Recursive Functions

We have seen that we can use TMs to compute number-theoretic functions $f: \mathbb{N}^k \longrightarrow \mathbb{N}$

According to the Church-Turing Thesis every algorithmically computable function can be computed by a TM.

Question: Which are the Turing-computable functions?

We study now functions themselves, rather than characterizing them through TMs. And we will outline an effective method for computing such functions.

We will study two important classes of functions:

1) primitive recursive functions (studied by Gödel (1931) and Kleene (1936))
2) μ-recursive functions

$\updownarrow$

TMs

Primitive recursive functions:

Definition: The class of primitive recursive functions (PRFs) is defined inductively

- base cases: The following are PRFs

1) The successor function S: $S(x) = x+1$, $\forall x \in \mathbb{N}$
2) The zero function Z: $Z(x) = 0$, $\forall x \in \mathbb{N}$
3) The projection functions $\pi_i^{(n)}$:

$$\pi_i^{(n)}(x_1, \dots, x_n) = x_i \quad \text{for } i \in \{1, \dots, n\}$$

$\forall x_1, \dots, x_n \in \mathbb{N}$

- inductive cases: PRFs are obtained from other PRFs through: 4) composition
  5) primitive recursion

Definition of composition of number-theoretic functions

Let $g_i : \mathbb{N}^k \to \mathbb{N}$ for $i \in \{1, \dots, n\}$

$h : \mathbb{N}^n \to \mathbb{N}$

Then $f : \mathbb{N}^k \to \mathbb{N}$ defined by

$$f(x_1, \dots, x_k) = h(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$$

is called the composition of h with $g_1, \dots, g_n$

We write also $f = h \circ (g_1, \dots, g_n)$.

The composition $f(x_1, \dots, x_k)$ is undefined if either

- $g_i(x_1, \dots, x_k) \uparrow$ for some $i \in \{1, \dots, n\}$, or
- $g_i(x_1, \dots, x_k) = y_i \neq \uparrow$ for $i \in \{1, \dots, n\}$ and $h(y_1, \dots, y_n) \uparrow$

If g_i, for $i \in \{1, \dots, n\}$ and h are PRFs, then also $f = h \circ (g_1, \dots, g_n)$ is a PRF.

Examples:

- One function $C_1(x) = 1$ for all x:

  $C_1 = S \circ Z$ $\qquad$ $C_1(x) = S(Z(x))$

- By repeated composition we can obtain a function C_i with $C_i(x) = i$ for every number i:

  $$C_i = \underbrace{S \circ S \circ \cdots \circ S}_{i\text{-times}} \circ Z$$

- $C_i^{(n)}(x_1, \dots, x_n) = i$ defined as $C_i^{(n)} = \underbrace{S \circ \cdots \circ S}_{i\text{-times}} \circ Z \circ P_1^{(n)}$

Definition of primitive recursion

Let $g : \mathbb{N}^n \to \mathbb{N}$

$h : \mathbb{N}^{n+2} \to \mathbb{N}$ be total functions

Then $f : \mathbb{N}^{n+1} \to \mathbb{N}$ defined by

$$\begin{cases} f(x_1, \dots, x_n, 0) = g(x_1, \dots, x_n) \\ f(x_1, \dots, x_n, y+1) = h(x_1, \dots, x_n, y, f(x_1, \dots, x_n, y)) \end{cases}$$

is said to be defined from g and h by primitive recursion: $f = PR[g, h]$

x_i ... parameters of the definition

y ... recursive variable

Notice that from the definition we get an algorithm for computing f (assuming g and h are computable).

Consider a fixed set of parameters $x_1, \dots, x_n$

- $f(x_1, \dots, x_n, 0)$ is obtained from $g(x_1, \dots, x_n)$
- $f(x_1, \dots, x_n, y+1)$ is obtained from h using
  - the parameters $x_1, \dots, x_n$
  - the previous value y of the recursive variable
  - the previous value $f(x_1, \dots, x_n, y)$ of f itself

Notice that the computation of f corresponds to an iterative process.

Note: a function defined by composition and primitive recursion from total functions is also total

Examples of PRFs:

- identity: $id(x) = x$ $\qquad id = \uparrow_1^{(1)}$
- addition: $add : \mathbb{N}^2 \to \mathbb{N}$ $\qquad add(x, y) = x + y$

$$\begin{cases} add(x, 0) = g(x) \\ add(x, y+1) = h(x, y, add(x, y)) \end{cases}$$

we take: $g(x) = id(x)$ $\qquad g = \uparrow_1^{(1)}$

$h(x, y, z) = s(z)$ $\qquad h = s \circ \uparrow_3^{(3)}$

$$\begin{aligned} add(5, 3) &= add(5, 2+1) = \\ &= s(add(5, 2)) = s(s(add(5, 1))) = \\ &= s(s(s(add(5, 0)))) = s(s(s(id(5)))) = \\ &= 8 \end{aligned}$$

Exercise: show that multiplication is a PRF.

