



Primitive and Partial Recursive Functions

Robert M. Keller
Harvey Mudd College
April 2010



What is this?

- An alternate approach to computability, based on **numeric** functions.
- Sometimes having this alternate viewpoint will be helpful, and often more elegant.
- Also, much common terminology is derived from this approach rather than from Turing machines.
- The family of primitive recursive functions is first defined, then partial recursive functions are built on that.



Numbers vs. Strings

- What do we lose with numbers? Nothing really, as strings over any alphabet can be regarded as numerals:

Over $\Sigma = \{a, b, c\}$ for example:

$\varepsilon \leftrightarrow 0$
 $a \leftrightarrow 1$
 $b \leftrightarrow 2$
 $c \leftrightarrow 3$
 $aa \leftrightarrow 4$
 $ab \leftrightarrow 5$
...

- For an n -letter alphabet, this is called the ***n-adic encoding***.



String Manipulation by Numbers

- In an n -adic encoding:
 - dividing by n is truncating the last character.
 - modding by n is getting the last character.
 - multiplying by n and adding the value of a character is like consing the a character on the end.
- A Turing machine tape can be represented as two stacks, then the stacks pushed and pop by the above numeric operations, to simulate a Turing machine.



Primitive Recursive Functions

- The set of **primitive recursive functions** is defined inductively.
- Every function is k -ary, for some $k > 0$.
- The domain and co-domain of each function is the set of natural numbers $\{0, 1, 2, 3, \dots\}$, or k -tuples in the case of domain.



Basis Functions (1 of 3)

- The **zero** function is primitive recursive:

$$\text{zero}(x) = 0$$



Basis Functions (2 of 3)

- The **projection** functions are all primitive recursive:

$$\pi_j^k(x_1, x_2, \dots, x_k) = x_j$$

for each arity $k \geq 1$ and each i , $1 \leq i \leq k$.



Basis Functions (3 of 3)

- The **successor** function is primitive recursive:

$$S(x) = x + 1$$



Induction Rules (1 of 2)

- The **composition** of primitive recursive functions is primitive recursive:

$$h(x_1, x_2, \dots, x_k) =$$
$$f(g_1(x_1, x_2, \dots, x_k),$$
$$g_2(x_1, x_2, \dots, x_k),$$
$$\dots,$$
$$g_r(x_1, x_2, \dots, x_k))$$

for each pair of arities $k, r > 0$.



Constant Functions

- A consequence of the rules up to this point is that **constant** functions are all primitive recursive:

$$C_c^k(x_1, x_2, \dots, x_k) = c$$

for each natural number c .

This is so because is just a composition of the zero and successor functions:

$$C_c^k(x_1, x_2, \dots, x_k) = S(S(\dots S(\text{zero}(\pi_1^k(x_1, x_2, \dots, x_k))) \dots))$$



Explicit Definition (ED)

- This is a convenient shorthand for combining compositions, projections, and constants. We can just use definitions such as:

$$f(x, y, z) = g(h(y, x), 5, k(z, z))$$

and know that if g , h , and k are primitive recursive, so is f ,

because we can exhibit the corresponding composition of zero, S , and projections.



Example of Explicit Definition

- $f(x, y, z) = g(h(y, x), 5, k(z, z))$

is equivalent to:

- $f(x, y, z) = g(h(\pi^3_2(x, y, z), \pi^3_1(x, y, z)),$
 $S(S(S(S(S(\text{zero}(\pi^3_1(x, y, z))))))),$
 $k(\pi^3_2(x, y, z), \pi^3_2(x, y, z)))$

- ED is also sometimes called ET (Explicit Transformation)



Induction Rules (2 of 2)

- A function **f** defined from already-defined primitive recursive functions **b** and **r** of the appropriate arities by the following **primitive recursion** pattern is primitive recursive:

$$f(0, x_1, x_2, \dots, x_k) = b(x_1, x_2, \dots, x_k)$$

$$f(n+1, x_1, x_2, \dots, x_k) =$$

$$r(x_1, x_2, \dots, x_k, n, f(n, x_1, x_2, \dots, x_k))$$

- Note that primitive recursion is a very stylized template. Not every recursion fits into this template.



