbb{N} \to \mathbb{N}$ is obtained from $g : \mathbb{N}^2 \to \mathbb{N}$ by primitive recursion if

$$\begin{aligned} h(0) &= k \\ h(t+1) &= g(t, h(t)) \end{aligned}$$

Theorem

If h is obtained from g by primitive recursion and g is computable then h is computable.

Recursion for $h : \mathbb{N} \to \mathbb{N}$

$h : \mathbb{N} \to \mathbb{N}$ is obtained from $g : \mathbb{N}^2 \to \mathbb{N}$ by primitive recursion if

$$\begin{aligned} h(0) &= k \\ h(t+1) &= g(t, h(t)) \end{aligned}$$

Theorem

If h is obtained from g by primitive recursion and g is computable then h is computable.

Proof.

The following program computes h:

$$\begin{array}{ll} & Y \leftarrow k \quad \text{(is a macro, it's easy to do)} \\ [A] & \text{IF } X = 0 \text{ GOTO } E \quad \text{(another macro, IF with =)} \\ & Y \leftarrow g(Z, Y) \\ & Z \leftarrow Z + 1 \\ & X \leftarrow X - 1 \\ & \text{GOTO } A \end{array}$$

If $f, g_1, \ldots, g_k$ are total then h is total. $\square$

Recursion for $h : \mathbb{N}^{n+1} \to \mathbb{N}$

$h : \mathbb{N}^{n+1} \to \mathbb{N}$ is obtained from $g : \mathbb{N}^{n+2} \to \mathbb{N}$ and $f : \mathbb{N}^{n} \to \mathbb{N}$ by primitive recursion if

$$
\begin{aligned}
h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\
h(x_1, \ldots, x_n, t+1) &= g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n)
\end{aligned}
$$

Recursion for $h : \mathbb{N}^{n+1} \to \mathbb{N}$

$h : \mathbb{N}^{n+1} \to \mathbb{N}$ is obtained from $g : \mathbb{N}^{n+2} \to \mathbb{N}$ and $f : \mathbb{N}^n \to \mathbb{N}$ by primitive recursion if

$$\begin{aligned} h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n, t+1) &= g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n) \end{aligned}$$

Theorem

If h is obtained from f and g by primitive recursion and f and g are computable then h is computable.

Another way to characterized computable functions?

Let's combine computable functions with the schemes of

- composition
- primitive recursion

to obtain new computable functions.

Another way to characterized computable functions?

Let's combine computable functions with the schemes of

- composition
- primitive recursion

to obtain new computable functions.

- We defined computable functions as the total functions where can be programmed in the simple imperative functiosn $\mathscr{S}$.

Another way to characterized computable functions?

Let's combine computable functions with the schemes of

- composition
- primitive recursion

to obtain new computable functions.

- We defined computable functions as the total functions where can be programmed in the simple imperative functiosn $\mathscr{S}$.
- Let's now think in a simple functional language (i.e., with a very restricted recursion)
  - Can we define computable functions though the schemes of composition and primitive recursion instead or using $\mathscr{S}$?
  - in other words, can we characterize computable functions using a simple functional language?

Another way to characterized computable functions?

Let's combine computable functions with the schemes of

- composition
- primitive recursion

to obtain new computable functions.

- We defined computable functions as the total functions where can be programmed in the simple imperative functiosn $\mathscr{S}$.
- Let's now think in a simple functional language (i.e., with a very restricted recursion)
  - Can we define computable functions though the schemes of composition and primitive recursion instead or using $\mathscr{S}$?
  - in other words, can we characterize computable functions using a simple functional language?

We need initial functions. Let's try with

- $s(x) = x + 1$
- $n(x) = 0$
- proyections: $u_i^n(x_1, \ldots, x_n) = x_i$ for $i \in \{1, \ldots, n\}$

PRC Classes

A class $\mathcal{C}$ of total functions is PRC (primitive recursive closed) if

1. the initial functions are in $\mathcal{C}$
2. if a function f is obtained from other functions already in $\mathcal{C}$ using composition and primitive recursion, then f is also in $\mathcal{C}$.

PRC Classes

A class $\mathcal{C}$ of total functions is PRC (primitive recursive closed) if

1. the initial functions are in $\mathcal{C}$
2. if a function f is obtained from other functions already in $\mathcal{C}$ using composition and primitive recursion, then f is also in $\mathcal{C}$.