We can consider a zero-argument function as a constant. Then we can define a one-argument function using $h : \mathbb{N}^2 \to \mathbb{N}$ and PR:

$$\begin{cases} f(0) = n_0 \quad \text{with } n_0 \in \mathbb{N} \\ f(y+1) = h(y, f(y)) \end{cases}$$

Example: factorial $\qquad fact(y) = \begin{cases} 1 & \text{if } y > 0 \\ \prod_{i=1}^{y} i & \text{if } y > 0 \end{cases}$

$$\begin{cases} fact(0) = 1 \\ fact(y+1) = h(y, fact(y)) = s(y) \cdot fact(y) \end{cases}$$

$h = mult \circ (s \circ \uparrow_1^{(2)}, \uparrow_2^{(2)})$

Theorem: Every PRF is Turing computable

Proof: We have to show that we can construct TMs that

1) compute the basic functions
2) compute the composition $h \circ (g_1, \dots, g_m)$
3) compute primitive recursion from g and h

For (1) and (2), see lab-exercise 4.

We show now (3), i.e. that Turing computable functions are closed under PR.

Let $g : \mathbb{N}^n \to \mathbb{N}$ be Turing-computable $\rightsquigarrow$ TM G
$h : \mathbb{N}^{n+2} \to \mathbb{N}$ $\rightsquigarrow$ TM H

and let f be defined from g, h by PR.

We show that also f is Turing-computable, i.e. we construct a TM F computing f.

The computation of $f(x_1, \dots, x_n, y)$ starts with

$\triangleright \bar{x}_1 \triangleright \bar{x}_2 \triangleright \dots \triangleright \bar{x}_n \triangleright \bar{y}$
↑
head

($\bar{x}$ denotes the unary representation of x)

1) a counter set to 0 is written to the right of the input

$\triangleright \bar{x}_1 \dots \bar{x}_n \triangleright \bar{y} \triangleright \bar{0} \triangleright$

2) the parameters are copied to the right of the counter

$\triangleright \bar{x}_1 \dots \bar{x}_n \triangleright \bar{y} \triangleright \bar{0} \triangleright \bar{x}_1 \triangleright \dots \triangleright \bar{x}_n \triangleright$

3) G is run on the final n values

$\triangleright \bar{x}_1 \dots \bar{x}_n \triangleright \bar{y} \triangleright \bar{0} \triangleright \bar{z}_0 \triangleright$

with $z_0 = g(x_1, \dots, x_n) = f(x_1, \dots, x_n, 0)$

4) The TM F starts now a loop, where at the i-th iteration the tape contains

$$\not b\, \overline{x_1}\, \not b \cdots \not b\, \overline{x_m}\, \not b\, \overline{y}\, \not b\, \overline{i}\, \not b\, \overline{z_i}\, \not b$$

with $z_i = f(x_1, \dots, x_m, i)$ for $i \in \{0, \dots, y\}$

i is compared with y

- if they are equal, $\overline{z_i}$ is moved to the beginning of the tape and the computation finishes
- if $i < y$, then the tape is transformed to

$$\not b\, \overline{x_1}\, \not b \cdots \not b\, \overline{x_m}\, \not b\, \overline{y}\, \not b\, \overline{i+1}\, \not b\, \overline{x_1}\, \not b \cdots \not b\, \overline{x_m}\, \not b\, \overline{i+1}\, \not b\, \overline{z_i}\, \not b$$

and H is run on the last $m+2$ values, producing

$$\not b\, \overline{x_1}\, \not b \cdots \not b\, \overline{x_m}\, \not b\, \overline{y}\, \not b\, \overline{i+1}\, \not b\, \overline{z_{i+1}}\, \not b$$

with $z_{i+1} = f(x_1, \dots x_m, i+1)$

Then the next loop iteration is performed q.e.d.

