Phil 320

Chapter 5, §5.3: The scope of abacus computability

Assume that in computing $f(x_1,\ldots, x_r)$, some stipulations are followed:

(a) The arguments $x_1,\ldots, x_r$ are stored in registers 1 to r.
(b) The output register must be r+1.
(c) If the computation halts, the original arguments are still in registers 1 to r.

We show that a broad class of functions (the *recursive* functions) are all abacus-computable.

A. Initial or **basic functions.**

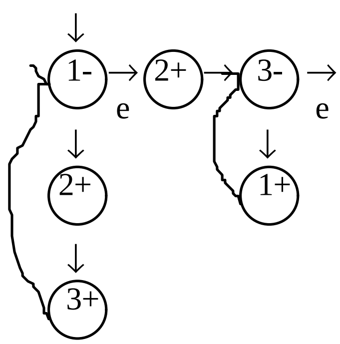
a) The *zero function*. $z(x) = 0$ for all $x \in N$.

b) The *successor function*. $s(x) = x + 1$

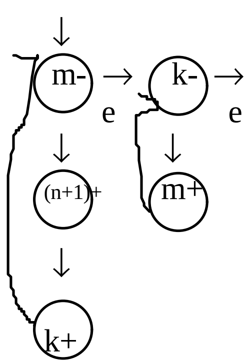
c) The *identity* or *projection functions*. $id_i^n(x_1,\ldots,x_n) = x_i$. The ith element of an n-tuple.

Examples: $id_1^1(x_1) = x_1$.
$id_3^5(x_1, x_2, \ldots, x_5) = x_3$
$id_3^3(4, 2, 25) = 25$

- z is computed by an empty abacus. For the abacus halts with 0 in R_2.
- s is computed by the abacus



- id_m^n is computed by (where $k \geq n+1$ is the index of an auxiliary register)



Next, we show that three important operations applied to functions already known to be abacus-computable will give us new functions that are still abacus-computable.

B. Composition

Example 1: $f(x) = x+2$. Write as $f(x) = s(s(x))$.
Example 2: $f(x) = 12$. Write as $f(x) = s(s(\ldots s(z(x))\ldots))$.
Example 3: $f(x_1, x_2, x_3) = x_2 + 1$. Write as $f(x_1, x_2, x_3) = s(id_2^3(x_1,x_2,x_3))$.

In general: suppose f is a function of m arguments and each of $g_1,\ldots, g_m$ is a function of n arguments. Then

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

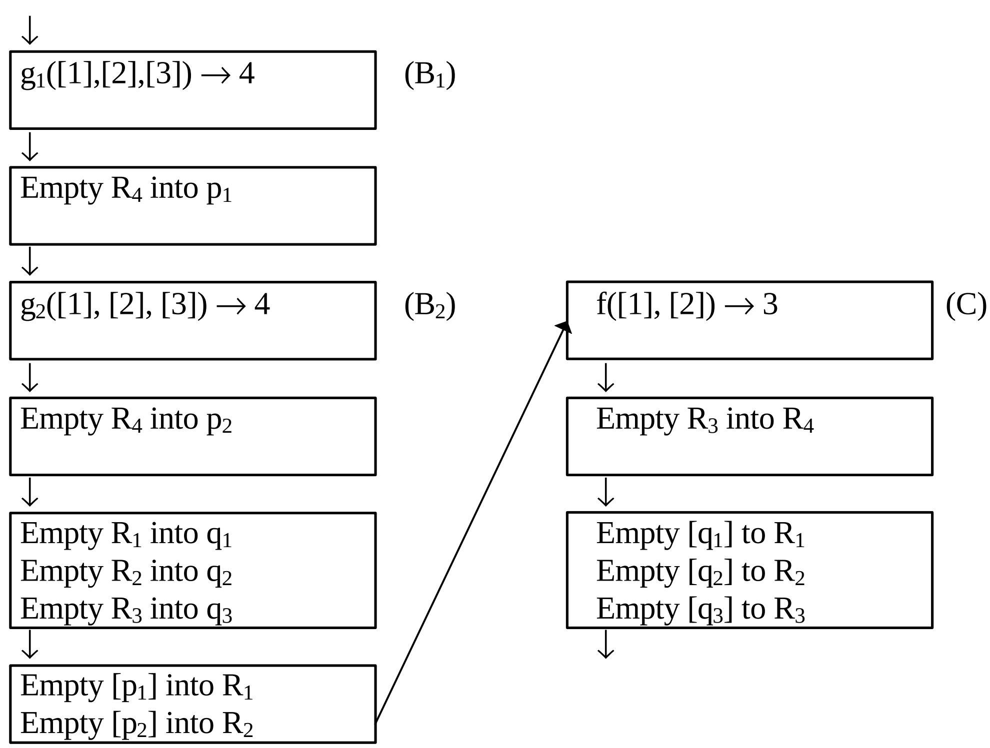
is defined by composition from f and $g_1, \ldots, g_m$.

