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

George Tourlakis

February 5, 2018

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.

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, goto instruction). What they do have is

(1) Each program references (uses) a finite number of 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** $X \dots$ **end**, where "$\dots$" 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).

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

Definition 1.1 Every 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

---

[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.

- P;Q, built by *superposition* (concatenation) and normally denoted vertically, without the separator ";", like this:

$$P$$
$$Q$$

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

- **Loop** X; P; **end**, called *loop closure* (of P), and normally written vertically without separators ";" like this:

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

∎

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] (and in class) and will not be repeated here. Similarly, the symbol $P_Y^{\vec{X}_n}$ is as for the URMs: 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. Of course, we know from [Tou12], and from class, that all such $P_Y^{\vec{X}_n}$ are total. Quite a few examples are given in loc. cit.

We defined the set of loop programmable functions, $\mathscr{L}$:

Definition 1.3 The symbol $\mathscr{L}$ stands for $\{P_Y^{\vec{X}_n} : P \in L\}$. ∎

2 $\mathscr{PR}$ vs. $\mathscr{L}$

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

Proof By induction over $\mathscr{PR}$ and brute-force programming:

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.

How does one compute $\lambda x.f(g(x))$ if g is G_X^X and f is F_X^X? One uses

$$\begin{pmatrix} G' \\ F \end{pmatrix}_X^X$$

where G' is G modified to avoid side-effects: One must ensure that all the variables of F *other than* X —which were referenced in G— are set to 0 upon exit from G because F expects all these variables to be 0 in order to compute f correctly. G' does that, if necessary, by us placing at the end of G several statements of the type $Y \leftarrow 0$.

The general case $\lambda\vec{x}_m.f\big(g_1(\vec{x}_m),\ldots,g_n(\vec{x}_m)\big)$ is programmed similarly.

Finally, we indicate in pseudo-code how to compute $f(x,\vec{y}_n)$ where

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

assuming we have loop programs H and G for h and g respectively. The pseudo-code is

$$z \leftarrow h(\vec{y}_n)$$
$$i \leftarrow 0$$
$$\textbf{Loop } x$$
$$z \leftarrow g(i, \vec{y}_n, z)$$
$$i \leftarrow i + 1$$
$$\textbf{end}$$

The pseudo-code above means —for example, by "$z \leftarrow h(\vec{y}_n)$"— that we rather have placed the program H in the place of that pseudo-instruction, with input variables $\vec{y}_n$ and output variable z. Similar comment for G.

Once again one has to eliminate side-effects. For example, neither H nor G are allowed to change $\vec{y}_n$. G must not change i either.[2] Any non input variables of G must be explicitly set to 0 ($W \leftarrow 0$) at the end of G —by a modified G if the original was not doing this— so that G correctly computes "according to its spec" every time we enter this sub program while we are looping around the loop x times. Indeed, any non input variables of G that occur in H must also be set explicitly to 0 at the end of H so that G computes correctly the first time we enter the loop. ∎

To handle the converse of the preceding theorem we define

Definition 2.2 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. ∎

Theorem 2.3 $\mathscr{L} \subseteq \mathscr{PR}$.

Proof The plan is to use induction over the definition of L (1.1) and prove that for every $P \in L$ and any Y in P, $P_Y \in \mathscr{PR}$.

Why *is the above plan sufficient, for what we want*, which is to show (1) below?

For all $P \in L$, and any input set $\vec{X}_n$ and output variable Y, all in P, that $P_Y^{\vec{X}_n} \in \mathscr{PR}$ (1)

---

[2]To make any variable "read only", for example i, it is very easy: Change all occurrences of i in G to a *new variable i'*. Add the instruction $i' \leftarrow i$ at the very beginning of the so modified G.

Because, say we picked a $P \in L$ and a Y in P as output variable. Say $\vec{X}_n, \vec{W}_m$ is the set of all variables of P. But then, if our plan succeeds we have that

$$P_Y = P_Y^{\vec{X}_n, \vec{W}_m} \in \mathscr{PR}$$

If we now set $P_Y = \lambda \vec{X}_n \vec{W}_m . f(\vec{X}_n, \vec{W}_m)$, we have $f \in \mathscr{PR}$, and —by Grzegorczyk substitution— that also

$$\lambda \vec{X}_n . f(\vec{X}_n, \underbrace{0, \dots, 0}_{m \text{ zeros}}) \in \mathscr{PR}$$

But

$$\lambda \vec{X}_n . f(\vec{X}_n, \underbrace{0, \dots, 0}_{m \text{ zeros}}) = P_Y^{\vec{X}_n}$$

On to our plan then!

For the *basis*, we have cases:

- P is $X \leftarrow 0$. Then $P_X = \lambda x.0 \in \mathscr{PR}$.

- P is $X \leftarrow Y$. Then $P_X = \lambda xy.y \in \mathscr{PR}$, while $P_Y = \lambda xy.y \in \mathscr{PR}$.

