Contents

1 Simple Programs **3**
1.1 Empty Function 3
1.2 General Assignments and Read-Only Inputs 4
1.3 Composition 7
1.4 $\mathcal{P}$ and $\mathcal{R}$ are Closed under Composition 10
1.5 Iteration 11
1.6 Unbounded Search 17
1.7 Successor and Projection functions 20

Chapter 1

Simple Programs

These SUB-Notes are derived from our Main Notes and the Book. By removing a lot of prose they allow more isolation and hence concentration in learning the URM Example Programs, and why they tick.

1.1 Empty Function

1.1.1 Example. Let M be the program

$1 : \mathbf{x} \leftarrow 0$
$2 : \mathbf{goto}\ 1$
$3 : \mathbf{stop}$

Then $M^{\mathbf{x}}_{\mathbf{x}}$ is the empty function $\emptyset$, sometimes written as $\lambda x. \uparrow$.

M above computes $M^x_x = \emptyset$.
Thus the empty function is partial computable but nontotal.

We have just established $\emptyset \in \mathcal{P} - \mathcal{R}$.

Hence $\mathcal{R} \subsetneqq \mathcal{P}$.

□

1.2 General Assignments and Read-Only Inputs

1.2.1 Example. Let M be the program *segment*

$$
\begin{aligned}
k-1 &: \mathbf{x} \leftarrow 0 \\
k &: \textbf{if } \mathbf{z} = 0 \textbf{ goto } k+4 \textbf{ else goto } k+1 \\
k+1 &: \mathbf{z} \leftarrow \mathbf{z} \dot{-} 1 \\
k+2 &: \mathbf{x} \leftarrow \mathbf{x} + 1 \\
k+3 &: \textbf{ goto } k \ \{\text{Loop until } \mathbf{z} \text{ is } 0\} \\
k+4 &: \ldots \ \{\text{at this point } \mathbf{x} \text{ has the original value of } \mathbf{z}\}
\end{aligned}
$$

What it does:

By the time the computation reaches instruction $k+4$, the program segment has set the value of $\mathbf{z}$ to 0, and has made the value of $\mathbf{x}$ equal to the value that $\mathbf{z}$ had when instruction $k-1$ was current.

In short, the above sequence of instructions simulates what we would have written, say, in FORTRAN or C as

$$
\begin{aligned}
L &: \quad \mathbf{x} \leftarrow \mathbf{z} \\
L+1 &: \mathbf{z} \leftarrow 0 \\
L+2 &: \ldots
\end{aligned}
$$

where the FORTRAN and C semantics of $L: \ \mathbf{x} \leftarrow \mathbf{z}$ *are standard in programming*:

They require that, upon execution of the instruction, the value of $\mathbf{z}$ is *copied* into $\mathbf{x}$ but the value of $\mathbf{z}$ remains unchanged.

We say that the "normal" assignment $\mathbf{x} \leftarrow \mathbf{z}$ is *non destructive*. □

1.2.2 Exercise. Write a program segment that simulates precisely the FORTRAN L : $\mathbf{x} \leftarrow \mathbf{z}$; that is, copy the value of $\mathbf{z}$ into $\mathbf{x}$ without causing $\mathbf{z}$ to change as a side-effect.

□



Read-Only inputs.



Because of Exercise 1.2.2 above, *without loss of generality*, one may assume that any input variable, $\mathbf{x}$, of a URM M is *read-only*.

This means that its value is retained throughout any computation of the program.



Why "*without loss of generality*"? Because if an input variable $\mathbf{x}$ is not such, then we can *make* it be!



Indeed, let's add a new variable as an *input* variable, $\mathbf{x}'$ *instead* of $\mathbf{x}$.

Then, in detail, do this to make $\mathbf{x}'$ a read-only input variable:

- Add at the very beginning of M the instruction $1 : \mathbf{x} \leftarrow \mathbf{x}'$ of Exercise 1.2.2.
- Adjust all the following labels consistently, including, of course, the ones referenced by *if-statements*—a tedious but straightforward task.
- Call M' the so-obtained URM.

