Recursive functions, BlooP, and FlooP

Dirk Schlimm

November 8, 2017

1 Primitive recursive functions

The primitive recursive functions are:

1. Zero: $Z(n) = 0$
2. Successor: $S(n) =$ the number following n in the natural number series
3. Projections: $P^i_k(\underbrace{x_1, \ldots, x_k}_{\overrightarrow{x}}) = P^i_k(\overrightarrow{x}) = x_i$ (for $1 \leq i \leq k$).

And functions that can be obtained from primitive recursive functions by means of:

4. *Composition*:
   Given: the function g with m variables, and
   the functions $h_1, \ldots, h_m$ with k variables each.
   Composition yields the function f with k variables $\overrightarrow{x}$:
   $$f(\overrightarrow{x}) = g(h_1(\overrightarrow{x}), \ldots, h_m(\overrightarrow{x}))$$
5. *Primitive recursion*:
   Given: $d \in \mathbb{N}$ and a primitive recursive function h with two variables.
   Schema for functions *with one variable* $f(x)$:
   $$\begin{array}{rcl} f(0) & = & d \\ f(n+1) & = & h(\ f(n),\ n\ ) \end{array}$$
   Schema for functions *with more than one variables* $f(x_0, \underbrace{x_1, \ldots, x_n}_{\overrightarrow{x}})$:
   $$\begin{array}{rcl} f(0, \overrightarrow{x}) & = & g(\overrightarrow{x}) \\ f(n+1, \overrightarrow{x}) & = & h(\ f(n, \overrightarrow{x}),\ n,\ \overrightarrow{x}\ ) \end{array}$$

Examples

a) Identity: $id(x) = P^1_1(x)$.

b) Addition: $\begin{array}{ll} plus\,(0, x) & = P^1_1(x) \\ plus\,(n+1, x) & = S(P^1_3(\ plus\,(n, x),\ n,\ x\ )) \end{array}$

To define *plus* according to the schema of primitive recursion, we have:
$g(y) = P^1_1(y)$ and $h(u, v, w) = S(P^1_3(\, u, v, w\,))$.

c) Multiplication, factorial, exponentiation. (Good exercises, try them yourself!)

d) Predecessor function, $pred\,(x)$:

$$\begin{array}{rcll} pred\,(0) & = & 0 & \\ pred\,(n+1) & = & n, & \text{where } h(v, w) = P^2_2(v, w). \end{array}$$

e) Limited (truncated) subtraction, $x \dot{-} y$, which is $x - y$ if $x > y$ and 0 otherwise.

f) The conditional function, $cond\,(x, y, z)$, defined by

$$cond\,(x, y, z) = \begin{cases} y & \text{if } x = 0, \\ z & \text{otherwise.} \end{cases}$$

is primitive recursive, since it can be defined by

$$\begin{array}{rcllll} cond\,(0, y, z) & = & y, & g = P^1_2(y, z) & f(0, \overrightarrow{x}) & = g(\overrightarrow{x}) \\ cond\,(x+1, y, z) & = & z & h = P^4_4(f, n, y, z) & f(n+1, \overrightarrow{x}) & = h(\ f(n, \overrightarrow{x}),\ n,\ \overrightarrow{x}\ ) \end{array}$$

This corresponds to '`if` $\underbrace{\ldots}_{x=0}$ `then` $\underbrace{\ldots}_{y}$ `else` $\underbrace{\ldots}_{z}$' constructions in programming languages.

g) Finite sums and products: If $f_i(\overrightarrow{x})$ are primitive recursive functions, for $1 \leq i \leq n$, then so is the function

$$\sum_{i=0}^{n} f_i(\overrightarrow{x}) \ = \ f_1(\overrightarrow{x}) + \cdots + f_n(\overrightarrow{x}),$$

as well as

$$\prod_{i=0}^{n} f_i(\overrightarrow{x}) \ = \ f_1(\overrightarrow{x}) \times \cdots \times f_n(\overrightarrow{x}).$$

The *characteristic function* of an n-ary predicate or relation $R(\overrightarrow{x})$ is:

$$C_R(\overrightarrow{x}) = \begin{cases} 1 & \text{if } R(\overrightarrow{x}), \\ 0 & \text{if not } R(\overrightarrow{x}). \end{cases}$$

Now, a *predicate* is primitive recursive, if its characteristic function is.