If all the g_i's and f are abacus-computable, so is h.
Here is an outline of the proof. Given arguments $x_1,\ldots, x_n$ in registers R_1 through R_n, apply each of $g_1, \ldots, g_m$ in turn and store the results in auxiliary registers. Clear everything and put these *m* numbers in R_1 through R_m. Now apply f, and finish up by putting the result in R_{n+1}. In effect, we just glue the different abacus machines together.

The method is illustrated by taking $r = 3$ (3 arguments for each *g*) and $m = 2$ (2 arguments for *f*). So

$$h(x_1,\ldots, x_3) = f(g_1(x_1,\ldots, x_3), g_2(x_1,\ldots, x_3)).$$

Here is the picture of the abacus that computes h, where B_1, B_2 are the abaci that compute g_1, g_2 and C is the abacus that computes f. (The description follows.)



Start with $x_1,\ldots, x_3$ in R_1, R_2, R_3. Select registers p_1, p_2 which are well beyond any registers used in any of the existing abacus machines. Select registers $q_1, \ldots, q_3$ which are also well beyond any registers used in the programs and beyond p_1, p_2.

Step 1: Compute and store the result of $g_1(x_1,\ldots, x_3)$ in p_1 and $g_2(x_1,\ldots, x_3)$ in p_2, in turn, and erase R_4 after each calculation. Temporarily store the arguments x_1, x_2, x_3 in the three q registers.

Step 2: Move the contents $[p_1]$ and $[p_2]$ into the first two registers and erase R_3. Compute f, put the result into R_4, and restore the arguments from $q_1, \ldots, q_3$ into $R_1, \ldots, R_3$.

C. Primitive recursion

We will have much more to say about this operation in chapter 6.

Example 1: Addition (sum). Compute sum(x, y) = x + y in the following way:

sum(x, 0) = x
sum(x, y+1) = sum(x, y)+1

To compute sum(x, y), we now just have to work our way through sum(x, 0), sum(x, 1), …, sum(x, y-1) and sum(x, y). For instance:

sum(3, 2) = sum(3, 1) + 1
sum(3, 1) = sum(3, 0) + 1
sum(3, 0) = 3

$\Rightarrow$ so sum(3, 2) = 5

Idea: There is a recipe that lets you compute the value when the second argument is y+1 if you know the value when the second argument is y. And there is a recipe for what to do if the second argument is 0. So we can work our way up to any second argument, one step at a time.

Example 2: Multiplication (prod)

Prod(x, 0) = 0
Prod(x, y+1) = sum(x, prod(x, y))

prod(3, 2) = sum(3, prod(3, 1))
prod(3, 1) = sum(3, prod(3, 0)) = sum(3, 0) = 3 $\quad\Rightarrow$ prod(3, 2) = 6

Precise definition (for 2-place function):