Clearly, $M'^{\mathbf{x}',\mathbf{y}_1,\ldots,\mathbf{y}_n}_{\mathbf{z}} = M^{\mathbf{x},\mathbf{y}_1,\ldots,\mathbf{y}_n}_{\mathbf{z}}$, and M' does not change $\mathbf{x}'$.

1.3 Composition

1.3.1 Example. (Composing Computable Functions)

Suppose that $\lambda x\vec{y}.f(x,\vec{y})$ and $\lambda\vec{z}.g(\vec{z})$ are *partial computable*, and say

$$f = F_{\mathbf{u}}^{\mathbf{x},\vec{\mathbf{y}}}$$

while

$$g = G_{\mathbf{x}}^{\vec{\mathbf{z}}}$$

We assume without loss of generality that $\mathbf{x}$ is the only variable common to F and G and is read-only in F. Thus, if we concatenate the programs G and F in that order, *and*

1. remove the last instruction of G (k : **stop**, for some k) —call the program segment that results from this G', and
2. renumber the instructions of F as $k, k+1, \ldots$ (and, as a result, renumber the references that if-statements of F make) in order to give $(G'F)$ the correct program structure,

then, $\lambda\vec{y}\vec{z}.f(g(\vec{z}),\vec{y}) = (G'F)_{\mathbf{u}}^{\vec{\mathbf{y}},\vec{\mathbf{z}}}$.

Note that all non-input variables of F will still hold 0 as soon as the execution of $(G'F)$ makes the first instruction of F current *for the first time*

because G, F have no common variables besides x, and x is READ ONLY for F.

□

Thus, we have, by repeating the above idea a finite number of times:

1.3.2 Proposition. *If* $\lambda\vec{y}_n.f(\vec{y}_n)$ *and* $\lambda\vec{z}.g_i(\vec{z})$, *for* $i = 1,\ldots,n$, *are partial computable, then so is* $\lambda\vec{z}.f(g_1(\vec{z}),\ldots,g_n(\vec{z}))$.



Note that

$$f(g_1(\vec{a}),\ldots,g_n(\vec{a}))\uparrow$$

if ANY $g_i(\vec{a})\uparrow$

Else $f(g_1(\vec{a}),\ldots,g_n(\vec{a}))\downarrow$ provided f is defined on all $g_i(\vec{a}_n)$.



For the record, we will define *composition* to mean the *somewhat rigidly defined operation* used in 1.3.2, that is:

1.3.3 Definition. Given any partial functions (computable or not) $\lambda\vec{y}_n.f(\vec{y}_n)$ and $\lambda\vec{z}.g_i(\vec{z})$, for $i = 1,\ldots,n$, we say that $\lambda\vec{z}.f(g_1(\vec{z}),\ldots,g_n(\vec{z}))$ is the result of their *composition*. □



We characterised the Definition 1.3.3 as "*rigid*". *Why?*

Indeed, note that it *requires all* the arguments of f to be substituted by some $g_i(\vec{z})$ —unlike Example 1.3.1, where we *substituted* a function call *only in* one *variable of* f there, and did nothing with the variables $\vec{y}$.

Also, for each call $g_i(\ldots)$ we require that the argument list, "...", *must be the same*; in 1.3.3 it was $\vec{z}$.

As we will show in examples later, this rigidity is only *apparent*.



1.4 $\mathcal{P}$ and $\mathcal{R}$ are Closed under Composition

We can rephrase 1.3.2, saying simply that

1.4.1 Theorem. $\mathcal{P}$ is *closed under* composition.

1.4.2 Corollary. $\mathcal{R}$ is closed under composition.

Proof. Let f, g_i be in $\mathcal{R}$.

Then they are in $\mathcal{P}$, hence so is $h = \lambda\vec{y}.f\Big(g_1(\vec{y}), \ldots, g_m(\vec{y})\Big)$ by 1.4.1.

