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

The halting problem

$\text{HALT}(x, y)$ is true iff the program with number y is not indefined when run with the number x , i.e.

$$\text{HALT}(x, y) = \begin{cases} 1 & \text{if } \Psi_P^{(1)}(x) \downarrow \\ 0 & \text{otherwise} \end{cases}$$

where P is the unique program such that $\#(P) = y$.

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

It is clear that

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if HALT}(x, x) \\ 0 & \text{otherwise} \end{cases}$$

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

It is clear that

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if HALT}(x, x) \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $\#(P) = e$.

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

It is clear that

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if HALT}(x, x) \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $\#(P) = e$. By definition of HALT,

$$\text{HALT}(x, e) \quad \text{iff} \quad P(x) \text{ halts}$$

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

It is clear that

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if HALT}(x, x) \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $\#(P) = e$. By definition of HALT,

$$\text{HALT}(x, e) \quad \text{iff} \quad P(x) \text{ halts} \quad \text{iff} \quad \neg\text{HALT}(x, x)$$

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

It is clear that

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if HALT}(x, x) \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $\#(P) = e$. By definition of HALT,

$$\text{HALT}(x, e) \quad \text{iff} \quad P(x) \text{ halts} \quad \text{iff} \quad \neg\text{HALT}(x, x)$$

e is fixed; x is variable.

HALT is not computable

Theorem

HALT *is not computable.*

Proof.

Suppose that it is. We can build the following program P:

$$[A] \qquad \text{IF HALT}(X, X) = 1 \text{ GOTO } A$$

It is clear that

$$\Psi_P^{(1)}(x) = \begin{cases} \uparrow & \text{if HALT}(x, x) \\ 0 & \text{otherwise} \end{cases}$$

Suppose that $\#(P) = e$. By definition of HALT,

$$\text{HALT}(x, e) \quad \text{iff} \quad P(x) \text{ halts} \quad \text{iff} \quad \neg\text{HALT}(x, x)$$

e is fixed; x is variable. In particular, for $x = e$:

$$\text{HALT}(e, e) \quad \text{iff} \quad P(e) \text{ halts} \quad \text{iff} \quad \neg\text{HALT}(e, e)$$

Church's Thesis

There are many different computation models.
It has been proved that they have the same power than $\mathscr{S}$

- C
- Java
- Haskell
- Turing machines
- ...

Church's Thesis

There are many different computation models.
It has been proved that they have the same power than $\mathscr{S}$

- C
- Java
- Haskell
- Turing machines
- ...

Church's Thesis. All algorithms to compute functions in the natural numbers can be programmed in $\mathscr{S}$.

Church's Thesis

There are many different computation models.
It has been proved that they have the same power than $\mathscr{S}$

- C
- Java
- Haskell
- Turing machines
- ...

Church's Thesis.All algorithms to compute functions in the natural numbers can be programmed in $\mathscr{S}$.

Hence, the halting problem says

> *there is no algorithm to decide the truth of falsity of* HALT(x, y)

Universality

For each $n > 0$ we define

$$\Phi^{(n)}(x_1, \ldots, x_n, e) \quad = \quad \text{the output of the program } e \text{ with input } x_1, \ldots, x$$

Universality

For each $n > 0$ we define

$$\begin{aligned}\Phi^{(n)}(x_1, \ldots, x_n, e) &= \text{the output of the program } e \text{ with input } x_1, \ldots, x_n \\ &= \Psi_P^{(n)}(x_1, \ldots, x_n) \qquad \text{where } \#(P) = e\end{aligned}$$

Universality

For each $n > 0$ we define

$$\begin{aligned}\Phi^{(n)}(x_1, \ldots, x_n, e) &= \text{the output of the program } e \text{ with input } x_1, \ldots, x_n \\ &= \Psi_P^{(n)}(x_1, \ldots, x_n) \qquad \text{where } \#(P) = e\end{aligned}$$

Theorem

For each $n > 0$ the function $\Phi^{(n)}$ is partially computable.

Observe that the program for $\Phi^{(n)}$ is a program interpreter.
I.e., it interprets the numerical encoding of programs.

To show the theorem we will build the program U_n that computes $\Phi^{(n)}$.

