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

George Tourlakis

October 9, 2002

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) $X = 0$

   (ii) $X = Y$

   (iii) $X = X + 1$

   (iv) **Loop** $X \ldots$ **end**, where "..." represents syntactically valid instructions. The **Loop** part is matched or balanced by the **end** part.

Informally, the structure of loop programs can be defined by induction: Every instruction of *type* (i)–(iii) *standing by itself* is a loop program. If we already have two loop programs P and Q we can build two new ones. One is built by *superposition* or concatenation:

$$\begin{matrix} P \\ Q \end{matrix}$$

*Lecture notes for CS4111 and CS6113; Fall 2002

[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', $Y_3''4$ for variables—i.e., we will continue using metanotation.

The other is built by *loop closure*. For any variable X (that may or may not be in P), we form the new program:

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

More formally we define the syntax of loop programs in BNF[2] as follows (the notation $\langle name\rangle$ is reserved to name ("name") syntactic categories, or, as we call these, *nonterminals*).

We start with the alphabet of symbols (*terminals*):

$$\Sigma = \{v, 1, =, +, 0, \mathbf{Loop}, \mathbf{end}, ;\} \tag{1}$$

The grammar rules are[3]

G1 (Variables):

$$\langle var\rangle \rightarrow v1 \Big| \langle var\rangle 1$$

where "$\rightarrow$" and "$\Big|$" are metanotational symbols (part of the BNF descriptive notation, not of Σ) and mean "is defined as" and "alternatively" respectively.

G2

$$\begin{aligned}\langle stmt\rangle \rightarrow & \langle var\rangle = 0\\ & \Big| \langle var\rangle = \langle var\rangle\\ & \Big| + \langle var\rangle\\ & \Big| \mathbf{Loop}\ \langle var\rangle; \langle prog\rangle; \mathbf{end}\end{aligned}$$

G3

$$\begin{aligned}\langle prog\rangle \rightarrow & \langle stmt\rangle\\ & \Big| \langle prog\rangle; \langle stmt\rangle\end{aligned}$$

Thus, **G1** describes that a variable *really* is a string like $v1$ or $v111$ or, in general, $v1^n$—where 1^n is a string of n "1" ($n \geq 1$).

G2 describes "single" statements. Just as in Algol descriptions (case of for-statements, blocks, etc.) we found it convenient to say that a whole construct such as "**Loop** X; P; **end**"—where P is some program—is a single statement. ";" acts a separator in our formal BNF, a

[2]Backus-Naur Form, that is, context free grammar notation that is used to describe (the "context free part" of) Algol-like languages like Pascal, PL/1 or Algol itself.

[3]Familiarity with COSC2001 or equivalent is taken as a fact.

symbol we normally suppress in informal discussions where we write loop programs vertically. Note that "$+X$" (where, as we have agreed, X is an informal variable name) is meant to stand for the informal "$X = X + 1$". The former syntax makes the definition context free no matter how variables are defined, thus avoiding the well-known non-context free construct $\{w\#w : w \in \{0,1\}^*\}$.[4]

G3 describes programs as nonempty finite sequences of statements (or instructions). The second alternative in the definition makes the separator ";" left associative thus guiding any (reasonable) translation scheme to execute statements from left to right (or top to bottom in a vertical layout).[5]

It is profitable to recast the above BNF in an ordinary inductive definition in the manner that, say, one defines the formulas of propositional or predicate calculus, or one defines $\mathcal{PR}$.

In theoretical discussions about loop programs we will use the shorthand notation

$$v_1, v_2, \ldots, v_i, \ldots$$

to denote the sequence

$$v1, v11, \ldots, v1^i, \ldots$$

of variables. We will always denote loop programs by the letters P, Q, R,[6] with or without primes and subscripts. We can now state:

Definition 1.1 The set of loop programs, L, is the smallest set of strings over Σ ((1), p.2) that includes all the *initial strings*[7]

$$\{v_i = 0 : i \geq 1\} \cup \{v_i = v_j : i \geq 1 \wedge j \geq 1\} \cup \{+v_i : i \geq 1\}$$

and is closed under the following string-operations:

1. If P is in, then so is

   Loop $v_i; P;$ **end**

   for any $i \geq 1$.

2. If P and Q are in,[8] then—for all i and j (both ≥ 1)—so are

   (a) $P; v_i = 0$

   (b) $P; v_i = v_j$

   (c) $P; +v_i$

---

[4]With variables defined to be $v1^n$, a somewhat different context free grammar can effectively allow "$X = X + 1$".

[5]It says that once we figured out that—in a string "$P; I$"—the P-part is a program, then if I is a statement we can proclaim that the whole thing is a program. This recognition has proceeded from left to right and correspondingly execution of these statements ought to proceed from left to right.

