A Programming Formalism for $\mathcal{PR}$*

A brief note that assumes access to [Tou12].

George Tourlakis

October 21, 2020

Lecture #9 (continued) Oct. 7.

1 Syntax and Semantics of Loop Programs

Loop programs were introduced by D. Ritchie and A. Meyer ([MR67]) as program-theoretic counterpart to the number theoretic introduction of the set of primitive recursive functions $\mathcal{PR}$.

This programming formalism among other things connected the definitional (or structural) complexity of primitive recursive functions with their (run time) computational complexity.

*Supplementary lecture notes for EECS2001B; Fall 2020

Loop programs are very similar to programs written in FORTRAN,

but have a number of *simplifications*,

notably they lack an unrestricted do-while instruction (equivalently, there is *NO goto instruction*).

What they do have is

(1) Each program references (uses) a finite number of *$\mathbb{N}$-valued variables* that we denote *metamathematically* by single letter names (upper or lower case is all right) with or without subscripts or primes.[1]

(2) Instructions are of the following types (X, Y could be any variables below, including the case of two identical variables):

(i) $X \leftarrow 0$

(ii) $X \leftarrow Y$

(iii) $X \leftarrow X + 1$

(iv) **Loop** $X \ldots$ **end**,

where "$\ldots$" represents a *sequence of syntactically valid instructions* (which in 1.1 will be called a "loop program"). The **Loop** part is matched or balanced by the **end** part as it will become evident by the inductive definition below (1.1).

[1]The precise syntax of variables will be given shortly, but even after this fact we will continue using signs such as X, A, Z', Y''_{34} for variables—i.e., we will continue using metanotation.

Informally, the structure of loop programs can be defined by induction:

Definition 1.1

- Every ONE instruction of *type* (i)–(iii) *standing by itself* is a *loop program.*

  If we already have two loop programs P and Q, then so are

- P;Q, built by *superposition* (concatenation)

  normally written vertically, without the separator ";", like this:

$$\begin{array}{c} P \\ Q \end{array}$$

and,

- for any variable X (that *may or may not* be in P),

  Loop X; P; **end**, is a program,

  called *the loop closure* (of P),

  and normally written vertically without separators ";" like this:

$$\begin{array}{l}\textbf{Loop } X \\ \quad P \\ \textbf{end}\end{array}$$

■

Lecture #10, Oct. 19

Definition 1.2 *The set of all loop programs will be denoted by L.* ■

The informal semantics of loop programs are precisely those given in [Tou12].

They are almost identical to the semantics of the URM programs.

1. A loop program **terminates** "if it has nothing to do", that is,

   If the current instruction is EMPTY.

2. *All three assignment statements behave as in any programming language,*

   and after execution of any such instruction*, the instruction below it (if any) is the next CURRENT instruction.*

3. When the instruction

   "**Loop** X; P; **end**"

   becomes current, its *execution* DOES (a) or (b) below:

   ► We view the **Loop-end** construct as an "instruction" just as a **begin-end** block is in, say, Pascal. ◄

   (a) *NOTHING, if $X = 0$ at that time*
   and program execution moves to the first instruction below the loop.

   (b) If $X = a > 0$ initially, then the instruction execution has the same effect as the program

$$a \text{ copies} \begin{cases} P \\ P \\ \vdots \\ P \end{cases}$$

So, the semantics of **Loop-end** *are such that the number of times around the loop is NOT affected if the program CHANGES X by an assignment statement inside the loop!*

The symbol $P_Y^{\vec{X}_n}$ has exactly the same meaning as for the URMs, but here "P" is some loop program

It is the function computed by loop program P if we use $\vec{X}_n = X_1, X_2, \ldots, X_n$ as the input and Y as the output variables.

All $P_Y^{\vec{X}_n}$ are total.

This is trivial to prove by induction on the formation of P — that ALL loop Programs Terminate.

Basis: Let P be a one-instruction program. By 1 and 3 of page 7, such a program terminates.

I.H. Fix and Assume for programs P and Q.

I.S.