U_n: an idea

U_n is the program that computes

$$\begin{aligned}\Phi^{(n)}(x_1, \ldots, x_n, e) &= \text{output of the program } e \text{ with input } x_1, \ldots, x_n \\ &= \Psi_P^{(n)}(x_1, \ldots, x_n) \qquad \text{where } \#(P) = e\end{aligned}$$

U_n needs

- know who is P (decodifying e)
- keep trac of the states of P at each step
  - it starts from the initial state of P when the input is $x_1, \ldots, x_n$
  - it codifyins the states as lists
  - For example $Y = 0, X_1 = 2, X_2 = 1$ is codified as $[0, 2, 0, 1] = 63$

In the code of U_n

- K indicates the number of the instruction that we are about to execute (in the simulation of P)
- S describe the state of P in each instant

Initialization

$//$ input $= x_1, \ldots, x_n, e$

$//$ $\#(P) = e = [i_1, \ldots, i_m] - 1$

$Z \leftarrow X_{n+1} + 1$

$//$ $Z = [i_1, \ldots, i_m]$

$$S \leftarrow \prod_{j=1}^{n} (p_{2j})^{X_j}$$

$//$ $S = [0, X_1, 0, X_2, \ldots, 0, X_n]$ is the initial state

$K \leftarrow 1$

$//$ the first instruction of P that we should analyze is 1

Main Cycle

// S codifies the state, K is the instruction number

// $Z = [i_1, \ldots, i_m]$

[C] IF $K = |Z| + 1 \vee K = 0$ GOTO F

// if I'm at the end, then finish (we will see $K = 0$ later)

// otherwise, let $Z[K] = i_K = \langle a, \langle b, c \rangle \rangle$

$U \leftarrow r(Z[k])$

// $U = \langle b, c \rangle$

$P \leftarrow p_{r(U)+1}$

// the variable that appears in i_K is the $c + 1$-th

// P is the prime for the variable that appears in i_K

Main Cycle (cont.)

// S codifies the state, K is the instruction number

// $Z = [i_1, \ldots, i_m], i_K = \langle a, \langle b, c\rangle\rangle, U = \langle b, c\rangle$

// P is the prime for the variable V that appears in i_K

IF $l(U) = 0$ GOTO N

// if it is the instruction $V \leftarrow V$ we go to N

IF $l(U) = 1$ GOTO S

// if it is the instruction $V \leftarrow V + 1$ we go to S

// otherwise, it is of the form $V \leftarrow V - 1$ or IF $V \neq 0$ GOTO L

IF $\neg(P|S)$ GOTO N

// if P divides S (i.e. V=0), jump to N

IF $l(U) = 2$ GOTO R

// $V \neq 0$ and it is the instruction $V \leftarrow V - 1$ jump to R

Case IF $V \neq 0$ GOTO L y $V \neq 0$

$//$ S codifies the state, K is the instruction number

$//$ $Z = [i_1, \ldots, i_m], i_K = \langle a, \langle b, c \rangle\rangle, U = \langle b, c \rangle$

$//$ P is the prime for the variable V that appears in i_K

$//$ $V \neq 0$ and it is the instruction IF $V \neq 0$ GOTO L

$//$ $b \geq 2$, and hence L is the $b - 2$-th label

$$K \leftarrow \min_{j \leq |Z|} (l(Z[j]) + 2 = l(U))$$

$//$ K is the first instruction with label L

$//$ if there is no such instruction then, $K = 0$ (go out of the cicle)

GOTO C

$//$ goes to the first instruction in the main cycle

Case R (Substraccion)

//	S codifies the state, K is the instruction number
//	$Z = [i_1, \ldots, i_m], i_K = \langle a, \langle b, c\rangle\rangle, U = \langle b, c\rangle$
//	P is the prime for the variable V that appears in i_K
//	we are considering $V \leftarrow V - 1$ with $V \neq 0$
[R]	$S \leftarrow S$ div P
	GOTO N
//	S=new state of P (substract 1 to V) and jumps to N

Caso S (Addition)