Examples: Odd$(\,x\,)$, equal$(\,x, y\,)$, less-than$(\,x, y\,)$, min$(\,x, y\,)$, etc.

h) Definition by cases:

Given n disjoint primitive recursive conditions (= predicates), $A_1, \ldots, A_n$, and n primitive recursive functions $h_1, \ldots, h_n$, we can define the following primitive recursive function:

$$f(x) = \begin{cases} h_1(x) & \text{if } x \text{ satisfies } A_1 \\ \vdots & \vdots \\ h_n(x) & \text{if } x \text{ satisfies } A_n \end{cases}$$

by

$$\begin{aligned} f(x) &= \underbrace{(h_1(x) \times C_{A_1})}_{\text{either } h_1(x) \text{ or } =0} + \cdots + (h_n(x) \times C_{A_n}) \\ &= \sum_{i=1}^{n} h_i \times C_{A_i} \end{aligned}$$

i) Logical operations ($\wedge$, $\vee$, $\supset$, $\sim$) on primitive recursive predicates.

j) Bounded minimization:

If h is a primitive recursive function, then f *is obtained from* h *by bounded minimization* if

$$f(\overrightarrow{x}) = \min y \leq n\big[h(\overrightarrow{x}, y) = 0\big],$$

which means: "the least y less or equal than n, such that $h(\overrightarrow{x}, y) = 0$ if there is one; n otherwise."

In other words, f yields the minimal value of y (up to the *bound* n) for which h holds.

k) Similarly, we have for predicates: Bounded existence and universality.

Given a primitive recursive predicate P, we can define new predicates

$$\exists y \leq n\ P(\overrightarrow{x}, y) \quad \equiv_{Def} \quad \text{there is a } y \leq n \text{ such that } P(\overrightarrow{x}, y) \text{ holds},$$

and

$$\forall y \leq n\ P(\overrightarrow{x}, y) \quad \equiv_{Def} \quad \text{for all } y \leq n, P(\overrightarrow{x}, y) \text{ holds},$$

and show that they are indeed primitive recursive.

l) Functions obtained by iteration and simultaneously defined functions.

2 BlooP programs

BlooP can be considered as a simple (computer) language for representing predictably terminable computations.

In the following the main constructs of the language BlooP are presented. Expressions in capital `TYPEWRITER` font are part of the language, expressions in ⟨ *angled brackets* ⟩ are meta-variables.

1. Procedures
   Hofstadter refers to this ability to define and call produres elsewhere in a program by *automatic chunking*.

   `DEFINE PROCEDURE "`⟨ *name* ⟩`" [`⟨ *input variable(s)* ⟩`]:`
   `    BLOCK 0: BEGIN`
   `      ⋮`
   `    BLOCK 0: END;`

2. Blocks

   In addition to the main block of a procedure, other blocks can be defined:

   `    BLOCK 1: BEGIN`
   `      ⋮`
   `    BLOCK 1: END;`

3. Input variables

   `M, N, ...`

   These are introduced in the definition of a procedure and can be used within that procedure.

4. Temporary variables

   `CELL(0), CELL(1), CELL(2), ...`

   Values are assigned to them as follows:

   `CELL(0)` ⇐ ⟨ *mathematical expression* ⟩ `;`

   A particular temporary variable is `OUTPUT`, which is by default set to `0` at the beginning of a block. It contains the *result* or *output* of the procedure.

5. Mathematical expressions

   Can refer to numbers, $+$, $\times$, temporary variables (including `OUTPUT`), or previously defined procedures.

   Mathematical expressions can also have the boolean values `YES` or `NO`.

6. Conditionals

   `IF` ⟨ *condition* ⟩`, THEN:`
   followed by either a single instruction or a block.

   If the condition is satisfied, the next instruction (or block) is executed, otherwise it is simply ignored.

7. Conditions

   Conditions can refer to numbers, $+$, $\times$, $<$, $>$, $=$, temporary variables (including `OUTPUT`), the boolean values `YES` or `NO`, or previously defined procedures.

   Also logical operations like `AND`, `OR`, and `NOT` are allowed. Statements can be groups with braces { and }.

8. Bounded loops

   ```
   LOOP ⟨ mathematical expression ⟩ TIMES:
   BLOCK ⟨ natural number ⟩:  BEGIN
     ⋮
   BLOCK ⟨ natural number ⟩:  END;
   ```

   If a loop can be aborted, it must be defined as:

   ```
   LOOP AT MOST ⟨ mathematical expression ⟩ TIMES:
   BLOCK ⟨ natural number ⟩:  BEGIN
     ⋮
   BLOCK ⟨ natural number ⟩:  END;
   ```

9. Exiting a block

   ```
   QUIT BLOCK ⟨ number ⟩;
   ```

   This command jumps to the end of block ⟨ *number* ⟩.

   If the end of block `0` is reached, the procedure terminates with the current value of `OUTPUT`.

10. Aborting a loop

    ```
    ABORT LOOP ⟨ number ⟩;
    ```

    This command means that we jump to the statement immediately following the `END` that marks the bottom of the loop's block.

    On the difference between `QUIT` and `ABORT`, see GEB, p. 412.