- What about the program

$$\begin{array}{c} P \\ Q \end{array}$$

  By the I.H. starting at the top of program P we eventually overshoot it and make the first instruction of Q current.

  By I.H. again, we eventually overshoot Q and the whole computation ends.

- What about the program

$$\textbf{Loop}X; P; \textbf{end}$$

Well, if $X = 0$ initially, then this terminates (does nothing).

So suppose X has the value $a > 0$ initially.
Then the program behaves like

$$a \text{ copies } \begin{cases} P \\ P \\ \vdots \\ P \end{cases}$$

By the I.H. for each copy of P above when started with its first instruction, the instruction pointer of the computation will eventually overshoot the copy's last instruction.

But then starting the computation with the 1st instruction of the 1st P, eventually the computation executes the 1st instruction of the 2nd P,

then, eventually, that of the 3rd P ...

and, then, eventually, that of the last (a-th) P.

We noted that each copy of P will be overshot by the computation; THUS the overall computation will be over after the LAST copy has been overshot. PROVED!

Definition 1.3 *We define the set of* <u>*loop programmable functions*</u>*, $\mathcal{L}$:*

The symbol $\mathcal{L}$ stands for $\{P_Y^{\vec{X}_n} : P \in L\}$. ■

Two examples. Refer the computation of $\lambda x.rem(x,2)$ and $\lambda x.\lfloor x/2 \rfloor$ earlier.

If we let $f = \lambda x.rem(x,2)$ we saw that the following *sim. recursion* computes f.

$$\begin{cases} f(0) & = 0 \\ g(0) & = 1 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) \end{cases} \tag{1}$$

As a loop program this is implemented as the program P below —that is, $f = P_F^X$.

$G \leftarrow G + 1$
Loop X
$T \leftarrow F$
$F \leftarrow G$
$G \leftarrow T$
end

As for $\lambda x.\lfloor x/2 \rfloor$ we saw earlier that if $f = \lambda x.\lfloor x/2 \rfloor$ then we have:

$$\begin{cases} f(0) & = 0 \\ g(0) & = 0 \\ f(x+1) & = g(x) \\ g(x+1) & = f(x) + 1 \end{cases} \tag{2}$$

Loop X
$T \leftarrow F$
$F \leftarrow G$
$T \leftarrow T + 1$
$G \leftarrow T$
end

If P is the name of the above program, then $P_F^X = f$.

Subtracting by adding!
The program Q_X^X below computes $\lambda x.x \dot{-} 1$.

How?

X lags from T by one. At the end of the loop T holds the original value of X, but X is ONE behind its original value!

$T \leftarrow 0$
Loop X
$X \leftarrow T$
$T \leftarrow T + 1$
end

Addition

Program P below computes $\lambda xy.x + y$ as P_Y^{XY}.

Loop X
$Y \leftarrow Y + 1$
end

Multiplication

Program Q below computes $\lambda xy.x \times y$ as Q_Z^{XY}.

Loop X
  Loop Y
  $Z \leftarrow Z + 1$
  end
end

Why? Because we add 1 —$X \times Y$ times— to Z that starts as 0.

2 $\mathcal{PR} \subseteq \mathcal{L}$

Theorem 2.1 $\mathcal{PR} \subseteq \mathcal{L}$.

Proof By induction over $\mathcal{PR}$ and brute-force programming we are proving THIS property of ALL $f \in \mathcal{PR}$:

"f is loop programmable".

Basis: $\lambda x.x + 1$ is P_X^X where P is $X \leftarrow X + 1$.

Similarly, $\lambda \vec{x}_n.x_i$ is $P_{X_i}^{\vec{X}_n}$ where P is

$$X_1 \leftarrow X_1; X_2 \leftarrow X_2; \ldots ; X_n \leftarrow X_n$$

The case of $\lambda x.0$ is as easy.

Propagation of the property we are proving with ***Grzegorczyk substitution***.

Just probe the function substitution case.

How does one compute $\lambda \vec{x}\vec{y}.f(g(\vec{x}), \vec{y})$ *if* $g = G_Z^{\vec{X}}$ *and* $f = F_W^{Z\vec{Y}}$ *?*

Same as with URM programs.