Theorem

the class of computable functions is a PRC class.

PRC Classes

A class $\mathcal{C}$ of total functions is PRC (primitive recursive closed) if

1. the initial functions are in $\mathcal{C}$
2. if a function f is obtained from other functions already in $\mathcal{C}$ using composition and primitive recursion, then f is also in $\mathcal{C}$.

Theorem

the class of computable functions is a PRC class.

Proof.

We already saw 2.

PRC Classes

A class $\mathcal{C}$ of total functions is PRC (primitive recursive closed) if

1. the initial functions are in $\mathcal{C}$
2. if a function f is obtained from other functions already in $\mathcal{C}$ using composition and primitive recursion, then f is also in $\mathcal{C}$.

Theorem

the class of computable functions is a PRC class.

Proof.

We already saw 2. Let's see 1:

- $s(x) = x + 1$ can be computed with the program

$$Y \leftarrow X + 1$$

PRC Classes

A class $\mathcal{C}$ of total functions is PRC (primitive recursive closed) if

1. the initial functions are in $\mathcal{C}$
2. if a function f is obtained from other functions already in $\mathcal{C}$ using composition and primitive recursion, then f is also in $\mathcal{C}$.

Theorem

the class of computable functions is a PRC class.

Proof.

We already saw 2. Let's see 1:

- $s(x) = x + 1$ can be computed with the program

$$Y \leftarrow X + 1$$

- $n(x) = 0$ is computed with the empty program

PRC Classes

A class $\mathcal{C}$ of total functions is PRC (primitive recursive closed) if

1. the initial functions are in $\mathcal{C}$
2. if a function f is obtained from other functions already in $\mathcal{C}$ using composition and primitive recursion, then f is also in $\mathcal{C}$.

Theorem

the class of computable functions is a PRC class.

Proof.

We already saw 2. Let's see 1:

- $s(x) = x + 1$ can be computed with the program

$$Y \leftarrow X + 1$$

- $n(x) = 0$ is computed with the empty program
- $u_i^n(x_1, \ldots, x_n) = x_i$ is computed with the program

$$Y \leftarrow X_i$$

$\square$

Primitive Recursive Functions

A function is primitive recursive (p.r.) if it can be obtained from initial functions by means of a finite number of applications of composition and primitive recursion.

Theorem

a function is p.r. iff it belongs to any PRC.

Primitive Recursive Functions

A function is primitive recursive (p.r.) if it can be obtained from initial functions by means of a finite number of applications of composition and primitive recursion.

Theorem

a function is p.r. iff it belongs to any PRC.

Proof.

($\Leftarrow$) The class of p.r. functions is a PRC class. Hence, if f is in any PRC class, in particular f is p.r.

Primitive Recursive Functions

A function is primitive recursive (p.r.) if it can be obtained from initial functions by means of a finite number of applications of composition and primitive recursion.

Theorem

a function is p.r. iff it belongs to any PRC.

Proof.

($\Leftarrow$) The class of p.r. functions is a PRC class. Hence, if f is in any PRC class, in particular f is p.r.

($\Rightarrow$) Let f be p.r. and let $\mathcal{C}$ be a PRC class. As f is p.r., there is a list

$$f_1, f_2, \ldots, f_n$$

such that

- $f = f_n$
- f_i is initial (hence it is in $\mathcal{C}$) or it is obtained by composition or primitive recursion from functions f_j, $j < i$ (hence it is also in $\mathcal{C}$).

But then, all the functions in the list are in $\mathcal{C}$.

Computable Functions = Primitive Recursive Functions?

Ok then, the class of p.r. functions is the smallest PRC class.

Computable Functions = Primitive Recursive Functions?

Ok then, the class of p.r. functions is the smallest PRC class.

Corollary

Every p.r. function is computable.

Proof.

We already proved that the class of computable functions is PRC. By the previous tehorem, if f is p.r., then f is in the class of computable functions. □

Computable Functions = Primitive Recursive Functions?

Ok then, the class of p.r. functions is the smallest PRC class.

Corollary

Every p.r. function is computable.

Proof.

We already proved that the class of computable functions is PRC. By the previous tehorem, if f is p.r., then f is in the class of computable functions. $\square$

Of course, no every partially computable function is p.r. because every p.r. function is total. But...