[6]These are informal meta-names entirely analogous to the meta-names X, Y, Z that we use for variables. In other contexts one acts analogously: For example, in logic one usually uses capital letters such as A, B, C—sometimes calligraphic, $\mathscr{A}, \mathscr{B}, \mathscr{C}$—to denote arbitrary formulas, p, q, r to denote Boolean variables and x, y, z, u, v, w to denote so-called "object variables".

[7]Compare in other analogous contexts: Initial functions in $\mathcal{PR}$; initial formulas ($p, q, r, \ldots$, *true*, *false*) in Boolean logic.

[8]Of course, P and Q may—but don't have to—name the same string (program). The same holds for v_i and v_j: It is allowed to have $i = j$.

(d) $P; \textbf{Loop } v_j; Q; \textbf{end}$ ■

The semantics of loop programs describe *what the execution of such a program does to the values (or "contents") of its list, or* vector*, of variables*. Throughout this note will use the notations "$[a_1, a_2, \ldots, a_r]$", "$a_1, a_2, \ldots, a_r$" and "$\vec{a}_r$" for vectors interchangeably. The first notation is typographically clearer in some contexts as—via bracketing—it emphasizes that we view the sequence $a_1, \ldots, a_r$ as one object; for example, as output of a vector-valued function.[9]

Definition 1.2 (Loop Program Semantics) Let the sequence $v_1, \ldots, v_n$ *include all* the variables referenced in a program P. We denote by $\lambda\vec{v}_n.P(\vec{v}_n)$ the *vector-valued* function from $\mathbb{N}^n \to \mathbb{N}^n$ defined by induction on the definition of P (using 1.1) as follows:

For all (values of) $v_1, \ldots, v_n$, if P is

1. $v_i = 0$: Then $P(\vec{v}_n) = [v_1, \ldots, v_{i-1}, 0, v_{i+1}, \ldots, v_n]$, for $1 \le i \le n$.
2. $v_i = v_j$: Then $P(\vec{v}_n) = [v_1, \ldots, v_{i-1}, v_j, v_{i+1}, \ldots, v_n]$, for $1 \le i \le n$.
3. $+v_i$: Then $P(\vec{v}_n) = [v_1, \ldots, v_{i-1}, v_i + 1, v_{i+1}, \ldots, v_n]$, for $1 \le i \le n$.
4. **Loop** $v_k; Q;$ **end**: Then $P(\vec{v}_n) = Q^{v_k}(\vec{v}_n)$,[10] where the vector valued iteration $Q^a(\vec{v}_n)$ is defined by
$$\begin{aligned} Q^0(\vec{v}_n) &= [v_1, \ldots, v_n] \\ Q^{a+1}(\vec{v}_n) &= Q\Big(Q^a(\vec{v}_n)\Big) \end{aligned} \tag{2}$$
5. $R; S$, where S is a program that consists of a single instruction, one among 1–4 above: Then
$$P(v_1, \ldots, v_n) = S\Big(R(v_1, \ldots, v_n)\Big) \tag{3}$$

■

Case 4 above warrants a comment. It says that if $v_k = a$ then the effect of $\textbf{Loop } v_k; Q; \textbf{end}$ is the same as that of the program
$$\underbrace{Q; Q; \cdots; Q}_{a \text{ copies of } Q}$$
regardless of what may be happening to v_k inside Q. That is, if v_k is changed by Q this does not affect the number of times the loop executes; this depends only on the value a of v_k *just prior to entering the loop*. In particular, if $a = 0$ the loop is skipped.

Note that the *vector* primitive recursion (2) above is shorthand for an ordinary simultaneous recursion of number theoretic functions (right field $\mathbb{N}$). Indeed, renaming $Q^a(\vec{v}_n)$ as $g(a, \vec{v}_n)$ and setting
$$g(a, \vec{v}_n) = [g_1(a, \vec{v}_n), \ldots, g_n(a, \vec{v}_n)]$$
and
$$Q(\vec{v}_n) = [q_1(a, \vec{v}_n), \ldots, q_n(a, \vec{v}_n)]$$

[9] In which case it is clearer to write, say, "$f(a) = [c, d, e, f]$" rather than "$f(a) = c, d, e, f$".
[10] In the case of functions, the notation f^k means composition k times; *not* exponentiation.

we have for $i = 1, \ldots, n$:

$$\begin{aligned} g_i(0, \vec{v}_n) &= v_i \\ g_i(a+1, \vec{v}_n) &= q_i\Big(g_1(a, \vec{v}_n), \ldots, g_n(a, \vec{v}_n)\Big) \end{aligned} \tag{2$'$}$$

Thus, if all the vector-components of the Q-function—the $\lambda \vec{v}_n.q_i(\vec{v}_n)$ above—are in $\mathcal{PR}$, then so are all the $\lambda a\vec{v}_n.g_i(a, \vec{v}_n)$, and therefore all the components of $\lambda \vec{v}_n.P(\vec{v}_n)$, namely the $\lambda \vec{v}_n.p_i(\vec{v}_n)$, because $P(\vec{v}_n) = Q^{v_k}(\vec{v}_n) = g(v_k, \vec{v}_n)$—hence $p_i(\vec{v}_n) = g_i(v_k, \vec{v}_n)$.

