Recursive Functions

Computability and Logic

What we're going to do

- We're going to define the class of recursive functions.
- These are all functions from natural numbers to natural numbers.
- We'll show that all recursive functions are Abacus-computable* (and therefore also Turing-computable)
- Later, we'll also show that all functions (from natural numbers to natural numbers) that are Turing-computable* are recursive.
- Hence, the class of recursive function will coincide with the set of all Turing-computable* functions (and therefore also with the set of all Abacus-computable* functions).

Primitive Functions

- z: the zero function
  - $z(x) = 0$ for all x
- s: the successor function
  - $s(x) = x + 1$ for all x
- id^n_m: the identity function
  - $id^n_m(x_1, \ldots, x_n) = x_m$ for any $x_1, \ldots, x_n$ and $1 \leq m \leq n$

- All primitive recursive functions are recursive functions

Operations on Functions

- On the next slides we'll cover 3 operations on functions that generate functions from functions:
  - Composition
  - Primitive Recursion
  - Minimization
- Performing any of these 3 operations on recursive functions generates a new recursive function.
- In fact, the set of recursive functions is defined as the set of all and only those functions that can be obtained from the primitive functions, and the processes of composition, primitive recursion, and minimization.

Composition

- Given functions $f(x_1, \ldots, x_m)$, $g_1(x_1, \ldots, x_n)$, …, $g_m(x_1, \ldots, x_n)$, the composition function $Cn[f, g_1, \ldots, g_n]$ is defined as follows:
  - $Cn[f, g_1, \ldots, g_m](x_1, \ldots, x_n) = f(g_1(x_1, \ldots, x_n), \ldots, g_m(x_1, \ldots, x_n))$

Examples Composition

- Cn[s,z]:
  - For any x: Cn[s,z](x) = s(z(x)) = s(0) = 1
  - So, where $const_1(x) = 1$ for all x, we now know that $const_1$ is a recursive function.
- Cn[s,$const_1$]
  - For any x: Cn[s,$const_1$](x) = s($const_1$(x)) = s(1) = 2
  - So, where $const_2(x) = 2$ for all x, we now know that $const_2$ is a recursive function.
- Etc.
- We thus have that for any n: where $const_n(x) = n$ for all x, $const_n$ is a recursive function.

Primitive Recursion

- Like composition, primitive recursion is a well-known operation:
  - Define the 0 case
  - Define the x+1 case in term of the x case

- However, before formally defining the operation of primitive recursion, let's start with some intuitive examples.

Addition

- We can define addition in the following recursive way:
  - $x + 0 = x$
  - $x + y' = (x + y)'$ (where x' is the successor of x)

- Somewhat more formally:
  - $sum(x,0) = x$
  - $sum(x,y') = sum(x,y)'$

Multiplication and Exponentiation

- We can likewise define multiplication recursively:
  - prod(x,0) = 0
  - prod(x,y') = prod(x,y) + x

- And exponentiation:
  - exp(x,0) = 1 (so this assumes we define $0^0 = 1$)
  - exp(x,y') = exp(x,y) * x

Formal Definition of Primitive Recursion

- Given functions $f(x_1, \ldots, x_n)$ and $g(x_1, \ldots, x_n, y, z)$, the primitive recursive function Pr[f,g] is defined as follows:
  - $Pr[f,g](x_1, \ldots, x_n, 0) = f(x_1, \ldots, x_n)$
  - $Pr[f,g](x_1, \ldots, x_n, y') = g(x_1, \ldots, x_n, y, Pr[f,g](x_1, \ldots, x_n, y))$
  - (so the 'y' is the variable you are recursing over down to 0. This recursion happens when we plug in $Pr[f,g](x_1, \ldots, x_n, y)$ for z in $g(x_1, \ldots, x_n, y, z)$)
- If f and g are recursive, then Pr[f,g] is recursive as well.

Examples Primitive Recursive Functions in Formal Format

- Addition:
  - sum(x,0) = x, i.e. f(x) = x, i.e. f = id^1_1
  - sum(x,y') = sum(x,y)', i.e. g(x,y,z) = s(z), i.e. g = Cn[s, id^3_3]
  - So, sum(x,y) = Pr[id^1_1, Cn[s, id^3_3]]

- Multiplication:
  - prod(x,0) = 0, i.e. f(x) = 0, i.e. f = z
  - prod(x,y') = prod(x,y) + x, i.e. g(x,y,z) = z + x, i.e. g = Cn[sum,id^3_3,id^3_1]
  - So, prod(x,y) = Pr[z, Cn[sum,id^3_3,id^3_1]]
  - Filling in sum: prod(x,y) = Pr[z, Cn[Pr[id^1_1, Cn[s, id^3_3]],id^3_3,id^3_1]]

- I'll spare you the formal exponentiation case!