Is every computable function p.r.?

Example of a p.r. function

Addition $add(x, y) = x + y$ is p.r.

Let's think how do we write it in a functional language

Example of a p.r. function

Addition $add(x, y) = x + y$ is p.r.

Let's think how do we write it in a functional language

```
add ::  nat -> nat
add x y = x + y
```

But let's not use the $+$ function; just $+1$

Example of a p.r. function

Addition $add(x, y) = x + y$ is p.r.

Let's think how do we write it in a functional language

```
add ::  nat -> nat
add x y = x + y
```

But let's not use the $+$ function; just $+1$

```
add ::  nat -> nat
add x 0 = x
add x (y+1) = (add x y) + 1
```

Example of a p.r. function

Addition $add(x, y) = x + y$ is p.r.

Let's think how do we write it in a functional language

```
add ::  nat -> nat
add x y = x + y
```

But let's not use the $+$ function; just $+1$

```
add ::  nat -> nat
add x 0 = x
add x (y+1) = (add x y) + 1
```

We can rewrite this as

$$\begin{aligned} add(x, 0) &= u_1^1(x) \\ add(x, y+1) &= g(y, add(x, y), x) \end{aligned}$$

where

$$g(x_1, x_2, x_3) = s(u_2^3(x_1, x_2, x_3))$$

Other p.r. functions

- $x \cdot y$
- $x!$
- x^y
- $x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$
- $\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$

Other p.r. functions

- $x \cdot y$
- $x!$
- x^y
- $x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y \\ 0 & \text{otherwise} \end{cases}$
- $\alpha(x) = \begin{cases} 1 & \text{if } x = 0 \\ 0 & \text{otherwise} \end{cases}$
- and many others. Are they all the computable functions?

Primitive recursive predicates

Predicates are just functions that take values in $\{0, 1\}$.

- 1 is interpreted as true.
- 0 is interpreted as false.

p.r. predicates are those which are represented by p.r. functions in $\{0, 1\}$.

Primitive recursive predicates

Predicates are just functions that take values in $\{0, 1\}$.

- 1 is interpreted as true.
- 0 is interpreted as false.

p.r. predicates are those which are represented by p.r. functions in $\{0, 1\}$.

For example, the predicate $x \leq y$ is p.r. because it can be defined as

$$\alpha(x \dot{-} y)$$

Logical operators

Theorem

Let $\mathcal{C}$ be a PRC class. If p and q are predicates in $\mathcal{C}$ then $\neg p$, $p \wedge q$ and $p \vee q$ are in $\mathcal{C}$.

Logical operators

Theorem

Let $\mathcal{C}$ be a PRC class. If p and q are predicates in $\mathcal{C}$ then $\neg p$, $p \wedge q$ and $p \vee q$ are in $\mathcal{C}$.

Proof.

- $\neg p$ can be defined as $\alpha(p)$
- $p \wedge q$ can be defined as $p \cdot q$
- $p \vee q$ can be defined as $\neg(\neg p \wedge \neg q)$

$\square$

Logical operators

Theorem

Let $\mathcal{C}$ be a PRC class. If p and q are predicates in $\mathcal{C}$ then $\neg p$, $p \wedge q$ and $p \vee q$ are in $\mathcal{C}$.

Proof.

- $\neg p$ can be defined as $\alpha(p)$
- $p \wedge q$ can be defined as $p \cdot q$
- $p \vee q$ can be defined as $\neg(\neg p \wedge \neg q)$

$\square$

Corollary

If p and q are p.r. predicates, then also $\neg p$, $p \vee q$ and $p \wedge q$ are p.r. predicates.

Logical operators

Theorem

Let $\mathcal{C}$ be a PRC class. If p and q are predicates in $\mathcal{C}$ then $\neg p$, $p \wedge q$ and $p \vee q$ are in $\mathcal{C}$.

Proof.

- $\neg p$ can be defined as $\alpha(p)$
- $p \wedge q$ can be defined as $p \cdot q$
- $p \vee q$ can be defined as $\neg(\neg p \wedge \neg q)$

$\square$

Corollary

If p and q are p.r. predicates, then also $\neg p$, $p \vee q$ and $p \wedge q$ are p.r. predicates.