17/11/2015

Some PRFs we are going to use:

- predecessor $\begin{cases} pred(0) = 0 \\ pred(y+1) = y \end{cases}$
- proper subtraction $sub(x, y)$ $\quad x \dot{-} y$ $\quad \begin{cases} sub(x, 0) = x \\ sub(x, y+1) = pred(sub(x, y)) \end{cases}$

Predicates are PRF that return 0, representing false, or 1 ... true

- sign $\begin{cases} sg(0) = 0 \\ sg(y+1) = 1 \end{cases}$ sign complement $\begin{cases} \overline{sg}(0) = 1 \\ \overline{sg}(y+1) = 0 \end{cases}$

- less than $lt(x,y) = sg(y \dot{-} x)$
  greater than $gt(x,y) = sg(x \dot{-} y)$
- equal to $eq(x,y) = cosg(lt(x,y) + gt(x,y))$
  not equal to $neq(x,y) = lt(x,y) + gt(x,y)$

Boolean operators: for r_1, r_2 boolean values (i.e. 0 or 1)

$not(r_1) = cosg(r_1)$

r_1 and r_2 $= r_1 \cdot r_2$

r_1 or r_2 $= sg(r_1 + r_2)$

We can use eq to specify the value of a function for a finite set of values

e.g.
$$f(x) = \begin{cases} 2 & \text{if } x = 0 \\ 5 & \text{if } x = 1 \\ x & \text{otherwise} \end{cases}$$

$$f(x) = eq(x,0) \cdot 2 + eq(x,1) \cdot 5 + gt(x,1) \cdot x$$

We can generalise this

<u>Theorem</u>: Let g be PRF and f a total function identical to g for all but a finite number of input values. Then f is a PRF.

Proof: exercise

<u>Theorem</u>: Let $g(x,y)$ be a PRF. Then the following functions obtained from g are also PRF

- $f(x, y, z_1, \dots, z_n) = g(x,y)$ ... adding dummy variables
- $f(x,y) = g(y,x)$ ... permuting variables
- $f(x) = g(x,x)$ ... identifying variables

Proof: exercise

Bounded operators:

We can easily construct a PRF that e.g. sums a fixed number of arguments.

What about a sum of a variable number of arguments,

e.g. $f(y) = \sum_{i=0}^{y} g(i) = g(0) + g(1) + \cdots + g(y)$

Let $\vec{x}$ denote $x_1, \ldots, x_n$

$f(\vec{x}, y) = \sum_{i=0}^{y} g(\vec{x}, i)$ is PR if g is PR.

Indeed $\begin{cases} f(\vec{x}, 0) = g(\vec{x}, 0) \\ f(\vec{x}, y+1) = f(\vec{x}, y) + g(\vec{x}, y+1) \end{cases}$

Similarly for the bounded product $\prod_{i=0}^{y} g(\vec{x}, i)$

Bounded minimization:

$$f(\vec{x}, y) = \mu z \leq y \, [r(\vec{x}, z)]$$

$$= \begin{cases} z & \text{if } r(\vec{x}, i) = 0 \text{ for } 0 \leq i < z \leq y \\ & \quad \text{and } r(\vec{x}, z) = 1 \\ y+1 & \text{otherwise} \end{cases}$$

i.e. the minimum $z \leq y$ satisfying $r(\vec{x}, z)$
if such a z exists, and
$y+1$ otherwise

Can be considered as a bounded search for z.

Theorem: Let $r(\vec{x}, y)$ be a PRF

Then $f(\vec{x}, y) = \mu z \leq y \, [r(\vec{x}, z)]$ is a PRF

Proof:

We define

$$g(\vec{x}, i) = \begin{cases} 1 & \text{if } \varphi(\vec{x}, j) = 0 \text{ for } 0 \le j \le i \\ 0 & \text{otherwise} \end{cases}$$

$$= \prod_{j=0}^{i} \operatorname{cosg}(\varphi(\vec{x}, j))$$

$g(\vec{x}, i)$ is PR since it is the bounded product of $\operatorname{cosg} \circ \varphi$.

Let us consider the bounded sum of $g(\vec{x}, i)$

e.g.			
$\varphi(\vec{x}, 0) = 0$	$g(\vec{x}, 0) = 1$	$\sum_{i=0}^{0} g(\vec{x}, i) = 1$
$\varphi(\vec{x}, 1) = 0$	$g(\vec{x}, 1) = 1$	$\sum_{i=0}^{1} \ldots = 2$
$\varphi(\vec{x}, 2) = 1$	$g(\vec{x}, 2) = 0$	$\sum_{i=0}^{2} \ldots = 2$
$\varphi(\vec{x}, 3) = 0$	$g(\vec{x}, 3) = 0$	$\sum_{i=0}^{3} \ldots = 2$
$\varphi(\vec{x}, 4) = 1$	$g(\vec{x}, 4) = 0$	$\sum_{i=0}^{4} \ldots = 2$
⋮	⋮	⋮

Let z be the first number with $\varphi(\vec{x}, z) = 1$.

