

Appendix: Loop-while programs

Recall from Chapter 1 that we have a programming language in which the variables are X_0, X_1, X_2, \dots . Here X_3 is really the string X''' , a fact that can usually be ignored.

A *program* is a string of commands, built up in accordance with a couple of syntactical rules. There are five kinds of commands:

1. $X_n \leftarrow 0$. This is the *clear* command; its effect is to assign the value 0 to X_n .
2. $X_n \leftarrow X_n + 1$. This is the *increment* command; its effect is to increase the value assigned to X_n by one.
3. $X_n \leftarrow X_m$. This is the *copy* command; its effect is just what the name suggests; in particular it leaves the value of X_m unchanged.
4. `loop X_n and endloop X_n` . A program of the form

$$\begin{array}{c} \text{loop } X_n \\ \mathcal{P} \\ \text{endloop } X_n \end{array}$$

causes the program \mathcal{P} to be executed the number of times given by the *initial* value of X_n . If the initial value of X_n is 0, then these commands will do nothing. The syntactical rule is that the loop and endloop commands must be used in pairs, as shown.

5. `while $X_n \neq 0$ and endwhile $X_n \neq 0$` . The program

$$\begin{array}{c} \text{while } X_n \neq 0 \\ \mathcal{P} \\ \text{endwhile } X_n \neq 0 \end{array}$$

causes the program \mathcal{P} to be executed repeatedly until such time as X_n is assigned the value 0, if that ever happens. If the *initial* value of X_n is 0, then these commands will do nothing. The syntactical rule is that the while and endwhile commands must be used in pairs, as shown.

The system is that initially the input is pre-loaded as the values of X_1, \dots, X_k (when we are computing a k -place partial function), the other variables are assigned 0, and the output is the value of X_0 when the program halts (if it ever does).

We say that a program \mathcal{P} *computes* a partial function f if whenever \mathcal{P} is started with input \vec{x} we have the following:

- If $f(\vec{x})$ is defined, then the program eventually halts, with X_0 assigned the value $f(\vec{x})$.
- If $f(\vec{x})$ is undefined, then the program never halts.

For example, the addition function $\langle x, y \rangle \mapsto x + y$ is loop-computed by the

following program:

```

 $X_0 \leftarrow X_1$ 
loop  $X_2$ 
   $X_0 \leftarrow X_0 + 1$ 
endloop  $X_2$ 

```

This program copies X_1 into X_0 , and increments X_0 the number of times given by X_2 . And then it stops (although there is no “halt” command).

For another example, albeit in the opposite direction, the following is a syntactically correct loop program:

```

 $X_2 \leftarrow 0$ 
loop  $X_1$ 
   $X_0 \leftarrow X_2$ 
   $X_2 \leftarrow X_2 + 1$ 
endloop  $X_1$ 

```

Being a program, it must compute some one-place function, and some two-place function, and so forth. It is left as an exercise to determine what these functions are.

Why *this* language? It is easy to spot features that are lacking in the language. But because we hope to prove statements *about* the language, we want it to be just as simple as possible, so as to make our proofs as simple as possible. And at the same time, we need a language that will be strong enough to compute all of the effectively calculable partial functions on \mathbb{N} .

The k -place function that is constantly 0 (i.e., $f(\vec{x}) = 0$) is loop-computed by the empty program. And it is loop-computed by the one-line program

$$X_0 \leftarrow 0$$

as well.

The successor function $S(x) = x + 1$ is loop-computed by the two-line program

$$\begin{aligned} X_0 &\leftarrow X_1 \\ X_0 &\leftarrow X_0 + 1 \end{aligned}$$

which also computes the two-place function $f(x, y) = x + 1$.

The projection function I_n^k defined by the equation $I_n^k(x_1, \dots, x_k) = x_n$ is loop-computed by the one-line program

$$X_0 \leftarrow X_n$$

for any $k \geq n$. (For $k < n$, this program computes the k -place zero function.)