By assumption, the f, g_i are total. So, for any $\vec{y}$, we have $g_i(\vec{y}) \downarrow$ —a number. Hence also $f\Big(g_1(\vec{y}), \ldots, g_m(\vec{y})\Big) \downarrow$.

That is, *h is total*, hence, *being in $\mathcal{P}$*, it *is also in $\mathcal{R}$*. □

Jan. 14, 2026

1.5 Iteration

Composing a number of times that *depends on the value of an input variable* —or as we may say, a *variable number of times*— is called *iteration*. The general case of iteration is called *primitive recursion*.

1.5.1 Definition. (Primitive Recursion) A number-theoretic function f is defined by *primitive recursion* from given functions $\lambda\vec{y}.h(\vec{y})$ and $\lambda x\vec{y}z.g(x,\vec{y},z)$ provided, *for all* $x,\vec{y}$, its values are given by the two equations below:

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

h is *called* the *Basis function*, while g is the *Iterator*.

We can take for granted a fundamental (but difficult) result (see EECS 1028, W20, course notes), that *a unique f that satisfies the above equations/requirements exists.*

Moreover, if both h and g are total, then so is f as it can easily be shown by induction on x.

□

Note that

1. $f(1,\vec{y}) = g(0,\vec{y},\overbrace{h(\vec{y})}^{f(0,\vec{y})})$,
2. $f(2,\vec{y}) = g(1,\vec{y},\overbrace{g(0,\vec{y},h(\vec{y}))}^{f(1,\vec{y})})$,
3. $f(3,\vec{y}) = g(2,\vec{y},\overbrace{g(1,\vec{y},g(0,\vec{y},h(\vec{y})))}^{f(2,\vec{y})})$, etc.

Thus the "x-value", 0, 1, 2, 3, etc., *equals the number of times we call g* (i.e., *the number of times we iterate g*).

With a little programming experience —but also see the preceding display-box on the previous page— it is easy to see that to compute $f(x, \vec{y})$ of 1.5.1 we can employ the simple pseudo code below:

Why "**pseudo**?"

$1 : z \leftarrow h(\vec{y})$
$2 :$ **for** $i = 0$ **to** $x \dot{-} 1$
$3 : z \leftarrow g(i, \vec{y}, z)$

At the end of the loop, z holds $f(x, \vec{y})$ —last value of i used in line 3 is $x \dot{-} 1$.

Here is how to *implement the above* as a URM:

1.5.2 Example. (Iterating Computable Functions)

Suppose that $\lambda x\vec{y}z.g(x, \vec{y}, z)$ and $\lambda\vec{y}.h(\vec{z})$ are partial computable, and, say, $g = G^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}_{\mathbf{z}}$ while $h = H^{\vec{\mathbf{y}}}_{\mathbf{z}}$.

By earlier remarks we may assume:

(i) The only variables that H and G have in common are $\mathbf{z}, \vec{\mathbf{y}}$.
(ii) The variables $\vec{\mathbf{y}}$ are read-only in both H and G.
(iii) $\mathbf{i}$ is read-only in G and does not appear in H.
(iv) $\mathbf{x}$ does not occur in any of H or G.

We can now see that the following URM program, let us call it F, computes the f of Definition 1.5.1 for which we wrote the easy pseudo code on page 13 (we *reproduce* it here for convenience):

$$\begin{array}{l} 1: z \leftarrow h(\vec{y}) \\ 2: \textbf{for } i = 0 \textbf{ to } x - 1 \\ 3: z \leftarrow g(i, \vec{y}, z) \end{array} \qquad (1)$$

LAST VALUE of i in the loop is $x-1$ (x being the original value of x).