One uses program concatenation and minds that Z is the only variable common between F and G.

$$\begin{pmatrix} G \\ F \end{pmatrix}_W^{\vec{X}\vec{Y}}$$

Propagation with primitive recursion.

So, say $h = H_Z^{\vec{Y}}$ and $g = G_Z^{X,\vec{Y},Z}$ where H and G are in L.

We indicate in pseudo-code how to compute $f = prim(h, g)$.

We have

$$\begin{aligned} f(0, \vec{y}_n) &= h(\vec{y}_n) \\ f(x+1, \vec{y}_n) &= g(x, \vec{y}_n, f(x, \vec{y}_n)) \end{aligned}$$

The pseudo-code is

$z \leftarrow h(\vec{y}_n)$ **Computed as** $H_Z^{\vec{Y}_n}$

$i \leftarrow 0$

Loop x

$z \leftarrow g(i, \vec{y}_n, z)$ **Computed as** $G_Z^{I,\vec{Y}_n,Z}$

$i \leftarrow i + 1$

end

See the similar more complicated programming for URMs to recall precautions needed to avoid side-effects. ■

3 $\mathcal{L} \subseteq \mathcal{PR}$

To handle the converse of 2.1 we will simulate the computation of loop program P by an array of primitive recursive functions.

Definition 3.1 *For any $P \in L$ and any variable Y in P, the symbol P_Y is an abbreviation of $P_Y^{\vec{X}_n}$, where $\vec{X}_n$ are* all *the variables that occur in P.* ■

Lemma 3.2 For any $P \in L$ and any variable Y in P, we have that $P_Y \in \mathcal{PR}$.

Proof

(A) For the ***Basis***, we have cases:

- P is $X \leftarrow 0$. Then $P_X = \lambda x.0 \in \mathcal{PR}$.
- P is $X \leftarrow Y$. Then $P_X = \lambda xy.y \in \mathcal{PR}$, while $P_Y = \lambda xy.y \in \mathcal{PR}$.
- P is $X \leftarrow X + 1$. Then $P_X = \lambda x.x + 1 \in \mathcal{PR}$

Let us next do the *induction step*:

(B) P is $Q; R$.

(i) **Case** where **NO variables are common** between Q and R.

Let the Q variables be $\vec{z}_k$ and the R variables be $\vec{u}_m$.

- What can we say about $\Big(Q; R\Big)_{z_i}$?

  Let $\lambda \vec{z}_k.f(\vec{z}_k) = Q_{z_i}$.

  $f \in \mathcal{PR}$ by the I.H.

  But then, so is $\lambda \vec{z}_k \vec{u}_m.f(\vec{z}_k)$ by Grzegorczyk Ops.

  But this is $\Big(Q; R\Big)_{z_i}$.

- Similarly we argue for $\Big(Q; R\Big)_{u_j}$.

Lecture #11. Oct. 21

(ii) **Case** where $\vec{y}_n$ are <u>common</u> between Q and R.

$\vec{z}$ and $\vec{u}$ —just as in case (i) above— are the <u>*NON*-common</u> variables.