Corollary

If p and q are computable predicates, then also $\neg p$, $p \vee q$ y $p \wedge q$ are computable predicates.

Definition by cases (2)

Theorem

Let $\mathcal{C}$ be a PRC class. Let $h, g : \mathbb{N}^n \to \mathbb{N}$ be functions in $\mathcal{C}$ and let $p : \mathbb{N}^n \to \{0, 1\}$ be a predicate in $\mathcal{C}$. Then the function

$$f(x_1, \ldots, x_n) = \begin{cases} g(x_1, \ldots, x_n) & \text{if } p(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

is in $\mathcal{C}$.

Proof.

$f(x_1, \ldots, x_n) =$
$g(x_1, \ldots, x_n) \cdot p(x_1, \ldots, x_n) + h(x_1, \ldots, x_n) \cdot \alpha(p(x_1, \ldots, x_n))$ $\square$

Definition by cases ($m+1$)

Theorem

Let $\mathcal{C}$ be a PRC class. Let $g_1, \ldots, g_m, h : \mathbb{N}^n \to \mathbb{N}$ be functions in $\mathcal{C}$ and let $p_1, \ldots, p_m : \mathbb{N}^n \to \{0,1\}$ be predicates in $\mathcal{C}$. Then the function

$$f(x_1, \ldots, x_n) = \begin{cases} g_1(x_1, \ldots, x_n) & \text{if } p_1(x_1, \ldots, x_n) \\ \vdots & \\ g_m(x_1, \ldots, x_n) & \text{if } p_m(x_1, \ldots, x_n) \\ h(x_1, \ldots, x_n) & \text{otherwise} \end{cases}$$

is in $\mathcal{C}$.

Primitive recursion

- we have not yet answer the question: p.r. = computable?
- and we can still not answer it

Observe that the recursion scheme

$$\begin{aligned} h(x_1,\ldots,x_n,0) &= f(x_1,\ldots,x_n) \\ h(x_1,\ldots,x_n,t+1) &= g(t,h(x_1,\ldots,x_n,t),x_1,\ldots,x_n) \end{aligned}$$

is very simple:

Primitive recursion

- we have not yet answer the question: p.r. = computable?
- and we can still not answer it

Observe that the recursion scheme

$$
\begin{aligned}
h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\
h(x_1, \ldots, x_n, t+1) &= g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n)
\end{aligned}
$$

is very simple:

- recursion is always done in the last parameter

Primitive recursion

- we have not yet answer the question: p.r. = computable?
- and we can still not answer it

Observe that the recursion scheme

$$
\begin{aligned}
h(x_1, \ldots, x_n, 0) &= f(x_1, \ldots, x_n) \\
h(x_1, \ldots, x_n, t+1) &= g(t, h(x_1, \ldots, x_n, t), x_1, \ldots, x_n)
\end{aligned}
$$

is very simple:

- recursion is always done in the last parameter
- the variant function of $h(x_1, \ldots, x_n, x_{n+1})$ is x_{n+1}

For-programs

If we go back to the imperative language known as Pascal, the p.r. functions are the onces that can be written using for-loops:

- the only type of looks are of the form

```
for i=1 to x {
    S(i)
}
```

- it is not the kind of cicles we have in $\mathscr{S}$ (which are rather `while` loops)

For-programs

If we go back to the imperative language known as Pascal, the p.r. functions are the onces that can be written using for-loops:

- the only type of looks are of the form

```
for i=1 to x {
    S(i)
}
```

- it is not the kind of cicles we have in $\mathscr{S}$ (which are rather `while` loops)

We know that:

- $\mathscr{S}$ can simulate `while`
- `for` loops can be simulated using `while`
- any for-program can be rewritten in $\mathscr{S}$

For-programs

If we go back to the imperative language known as Pascal, the p.r. functions are the onces that can be written using for-loops:

- the only type of looks are of the form

```
for i=1 to x {
    S(i)
}
```

- it is not the kind of cicles we have in $\mathscr{S}$ (which are rather `while` loops)

We know that:

- $\mathscr{S}$ can simulate `while`
- `for` loops can be simulated using `while`
- any for-program can be rewritten in $\mathscr{S}$

Questions:

- Any $\mathscr{S}$ program can be rewritten as a for-program?
- Is primitive recursion as expressive as general recursion in a functional language?
- computable = primitive recursive?