Then $$g(\vec{x}, i) = \begin{cases} 1 & \text{for } i < z \\ 0 & \text{for } i \ge z \end{cases}$$

Hence $$\sum_{i=0}^{y} g(\vec{x}, i) = \begin{cases} y+1 & \text{for } y < z \\ z & \text{otherwise} \end{cases}$$

(Note that the first case covers also the case where z does not exist.)

We get that $$f(\vec{x}, y) = \mu z \le y [\varphi(\vec{x}, y)] = \sum_{i=0}^{y} g(\vec{x}, i)$$

Hence bounded minimization is PR.

Exercise: Show that the following are PRFs:

1) $f_1(x, y_0, y)$ = the first value z in $[y_0, y]$ for which $\varphi(x, z)$ is true
2) $f_2(x, y)$ = the second value z in $[0, y]$ — " —
3) $f_3(x, y)$ = the largest value z in $[0, y]$ — " —

If there is no value z in the range s.t. $\varphi(x, z)$ is true, then f_i is $y+1$.

Exercise: Consider integer division $div(x,y)$.

$div(x,y)$ is not defined for $y=0$, hence not total, hence not PR.

Let $quo(x,y) = \begin{cases} 0 & \text{if } y=0 \\ div(x,y) & \text{if } y \neq 0 \end{cases}$

1) Define $quo(x,y)$ using bounded minimization
2) Show that remainder, divides, number of divisors, and prime are PRFs.

We have shown that every PRF is Turing computable. This shows that many useful functions that can easily be defined as PRFs can be computed by a TM (without explicitly constructing the TM).

What about computations that should return more than a single value, or that should take as input a variable number of parameters (e.g. a sorting algorithm)?

We introduce now a coding scheme to encode a sequence of numbers in a single number (and that is PR).

Gödel numbering

makes use of the unique decomposition of a natural number into a product of primes.

Consider $x_0, x_1, \dots, x_{n-1}$ ... n natural numbers

It is encoded as $\uparrow_0^{x_0+1} \cdot \uparrow_1^{x_1+1} \cdot \dots \cdot \uparrow_{n-1}^{x_{n-1}+1} = 2^{x_0+1} \cdot 3^{x_1+1} \cdot \dots \cdot \uparrow_{n-1}^{x_{n-1}+1}$

where $\uparrow_i$ is the i-th prime.

E.g. $1,2 \rightsquigarrow 2^2 \cdot 3^3 = 108$

$0,1,3 \rightsquigarrow 2^1 \cdot 3^2 \cdot 5^4 = 11250$

$0,1,0,1 \rightsquigarrow 2^1 \cdot 3^2 \cdot 5^1 \cdot 7^2 = 4410$

... Gödel number of $x_0, \dots, x_{n-1}$

Note: we use x_i+1 (instead of x_i) to make sure that $\uparrow_i$ appears also for $x_i = 0$.

can we compute the encoding of a sequence of numbers?

- we can compute the i-th prime by a PRF $pr(i)$

Exercise: show that $pr(i)$ is a PRF

- define the PRFs: quotient(x, y)
  remainder(x, y)
  divides(x, y) (is a predicate)
  ndivisors(x) (number of divisors)
  prime(x)

- define $pr(i)$ using - prime(x)
  - bounded minimization
  - primitive recursion

  and exploiting the fact that $pr(i+1) \leq pr(i)! + 1$

- function that encodes a sequence of numbers of fixed length:

$$pr_k(x_0, \dots, x_k) = pr(0)^{x_0+1} \cdot \dots \cdot pr(k)^{x_k+1} = \prod_{i=0}^{k} pr(i)^{x_i+1}$$

- given a number x encoding a sequence of numbers $x_0, \dots, x_k$, we can extract x_i from x by means of a decoding function

$$dec(i, x) = \mu z \leq x \left[not\left(divides\left(x, pr(i)^{z+1}\right)\right)\right] \dot{-} 1$$

Note: $dec(i, x)$ returns 0 for every $pr(i)$ that does not occur in the prime decomposition of x.

Hence, using PRFs, we can go back and forth between sequences of numbers and their encodings.

When defining a function through PR, in the recursive case we make use of the result of the function on the previous value of the recursive variable.