Examples of Primitive Recursive Functions

- $\text{add}(x, y)$: addition
- $\text{mult}(x, y)$: multiplication
- $\text{pred}(x)$: predecessor
- $\text{sub}(x, y)$: proper subtraction*
- $\text{mod}(x, y)$: modulus
- $\text{div}(x, y)$: integer division (quotient)
- $\text{sqrt}(x)$: integer square root

* $\text{sub}(x, y)$ is 0 if $x < y$



rex implementations

- I will demonstrate some of these using explicit definition in rex. This allows the definitions to be tested readily.
- rex does not restrict to natural numbers and does not enforce a primitive recursive formalism, so we have to be careful not to “cheat”.



add implementation in rex

- $S(n) = n + 1$; // pretend this definition is built in
- $\text{add}(0, y) \Rightarrow y$;
- $\text{add}(n+1, y) \Rightarrow S(\text{add}(n, y))$;
- For reference (identify b and r above):

$$f(0, x_1, x_2, \dots, x_k) = b(x_1, x_2, \dots, x_k)$$

$$f(n+1, x_1, x_2, \dots, x_k) =$$

$$r(x_1, x_2, \dots, x_k, n, f(n, x_1, x_2, \dots, x_k))$$



mult implementation

- $\text{mult}(0, y) \Rightarrow 0;$
- $\text{mult}(n+1, y) \Rightarrow \text{add}(y, \text{mult}(n, y));$
- For reference (identify b and r above):

$$f(0, x_1, x_2, \dots, x_k) = b(x_1, x_2, \dots, x_k)$$

$$f(n+1, x_1, x_2, \dots, x_k) =$$

$$r(x_1, x_2, \dots, x_k, n, f(n, x_1, x_2, \dots, x_k))$$



pred (predecessor) implementation

- informally $\text{pred}(y) = y = 0? 0 : y-1;$
- $\text{pred}(0) =>$
- $\text{pred}(n+1) =>$
- For reference (identify b and r above):

$$f(0, x_1, x_2, \dots, x_k) = b(x_1, x_2, \dots, x_k)$$

$$f(n+1, x_1, x_2, \dots, x_k) =$$

$$r(x_1, x_2, \dots, x_k, n, f(n, x_1, x_2, \dots, x_k))$$



sub implementation

- sub is **proper** subtraction (aka "monus"):
If $a \geq b$, then $\text{sub}(a, b) = a - b$.
If $a < b$, then $\text{sub}(a, b) = 0$.
- $\text{sub}(y, 0) \Rightarrow$
- $\text{sub}(y, n+1) \Rightarrow$



Primitive Recursive *Predicates*

- For some definitions we want to have predicates, which we can equate to functions that return only values 0 (false) and 1 true.
- $\text{sgn}(0) \Rightarrow 0$;
- $\text{sgn}(n+1) \Rightarrow 1$;
- sgn converts arbitrary values to $\{0, 1\}$.



Negation

- $\text{not}(0) \Rightarrow 1;$
- $\text{not}(n+1) \Rightarrow 0;$



Equality Predicate

- $\text{eq}(x, y) = \text{not}(\text{add}(\text{sub}(x, y), \text{sub}(y, x)))$;



if-then-else function

- $\text{ifthenelse}(0, x, y) \Rightarrow y;$
- $\text{ifthenelse}(n+1, x, y) \Rightarrow x;$



mod and div

- $\text{mod}(0, y) \Rightarrow 0;$
- $\text{mod}(n+1, y) \Rightarrow \text{ifthenelse}(\text{eq}(\text{S}(\text{mod}(n, y)), y),$
 $0,$
 $\text{S}(\text{mod}(n, y)));$
- $\text{div}(0, y) \Rightarrow 0;$
- $\text{div}(n+1, y) \Rightarrow \text{ifthenelse}(\text{eq}(\text{S}(\text{mod}(n, y)), y),$
 $\text{S}(\text{div}(n, y)),$
 $\text{div}(n, y));$



Pragmatic Perspective

- Primitive recursive functions are functions that can be defined using only **definite iteration** (e.g. the equivalent of a for-loop with upper bound pre-determined)

and **not** requiring **indefinite iteration** (while-loops) or the full power of recursion.