//	S codifies the state, K is the instruction number
//	$Z = [i_1, \ldots, i_m], i_K = \langle a, \langle b, c \rangle \rangle, U = \langle b, c \rangle$
//	P is the prime for the variable V that appears in i_K
//	we are considering $V \leftarrow V + 1$
$[S]$	$S \leftarrow S \cdot P$
	GOTO N
//	S=new state of P (adds 1 a V) and jumps to N

Case N (Nil)

//	S codifies the state, K is the instruction number
//	$Z = [i_1, \ldots, i_m], i_K = \langle a, \langle b, c \rangle \rangle, U = \langle b, c \rangle$
//	P is the prime for the variable V that appears in i_K
//	the instruction does not change the state
$[N]$	$K \leftarrow K + 1$
	GOTO C
//	S is unchanged
//	K goes to the next instruction
//	back to the main cycle

Returning the result

//	S codifies the final state of P
//	we are living teh main cycle
$[F]$	$Y \leftarrow S[1]$
//	Y=the value of the variable Y when P halts

Everything together

$$
\begin{array}{ll}
 & Z \leftarrow X_{n+1} + 1 \\
 & S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i} \\
 & K \leftarrow 1 \\
[C] & \text{IF } K = |Z| + 1 \vee K = 0 \text{ GOTO } F \\
 & U \leftarrow r(Z[k]) \\
 & P \leftarrow p_{r(U)+1} \\
 & \text{IF } l(U) = 0 \text{ GOTO } N \\
 & \text{IF } l(U) = 1 \text{ GOTO } S \\
 & \text{IF } \neg(P|S) \text{ GOTO } N \\
 & \text{IF } l(U) = 2 \text{ GOTO } R \\
 & K \leftarrow \min_{i \leq |Z|} (l(Z[i]) + 2 = l(U)) \\
 & \text{GOTO } C
\end{array}
$$

$$
\begin{array}{ll}
[R] & S \leftarrow S \text{ div } P \\
 & \text{GOTO } N \\
[S] & S \leftarrow S \cdot P \\
 & \text{GOTO } N \\
[N] & K \leftarrow K + 1 \\
 & \text{GOTO } C \\
[F] & Y \leftarrow S[1]
\end{array}
$$

Notation

Sometimes we write

$$\Phi_e^{(n)}(x_1, \ldots, x_n) = \Phi^{(n)}(x_1, \ldots, x_n, e)$$

Sometimes we drop the superindex when $n = 1$

$$\Phi_e(x) = \Phi(x, e) = \Phi^{(1)}(x, e)$$

Step Counter

Let's define

$\mathrm{STP}^{(n)}(x_1, \ldots, x_n, e, t)$	iff	program e halts in t or less steps with input $x_1, \ldots, x_n$
	iff	there is a computation of program e of length $\leq t + 1$, when started with input $x_1, \ldots, x_n$

Theorem

For each $n > 0$, the predicate $\mathrm{STP}^{(n)}(x_1, \ldots, x_n, e, t)$ is p.r.

Snapshot

Let's define

$$\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, e, t) \quad = \quad \begin{array}{l}\text{representation of the instant configuration} \\ \text{of the program } e \\ \text{with input } x_1, \ldots, x_n \text{ in step } t\end{array}$$

The instant configuration can be represented by

$$\langle \text{instruction number}, \text{list representing the state} \rangle$$

Theorem

For each $n > 0$, the predicate $\mathrm{SNAP}^{(n)}(x_1, \ldots, x_n, e, t)$ is p.r.

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

Let's define $f : \mathbb{N} \to \mathbb{N}, f(x) = \tilde{\Phi}_x(x) + 1$

- f is computable because $\tilde{\Phi}$ is computable

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

Let's define $f : \mathbb{N} \to \mathbb{N}, f(x) = \tilde{\Phi}_x(x) + 1$

- f is computable because $\tilde{\Phi}$ is computable
- f is not p.r. because suppose it is p.r.
  - there would be e such that $\tilde{\Phi}_e = f$

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

Let's define $f : \mathbb{N} \to \mathbb{N}, f(x) = \tilde{\Phi}_x(x) + 1$

- f is computable because $\tilde{\Phi}$ is computable
- f is not p.r. because suppose it is p.r.
  - there would be e such that $\tilde{\Phi}_e = f$
  - and then $\tilde{\Phi}_e(x) = f(x) = \tilde{\Phi}_x(x) + 1$