The sequence of z-numbers computed is:

$$\overbrace{f(0,\vec{y})}^{f(0,\vec{y})=h(\vec{y})}, \overbrace{g(0,\vec{y},f(0,\vec{y}))}^{f(1,\vec{y})}, \overbrace{g(1,\vec{y},f(1,\vec{y}))}^{f(2,\vec{y})}, \ldots, \overbrace{g(x-1,\vec{y},f(x-1,\vec{y}))}^{f(x,\vec{y})}$$

The "**for**"-line ensures we iterate while indexing our iterations $i = 0, 1, \ldots, x \dot{-} 1$ —that is, x times (0 to $x \dot{-} 1$).

Instead of using a **for**-loop we can program (1) in URM Programming Language using "**goto**".

Incidentally, "{...}" indicates a comment in a program and so does /*...*/

$$\begin{array}{ll} 1: z \leftarrow & h(\vec{y}) \ \{= f(0,\vec{y})\} \\ 2: i \leftarrow & 0 \\ 3: \textbf{if } x = 0 \textbf{ goto } 8 \ \{\text{exit the loop}\} \textbf{ else goto } 4 & \\ 4: x \leftarrow & x \dot{-} 1 \\ 5: z \leftarrow & g(i,\vec{y},z) \ \{\text{1st time: } i = 0, \text{ 2nd time: } i = 1; \text{ 3rd time } i = 2, \text{ etc.}\} \\ 6: i \leftarrow & i + 1 \ \{i\text{-value used in line "5:" is the one before increase here!}\} \\ 7: \textbf{goto } 3 & \\ 8: \textbf{stop} & \end{array}$$

The URM F below uses URMs H and G for h and g and directly translates the above.

$\boxed{H'}$ is program H with the **stop** instruction removed, $\boxed{G'}$ is program G that has the **stop** instruction removed, and instructions renumbered (and if-statements adjusted) as needed:

	$\boxed{H'}_{\mathbf{z}}^{\vec{\mathbf{y}}}$
$r:$	$\mathbf{i} \leftarrow 0$ {You need the i to be faithful to (1)}
$r+1:$	**if** $\mathbf{x} = 0$ **goto** $k+m+2$ **else goto** $r+2$ $\{x \searrow 0$ as $i \nearrow x-1\}$
$r+2:$	$\mathbf{x} \leftarrow \mathbf{x} \dot{-} 1$
	$r{+}3{:}\boxed{G'}_{\mathbf{z}}^{\mathbf{i},\vec{\mathbf{y}},\mathbf{z}}$
$k:$	$\mathbf{i} \leftarrow \mathbf{i}+1$ {i-value used in G' above is the one before advancing it}
$k+1:$	$\mathbf{w}_1 \leftarrow 0$
$\vdots$	$\vdots$
$k+m:$	$\mathbf{w}_m \leftarrow 0$
$k+m+1:$	**goto** $r+1$
$k+m+2:$	**stop** /*$\mathbf{x} = 0$ and $\mathbf{i} =$ orig. $\mathbf{x}$; last $\mathbf{i}$-value *in* G' is $\mathbf{x} \dot{-} 1$*/

The instructions $\mathbf{w}_i \leftarrow 0$ set explicitly to zero all the variables of G' other than $\mathbf{i}, \mathbf{z}, \vec{\mathbf{y}}$ to ensure correct behavior of G'. Note that the $\mathbf{w}_i$ are *implicitly* (by the I/O Agent) initialized to zero *ONLY* the *FIRST* time G' is executed. Clearly, the URM F simulates the pseudo program above, thus $f = F_{\mathbf{z}}^{\mathbf{x},\vec{\mathbf{y}}}$. □

We just proved:

1.5.3 Proposition. *If f, g, h relate as in Definition 1.5.1 and h and g are in $\mathcal{P}$, then so is f. We say that $\mathcal{P}$ is* closed under primitive recursion.

1.5.4 Corollary. *If f, g, h relate as in Definition 1.5.1 and h and g are in $\mathcal{R}$, then so is f. We say that $\mathcal{R}$ is* closed under primitive recursion.