- Primitive recursion *as given* is **not** a special case of **tail recursion**, although there is an equivalent version that is.
- The standard version of primitive recursion is “top-down”, whereas tail-recursion is “bottom-up”.



Primitive Recursion = Definite Iteration

- The function f defined in the primitive recursion scheme can be computed by the following for-loop:

```
// To compute  $acc == f(n, x_1, x_2, \dots, x_k)$   
// where  $f$  is defined by primitive recursion  
// from  $b$  and  $r$ 
```

```
acc :=  $b(x_1, x_2, \dots, x_k)$ ;
```

```
for(  $j := 0; j < n; j++$  )  
  {  
    acc :=  $r(x_1, x_2, \dots, x_k, j, acc)$ ;  
  }
```



Proof by Invariant

- The function f defined in the primitive recursion scheme can be computed by the following for-loop:

```
// To compute  $acc == f(n, x_1, x_2, \dots, x_k)$   
// where  $f$  is defined by primitive recursion  
// from  $b$  and  $r$ 
```

```
acc :=  $b(x_1, x_2, \dots, x_k)$ ;
```

```
for(  $j := 0; j < n; j++$  )  
    invariant:  $acc = f(j, x_1, x_2, \dots, x_k)$   
    {  
         $acc := r(x_1, x_2, \dots, x_k, j, acc)$ ;  
    }
```



Tail-Recursion Theorem

- The function $f(n, x_1, x_2, \dots, x_k)$ defined by primitive recursion can be computed as $t(n, b(x_1, x_2, \dots, x_k))$ where t is defined in the following tail-recursion:

$$t(0, \text{acc}) \Rightarrow \text{acc};$$

$$t(n+1, \text{acc}) \Rightarrow r(x_1, x_2, \dots, x_k, n, \text{acc});$$

- Proof: This version can be “read off” from the previous loop version. The connection to the original primitive recursion was established by the loop invariant.



Example: Factorial

- Primitive-recursive version
(uses the primitive-recursion pattern):

$$\begin{aligned} \text{fac}(0) &=> 1; \\ \text{fac}(n+1) &=> \text{mult}(n+1, \text{fac}(n)); \end{aligned}$$

- Tail-recursive version
(doesn't use the pattern, but equivalent):

$$\begin{aligned} \text{fac_tr}(n) &= \text{t}(n, 1); \\ \text{t}(0, \text{acc}) &=> \text{acc}; \\ \text{t}(n+1, \text{acc}) &=> \text{t}(n, \text{mult}(n+1, \text{acc})); \end{aligned}$$



Totality Theorem

- Every primitive recursive function is a total function.
- Two levels of induction are involved:
 - For each individual use of the primitive-recursion pattern, there is an induction to show that f is defined for all n , assuming that b and r are total.
 - Structural induction is used to ascertain that anything defined from the derivation rules is a function.



Computability Theorem

- Every primitive-recursive function is computable by a Turing machine.
- This follows from the Church/Turing thesis.
- It can also be shown in significant detail by showing how a Turing machine can be constructed by composing functions using the basis functions and induction rules.



Primitive Recursion Diagonalization Theorem

- There is a computable function that is not primitive recursive.
- Proof: A Turing machine can effectively enumerate the primitive recursive functions of one argument, by applying the rules in some orderly fashion:

$$p_0, p_1, p_2, \dots$$

Then define $q(x) = p_x(x) + 1$. This function is clearly total, since each p_x is, but q cannot be p_k for any k .



The Ackermann Hierarchy

- We notice that add and mult have similar definitions.
 - add uses S as a base
 - mult uses add as a base
- We can go on to define exp analogously:
 - exp uses mult as a base
- When does this stop?
- Never, but we quickly reach functions that have very large values for small arguments.
- Ackermann observed that is possible to diagonalize over this hierarchy.



The Ackermann Hierarchy

- $A_0(m) = S(m)$
- $A_{n+1}(0) = A_n(1)$
- $A_{n+1}(m+1) = A_n(A_{n+1}(m))$
- In effect, $A_{n+1}(m+1) = A_n^m(1)$, the m -fold application of A_n .
- Each function in the list: A_0, A_1, A_2, \dots is clearly primitive-recursive.
- Define $A(n, m) = A_n(m)$ (called Ackermann's Function)
- It can be proved that for any primitive recursive function p of one variable, there is an n such that

$$\forall m \in \mathbb{N} \quad p(m) < A(n, m)$$

- Then the function $q(m) = A(m, m)$ cannot be primitive recursive.