What if we need more than one previous value of the function? (possibly all previous values)

Example: Fibonacci numbers 0 1 1 2 3 5 8 13 ...

$$\begin{cases} f(0) = 0 \\ f(1) = 1 \\ f(y+1) = f(y) + f(y-1) & \text{for } y \geq 1 \end{cases}$$

This is not a definition by PR.

To provide a def. by PR, we use an auxiliary function h s.t. $h(y) = [f(y), f(y+1)] = gn_1(f(y), f(y+1))$

$$h(0) = [f(0), f(1)] = [0, 1] = gn_1(0,1) = 2^1 \cdot 3^2 = 18$$

$$h(y+1) = [f(y+1), f(y+2)] =$$

$$= [f(y+1), f(y+1) + f(y)] =$$

$$= [dec(1, h(y)), dec(1, h(y)) + dec(0, h(y))]$$

We get the sequence of values $h(0), h(1), h(2)$, i.e.

$$[f(0), f(1)], [f(1), f(2)], [f(2), f(3)], \ldots$$

To obtain the Fibonacci numbers, we take

$$f(y) = dec(0, h(y))$$

This can be generalized to recursive definitions that make use of all previous values of the function.

Let $f : \mathbb{N}^{n+1} \to \mathbb{N}$ and consider the sequence $f(\vec{x}, i)$, for $i \geq 0$

We define: $$gn_f(\vec{x}, y) = \prod_{i=0}^{y} pr(i)^{f(\vec{x}, i)+1}$$

Theorem: gn_f is PR iff f is PR.

Proof: "$\Leftarrow$" follows immediately from the def. of gn_f and the fact that the bounded product is PR

"$\Rightarrow$" follows from the fact that we can extract f from gn_f

$$f(\vec{x}, y) = dec(y, gn_f(\vec{x}, y))$$

□

We can use gm_f to define functions by so-called course-of-values recursion:

Definition: Let $g: \mathbb{N}^n \to \mathbb{N}$, $h: \mathbb{N}^{n+2} \to \mathbb{N}$ be total functions. Then $f: \mathbb{N}^{n+1} \to \mathbb{N}$ defined by

$$\begin{cases} f(\vec{x}, 0) = g(\vec{x}) \\ f(\vec{x}, y+1) = h(\vec{x}, y, gm_f(\vec{x}, y)) \end{cases}$$

is said to be obtained from g and h by course-of-values recursion.

Theorem: Let $g: \mathbb{N}^n \to \mathbb{N}$, $h: \mathbb{N}^{n+2} \to \mathbb{N}$ be PRFs. Then $f: \mathbb{N}^{n+1} \to \mathbb{N}$ defined from g, h by course-of-values recursion is a PRF.

Proof: We define gm_f by PR from g and h.

$$gm_f(\vec{x}, 0) = pr(0)^{f(\vec{x},0)+1} = 2^{g(\vec{x})+1}$$

$$gm_f(\vec{x}, y+1) = gm_f(\vec{x}, y) \cdot pr(y+1)^{f(\vec{x}, y+1)+1} =$$

$$= gm_f(\vec{x}, y) \cdot pr(y+1)^{h(\vec{x}, y, gm_f(\vec{x}, y))+1}$$

Note: the evaluation of $gm_f(\vec{x}, y+1)$ uses only

- the parameters $\vec{x}$
- the previous value y of the recursive variable
- the ... $gm_f(\vec{x}, y)$ of gm_f
- the PRFs h, pr, $+$, $\cdot$, exponentiation

Hence, gm_f is a PRF.

By the previous theorem, also f is a PRF. □

Note: Gödel numbering gives function computation the equivalent of unlimited memory, since a single number can store an arbitrary sequence of numbers.

Computable partial functions:

We have seen that all PRFs are total.

Questions: 1) Are all computable total functions also PR?
2) Should we consider also non-total functions?

Theorem: There are total effectively computable functions that are not PR.