11. Test procedures

    Procedures that return a `YES/NO` value are called *tests*. They are defined like (function) procedures (see item 1), but their name must end with a question mark. For example:

    ```
    DEFINE PROCEDURE "PRIME?" [N]:
    ```

12. Procedure calls

    To obtain the output of a procedure the procedure has to be called with the correct number of inputs.

    For example, the call `PRIME? [7]` yields the output `YES`.

BlooP is characterized by the *boundedness* of its loop. Because of this, BlooP functions correspond to the class of primitive recursive functions.

However, we can show that some computable functions are *not* primitive recursive!

3 FlooP programs

In the FlooP language this restriction is lifted by addition of a new kind of loop: 'free' loops, without ceiling (bound).

13. μ-loop

    ```
    MU-LOOP:
    BLOCK ⟨ natural number ⟩:  BEGIN
      ⋮
    BLOCK ⟨ natural number ⟩:  END;
    ```

This allows us to write programs that do not halt.

Terminating and non-terminating programs. FlooP programs are called

- *Terminators* if they always halt, and
- *Non-Terminators* if they are undefined (i. e., they loop forever) for at least one input.

Correspondences:

GEB	Functions	Turing Machines
BlooP	primitive recursive	—
FlooP (terminating)	(total) recursive	TMs that always halt
FlooP (non-terminating)	partial recursive	Turing Machines

4 TisLoopy

You can enter, edit, and run BlooP and FlooP programs following this link:

`http://www.cs.mcgill.ca/~cs230/TisLoopy`

You can copy-and-paste your programs into the white space on the left side of the page or edit them directly there. To run them, press 'Compile' and choose the function you would like to execute (with the appropriate parameters). To continue editing your program, click on 'Back to Editor'.

After compiling a program, the URL will change and it will encode your entire program in base-64 encoding. What this allows you to do is the following: If you bookmark the page and return to it later, it will still show your current program! You can also copy the URL, email it to somebody else, and if that person clicks on the link she will see exactly the same program in the editor that you had when you copied the link. The URL is like a Gödel number for your program!

See Section 8, below, for some sample BlooP programs.

5 Partial recursive functions

The *least search operator (μ-operator)* is defined as:

$$\mu y\,[f(\overrightarrow{x},y)=0]=z \text{ iff } \begin{cases} f(\overrightarrow{x},z)=0 \text{ and} \\ \text{for every } y<z, f(\overrightarrow{x},y) \text{ is defined, and } >0. \end{cases}$$

The μ-operator returns the smallest value for z, which makes $f(\overrightarrow{x},z)=0$, and for which all smaller values the function is defined. If there is no such value, the μ-operator is *undefined* (in this case the μ-operator still searches...).

The μ-operator is computable (determinate), i. e., it can be implemented by a TM.

Now, we can use the μ-operator to go beyond the class of primitive recursive functions:

The *partial recursive functions* (p.r.) are the *smallest* class containing

1. zero,
2. successor,
3. projection functions,

and closed under

4. composition,
5. primitive recursion, and
6. the μ-operator.

Note: Talking about "the smallest class" satisfying certain conditions, is equivalent to giving an inductive definition. The "*smallest*" corresponds to the *final clause*.

Recall, that a function is *total* if it is defined for all inputs. Thus, the *(total) recursive functions* are those partial recursive functions that are total.

6 Some fun theorems

For a partial recursive function φ_n, we cannot check in general whether $\varphi_n(b)\downarrow= r$, since the Halting Problem is unsolvable.

But, given n, we can decode it, and compute $\varphi_n(b)$ using an arbitrary number q as the upper bound for all searches (i. e., applications of the μ-operator).

Thus, we can define a predicate (the *Universal Computation Predicate*)

$$C(n,b,r,q),$$

$$C\,(\overbrace{\text{number of function } \varphi_n}^{n},\ \overbrace{\text{input(s) } \overrightarrow{x}}^{b},\ \overbrace{\text{result}}^{r},\ \overbrace{\text{max. bound}}^{q})$$

that holds if $\varphi_n(b)\downarrow= r$ and all searches are bounded by q.

Think of replacing the μ-operator by bounded minimization with bound q.

Since all searches are bounded, this predicate is *primitive recursive*!

Moreover, if $C(n,b,r,q)$, then $\forall w\geq q.\ C(n,b,r,w)$.

Theorem 1 (The Universal Computation Predicate) *There is a primitive recursive predicate C such that*

$$\varphi_n^k(b_1,\ldots,b_k)=r \quad \textit{iff} \quad \exists q\ C(n,\langle b_1,\ldots,b_k\rangle,r,q)$$

where k is the number of arguments.