Proof. As $\mathcal{R} \subseteq \mathcal{P}$, we have $f \in \mathcal{P}$.

But we noted earlier that if h and g is total, then so is f.

So, $f \in \mathcal{R}$. □

1.6 Unbounded Search

What does the following pseudo program do, if $g = G_{\mathbf{z}}^{\mathbf{x},\vec{\mathbf{y}}}$, for some URM G and read only $\mathbf{x}, \vec{\mathbf{y}}$?

$$\begin{aligned} &1 : \mathbf{x} \leftarrow 0 \\ &2 : \textbf{while } g(\mathbf{x}, \vec{\mathbf{y}}) \neq 0 \textbf{ do} \\ &3 : \mathbf{x} \leftarrow \mathbf{x} + 1 \end{aligned} \tag{1}$$

OK. Fix an input $\vec{\mathbf{y}}$.

We are **out of the loop** here (exited the **while**-loop) precisely *because*

- Testing for $g(\mathbf{x}, \vec{\mathbf{y}}) \neq 0$ *never* got stuck in the loop as a result of calling g with some $\mathbf{x} = m$ that makes $g(m, \vec{\mathbf{y}}) \uparrow$.
- The loop kicked us out *exactly* when $g(k, \vec{\mathbf{y}}) = 0$ was detected, for some $\mathbf{x} = k$, *for the first time*, in the **while**-test.

In short, the k satisfies

- $$k = \textit{smallest} \text{ such that } g(k, \vec{\mathbf{y}}) = 0$$
- and $(\forall z)(z < k \rightarrow g(z, \vec{\mathbf{y}}) \downarrow)$

Now, this k *depends* on $\vec{\mathbf{y}}$ so we may define it as a function f, *for any INPUT* $\vec{a}$ *assigned into* $\vec{\mathbf{y}}$, by:

$$k = f(\vec{a}) \stackrel{Def}{=} \min\Big\{x : g(x, \vec{a}) = 0 \wedge (\forall y)\big(y < x \rightarrow g(y, \vec{a}) \downarrow\big)\Big\} \quad \square$$

Kleene has suggested the symbol "(μy)" to denote the "find the minimum y" operation above, thus the above is rephrased as

$$f(\vec{a}) = (\mu x)g(x,\vec{a}) \stackrel{Def}{=} \begin{cases} \min\left\{x : g(x,\vec{a}) = 0 \wedge (\forall w)_{<y} g(w,\vec{a}) \downarrow \right\} \\ \uparrow \text{ if the min above does not exist} \end{cases} \tag{2}$$

where $(\forall y)_{<x} R(y,\ldots)$ or also $(\forall y)_{y<x} R(y,\ldots)$

is short for $(\forall y)(y < x \to R(y,\ldots))$.

We call the operation $(\mu x)g(x,\vec{a})$ —equivalently, the **program segment** "**while** $g(\mathbf{x},\vec{a}) \neq 0$ **do**"— *unbounded search*.

Why "*unbounded*" search? Because we do not know *a priori* how many times we have to go around the loop. This depends on the behaviour of g.

We saw how the minimum can fail to exist if $g(x,\vec{a}) \downarrow$ for all x but we *never* get $g(x,\vec{a}) = 0$; that is, we stay in the loop *going round and round forever because g-values are NONzero*.

Can we implement the pseudo-program (1) (reproduced below) as a URM F?

YES!

$$\begin{aligned} &1 : \mathbf{x} \leftarrow 0 \\ &2 : \textbf{while } g(\mathbf{x}, \vec{\mathbf{y}}) \neq 0 \textbf{ do} \\ &3 : \mathbf{x} \leftarrow \mathbf{x} + 1 \end{aligned} \tag{1}$$

1.6.1 Example. (Unbounded Search on a URM) So suppose again that $\lambda x\vec{y}.g(x, \vec{y})$ is total computable, and, say, $g = G^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$.

By earlier remarks we may assume that $\vec{\mathbf{y}}$ and $\mathbf{x}$ are read-only in G and that $\mathbf{z}$ is *not* one of them.

