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

A brief note that assumes access to [Tou12].

George Tourlakis

February 2, 2014

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** $X \ldots$ **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).

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

[*]Supplementary lecture notes for CS4111/5111; Winter 2014

[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:

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

  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:

$$\begin{array}{c} \textbf{Loop}\ X \\ P \\ \textbf{end} \end{array}$$

■

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 $\{P_Y^{\vec{X}_n} : P \in L\}$. ■

Remark. 1.4 It will be useful for Section 3 to disclose the true nature of loop program variables and carefully recast the definition of loop programs via a *context free grammar* (see your old course materials from CSE2001 3.00!).

The finite alphabet over which loop programs are built is

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

Bold type in **Loop** and **end** signifies that each of these keywords is a *single symbol* (on the blackboard we use underlining). We fix the lexicographic (alphabetic) order of the symbols in Σ as above, thus

$$\textbf{Loop} < \textbf{end} < X < 1 < \leftarrow < + < ; < 0 \tag{1}$$

The variables are strings of the *form* $X1^l$, where $l > 0$ and

$$1^l = \underbrace{ll\ldots l}_{l \text{ copies}}$$

Thus, using the name —or as we say in language theory, *nonterminal*— $\langle var \rangle$ to denote the set of variables, we know from CSE2001 that we can use the recursive definition below (in Backus Naur Form (BNF) notation, as it is called) *to define the contents of* $\langle var \rangle$:

$$\langle var \rangle ::= X1 \Big| \langle var \rangle 1$$

read as

"a variable is (symbol $::=$) the string '$X1$' or (symbol $\Big|$) it is a variable followed by a '1' "

Thus, using the nonterminal $\langle prog\rangle$ to name the set L, our earlier definition re-reads:

$$\begin{aligned}\langle prog\rangle ::= &\langle var\rangle \leftarrow 0 \Big| \\ &\langle var\rangle \leftarrow \langle var\rangle \Big| \\ &+\langle var\rangle \Big| \\ &\langle prog\rangle ; \langle prog\rangle \Big| \\ &\textbf{Loop}\langle var\rangle ; \langle prog\rangle ; \textbf{end}\end{aligned}$$

Note that "$+\langle var\rangle$" is used instead of the one we are using informally, "$\langle var\rangle \leftarrow \langle var\rangle + 1$". This is because context free grammars *cannot* generate strings of the type $z\#w$ where $z = w$ as strings —that is, they cannot in general compare strings for equality (again, recall CSE2001 and the *pumping lemma* for context free languages). ■

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 $P^{\vec{X}_n}_{X_i}$ 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

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

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

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

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

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

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

[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.

But

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

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 $Q; R$.

Pick a variable w_j in P. Let $\vec{y}_n$ be all the variables of Q, and $\vec{w}$ all those in R.[3]

Now, in one extreme case none of the y_i occur in $\vec{w}$, in which case $P_{w_j} = R_{w_j}$ and we are done by the I.H.

The interesting case is that some y_i are in $\vec{w}$, and thus we will assume that all are, as this will not restrict generality as we will explain shortly.

$$\text{Thus, } \vec{w}_{n+m} = \vec{y}_n; \vec{z}_m \tag{1}$$

where the z_j are *not* in Q. Next, a bit of notation that helps:

We let

$$\lambda\vec{y}_n\vec{z}_m.g_j(\vec{y}_n, \vec{z}_m) \overset{Def}{=} R_{w_j} \tag{2}$$

and

$$\lambda\vec{y}_n.f_i(\vec{y}_n) \overset{Def}{=} Q_{y_i}, \text{ for } i = 1, \ldots, n \tag{3}$$

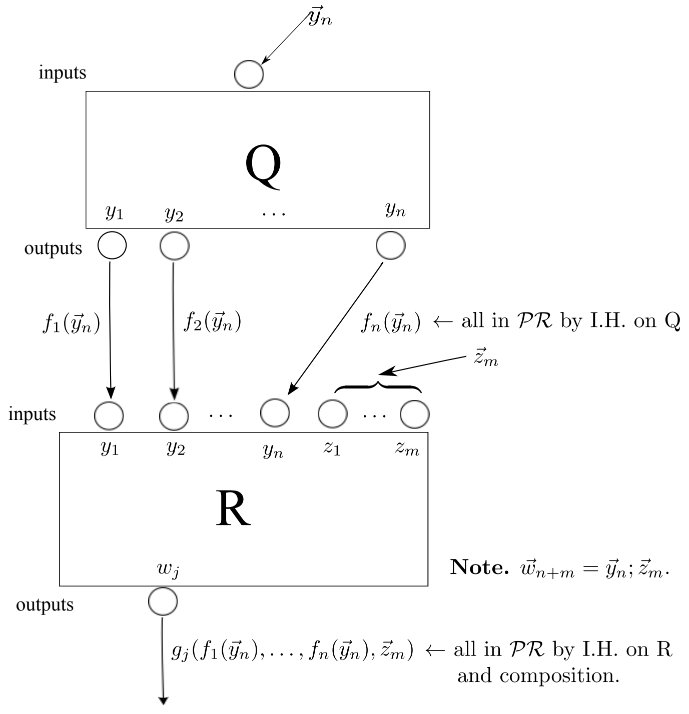
$$\text{By the I.H., the } f_i \text{ and } g_j \text{ are all in } \mathscr{PR} \tag{4}$$

Now imagine the computation of P and refer to the figure below as well: Clearly, *as soon as the computation exits* Q and enters R in the overall program P, all the y_i in R —*that are actually present*— will receive the value $f_i(\vec{y}_n)$.