Proof: we use a diagonalization argument

- the PRFs can be represented as strings over a suitable alphabet $\Sigma = \{\langle, \rangle, 2, 0, \dots, 9, (,), ,, :, <, >\}$
  - $\langle\rangle, \langle 2\rangle, \langle i \rangle^{(j)}$ are represented as $\langle\langle\rangle\rangle$, $\langle\langle 2\rangle\rangle$, $\langle\langle i\rangle(\langle j\rangle)\rangle$
  - composition $h \circ (g_1, \dots, g_n)$ is represented as $\langle \hat{h} \circ \langle \hat{g}_1, \dots, \hat{g}_n \rangle\rangle$ with $\hat{h}$, $\hat{g}_i$ representing h and g_i
  - function defined by PR from g and h is represented as $\langle \hat{g} : \hat{h} \rangle$
- We can enumerate all strings over Σ lexicographically, and filter out all malformed strings not representing a PRF. We get an enumeration of all PRFs, and among those, let $f_i^{(1)}$ denote the i-th 1-variable PRF.
- Now we can define the total 1-variable function

$$g(\vec{n}) = f_{\vec{n}}^{(1)}(\vec{n}) + 1$$

g is effectively computable: to obtain $g(\vec{n})$
- determine $f_{\vec{n}}^{(1)}$
- compute $f_{\vec{n}}^{(1)}(\vec{n})$
- add 1

However, $g(\vec{n}) \neq f_{\vec{n}}^{(1)}(\vec{n})$ for $|\vec{n}| \geq 0$

So, $g(\vec{n})$ is total and effectively computable, but not PR. □

We can also provide a direct definition of a total computable function that is not PR.

Ackermann's function:

$$\begin{cases} A(0,y) = y+1 \\ A(x+1,0) = A(x,1) \\ A(x+1,y+1) = A(x, A(x+1,y)) \end{cases}$$

$A(x,y)$ is defined for every pair $x,y \in \mathbb{N}$ [Exercise]

Moreover $A(x,y)$ can be effectively computed

e.g. $A(1,1) = A(0, A(1,0)) = A(0, A(0,1)) = A(0,2) = 3$

Ackermann's function grows incredibly fast:
if we fix the first variable, we get the functions

$$\begin{aligned} A(0,y) &= y+1 \\ A(1,y) &= y+2 \\ A(2,y) &= 2y+3 \\ A(3,y) &= 2^{y+3} - 3 \\ A(4,y) &= \underbrace{2^{2^{\cdot^{\cdot^{2^{16}}}}}}_{y\ 2's} - 3 \\ &\vdots \end{aligned}$$

Theorem: For every PRF f, there is some $i \in \mathbb{N}$ such that

$$f(i) < A(i,i)$$

This means that $A(i,i)$ grows faster than any PRF.
Hence it cannot be a PRF, and so also $A(x,y)$ is not.

The proof of this result is quite complex.

Can we extend the set of PRFs to include all total computable functions?

The answer is no. Regardless of how we extend the set of total functions that we consider computable, the previous diagonalization argument applies.

Does this mean that we cannot generate all computable functions?

- If we consider only total functions, we cannot
- But if we include in the definition also partial functions, then we can avoid the diagonalization argument:

  We claimed that g is not one of the f_i's because $g(i) \neq f_i^{(1)}(i)$.

  But, if $f_i^{(1)}(i)\uparrow$, then also $g(i) = f_i^{(1)}(i) + 1$ is undefined.

  Hence, we cannot say anymore that g is different from all f_i's.

μ-recursive functions

We need to characterize when a function defined by composition or PR is undefined

- for composition this was already done
- for PR: f defined by PR from g, h is defined only if the following holds

$$f(\vec{x}, 0)\downarrow \text{ if } g(\vec{x})\downarrow$$

$$f(\vec{x}, y+1)\downarrow \text{ if } f(\vec{x}, i)\downarrow \text{ for } 0 \leq i \leq y \text{ and } h(\vec{x}, y, f(\vec{x}, y))\downarrow$$

Hence, if $f(\vec{x}, y)\uparrow$, then $f(\vec{x}, i)\uparrow$ for all $i \geq y$.

Definition: The family of μ-recursive functions (μRF) is defined inductively as follows:

- 0, s, $\pi_n^{(j)}$ are μRFs.
- if $h: \mathbb{N}^m \to \mathbb{N}$ and $g_1, \dots, g_m : \mathbb{N}^k \to \mathbb{N}$ are μRFs, then $f = h \circ (g_1, \dots, g_m)$ is a μRF
- if $g: \mathbb{N}^n \to \mathbb{N}$ and $h: \mathbb{N}^{n+2} \to \mathbb{N}$ are μRFs, then $PR[g, h]$ is a μRF
- if $\varphi: \mathbb{N}^{n+1} \to \mathbb{N}$ is a total μRF (a predicate), then $f: \mathbb{N}^n \to \mathbb{N}$ defined by $f(\vec{x}) = \mu z[\varphi(\vec{x}, z)]$ is a μRF.
- Nothing else is a μRF.