- P is $X \leftarrow X + 1$. Then $P_X = \lambda x.x + 1 \in \mathscr{PR}$

Let next do the induction step:

(A) 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 $\left(Q; R\right)_{z_i}$?

          If we let $\lambda \vec{z}_k . f(\vec{z}_k)$ stand for Q_{z_i}, then $f \in \mathcal{PR}$ 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 $\left(Q; R\right)_{z_i}$.
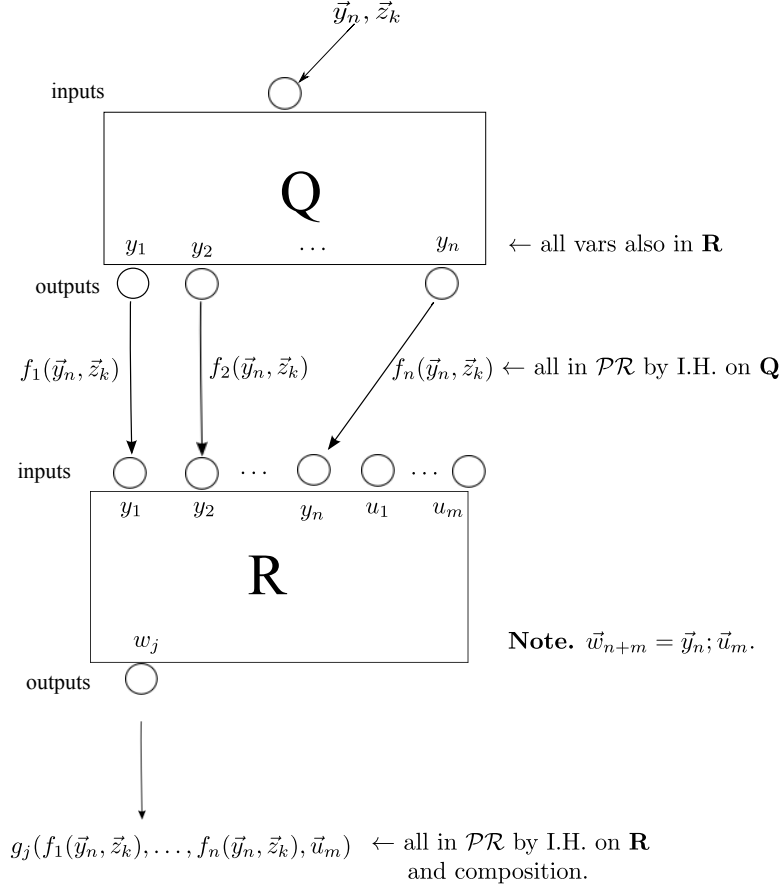
        - Similarly we argue for $\left(Q; R\right)_{u_j}$.

    (ii) Case where $\vec{y}_n$ are common between Q and R. $\vec{z}$ and $\vec{u}$ are the NON-common variables.

          ▶ Thus the set of variables of $\left(Q; R\right)$ 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).
        - 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 $\ldots$ 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

y_1 y_2 $\ldots$ y_n u_1 $\ldots$ u_m

R

w_j **Note.** $\vec{w}_{n+m} = \vec{y}_n; \vec{u}_m.$

outputs

$g_j(f_1(\vec{y}_n, \vec{z}_k), \ldots, f_n(\vec{y}_n, \vec{z}_k), \vec{u}_m) \quad \leftarrow$ all in $\mathcal{PR}$ by I.H. on **R**
and composition.

(B) 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 being one of them. Let

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

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

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

Moreover, let

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

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 $i > 0$.

5

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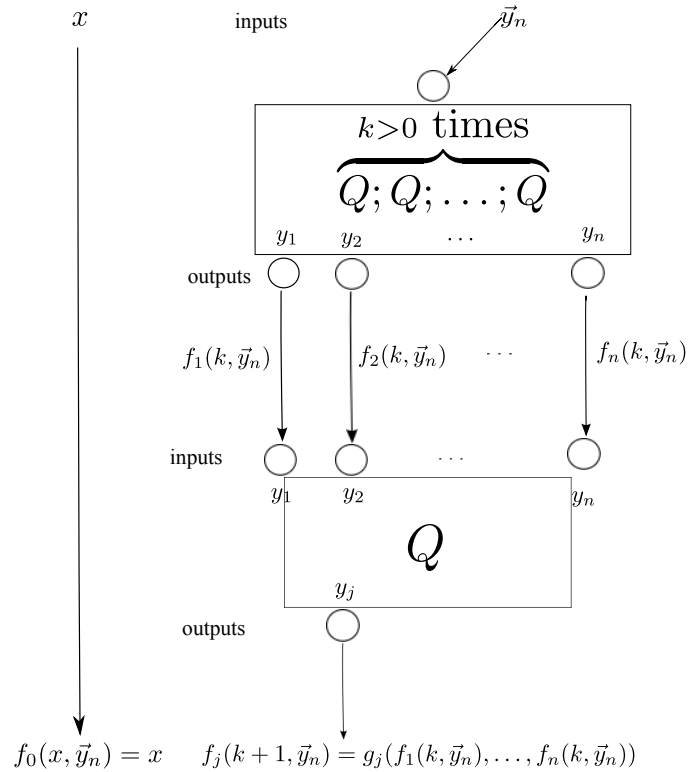
$$f_0(0, \vec{y}_n) = 0 \tag{8}$$