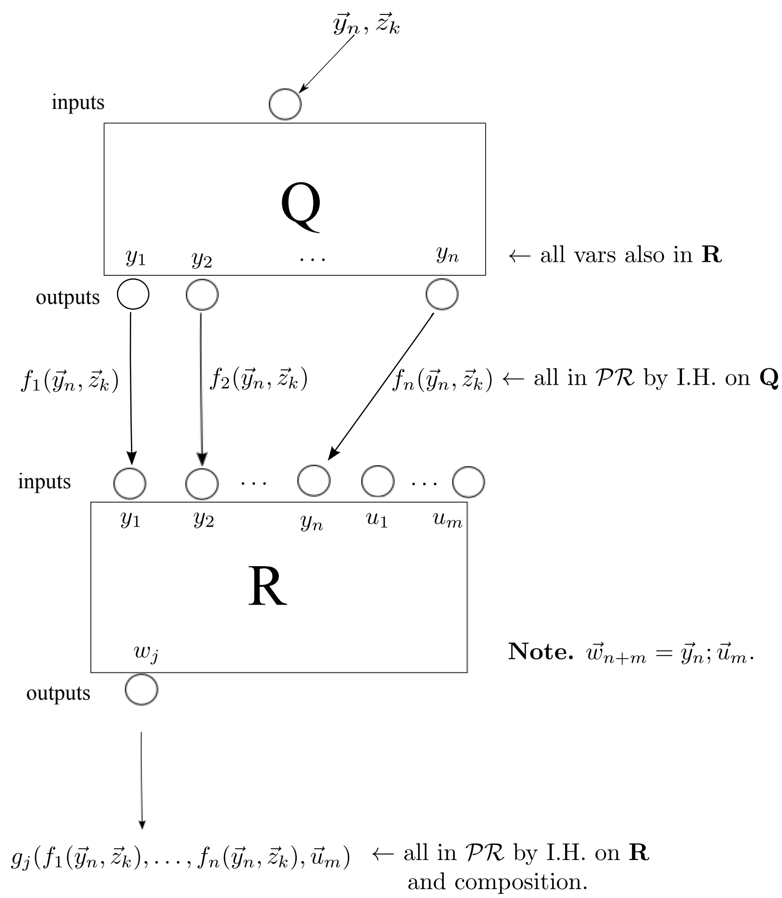
▶ Thus the set of variables of $\Big(Q; R\Big)$ is $\vec{y}_n \vec{z}_k \vec{u}_m$

Now, pick an output variable w_i.

- If w_i is among the z_j, then we are back to the first bullet of case (i).

  <u>Nothing that R does can change z_j.</u>

- So let the w_i be a component of the vector $\vec{y}_n \vec{u}_m$ instead. This case is fully captured by the figure below.



$\vec{y}_n, \vec{z}_k$
inputs
Q
y_1 y_2 $\cdots$ y_n
$\leftarrow$ all vars also in **R**
outputs
$f_1(\vec{y}_n, \vec{z}_k)$
$f_2(\vec{y}_n, \vec{z}_k)$
$f_n(\vec{y}_n, \vec{z}_k) \leftarrow$ all in $\mathcal{PR}$ by I.H. on **Q**
inputs
$\cdots$ $\cdots$
y_1 y_2 y_n u_1 u_m
R
w_j
Note. $\vec{w}_{n+m} = \vec{y}_n; \vec{u}_m$.
outputs
$g_j(f_1(\vec{y}_n, \vec{z}_k), \ldots, f_n(\vec{y}_n, \vec{z}_k), \vec{u}_m)$ $\leftarrow$ all in $\mathcal{PR}$ by I.H. on **R**
and composition.

(C) P is **Loop** $x; Q;$ **end**.

There are two subcases: x in Q; or NOT.

(a) x not in Q:

So, let $\vec{y}_n$ be all the variables of Q; x is NOT one of them. Let

$$\boxed{\lambda x\vec{y}_n.f_0(x, \vec{y}_n)} \text{ denote } P_x \tag{5}$$

and, for $i = 1, \ldots, n$,

$$\boxed{\lambda x\vec{y}_n.f_i(x, \vec{y}_n)} \text{ denote } P_{y_i} \tag{6}$$

where x —being an input variable— holds the initial value we give to it before the program P starts.

In what follows we will refer to this initial value of x as "k".

Moreover, let

$$\boxed{\lambda\vec{y}_n.g_i(\vec{y}_n)} \text{ denote } Q_{y_i} \tag{7}$$

- By the I.H., the g_i are in $\mathcal{PR}$ for $i = 1, 2, \ldots, n$.

We want to prove that the functions in (5) and (6) are also in $\mathcal{PR}$.

Since $f_0 = \lambda x\vec{y}_n.x$ (Why?),

we only deal with the f_i for $i > 0$.

The plan is to set up a simultaneous recursion that produces the f_i from the g_i.

Now imagine the computation of P with input $x, y_1, \ldots, y_n$.