$f(\vec{x}) = \mu z[\varphi(\vec{x}, z)]$ is defined from $\varphi(\vec{x}, y)$ by (unbounded) minimization

... denotes the smallest z such that $\varphi(\vec{x}, z) = 1$.

Can be considered as an (unbounded) search over $\mathbb{N}$.

If $\varphi(\vec{x}, z) = 0$ for all $z \in \mathbb{N}$, then $\mu z[\varphi(\vec{x}, z)] \uparrow$.

Example: $f(x) = \mu z[eq(x, z \cdot z)]$

- if x is a perfect square, then $f(x)$ returns the square root of x
- otherwise $f(x)$ is undefined

We can now establish the correspondence between (partial) functions and TMs.

Theorem: Every μRF is Turing computable

Proof: We have already shown that

1) the basic functions $\Delta, \Sigma, \pi_i^{(j)}$ are Turing computable
2) that Turing computable total functions are closed under composition and primitive recursion

For (2), the TM that we have constructed establishes also closure for partial functions.

We show now that if $p(\vec{x}, y)$ is a total Turing computable predicate, then also $f(\vec{x}) = \mu z [p(\vec{x}, z)]$ is Turing computable

Let P be a TM computing p. We construct a TM for f.

1) Initially, the tape contains $\not\triangleright \bar{x}_1 \not\triangleright \dots \not\triangleright \bar{x}_n \not\triangleleft$, abbreviated $\not\triangleright \vec{\bar{x}} \not\triangleleft$

2) We add to the end $\bar{0}$, representing the initial value of an index j used for the search of the minimal z
$\not\triangleright \vec{\bar{x}} \not\triangleright \bar{0} \not\triangleleft$

3) At the generic step of the search:
$\not\triangleright \vec{\bar{x}} \not\triangleright \bar{j} \not\triangleleft$

We copy $\vec{x}$ and j
$\not\triangleright \vec{\bar{x}} \not\triangleright \bar{j} \not\triangleright \vec{\bar{x}} \not\triangleright \bar{j} \not\triangleleft$

4) We run P on the copy of $\vec{x}$ and j
$\not\triangleright \vec{\bar{x}} \not\triangleright j \not\triangleright \bar{t} \not\triangleleft$ where $t = p(\vec{x}, j)$

5) If $t = p(\vec{x}, j) = 1$ the value of $f(\vec{x})$ is j.
Otherwise, we erase $\bar{t}$, increment j, and continue with (3).

If no appropriate j is defined, the TM loops, i.e. $f(\vec{x}) \uparrow$. □

What about the other direction of the correspondence between TMs and μRFs?

12/11/2014

We show now that we can design a μRF that simulates the computation of a TM. This requires to transform TM computations into numbers, which is known as:

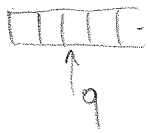
Arithmetization of TMs:

Let $M = (Q, \Sigma, \Gamma, \delta, q_0, \not b, F)$ be a TM computing a number-theoretic function with one argument

W.l.o.g. we assume that:

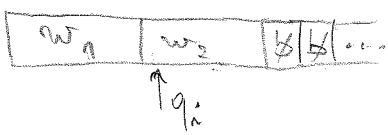
- $Q = \{q_0, q_1, \dots, q_m\}$
- $F = \{q_m\}$
- $\Gamma = \{a_0 = \not b, a_1 = 1, a_2, \dots, a_k\}$
- all transitions move the head either left or right

We associate to each configuration of M a number.
Differently from what done till now, we assume that M works on a semi-infinite tape:



Such a TM can simulate an ordinary TM working on a doubly-infinite tape [Exercise].

For a TM with a semi-infinite tape, a configuration is still a string $w_1 q_i w_2$ with



The content of the tape $w_1 w_2 = a_{i_0} a_{i_1} \dots a_{i_n}$ is encoded by the Gödel number for the sequence $i_0 i_1 \dots i_n$ ... tape number

e.g. $\not b\,1\,1\,\not b\,\not b\,\dots$ $\rightarrow 2^1 \cdot 3^2 \cdot 5^2 = 450$

or $2^1 \cdot 3^2 \cdot 5^2 \cdot 7^1 = 3150$

We can encode in the tape number any number of ⊔ to the right. Since ⊔ is represented by 0, this will not affect the decoding $dec(i,z)$ of tape position i from tape number z.

A configuration of a TM is represented by:
- the state number s
- the tape head position h (counting from the leftmost position)
- the tape number t

Hence, we use $gn_2(s,h,t)$

Since the TM moves right or left at each transition two consecutive configurations are different, and so are the configuration numbers.