  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

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

What is $f(k+1, \vec{y}_n)$? Well, consult the picture below:

$$x \qquad \text{inputs} \qquad \vec{y}_n$$

$$\overbrace{Q; Q; \ldots; Q}^{k>0 \text{ times}}$$

$$y_1 \quad y_2 \quad \cdots \quad y_n$$

outputs

$$f_1(k, \vec{y}_n) \qquad f_2(k, \vec{y}_n) \quad \cdots \quad f_n(k, \vec{y}_n)$$

inputs

$$y_1 \quad y_2 \quad \cdots \quad y_n$$

$$Q$$

$$y_j$$

outputs

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

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

(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

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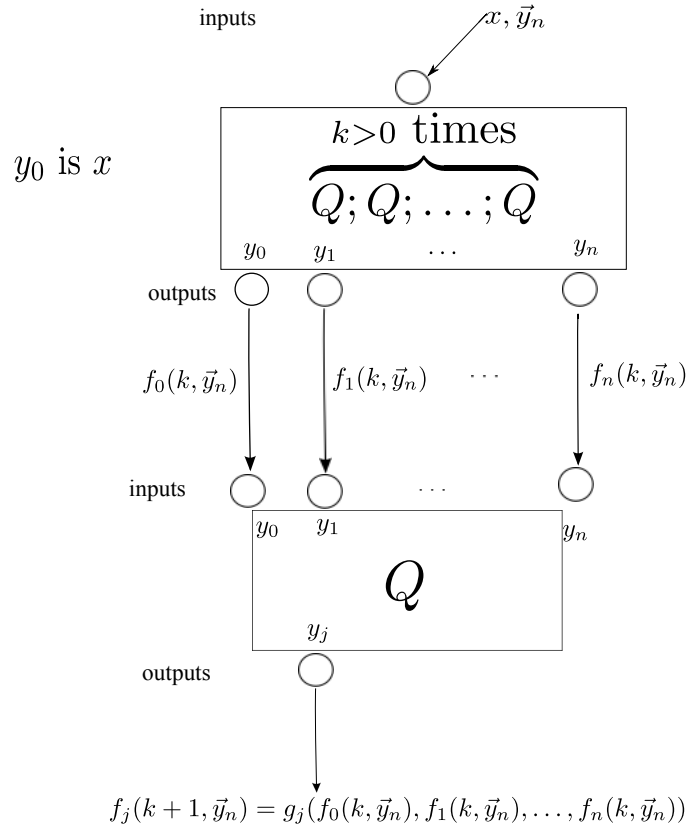
$$f_j(k + 1, \vec{y}_n) = g_j(f_0(k, \vec{y}_n), f_1(k, \vec{y}_n), \ldots, f_n(k, \vec{y}_n)), j = 0, \ldots, n$$

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

All in all, we have that

$$\mathscr{PR} = \mathscr{L}$$

3 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) Since the language L is context free, we can decide (algorithmically, intuitively speaking) for any string α whether it belongs to L (is a well-formed program) or not.

(B) We can algorithmically build the list, $List_1$, of all strings over Σ: List by length; in each length group lexicographically.[3]

(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 as "$P; X; Y$").

At the end of all this we have an algorithmic list of <u>all</u> the functions $\lambda x.f(x)$ of $\mathcal{PR}$, listed by their aliases, the P_Y^X. Let us call this list

$$f_0, f_1, f_2, \ldots, f_x, \ldots$$

By Cantor's "diagonalization method" we define a new function d for all x as follows:

$$d(x) = f_x(x) + 1 \tag{1}$$

Two observations:

1. d is total (obvious, since each f_x is) and <u>intuitively</u> computable. Indeed, to compute $d(a)$ generate the lists long enough until you have the a-th item (counting as in $0, 1, 2, \ldots, a$) in $List_3$. This item has the format P_Y^X. I.e., we have a loop program and designated input (one) and output variables. Start this program with input the value a (in X). On termination add 1 to what Y holds and return. This is $d(a)$.

2. d is not in the list! For otherwise, $d = f_i$ for some $i \geq 0$. We get a contradiction:

$$f_i(i) \stackrel{\text{by } d = f_i}{=} d(i) \stackrel{\text{by (1) above}}{=} f_i(i) + 1$$

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.

---

[3]Fix the ordering of Σ as listed in (1) on p.**??**.