Similar comments apply to (3) above: Let us write p for the vector valued function $\lambda \vec{v}_n.P(\vec{v}_n)$ and similarly r for $\lambda \vec{v}_n.R(\vec{v}_n)$ and s for $\lambda \vec{v}_n.S(\vec{v}_n)$. Then $p = [p_1, \ldots, p_n]$, $q = [q_1, \ldots, q_n]$ and $s = [s_1, \ldots, s_n]$. Thus each p_i, q_i, s_i is a number theoretic function of n variables (i.e., from $\mathbb{N}^n$ to $\mathbb{N}$). Clearly, $p_i = s_i \circ [r_1, \ldots, r_n]$, where "$\circ$" denotes function composition. Thus, if all the r_i and all the s_i are in $\mathcal{PR}$, then so are all the p_i functions.

We have done all the work that allows us to now state

Theorem 1.3 For any loop program P whose variables are among $\vec{v}_n$, each of the functions $\lambda \vec{v}_n.u_i^n\big(P(\vec{v}_n)\big)$—for $i = 1, \ldots, n$—is in $\mathcal{PR}$.

Proof Induction of programs P. If P corresponds to 1–3 in Definition 1.2, then each of $\lambda \vec{v}_n.u_i^n\big(P(\vec{v}_n)\big)$ is initial.

For case 4, we are done by the I.H. (induction hypothesis) on P and earlier remarks.

For case 5, if S is of type 1-3, then we are done by (3) and the I.H. on R. If S is of type 4, then we are done by (3) and the I.H. on R and Q. ■

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

We next define what it means for a program P—whose list of variables is $v_1, \ldots, v_n$—to compute a number theoretic function: First we decide which ones among the $\vec{v}_n$ we want to be the *input variables*; say[11]

$$v_1, \ldots, v_r \tag{4}$$

We also decide on *one output variable*; say, v_k.

We assume that the agent that executes a loop program, *implicitly—i.e.,* **not** *via instructions that are contained in the program—initializes* to 0 all the variables **other than those in the list (4)** and **then** starts to execute P.[12] At termination[13] we read off what v_k holds. This correspondence, $\vec{v}_r \mapsto v_k$, induced by the execution of P is the function that P computes with input $\vec{v}_r$ and output v_k. In symbols $P_{v_k}^{\vec{v}_r}$. This informal description is captured as follows:

Definition 2.1 For a program P whose variable list is $\vec{v}_n$ we define the symbol $P_{v_k}^{\vec{v}_r}$, where $1 \le r, k \le n$, to mean

$$\lambda \vec{v}_r.u_k^n\Big(P(\vec{v}_r, \underbrace{0, \ldots, 0}_{n-r \text{ 0's}})\Big)$$

■

[11] More generally, we could have chosen $v_{i_1}, \ldots, v_{i_r}$ for input. Since renaming of variables is up to us we can avoid the ugly notational acrobatics that this choice entails.

[12] Compare: A Turing machine is initialized with the input anywhere on its two-way infinite tape. It is *assumed* that the tape is automatically initialized everywhere else with the blank symbol. We then let the machine run.

[13] Since $\lambda \vec{v}.P(\vec{v})$ has all its components in $\mathcal{PR}$ for all choices of P, all loop programs terminate.

We also define

Definition 2.2

$$\mathscr{L} = \{P^{\vec{v}_r}_{v_k} : P \in L \wedge \text{ the } \vec{v}_r \text{ and } v_k \text{ occur in } P\}$$

■

By 1.3, we have at once

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

The converse is true

Theorem 2.4 $\mathcal{PR} \subseteq \mathscr{L}$.

Proof By induction on $\mathcal{PR}$ and brute-force programming:

Basis: $\lambda x.x + 1$ is P^X_X where P is $X = X + 1$ (we have reverted to the "relaxed" metanotation). Similarly, $\lambda \vec{x}_n.x_i$ is $P^{\vec{X}_n}_{X_i}$ where P is

$$X_1 = X_1; X_2 = X_2; \ldots ; X_n = X_n$$

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 G other than X are set to 0 upon exit from G, for F expects all these variables to be 0 in order to compute f correctly. G' does that, if necessary, by placing at the end of G several statements of the type $Y = 0$.

The general case $f \circ (g_1, \ldots , g_n)$ 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 for h and g. The pseudo-code is

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

Once again one has to eliminate side-effects. For example, $\vec{y}_n$ must not change. Program G (for g) expects all the variables other than $i, \vec{y}_n, z$ to hold 0 at each invocation. This must be ensured by explicit programming. ■

All in all

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

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.[14]

(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:

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

References

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

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