Factorial

- Let's do the 'classic' recursive definition.
- Factorial:
  - fac(0) = 1
  - fac(y') = fac(y) * y'
- If we try to put this into a formal format, we find we have a problem: for f, we need to use a const1 function, but … there is no x to use as its argument!

Using Dummy Variables

- So, we'll add a 'dummy' variable:
  - dummyfac(x,0) = 1
  - dummyfac(x,y') = dummyfac(x,y) * y'
  - So, dummyfac(x,y) = Pr[$const_1$,Cn[prod,id^3_3,Cn[s,id^3_2]]]
  - fac(x) = dummyfac(x,x), i.e. fac(x) = Cn[dummyfac,id^1_1,id^1_1]
  - So, fac(x) = Cn[Pr[$const_1$,Cn[prod,id^3_3,Cn[s,id^3_2]]],id^1_1,id^1_1] = Cn[Pr[Cn[s,z],Cn[Pr[z, Cn[Pr[id^1_1, Cn[s, id^3_3]], id^3_3, id^3_1]], id^3_3, Cn[s,id^3_2]]], id^1_1, id^1_1]
- Yikes!
- From now on, we'll accept the informal (and certainly much more readable!) explication of the factorial function as a demonstration that the factorial is a recursive function.

Bounded Sum

- Where f(x_1, …,x_n, y) is a function, define Sum[f] as follows:

$$\text{Sum}[\,f\,](x_1,\ldots x_n,y)=\sum_{i=0}^{y} f\,(x_1,\ldots,x_n,i)$$

- If f is recursive, then Sum[f] is recursive, since:

$$\text{Sum}[\,f\,](x_1,\ldots x_n,0)=\ f\,(x_1,\ldots x_n,0)$$

$$\text{Sum}[\,f\,](x_1,\ldots x_n,y')=\text{Sum}[\,f\,](x_1,\ldots x_n,y)+\ f\,(x_1,\ldots x_n,y')$$

Bounded Product

- Where f(x_1, …,x_n, y) is a function, define Prod[f] as follows:

$$\text{Prod}[\, f\,](x_1,\ldots x_n, y) = \prod_{i=0}^{y} f\,(x_1,\ldots, x_n, i)$$

- If f is recursive, then Prod[f] is recursive, since:

$$\text{Prod}[\, f\,](x_1,\ldots x_n, 0) = f\,(x_1,\ldots x_n, 0)$$

$$\text{Prod}[\, f\,](x_1,\ldots x_n, y') = \text{Prod}[\, f\,](x_1,\ldots x_n, y) \times f\,(x_1,\ldots x_n, y')$$

Modified Predecessor

- Let pred(x) = x-1 for x > 0, and pred(0) = 0
- pred(x) is recursive, since:
  - pred(0) = 0
  - pred(y') = y

- So notice that we can fit the pred function into the (informal) format of the primitive recursion operation, even though it doesn't actually recurse in any way!

Modified Difference

- Define diff(x,y):
  - diff(x,y) = x – y if y ≤ x
  - diff(x,y) = 0 otherwise.

- diff(x,y) is recursive, since:
  - diff(x,0) = x
  - diff(x,y') = pred(diff(x,y))

Signum Functions

- Define sg(x) = 1 if x > 0, and sg(0) = 0.
- Define sg'(x) = 0 if x > 0, and sg'(0) = 1.

- These are recursive, because:
  - sg'(x) = diff(1,x)
  - sg(x) = diff(1,sg'(x))

Primitive Recursive Functions

- The set of all primitive recursive functions is the set of all and only those functions that can be obtained from the primitive functions, and the processes of composition and primitive recursion (so no minimization!)
- Clearly:
  - All primitive recursive functions are recursive functions
  - In fact, all functions we have seen so far are primitive recursive functions
  - In particular, all primitive functions (z,s,id) are primitive recursive functions
- Also, all primitive recursive functions are total functions
  - So for a computable partial function to be recursive, we need to expand our definition of recursive functions

Minimization

- Where $f(x_1, \ldots, x_n, y)$ is a function, define Mn[f] as follows:
  - $Mn[f](x_1, \ldots, x_n) = y$ if $f(x_1, \ldots, x_n, y) = 0$, and for all $x < y$: $f(x_1, \ldots, x_n, x) > 0$
  - $Mn[f](x_1, \ldots, x_n)$ = [undefined] otherwise

- By definition, if f is recursive, then Mn[f] is recursive as well.

- Notice that $Mn[f](x_1, \ldots, x_n)$ can be undefined for 2 reasons:
  - If is a total function, but for all y: $f(x_1, \ldots, x_n, y) \neq 0$
  - If is a partial function, and you get to an undefined value before you get to a 0