Partial-Recursive Functions

- These extend the primitive recursive functions by using the “ μ operator”.
- They are sometimes therefore called the μ **Recursive Functions**.



Partial-Recursive Functions

- Start with the primitive-recursive functions as a base.
- Add one more induction rule: If h is a $k+1$ ary partial-recursive function (prf), then f is a k -ary prf:

$$f(x_1, x_2, \dots, x_k) = \mu x_0 [h(x_0, x_1, \dots, x_k) = 0]$$

- “the least value of x_k such that $h(x_0, x_1, \dots, x_k) = 0$ ”.
- **Note:** It is understood that if $h(x_0, x_1, \dots, x_k)$ is undefined for any $y < \text{the least } x_k$, then the value of $f(x_1, x_2, \dots, x_k)$ is also undefined.
 - μ is called the “minimalization operator”.



Example of Using the μ Operator

- Suppose we want to compute the integer square root of a number. We could define

$$\text{sqrt}(n) = \mu k [\text{sub}(n, \text{mult}(k, k)) = 0]$$

- It turns out that this particular use of μ is **not essential**; sqrt can be computed by primitive-recursive means. Still, it is convenient.



Example of Non-Total Functions Using μ Operator

- Consider

$$\text{diverge}(n) = \mu k [\text{sub}(k+1, k) = 0]$$

$\text{diverge}(n)$ is undefined for all n .

- Consider

$$\text{strange}(m, n) = \mu k [\text{not}(\text{eq}(k+m, n)) = 0]$$



Note on μ Operator and Ackermann

- It is not obvious why the μ operator would give us a way to compute Ackermann's function.
- The "double-recursion" equations given for Ackermann's function actually fit within a different formalism, Herbrand-Gödel-Kleene **general recursive functions** (GRF) rather than the partial recursive functions. [The formalism is similar to a set of rex definitions over functions on the natural numbers.]
- The two formalisms are equivalent, but this is often proved in a way that does not make a clear connection that bridges the gap between primitive and partial recursive functions in a manner applicable to Ackermann's function.



Computability Theorem for Partial-Recursive Functions

- Again we can appeal to the Church-Turing thesis to convince ourselves that the partial-recursive functions are computable partial functions.
- An explicit construction can also be given. Please think about how this could be done.
- It is clear that partial-recursive functions are not always total.



Converse of the Computability Theorem

- Every Turing computable partial function is computable by a partial-recursive function.
- Moreover, the μ operator needs to be used only **once** to achieve any partial-recursive function.



Importance of the Computability Theorem and its Converse

- Turing-computable partial functions and partial-recursive functions are established as being **virtually the same thing**.
- One was defined using **strings**, the other using **numbers**.



Strings vs. Numbers

- We recognize that natural numbers and strings are equivalent.
- Strings can be enumerated in a straightforward way, for example the strings over a 2-letter alphabet $\{a, b\}$:
 - 0 \leftrightarrow Λ
 - 1 \leftrightarrow a
 - 2 \leftrightarrow b
 - 3 \leftrightarrow aa
 - 4 \leftrightarrow ab
 - 5 \leftrightarrow ba
 - ...
- So a *set of numbers* is equivalent to a language (set of strings).



Establishing the Converse

- The converse shows that any Turing-computable partial function is a partial-recursive function.
- To do this involves **encoding** TM tapes and configurations as numbers.
- Then it can be shown that there are primitive recursive functions that:
 - Simulate a single step of a Turing machine.
 - Tell whether an encoded configuration is halting.



Primitive Recursive Functions for TMs

- $R(x)$ is the encoding of the configuration resulting after 1 step from encoded configuration x .
- $T(i, x)$ is the encoding of the configuration resulting from encoded configuration x after i steps.
- $P(x)$ indicates whether or not an encoded configuration is halting (0 or 1).