We have two sub-subcases:

- $x = 0$.

  In this sub-subcase, the loop is skipped and no variables are changed by the program. In terms of (5) and (6), what I just said translates into

  $$f_0(0, \vec{y}_n) = 0 \tag{8}$$

  and

  $$f_i(0, \vec{y}_n) = y_i, \text{ for } i = 1, \ldots, n \tag{9}$$

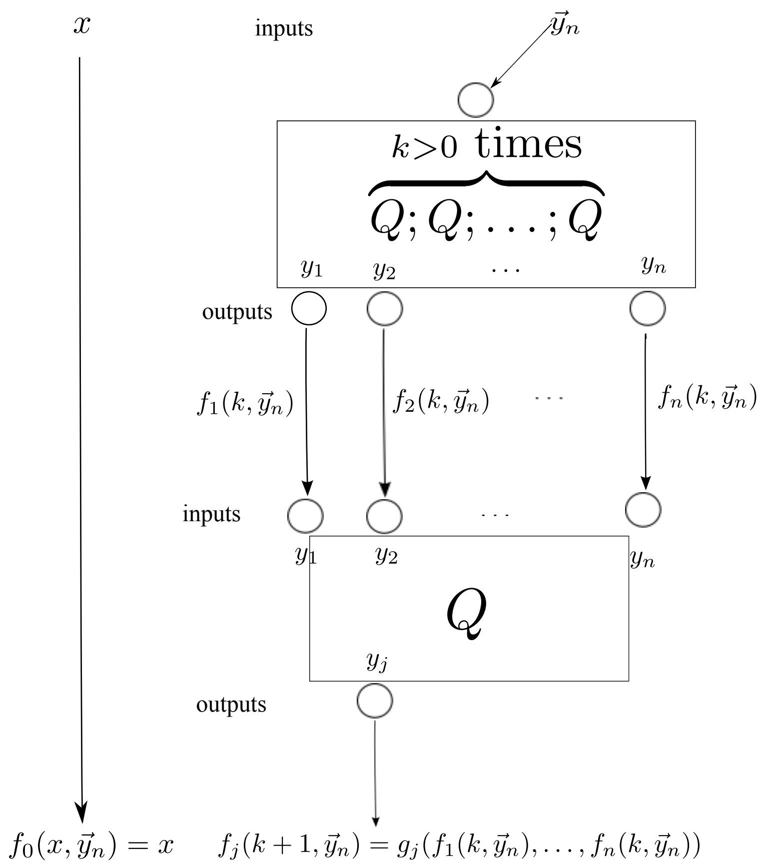
- $x = k + 1$, i.e., positive.

  The effect of P is

  $$k \text{ copies} \begin{cases} Q \\ Q \\ Q \\ \vdots \\ Q \end{cases} \\ Q \tag{10}$$

What is $f_i(k + 1, \vec{y}_n)$, for $i > 0$?

Well, consult the picture below:



We now have a simultaneous primitive recursion that yields the f_i from the g_i. The g_i being in $\mathcal{PR}$ by the I.H. on Q, so are the f_i.

(b) x in Q:

So, let $x, \vec{y}_n$ be all the variables of Q. Let

$$\lambda x\vec{y}_n.f_0(x, \vec{y}_n) \text{ denote } P_x \tag{11}$$

and, for $i = 1, \ldots, n$,

$$\lambda x\vec{y}_n.f_i(x, \vec{y}_n) \text{ denote } P_{y_i} \tag{12}$$

Moreover, let

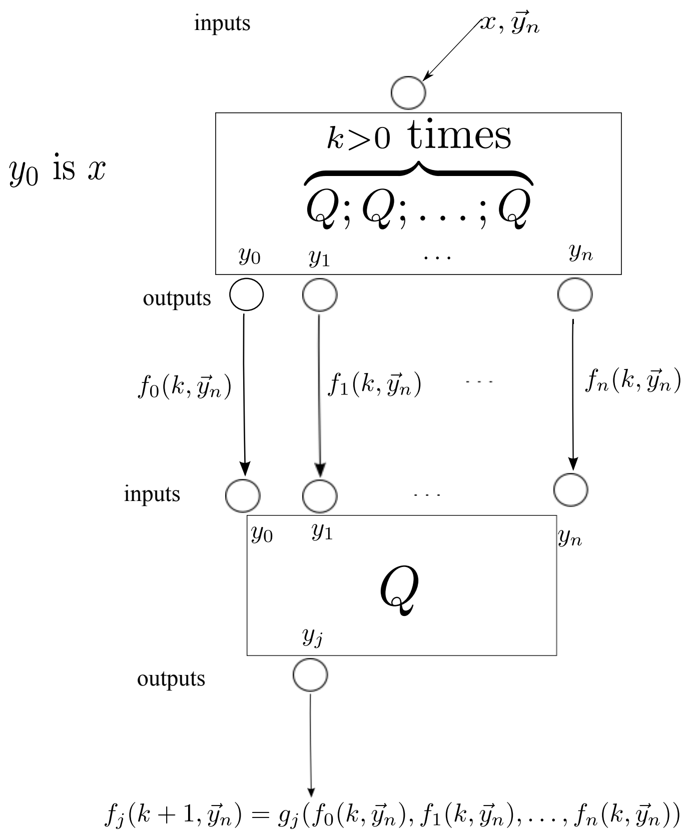
$$\lambda x\vec{y}_n.g_0(x, \vec{y}_n) \text{ denote } Q_x \tag{13}$$

$$\lambda x\vec{y}_n.g_i(x, \vec{y}_n) \text{ denote } Q_{y_i} \tag{14}$$

By the I.H., the g_i are in $\mathcal{PR}$ for $i = 1, 2, \ldots, n$.

We want to prove that the functions in (11) and (12) are also in $\mathcal{PR}$ by employing an appropriate simultaneous recursion. The basis equations are the same as (8) and (9).

For $x = k + 1$ we simply consult the figure below, to yield the recurrence equations



$$f_j(k+1, \vec{y}_n) = g_j(f_0(k, \vec{y}_n), f_1(k, \vec{y}_n), \ldots, f_n(k, \vec{y}_n)), j = 0, \ldots, n$$

As the g_j are in $\mathcal{PR}$, so are the f_j.

At the end of all this we have the proof of the Lemma.

$\blacksquare$

We can now prove

Theorem 3.3 $\mathcal{L} \subseteq \mathcal{PR}$.

Proof We must show that if $P \in L$ then for any choice of $\vec{X}_n, Y$ in P we have

$$P_Y^{\vec{X}_n} \in \mathcal{PR}$$

So pick a P and also $\vec{X}_n, Y$ in it.

Let $\vec{Z}_m$ the rest of the variables (the non-input variables) of P, and let

$$f = P_Y = P_Y^{\vec{X}_n, \vec{Z}_m}$$

and

$$g = P_Y^{\vec{X}_n}$$

By the lemma, $f \in \mathcal{PR}$.

But

$$g(\vec{X}_n) = f(\vec{X}_n, \overbrace{0, \ldots, 0}^{m\ zeros})$$

By Grzegorczyk substitution, $g = P_Y^{\vec{X}_n} \in \mathcal{PR}$. ■

All in all, we have that

$$\mathcal{PR} = \mathcal{L}$$

4 Incompleteness of $\mathcal{PR}$

We can now see that $\mathcal{PR}$ cannot possibly contain all the *intuitively computable* total functions. We see this as follows:

(A) It is immediately believable that we can write a program that checks if a string over the alphabet

$$\Sigma = \{X, 0, 1, +, \leftarrow, ;\ , \mathbf{Loop}, \mathbf{end}\}$$

of loop programs is a correctly formed program or not.

BTW, the symbols X and 1 above generate *all* the variables,

$$X1, X11, X111, X1111, \ldots$$

We will not ever write variables down as what they really are —"$X\,\underbrace{1\ldots1}_{k\ 1s}$"— but we will continue using *metasymbols* like

$$X, Y, Z, A, B, X'', Y'''_{23}, x, y, z'''_{15}$$

etc., for variables!