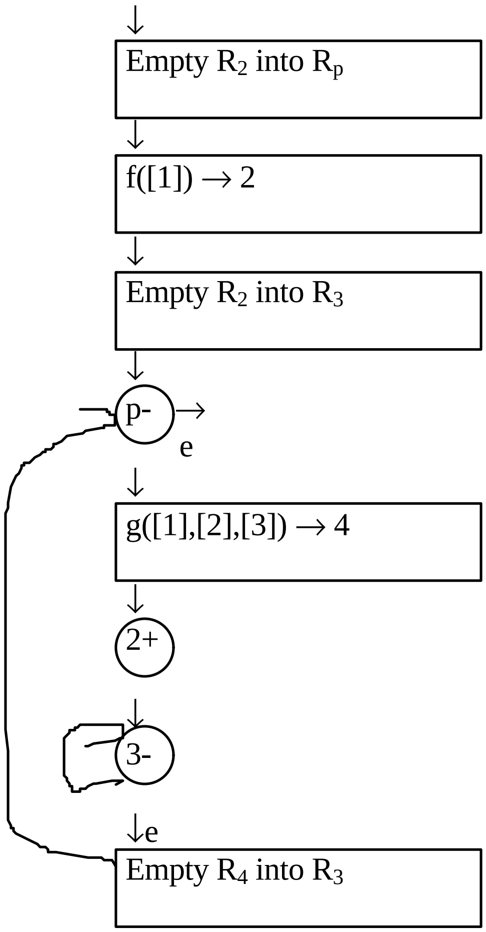
h is defined by primitive recursion from 1-place f and 3-place g if:

h(x, 0) = f(x) and
h(x, y+1) = g(x, y, h(x, y)).

If f and g are abacus-computable, so is h.

Proof: Suppose abacus *B* computes *f* (with result in register 2) and *C* computes *g* (with result in register 4). Show that there is an abacus *A* that computes *h* (with result in register 3).

Select a register R_p not used by f or g. Store x, y in registers 1 and 2, with 0 elsewhere.



(B)

$h(x,0) = f(x)$ is now in R_3

(C) – on first pass, R_4 stores $h(x,1)$. Later, stores $h(x, 2)$, $h(x, 3)$, etc.

Add 1 to R_2.

Empty R_3.

[1],[2],[3] are $x,1,h(x,1)$, then $x,2,h(x,2)$, etc.

The calculation starts by putting f(x) in R_3 and increases R_2 from 0 up to y.

D. Minimization

Here, we will only define this operation for a 2-place function f(x, y), but it can be defined more generally, and the proof below can also be made more general.

Example: Let $f(x, y) = \{x^2 - y^2$, if this is positive
$\{0$, if negative

Define

$h(x) = \{$smallest y such that $f(x, y) = 0$ IF there is such a y and $f(x, 0), f(x, 1), \ldots, f(x, y-1)$ are all defined
$\{$undefined, otherwise

Then $h(x) = x$.

Example: Let $f(x, y) = \{x - y^2$, if positive
$\{0$, if negative

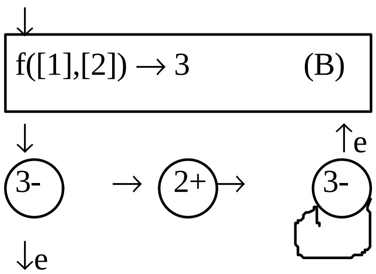
Define h as before. Then

$h(x) = \sqrt{x}$, rounded up to nearest integer.

In general, h(x) is defined as above. Note that h(x) is undefined if either f(x, y) is never 0 or f(x, i) is undefined for some i < y. h is said to be obtained from f by *minimization*.

To show computability:

Suppose abacus B computes f(x, y) and consider 1-place h obtained by minimization. Initially all registers but R_1 are empty; the solution $h(x)$ will go in R_2 (if defined).



Start with 0 for y in [2]; compute f. If val is 0, stops. Else, add 1 to y, compute f again, etc.

Empty out 3; add 1 to y and repeat the computation.

If the program stops, it can only be at the least y such that f(x, y) = 0 and f(x, t) is defined and positive for t < y; for if f is undefined somewhere along the way to y, then by definition the abacus B will not halt in computing f, so that the function defined by the above abacus will be undefined at (x,y).

RESULT:

The class of functions defined from the initial functions using composition, primitive recursion and minimization is called the *recursive functions* R. We have proved:

Theorem 5.8: All recursive functions are abacus computable.