Consider the following program $F^{\vec{\mathbf{y}}}_{\mathbf{x}}$, where $\boxed{G'}$ is the program G with the **stop** instruction removed, *and* instructions have been renumbered (and if-statements adjusted) as needed so that its first instruction has label 2.

1 : $\mathbf{x} \leftarrow 0$
2: $\boxed{G'}^{\mathbf{x},\vec{\mathbf{y}}}_{\mathbf{z}}$
k : **if** $\mathbf{z} = 0$ **goto** $k + l + 3$ **else goto** $k + 1$
$k + 1$: $\mathbf{w}_1 \leftarrow 0$ {**Comment.** Setting *all non-input variables* to 0; cf. 1.5.2.}
$\vdots$
$k + l$: $\mathbf{w}_l \leftarrow 0$ {**Comment.** Setting *all non-input variables* to 0; cf. 1.5.2.}
$k + l + 1$: $\mathbf{x} \leftarrow \mathbf{x} + 1$
$k + l + 2$: **goto** 2
$k + l + 3$: **stop** {**Comment.** Read answer off $\mathbf{x}$. This is the last $\mathbf{x}$-value used by G'}

□

1.7 Successor and Projection functions

The functions $\lambda x.0$, also denoted by Z, $\lambda x.x+1$, also denoted by S, and $\lambda \vec{y}_n.y_i$ also denoted by U_i^n are computable —and being total— are both recursive:

URMs for these three that prove recursiveness are

1.7.1 Example. (For Z) We compute using $M_{\mathbf{x}}^{\mathbf{x}}$ below:

$$1 : \mathbf{x} \leftarrow 0$$
$$2 : \textbf{stop}$$

□

1.7.2 Example. (For S) We compute using $N_{\mathbf{x}}^{\mathbf{x}}$ below:

$$1 : \mathbf{x} \leftarrow \mathbf{x} + 1$$
$$2 : \textbf{stop}$$

□

1.7.3 Example. (For U_i^n) We compute using $P_{\mathbf{z}}^{\vec{\mathbf{x}}_n}$ below:

$$P\begin{cases} 1: & \mathbf{w} \leftarrow \mathbf{x}_1 \\ 2: & \mathbf{w} \leftarrow \mathbf{x}_2 \\ 3: & \mathbf{w} \leftarrow \mathbf{x}_3 \\ & \vdots \\ i: & \mathbf{z} \leftarrow \mathbf{x}_i \\ i+1: & \mathbf{w} \leftarrow \mathbf{x}_{i+1} \\ & \vdots \\ n-1: & \mathbf{w} \leftarrow \mathbf{x}_{n-1} \\ n: & \mathbf{w} \leftarrow \mathbf{x}_n \\ n+1: & \textbf{stop} \end{cases}$$

The U_i^n functions are normally called "projection functions".

A better name is "generalised identities", as they are obtained from $\lambda x.x$ by adding input variables that are NOT used (some people call these unused input variables "**dummy** variables"). □

1.7.4 Example. Let the function $\lambda\vec{x}_n.f(\vec{x}_n)$ be (partial[†]) computable.

Then so is the function $\lambda x_2 x_3 \dots x_{n-1} x_n.f(5, x_2, x_3, \dots, x_{n-1}, x_n)$.

Indeed, by 1.7.1, 1.7.2 and 1.7.3, and closure of $\mathcal{P}$ under composition,

$$g = \lambda x_2 x_3 \dots x_{n-1} x_n.f\Big(\overbrace{S(S(S(S(S(Z(x_2))))))}^{5}, x_2, \dots, x_n\Big)$$

is computable (by substitution/composition). □

[†]Includes the subcase of being *total* computable!