Now suppose that the k -place function h is obtained by composition from f, g_1, \dots, g_n :

$$h(\vec{x}) = f(g_1(\vec{x}), \dots, g_n(\vec{x})).$$

Further suppose that we have programs $\mathcal{Q}, \mathcal{P}_1, \dots, \mathcal{P}_n$ that compute f, g_1, \dots, g_n , respectively. We want to make a program that computes h .

For concreteness, we will take the particular situation in which $n = 3$:

$$h(\vec{x}) = f(g_1(\vec{x}), g_2(\vec{x}), g_3(\vec{x}))$$

but the method is entirely general. In outline, we want the program

$$\begin{array}{l} U \leftarrow g_1(\vec{x}) \\ V \leftarrow g_2(\vec{x}) \\ W \leftarrow g_3(\vec{x}) \\ X_0 \leftarrow f(U, V, W) \end{array}$$

for suitable variable U , V , and W . But to this outline we need to add principles of tidiness and cleanliness: We need to store all the information in safe and appropriate locations, and we need to provide the programs with the conditions they expect—the inputs starting in X_1 and the other variables set to 0.

Let X_M be the last variable used in the programs $\mathcal{Q}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$. (Here “last” refers the obvious ordering on the variables; we may suppose that M is at least k .) The table shown gives a program to compute h .

The programs stores \vec{x} in the “safe” locations X_{M+1} to X_{M+k} , and eventually stores $g_1(\vec{x})$ in X_{M+k+1} , $g_2(\vec{x})$ in X_{M+k+2} , and $g_3(\vec{x})$ in X_{M+k+3} .

Observe that the only commands that have been added to $\mathcal{Q}, \mathcal{P}_1, \mathcal{P}_2, \mathcal{P}_3$ are copy and clear commands. No loop or while commands were added.

Finally, we come to the case of primitive recursion. Let’s start with the simplest case, where the one-place function h is obtained by primitive recursion from the two-place function g , by using the number m :

$$\begin{array}{l} h(0) = m \\ h(x+1) = g(h(x), x). \end{array}$$

In outline form, we want the program:

$$\begin{array}{l} T \leftarrow 0 \\ X_0 \leftarrow m \\ \text{loop } x \\ \quad X_0 \leftarrow g(h(T), T) \\ \quad T \leftarrow T + 1 \\ \text{endloop } x \end{array}$$

To this outline, we again add principles of tidiness and cleanliness. Assume that we have a program \mathcal{Q} that computes g . Again, let X_M be the last variable used in \mathcal{Q} . Then the program shown in the table computes h .

To verify the correctness of the program, we need to check that after the loop has been executed t times, X_0 contains $h(t)$ and X_{M+1} contains t . We can check this claim by induction on t . Initially, when $t = 0$, the program \mathcal{P} has put $h(0)$ into X_0 , and the program has cleared X_{M+1} .

Suppose, as the inductive hypothesis, the loop has been executed t times and we do indeed have $h(t)$ in X_0 and t in X_{M+1} . What happens on the next time (the $(t+1)$ st time) through the loop? This time, the program loads $h(t)$

$$\begin{array}{l}
X_{M+k} \leftarrow X_k \\
\cdots \quad \text{Save a copy of } \vec{x} \\
X_{M+2} \leftarrow X_2 \\
X_{M+1} \leftarrow X_1 \\
\mathcal{P}_1 \\
X_{M+k+1} \leftarrow X_0 \quad [\text{Save } g_1(\vec{x}).] \\
X_0 \leftarrow 0 \\
X_1 \leftarrow 0 \\
\cdots \quad [\text{Clean up.}] \\
X_M \leftarrow 0 \\
X_1 \leftarrow X_{M+1} \\
X_2 \leftarrow X_{M+2} \\
\cdots \quad [\text{Restore } \vec{x}.] \\
X_k \leftarrow X_{M+k} \\
\mathcal{P}_2 \\
X_{M+k+2} \leftarrow X_0 \quad [\text{Save } g_2(\vec{x}).] \\
X_0 \leftarrow 0 \\
X_1 \leftarrow 0 \\
\cdots \quad [\text{Clean up.}] \\
X_M \leftarrow 0 \\
X_1 \leftarrow X_{M+1} \\
X_2 \leftarrow X_{M+2} \\
\cdots \quad [\text{Restore } \vec{x}.] \\
X_k \leftarrow X_{M+k} \\
\mathcal{P}_3 \\
X_{M+k+3} \leftarrow X_0 \quad [\text{Save } g_3(\vec{x}).] \\
X_0 \leftarrow 0 \\
X_1 \leftarrow 0 \\
\cdots \quad [\text{Clean up.}] \\
X_M \leftarrow 0 \\
X_1 \leftarrow X_{M+k+1} \\
X_2 \leftarrow X_{M+k+2} \\
X_3 \leftarrow X_{M+k+3} \\
Q
\end{array}$$