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

Let's define $f : \mathbb{N} \to \mathbb{N}, f(x) = \tilde{\Phi}_x(x) + 1$

- f is computable because $\tilde{\Phi}$ is computable
- f is not p.r. because suppose it is p.r.
  - there would be e such that $\tilde{\Phi}_e = f$
  - and then $\tilde{\Phi}_e(x) = f(x) = \tilde{\Phi}_x(x) + 1$
  - e is fixed but x is variable

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

Let's define $f : \mathbb{N} \to \mathbb{N}, f(x) = \tilde{\Phi}_x(x) + 1$

- f is computable because $\tilde{\Phi}$ is computable
- f is not p.r. because suppose it is p.r.
  - there would be e such that $\tilde{\Phi}_e = f$
  - and then $\tilde{\Phi}_e(x) = f(x) = \tilde{\Phi}_x(x) + 1$
  - e is fixed but x is variable
  - instantiating $x = e$, $\tilde{\Phi}_e(e) = f(e) = \tilde{\Phi}_e(e) + 1$

A computable function which is not primitive recursive

- we can codify programs of $\mathscr{S}$ with constructors and proyectors which are p.r.
- we can codify the definitions of p.r. functions with constructors and proyectors p.r.
- There is $\Phi_e^{(n)}(x_1, \ldots, x_n)$ partially computable that simulates the e-th program with input $x_1, \ldots, x_n$
- There is $\tilde{\Phi}_e^{(n)}(x_1, \ldots, x_n)$ computable that simulates the e-th p.r. function with input $x_1, \ldots, x_n$.

Let's define $f : \mathbb{N} \to \mathbb{N}, f(x) = \tilde{\Phi}_x(x) + 1$

- f is computable because $\tilde{\Phi}$ is computable
- f is not p.r. because suppose it is p.r.
  - there would be e such that $\tilde{\Phi}_e = f$
  - and then $\tilde{\Phi}_e(x) = f(x) = \tilde{\Phi}_x(x) + 1$
  - e is fixed but x is variable
  - instantiating $x = e$, $\tilde{\Phi}_e(e) = f(e) = \tilde{\Phi}_e(e) + 1$
- this same prove shows that $\tilde{\Phi}$ is not p.r.

The Ackermann function (1928)

$$A(x,y,z) = \begin{cases} y+z & \text{if } x=0 \\ 0 & \text{if } x=1 \text{ and } z=0 \\ 1 & \text{if } x=2 \text{ y } z=0 \\ A(x-1,y,A(x,y,z-1)) & \text{if } x,z>0 \end{cases}$$

- $A_0(y,z) = A(0,y,z) = y+z$
- $A_1(y,z) = A(1,y,z) = y \cdot z$
- $A_2(y,z) = A(2,y,z) = y \uparrow z$
- $A_3(y,z) = A(3,y,z) = y \uparrow\uparrow z$
- $\ldots$

$A : \mathbb{N}^3 \to \mathbb{N}$ is not p.r. but for each i, $A_i : \mathbb{N}^2 \to \mathbb{N}$ is p.r.

Version of Robinson & Peter (1948)

$$B(m,n) = \begin{cases} n+1 & \text{if } m = 0 \\ B(m-1,1) & \text{if } m > 0 \text{ y } n = 0 \\ B(m-1, B(m, n-1)) & \text{if } m > 0 \text{ y } n > 0 \end{cases}$$

- $B_0(n) = B(0,n) = n+1$
- $B_1(n) = A(1,n) = 2 + (n+3) - 3$
- $B_2(n) = A(2,n) = 2 \cdot (n+3) - 3$
- $B_3(n) = A(3,n) = 2 \uparrow (n+3) - 3$
- $B_4(n) = A(4,n) = 2 \uparrow\uparrow (n+3) - 3$
- $\dots$

$B : \mathbb{N}^2 \to \mathbb{N}$ is not p.r. but each $B_i : \mathbb{N} \to \mathbb{N}$ is p.r.

A and B grow faster than any p.r. function.