1.7.5 Example. (Expansion of Argument List) Let the function $\lambda\vec{x}_n.f(\vec{x}_n)$ be (partial) computable. Then so is

$$\lambda\vec{x}_n\vec{\mathbf{w}}_m.f(\vec{x}_n) \tag{1}$$

To acknowledge the inputs $\vec{\mathbf{w}}_m$ we may introduce a new function letter, say g, and define the function it represents to be the same as (equal to) (2) below.

Indeed,

$$\begin{aligned}\lambda\vec{x}_n\vec{\mathbf{w}}_m.g(x_1,\ldots,x_n,\vec{\mathbf{w}}_1,\ldots,\vec{\mathbf{w}}_m) = \\ \lambda\vec{x}_n\vec{\mathbf{w}}_m.f\Big(U_1^{n+m}(\vec{x}_n,\vec{\mathbf{w}}_m),\ldots,U_n^{n+m}(\vec{x}_n,\vec{\mathbf{w}}_m)\Big)\end{aligned} \tag{2}$$

□

1.7.6 Example. (Identification of Two Variables) Suppose

$$f = \lambda\vec{x}_n.f(x_1,\ldots,\overbrace{x_i,\ldots,x_k}^{identify\ x_i\ and\ x_k},\ldots x_n) \tag{1}$$

is computable. How about

$$\begin{aligned}g = \lambda z\vec{x}_n.f(x_1,\ldots,\overbrace{z,\ldots,z}^{identified\ x_i\ and\ x_k\ above\ as\ new\ z},\ldots x_n)\\ = \lambda z\vec{x}_n.f\Big(\overbrace{x_1}^{pos\ 1},\ldots,U_1^1(\overbrace{z}^{pos\ i\ was\ x_i}),\ldots U_1^1(\overbrace{z}^{pos\ k\ was\ x_k}),\\ \ldots,\overbrace{x_n}^{pos\ n}\Big)\end{aligned}$$

Above we substituted two computable function calls, $U_1^1(z)$, into two variables, x_i and x_k. □

1.7.7 Example. (Swapping Two Variables) Suppose

$$f = \lambda\vec{x}_n.f(x_1,\ldots,\overbrace{x_i,\ldots,x_k}^{swap\ x_i\ and\ x_k},\ldots x_n) \tag{1}$$

is computable.

I want:

$$f = \lambda \vec{x}_n.f(\ldots, \overbrace{x_k}^{\text{subst. of } U_k^n(\vec{x}_n) \text{ into } x_i}, \ldots, \overbrace{x_i}^{\text{subst. of } U_i^n(\vec{x}_n) \text{ into } x_k}, \ldots) \quad (2)$$

□

1.7.8 Example. (The Add Function) The function below is computable. Show by providing a primitive recursive definition:

$$Add(0, y) = y$$
$$Add(x + 1, y) = Add(x, y) + 1$$

Fully "dressed" this looks like a "proper" Primitive Recursion with "h" and "g" and all!

$$Add(0, y) = \overbrace{U_1^1}^{\text{“}h\text{”}}(y)$$
$$Add(x + 1, y) = \overbrace{SU_3^3}^{\text{“}g\text{”}}\Big(x, y, Add(x, y)\Big)$$

□

1.7.9 Example. (The Switch Function) The function below is computable. Show by providing a primitive recursive definition::

$$switch(x, y, z) = \textbf{if } x = 0 \textbf{ then } y \textbf{ else } z$$

To rephrase,

$$switch(0, y, z) = y$$
$$switch(x + 1, y, z) = z$$

The above "looks like" a primitive recursion, but the format is incorrect: We need an "h-part" with **two** variables, y, z and a "g-part" with **four** variables, x, y, z, w. Here it is:

$$switch(0, y, z) = \overbrace{U_1^2}^{h}(y, z)$$
$$switch(x + 1, y) = \overbrace{U_3^4}^{g}(x, y, z, switch(x, y, z))$$

Fully "dressed" this looks like a "proper" Primitive Recursion with "h" and "g" and all!