(B) We can algorithmically build the list, $List_1$, of ALL strings over Σ:

List by length; and in each length group **lexicographically**.[2]

(C) Simultaneously to building $List_1$ build $List_2$ as follows:

For every string α generated in $List_1$, copy it into $List_2$ iff $\alpha \in L$ (which we can test by (A)).

(D) Simultaneously to building $List_2$ build $List_3$:

For every P (program) copied in $List_2$ copy all the finitely many strings P_Y^X (for all choices of X and Y in P) alphabetically (think of the string P_Y^X as "$P; X; Y$").

At the end of all this we have an algorithmic list of all the functions $\lambda x.f(x)$ of $\mathcal{PR}$,

listed by their aliases, the P_Y^X programs.

Let us call this list of ALL the one-argument $\mathcal{PR}$ FUNCTIONS

$$f_0, f_1, f_2, \ldots, f_x, \ldots \tag{1}$$

Each f_i is a $\lambda x.f_i(x)$

[2]Fix the ordering of Σ as listed above.

4.1 A Universal function for unary $\mathcal{PR}$ functions

At the end of all this we got a *universal* or *enumerating* function $U^{(PR)}$ for *all* the unary functions functions in $\mathcal{PR}$.

That is the function of TWO arguments

$$U^{(PR)} = \lambda ix.f_i(x) \tag{2}$$

$U^{(PR)}(i, x) = f_i(x)$.

What do I mean by "Universal"?

Definition 4.1 $U^{(PR)}$ of (2) is *universal* or *enumerating* for all the unary functions of $\mathcal{PR}$ meaning it has two properties:

1. If $g \in \mathcal{PR}$ is unary, then there is an i such that

$$g = \lambda x.U^{(PR)}(i, x)$$

and

2. Conversely, for every $i \in \mathbb{N}$, $\lambda x.U^{(PR)}(i, x) \in \mathcal{PR}$. ■

Theorem 4.2 The function of two variables, $\lambda ix.U^{(PR)}(i,x)$ is computable informally.

Proof Here is how to calculate $U^{(PR)}(i,x)$ for each given i and a:

1. *Find the i-th P_Y^X in the enumeration (1) that we have built in (D) above. That is, the f_i in $List_3$.*

   This does NOT mean we HAVE an infinite List sitting there:

   It means: ***build*** $List_1$ and *simultaneously* the lists $List_2$ and $List_3$ and stop once you got the i-th element of the latter List enumerated.

2. Now, run the P_Y^X you just found with input a into X. This terminates!

   After termination Y holds $f_i(a) = U^{(PR)}(i,a)$. ■

Important. *We* repeat *for posterity TWO by-products of 4.1 and 4.2:*

- The informally computable *Enumeration function* $U^{(PR)}$ is total.
- $\lambda x.U^{(PR)}(i,x) = f_i$ for all i.

Theorem 4.3 $U^{(PR)}$ is NOT primitive recursive.

Proof If it is, *then so is* $\lambda x.U^{(PR)}(x,x)+1$ by Grzegorczyk operations. As this is a unary $\mathcal{PR}$ function, we must have an i such that

$$U^{(PR)}(x,x)+1 = U^{(PR)}(i,x), \text{ for all } x \tag{3}$$

Setting i into x in (3) we get the contradiction

$$U^{(PR)}(i,i)+1 = U^{(PR)}(i,i) \qquad \square$$

Remark. 4.4 Thus $\lambda ix.U^{(PR)}(i,x)$ acts as the *COMPILER* of a *stored program computer*:

You give it a (pointer to a) **PROGRAM** i and *DATA* x and it *simulates* the **Program** (at address) i on the **Data** x!

We have just learnt in the above theorem that this compiler **CANNOT be programmed in the Loop-Programs Programming Language**! ■

References

[MR67] A. R. Meyer and D. M. Ritchie, *Computational complexity and program structure*, Technical Report RC-1817, IBM, 1967.

[Tou12] G. Tourlakis, *Theory of Computation*, John Wiley & Sons, Hoboken, NJ, 2012.