If a y_i is *not* present in R, then the output $f_i(\vec{y}_n)$ of Q will not affect R's output.

[3]Recall that in our metanotation, we normally use as variable names x, X, y, Y, z, Z, w, W with or without primes or subscripts!

Thus $P_{w_j} = \lambda \vec{y}_n \vec{z}_m . g_j\big(f_1(\vec{y}_n), \ldots, f_n(\vec{y}_n), \vec{z}_m\big)$, which is in $\mathscr{PR}$ by (4) and substitution. See also the figure below.



$\vec{y}_n$
inputs
Q
y_1
y_2
$\ldots$
y_n
outputs
$f_1(\vec{y}_n)$
$f_2(\vec{y}_n)$
$f_n(\vec{y}_n)$ ← all in $\mathcal{PR}$ by I.H. on Q
$\vec{z}_m$
inputs
y_1
y_2
y_n
z_1
z_m
R
w_j
Note. $\vec{w}_{n+m} = \vec{y}_n; \vec{z}_m$.
outputs
$g_j(f_1(\vec{y}_n), \ldots, f_n(\vec{y}_n), \vec{z}_m)$ ← 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.

We will only work with the first subcase and leave the other as an exercise.

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

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

$$\text{By the I.H., the } g_i \text{ are in } \mathcal{PR} \text{ for } i = 1, 2, \ldots, n. \tag{8}$$

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 $f_i(k+1, \vec{y}_n)$ from the $g_i(\vec{y}_n)$ and the $f_i(k, \vec{y}_n)$.

Well, for the basis, $f_i(0, \vec{y}_n) = y_i$ since no y_i changes if $x = k = 0$.

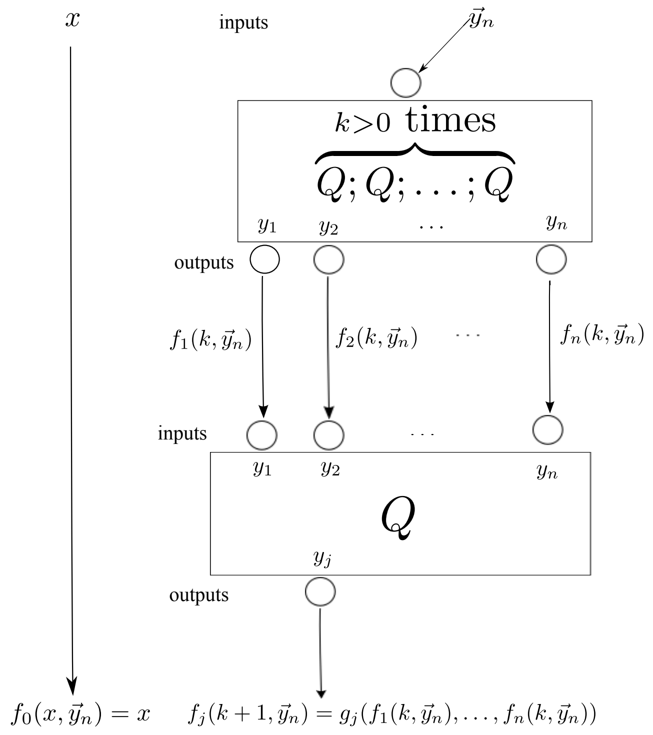
Now, according to the semantics of **Loop** $x; Q;$ **end**, $f_i(k+1, \vec{y}_n)$ is the output in the variable y_i of the program

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

while $f_i(k, \vec{y}_n)$ is the output in the variable y_i of the program

$$k \text{ copies} \begin{cases} Q \\ Q \\ Q \\ \vdots \\ Q \end{cases} \tag{9}$$

The following figure is a graphical representation of the recurrence equations that we will propose in (10). The figure is a straightforward adaptation of the superposition case (A) above, where the top program is that in (9), and the bottom one is Q.



To sum up, we have a simultaneous recursion from the $\mathcal{PR}$ functions g_j:

- Basis:
$$f_0(0, \vec{y}_n) = 0$$
and
$$f_i(0, \vec{y}_n) = y_i, \text{ for } i = 1, \ldots, n$$
- $$f_0(k+1, \vec{y}_n) = k+1$$
and
$$f_j(k+1, \vec{y}_n) = g_j\big(f_1(k, \vec{y}_n), \ldots, f_n(k, \vec{y}_n)\big), \text{ for } j = 1, \ldots, n \tag{10}$$

All in all, we have established 2.3, and thus 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 (i.e., whether α 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 list lexicographically.[4]

(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 algorithmically 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 all 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 intuitively 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:
$$f_i(i) \overset{\text{by } d=f_i}{=} d(i) \overset{\text{by (1) above}}{=} f_i(i) + 1$$

[4]Fix the ordering of Σ as listed in (1) on p.2.

References

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

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