We construct a function $tr_M(x,i)$ to trace the computation of M:

$tr_M(x,i)$ ... configuration number after i transitions when M is run on input x

Initial configuration $q_0 \sqcup \bar{x} \sqcup$

$$tr_M(x,0) = gn_2(0,0,2^{0+1} \cdot \prod_{i=1}^{x+1} pr(i)^2)$$

(2^{0+1}: $\sqcup$; $\prod_{i=1}^{x+1}$: since $\bar{x} = \underbrace{1\cdots1}_{x+1\ 1's}$)

We show how to compute $tr_M(x,y+1)$ from $tr_M(x,y)$.

Let z be a configuration number. We can extract from it
- the state number: $cs(z) = dec(0,z)$ ... current state
- the tape head position: $ctp(z) = dec(1,z)$ ... current tape position
- the current symbol under the tape head: $cts(z) = dec(ctp(z), dec(2,z))$ ... current tape symbol

We define functions to simulate the effect of transitions of M: let $\delta(q_{i_k}, b_{i_k}) = (q_{j_k}, c_k, d_k)$ for $0 \leq k \leq m$ be the transition of M

We can define the following functions

- new state:

$$ns(z) = \begin{cases} j_k & \text{if } cs(z) = i_k \text{ and } cts(z) = n(b_k) \\ & \text{for } 0 \leq k \leq m \\ cs(z) & \text{otherwise} \end{cases}$$

($n(b_k)$ denotes the number associated to symbol b_k)

Note: $ns(z)$ is defined by mutually exclusive cases with PR predicates $\Rightarrow$ $ns(z)$ is PR

- new tape symbol $nts(z)$
  analogous to $ns(z)$
- new tape position $ntp(z)$
  we increment the tape position by $n(d)-1$

$$\text{with } n(d) = \begin{cases} 0 & \text{if } d = L \\ 2 & \text{if } d = R \end{cases}$$

In a transition, the tape symbol to be written at position $ctp(z)$ is represented numerically by $nts(z)$.

To obtain the new tape number, we have to change the power of $pr(ctp(z))$ from $cts(z)+1$ to $nts(z)+1$.

Function for the new tape number

$$ntn(z) = quo(prod(ctn(z), pr(ctp(z))^{nts(z)+1}), pr(ctp(z))^{cts(z)+1})$$

We can now define the function $tr_M(x, y)$ returning the tape number after y transitions by PR:

$$\begin{cases} tr_M(x, 0) = qn_2(0, 0, 2^1 \cdot \prod_{i=1}^{x+1} qn(i)^2) \\ tr_M(x, y+1) = qn_2(ns(tr_M(x, y)), \\ \qquad\qquad ntp(tr_M(x, y)), \\ \qquad\qquad ntm(tr_M(x, y))) \end{cases}$$

Since all functions used are PRFs, so is tr_M.

We still need to get the result of the computation of M on x. Let $f_M(x)$ denote the function returning such a result:

- if M does not terminate on input x, then
  $f_M(x) \uparrow$, and
  we never have that $tr_M(x, y) = tr_M(x, y+1)$
- if M terminates after n transitions, then
  $f_M(x) \downarrow$ and
  $tr_M(x, n) = tr_M(x, n+1)$

We have no bound on the number of transitions that M can make before terminating:

$$term(x) = \mu z [eq(tr_M(x, z), tr_M(x, z+1))]$$

When M terminates on x, the tape contains $\not\vartriangleright \overline{f_M(x)} \not\vartriangleleft$, and the terminal tape number is given by:

$$ttm(x) = dec(2, tr_M(x, term(x)))$$

The result of the computation is given by the number of 1's on the tape:

$$sim_M(x) = \left(\sum_{i=0}^{k} eq(1, dec(i, ltm(x))) \right) \dot{-} 1$$

since $f_M(x)$ is $\underbrace{1 \cdots 1}_{f_M(x)+1 \text{ 1's}}$

where k is the length $gdln(ltm(x))$ of the sequence encoded by the Gödel number $ltm(x)$

Hence: - if $f_M(x)$ is defined for input x, then $sim_M(x) = f_M(x)$
- if $f_M(x)$ is not defined for input x (since M loops) then $sim_M(x) \uparrow$

We get:

Theorem: Every Turing computable function is μ-recursive

This allows us to restate the Church-Turing thesis:

A number-theoretic function is computable iff it is μ-recursive.