Recursive TM equivalents, using μ

- Halting in i steps is expressed by:

$$\mu_i [P(T(i, x_0)) = 0]$$

- The halting configuration, if any, resulting from x_0 is:

$$T(\mu_i [P(T(i, x_0)) = 0], x_0)$$



Encodings

- Using **primitive** recursive functions to encode and decode tapes and configurations requires a lengthy, but interesting, excursion.
- One way (but not the only way) to encode arbitrary sequences of numbers is to use “Gödel numbering”:

Any sequence of natural numbers
 (x_1, x_2, \dots, x_k)

can be encoded as a **single** natural number:

$$p_1^{1+x_1} p_2^{1+x_2} \dots p_k^{1+x_k}$$



Universal Partial-Recursive Functions

- Most results for Turing machines have parallels for the partial-recursive functions.
- The partial-recursive functions are programs that can be coded and **effectively enumerated** just like Turing machines can:

$$\varphi^k_0, \varphi^k_1, \varphi^k_2, \varphi^k_3, \dots$$

are the k-ary partial-recursive functions for any fixed k.

- “Effective” here means that there is an algorithm that, given i , can construct φ^k_i .



Kleene's Normal Form Theorem

- For each $k \geq 1$, there exists a 1-ary primitive recursive function U and a $(k+2)$ -ary primitive recursive predicate T_k such that
 - $\varphi_n^k(x_1, x_2, \dots, x_k)$ converges iff $(\exists z) T(n, x_1, x_2, \dots, x_k, z)$
$$\varphi_n^k(x_1, x_2, \dots, x_k) = U(\mu z [T(n, x_1, x_2, \dots, x_k, z) = 0])$$
 - Essentially, T is like the function that tells whether the n^{th} configuration of a TM computation is halting, while U gives the result from that halting configuration.
 - The numbers z code both the program for the partial recursive function in question **and** the number of steps.



Universal Partial-Recursive Functions

- For each k , there is a partial-recursive function ψ of $k+1$ variables such that

$$\psi(n, x_1, x_2, \dots, x_k) = \varphi_n^k(x_1, x_2, \dots, x_k)$$

- ψ is a universal function for k arguments.



Important: Terminology

- Henceforth, Turing-computable and “recursive” are used interchangeably:
 - Partial-recursive function = partial function computable by a Turing machine
 - Recursive function = total function computable by a Turing machine.
- These are not to be confused with “recursive” as used in programming language parlance.



Languages of Indices

- Set of indices of Turing machines (equivalently partial-recursive functions) provide a good testing ground for understanding the distinctions between recursive and recursive-enumerable languages.
- Suppose that $\varphi^k_0, \varphi^k_1, \varphi^k_2, \varphi^k_3, \dots$ is an effective enumeration of all (k-ary) partial recursive functions.



Divergence Notation

- $\varphi(x)\downarrow$ is used to mean that φ is **defined** for argument x .
- $\varphi(x)\uparrow$ is used to mean that φ **diverges** on argument x .



Divergence Problem Re-Cast

- The set $D = \{j \in \mathbb{N} \mid \varphi_j(j) \uparrow\}$ is **not** recursively-enumerable; this is the **divergence problem**.
- Suppose that D were r.e. Then by the **alternate characterization**, there is a k such that φ_k has D as its domain.
- By definition of D , $k \in D$ iff $\varphi_k(k) \uparrow$.
- But since D is the domain of φ_k , $k \notin D$ iff $\varphi_k(k) \uparrow$, by definition of “domain”.



Halting vs. Divergence

- The set $H = \{j \mid \varphi_j(j) \downarrow\}$ **is** recursively-enumerable (why?).
- But H is not recursive; this is the **halting problem**.
- If H were recursive, then so would its complement be.
- But its complement is D on the previous slide, which is not even recursively-enumerable.



The Set of Indices of **Total** Recursive Functions is not Recursively-Enumerable

- Let $A = \{j \mid \forall x \ \varphi_j(x) \downarrow\}$.
- Suppose that A is r.e.
- Let T be a total recursive function that enumerates A , i.e. $A = \{T(0), T(1), \dots\}$.
- Then the function T' defined by:
$$\forall j \ T'(j) = \varphi_{T(j)}(j) + 1$$
is also **total** and obviously computable (**recursive**).
- Thus T' has an index $k \in A$:
$$T' = \varphi_k$$
- But then $T'(k) = \varphi_k(k) = \varphi_{T(k)}(k) + 1 = T'(k) + 1$, which is contradictory.