□

1.7.10 Example. (The $x \dot{-} 1$ and $1 \dot{-} x$ Functions) (a) $\lambda x.x \dot{-} 1$ is total computable.

We can see it is total (defined for all x) so let us use a primitive recursion to compute it and thus show it is computable. Plainly,

$$0 \dot{-} 1 = 0$$
$$(x+1) \dot{-} 1 = x \text{ \{Redundant brackets used for clarity only\}}$$

To "dress it up" we need to add a variable —y— that does nothing ("dummy variable") since h-parts in primitive recursions must have at least on variable. So compute this function instead

$$f = \lambda xy.x \dot{-} 1 \tag{1}$$

Here it is:

$$f(0, y) = Z(y) \text{ \{Recall: } Z \text{ stands for the zero function\}}$$
$$f(x+1, y) = U_1^3(x, y, f(x))$$

The function $\lambda x.x \dot{-} 1$ we shall call "*Pred*" or "Predecessor".

(b) For $\lambda x.1 \dot{-} x$ we can use the switch function that we proved computable, noting

$$1 \dot{-} x = switch(x, 1, 0)$$

HMMM! Is it true that if I plug constants into variables of a computable function I get a new computable one? WHY? (shhhh... because I can view 1 and 0 as $\lambda x.1$ and $\lambda x.0$! SO???)

□

1.7.11 Example. (The $x \dot{-} y$ Function) We have done the "$x+y$" function ("*Add*"). It is fair to now do $x \dot{-} y$ and learn something that was nagging at us (WAS it?)

Here is a primitive recursion that does it:

$$\begin{aligned} x \dot{-} 0 &= x \\ x \dot{-} (y+1) &= (x \dot{-} y) \dot{-} 1 = Pred(x \dot{-} y) \end{aligned} \tag{$\dagger$}$$

Before I dress the above up with all the bells and whistles, here is the *nagging* question: Am I allowed to do my prim. recursion along a variable OTHER THAN THE FIRST? Well, "first", "second" etc. are due to an arbitrary toss around of the input variables in the λ-list.

If I view the function that outputs $x \dot{-} y$ with a "$\lambda yx.$" input list —rather than a "$\lambda xy.$" one— then y is *the* "first", and I am consistent with the Prim. Rec. format of equations the function if I write down the recursion †. So, $\lambda yx.x \dot{-} y$ is OK!

So, dressing up the recursion, and letting $Diff(y,x) \overset{Def}{=} x \dot{-} y$, then I get the equations of proper format

$$\begin{aligned} Diff(0,x) &= x \\ Diff(y+1,x) &= Pred\Big(U_3^3(y,x,Diff(y,x))\Big) \end{aligned}$$

> All these dressing up games are for us to be comfortable with the proposition that we are not cheating in our use of primitive recursion. We will skip the "dress-up" part, if we are not asked.

□

1.7.12 Example. (The Simple Exponential and an Enormous Function) By simple exponential i mean $\lambda x.2^x$.

Ok, computable for sure; here is a primitive recursion that does it:

$$\begin{aligned} 2^0 &= 1 \\ 2^{x+1} &= 2^x + 2^x = Add(2^x, 2^x) \end{aligned} \tag{1}$$

We can now define via a primitive recursion the function $\lambda x.Gi(x)$ —"Gi" for **Gigantic**!

$$\begin{aligned} Gi(0) &= 1 \\ Gi(x+1) &= 2^{Gi(x)} \end{aligned} \tag{2}$$

HOMEWORK (Ungraded, for now) Prove by induction on x that

$$Gi(x) = 2^{2^{\cdot^{\cdot^{\cdot^{2}}}}} \Big\} \; x \text{ 2's}$$

For intuition's sake: Note that
$Gi(0)$ = ladder of zero "2", coded as "1"
$Gi(1)$ = ladder of one "2"
$Gi(2)$ = ladder of two "2^2"
Take it from here!

□

Bibliography

[Dav65] M. Davis, *The undecidable*, Raven Press, Hewlett, NY, 1965.