Theorem 2 ((Kleene's) Normal Form theorem) *For $\overrightarrow{x}=x_1,\ldots,x_k$ the function*

$$f(n,\overrightarrow{x})=(\mu\, q\big[C(n,\langle\overrightarrow{x}\rangle,(q)_0,q)\big])_0$$

is partial recursive *and is* universal *for the partial recursive functions of k variables.*

- $(q)_0$ returns the first element in a sequence: If q codes the tuple $\langle a,b\rangle$, then $(q)_0=a$.
- q is a number, which is also the Gödel number of a pair $\langle r,s\rangle$.
- The μ-operator thus also yields a pair, of which the first component is the result of φ_n.
- This allows us to search for both the *result* and an *upper bound* of computation steps with one single μ-operator!

The theorem says, in other words, that if φ is a partial recursive function of k variables, then *for some* n and all $\overrightarrow{x}$,

$$\begin{aligned}\varphi(\overrightarrow{x}) &\simeq (\mu\, q\ \big[C(n,\langle\overrightarrow{x}\rangle,(q)_0,q)\big]\ )_0\\ &\simeq (\mu\, q\ \big[\varphi_C^4(n,\langle\overrightarrow{x}\rangle,(q)_0,q)=0\big]\ )_0\end{aligned}$$

φ_C^4 is the characteristic function of the Universal Computation Predicate C, which has 4 arguments.

Corollary 3 *Every partial recursive function may be defined with* at most one use *of the μ-operator.*

Since φ_C^4 (the characteristic function of the predicate C) is primitive recursive. This tells us that there is a *canonical, normal form* for each partial recursive function.

For programmers: One while-loop is enough to write any program that implements a function from $\mathbb{N}$ to $\mathbb{N}$!

7 GlooP... is a myth

Hofstadter mentions an hypothetical GlooP language that would extend FlooP and lift off some of its restrictions. However, according to the *Church-Turing Thesis*, such a language is not computable. By the CT-thesis, the class of computable functions corresponds exactly to those which are FlooP-computable.

8 Some BlooP programs

The text following in a line that begins with 'NB.' is a comment and does not have any effect on the program.

```
NB. The factorial procedure calculates N!.
DEFINE PROCEDURE ''FACTORIAL'' [N]:
BLOCK 0: BEGIN
    OUTPUT <= 1;
    CELL(0) <= 1;
    LOOP N TIMES:
    BLOCK 1: BEGIN
        OUTPUT <= OUTPUT x CELL(0);
        CELL(0) <= CELL(0) + 1;
    BLOCK 1: END;
BLOCK 0: END.

NB. The minus procedure calculates M-N.
DEFINE PROCEDURE ''MINUS'' [M, N]:
BLOCK 0: BEGIN
    OUTPUT <= 0;
    IF M < N, THEN:
    QUIT BLOCK 0;
    LOOP AT MOST M + 1 TIMES:
    BLOCK 1: BEGIN
        IF OUTPUT + N = M, THEN:
        ABORT LOOP 1;
        OUTPUT <= OUTPUT + 1;
    BLOCK 1: END;
BLOCK 0: END.

NB. The divide procedure calculates P/Q.
DEFINE PROCEDURE ''DIVIDE'' [P, Q]:
BLOCK 0: BEGIN
    OUTPUT <= 0;
    IF Q < 1, THEN:
    QUIT BLOCK 0;
    CELL(0) <= P;
    LOOP AT MOST P TIMES:
    BLOCK 1: BEGIN
        IF CELL(0) < Q, THEN:
        QUIT BLOCK 0;
        CELL(0) <= MINUS [CELL(0), Q];
        OUTPUT <= OUTPUT + 1;
    BLOCK 1: END;
BLOCK 0: END.
```



```
NB. The prime procedure returns 1 iff N is prime.
DEFINE PROCEDURE ''PRIME'' [N]:
BLOCK 0: BEGIN
    IF N = 0, THEN:
    QUIT BLOCK 0;
    CELL(0) <= 2;
    LOOP AT MOST MINUS [N, 2] TIMES:
    BLOCK 1: BEGIN
        IF REMAINDER [N, CELL(0)] = 0, THEN:
        QUIT BLOCK 0;
        CELL(0) <= CELL(0) + 1;
    BLOCK 1: END;
    OUTPUT <= 1;
BLOCK 0: END.

NB. The tortoise pair procedure returns 1 iff M
NB. is prime and M+N is prime.
DEFINE PROCEDURE ''TORTOISEPAIR'' [M, N]:
BLOCK 0: BEGIN
    IF PRIME [M] = 1, THEN:
    IF PRIME [M + N] = 1, THEN:
    OUTPUT <= 1;
BLOCK 0: END.
```

To write, compile, and run your own programs, go to:

`http://www.cs.mcgill.ca/~cs230/TisLoopy`