A program for composition

```

 $X_{M+2} \leftarrow X_1$     [Save  $x$ .]
 $X_{M+1} \leftarrow 0$     [Redundant;  $t = 0$ .]
 $X_0 \leftarrow 0$       [Redundant.]
 $X_0 \leftarrow X_0 + 1$ 
 $X_0 \leftarrow X_0 + 1$ 
 $\dots$                 [Load  $m$ .]
 $X_0 \leftarrow X_0 + 1$ 
loop  $X_{M+2}$           [Do loop  $x$  times.]
   $X_1 \leftarrow X_0$     [This is  $h(t)$ .]
   $X_2 \leftarrow X_{M+1}$   [This is  $t$ .]
   $X_0 \leftarrow 0$ 
   $X_3 \leftarrow 0$ 
   $\dots$                 [Clean up.]
   $X_M \leftarrow 0$ 
   $\mathcal{Q}$                 [This gives  $h(t+1)$ .]
   $X_{M+1} \leftarrow X_{M+1} + 1$   [Increment  $t$ .]
endloop  $X_{M+2}$ 

```

A program for primitive recursion ($k = 0$)

and t into X_1 and X_2 and then computes $g(h(t), t)$ (which is $h(t+1)$) and puts it in X_0 , and increments t (in X_{M+1}). So the claim still holds, after $t+1$ times through the loop.

The program halts after doing the loop x times. So at this point, we have from our claim that X_0 contains $h(x)$, which is what we wanted.

Next, we want to add a list of parameters to these functions. That is, suppose that h is the $(k+1)$ -place function obtained by primitive recursion from the k -place function f and the $(k+2)$ -place function g

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

and that \mathcal{P} computes f and that \mathcal{Q} computes g . In outline form, we want the program:

```

 $T \leftarrow 0$ 
 $X_0 \leftarrow f(\vec{x}) = h(\vec{x}, 0)$ 
loop  $y$ 
   $X_0 \leftarrow g(h(\vec{x}, T), \vec{x}, T)$ 
   $T \leftarrow T + 1$ 
endloop  $y$ 

```

To this outline, we again add principles of tidiness and cleanliness. See the table shown for a program to compute h .

Again, we can verify the correctness of the program. After the loop has been executed t times, X_0 contains $h(\vec{x}, t)$ and X_{M+k+1} contains t . We can check this by induction on t as before.

```

 $X_{M+k+2} \leftarrow X_{k+1}$     [Save  $y$ .]
 $X_{M+k} \leftarrow X_k$ 
 $\dots$     [Save a copy of  $\vec{x}$ .]
 $X_{M+2} \leftarrow X_2$ 
 $X_{M+1} \leftarrow X_1$ 
 $X_{M+k+1} \leftarrow 0$     [Redundant;  $t = 0$ .]
 $X_{k+1} \leftarrow 0$ 
 $\mathcal{P}$     [Compute  $h(\vec{x}, 0)$ .]
loop  $X_{M+k+2}$     [Do loop  $y$  times.]
   $X_1 \leftarrow X_0$     [This is  $h(\vec{x}, t)$ .]
   $X_2 \leftarrow X_{M+1}$ 
   $\dots$     [Restore  $\vec{x}$ .]
   $X_{k+1} \leftarrow X_{M+k}$ 
   $X_{k+2} \leftarrow X_{M+k+1}$     [This is  $t$ .]
   $X_0 \leftarrow 0$ 
   $X_{k+3} \leftarrow 0$ 
   $\dots$     [Clean up.]
   $X_M \leftarrow 0$ 
   $\mathcal{Q}$     [This gives  $h(\vec{x}, t + 1)$ .]
   $X_{M+k+1} \leftarrow X_{M+k+1} + 1$     [Increment  $t$ .]
endloop  $X_{M+k+2}$ 

```

A program for primitive recursion ($k > 0$)

Observe that no while commands have been added. Hence if both \mathcal{P} and \mathcal{Q} are loop programs, then so is the given program for h .

Theorem. Each primitive recursive function is computed by some loop program.

Proof. We have programs for the initial functions (the zero functions, the successor function, and the projection functions). We have shown that the class of loop-computable functions is closed under composition and primitive recursion. \dashv

This theorem tells us that the class of loop-computable functions includes all of the primitive recursive functions. (In particular, all the functions found in the previous chapter to be primitive recursive are loop-computable.) In fact, the opposite inclusion also holds: The class of primitive recursive functions includes all loop-computable functions. Consequently, the two classes coincide. This gives some intuitive significance to the class of primitive recursive functions: It is, roughly speaking, the class of functions that can be computed by structured programs without GOTO commands, or anything resembling a GOTO command.

But we are not yet done writing programs. Suppose that a k -place function h is obtained from the $k + 1$ -place function g by minimalization:

$$h(\vec{x}) = \mu y [g(\vec{x}, y) = 0]$$

where these functions might or might not be total. Further suppose that we have a program \mathcal{P} that while-computes g . Then the following program while-computes h :

```

Y ← 0
S ← S + 1
while S ≠ 0
  S ← g( $\vec{x}$ , Y)
  X0 ← Y
  Y ← Y + 1
endwhile S ≠ 0

```

This program computes successively $g(\vec{x}, 0), g(\vec{x}, 1), \dots$ searching for 0. The unabbreviated expansion of the program is given in the table.

Theorem. Each general recursive partial function is computed by some while program.

Proof. The previous theorem gave us the primitive recursive functions. We have now added minimalization. \dashv

So the class of while-computable partial functions includes all of the general recursive partial functions. Again, the opposite inclusion also holds.

```

 $X_{M+k} \leftarrow X_k$ 
 $\dots$  [Save a copy of  $\vec{x}$ .]
 $X_{M+2} \leftarrow X_2$ 
 $X_{M+1} \leftarrow X_1$ 
 $X_{M+k+1} \leftarrow 0$  [Initialize  $y$ .]
 $X_{k+1} \leftarrow X_{M+k+1}$  [Copy  $y$ .]
 $X_{M+k+2} \leftarrow X_{M+k+2} + 1$  [Set switch.]
while  $X_{M+k+2} \neq 0$ 
   $X_0 \leftarrow 0$ 
   $\mathcal{P}$  [Compute  $g(\vec{x}, y)$ .]
   $X_{M+k+2} \leftarrow X_0$  [ $S = g(\vec{x}, y)$ .]
   $X_0 \leftarrow X_{M+k+1}$  [ $X_0 = y$ .]
   $X_{M+k+1} \leftarrow X_{M+k+1} + 1$  [Increment  $y$ .]
   $X_1 \leftarrow X_{M+1}$ 
   $X_2 \leftarrow X_{M+2}$ 
   $\dots$  [Restore  $\vec{x}$ .]
   $X_k \leftarrow X_{M+k}$ 
   $X_{k+1} \leftarrow X_{M+k+1}$  [Copy  $y$ .]
   $X_{k+2} \leftarrow 0$ 
   $X_{k+3} \leftarrow 0$ 
   $\dots$  [Clean up.]
   $X_M \leftarrow 0$ 
endwhile  $X_{M+k+2} \neq 0$ 

```

A program for minimalization