Primitive and Partial Recursive Functions

Robert M. Keller
Harvey Mudd College
10 November 2003
What is this?

• An alternate approach to defining the computable functions, one based on composition of numeric functions.

• Sometimes having this alternate viewpoint will be helpful.

• The family of primitive recursive functions is first defined, then partial recursive functions are built on that.
Primitive Recursive Functions

- The set of primitive recursive functions is given inductively.

- Every function is $k$-ary, for some $k \geq 0$.

- The domain and co-domain of each function is the set of natural numbers $\{0, 1, 2, 3, \ldots\}$ or $k$-tuples thereof.
Basis Functions (1 of 3)

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

\[ Z^k(x_1, x_2, \ldots, x_k) = 0 \]

for each arity \( k \geq 0 \).
Basis Functions (2 of 3)

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

\[ p_k^j(x_1, x_2, \ldots, x_k) = x_j \]

for each arity \( k > 0 \) 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:

\[
    h(x_1, x_2, \ldots, x_k) = \\
    f(g_1(x_1, x_2, \ldots, x_k), \\
    g_2(x_1, x_2, \ldots, x_k), \\
    \ldots \\
    g_r(x_1, x_2, \ldots, x_k))
\]

for each pair of arities \(k, r \geq 0\).
Constant Functions

- A consequence of the rules up to this point is that \textbf{constant} functions are all primitive recursive:

\[ C^k_c(x_1, x_2, \ldots, x_k) = c \]

for each natural number \( c \).

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

\[ C^k_c(x_1, x_2, \ldots, x_k) = S(S(\ldots S(\text{zero}(x_1, x_2, \ldots, x_k)) \ldots)) \]
Explicit Definition (ED)

- This is a convenience in lieu of showing stacks of 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 \( f \).
Induction Rules (2 of 2)

- A function defined from primitive recursive functions
  by the following **primitive recursion pattern** is
  primitive recursive:

  \[
  h(0, x_1, x_2, \ldots, x_k) = g(x_1, x_2, \ldots, x_k)
  \]

  \[
  h(n+1, x_1, x_2, \ldots, x_k) =
  f(x_1, x_2, \ldots, x_k, n, h(n, x_1, x_2, \ldots, x_k))
  \]

- Here \( h \) is being defined from \( g \) and \( h \), which are
  known to be primitive recursive.
Examples of Primitive Recursive Functions

- add(x, y): addition
- mult(x, y): multiplication
- pred(x): predecessor
- sub(x, y): proper subtraction
- mod(x, y): modulus
- div(x, y): integer division (quotient)
- sqrt(x): integer square root
rex implementations

- I will demonstrate some of these using rex rules. This allows the definitions to be tested readily.

- rex does not enforce the primitive recursive formalism, so we have to be careful not to “cheat”.
add implementation

- $S(n) = n + 1; \quad // \text{pretend this is built in}$

- $\text{add}(0, y) \Rightarrow y;$

- $\text{add}(n+1, y) \Rightarrow S(\text{add}(n, y));$

- For reference
  \[
  h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)
  \]
  \[
  h(n+1, x_1, x_2, \ldots x_k) =
  f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))
  \]
mult implementation

• \( \text{mult}(0, y) => 0; \)

• \( \text{mult}(n+1, y) => \text{add}(y, \text{mult}(n, y)); \)

• For reference
  \[
  h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)
  \]

  \[
  h(n+1, x_1, x_2, \ldots x_k) = 
  f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))
  \]
pred implementation

- pred(0, y) =>

- pred(n+1, y) =>

- For reference
  \[ h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k) \]
  \[ h(n+1, x_1, x_2, \ldots x_k) = \]
  \[ f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k )) \]
sub implementation

- sub is **proper** subtraction (aka “monus”):
  If \( a < b \), then \( \text{sub}(a, b) = 0 \).

- \( \text{sub}(y, 0) \Rightarrow \)

- \( \text{sub}(y, n+1) \Rightarrow \)

- For reference
  \[
  h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)
  \]
  \[
  h(n+1, x_1, x_2, \ldots x_k) =
  f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))
  \]
Primitive Recursive Predicates

- For some definitions we want to have predicates, which we can equate to functions return only values 0 (false) and 1 true.

  - \( \text{sgn}(0) \Rightarrow 0; \)

  - \( \text{sgn}(n+1) \Rightarrow 1; \)

  - \( \text{sgn} \) converts arbitrary values to \( \{0, 1\}. \)
Negation

- not(0) => 1;
- not(n+1) => 0;
Equality Predicate

- \( \text{eq}(x, y) = \neg(\text{add}(\text{sub}(x, y), \text{sub}(y, x))) \);
if-then-else function

- ifthenelse(0, x, y) => y;
- ifthenelse(n+1, x, y) => x;

- This can be inefficient if all arguments must be computed to get a result (applicative order). It should be taken as an “academic” version, perhaps.
mod and div

- \(\text{mod}(0, y) \Rightarrow 0;\)

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

- \(\text{div}(0, y) \Rightarrow 0;\)

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

- Primitive recursive functions are functions that can be defined using only **definite iteration** (e.g. 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.
Primitive Recursion = Definite Iteration

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

```plaintext
acc = g(x_1, x_2, \ldots x_k);
for( j = 0; j < n; j++ )
    acc = f(x_1, x_2, \ldots x_k, j, acc);

// acc == h(n, x_1, x_2, \ldots x_k)
```
Tail-Recursive Version

• The function $h(n, x_1, x_2, \ldots, x_k)$ can be computed as $t(n, g(x_1, x_2, \ldots, x_k))$ where $t$ is defined in the following tail-recursion:

\[
\begin{align*}
t(0, \text{acc}) &= \text{acc}; \\
t(n+1, \text{acc}) &= f(x_1, x_2, \ldots, x_k, n, \text{acc});
\end{align*}
\]
Example: Factorial

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

  \[
  \begin{align*}
  \text{fac}(0) & = 1 \\
  \text{fac}(n+1) & = \text{mult}(n+1, \text{fac}(n))
  \end{align*}
  \]

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

  \[
  \begin{align*}
  \text{fac}_\text{tr}(n) & = t(n, 1) \\
  t(0, \text{acc}) & = \text{acc} \\
  t(n+1, \text{acc}) & = t(n, \text{mult}(n+1, \text{acc}))
  \end{align*}
  \]
Tail-Recursion Theorem

- If h is defined from f and g using primitive recursion, then h can also be defined from f and g using tail recursion.
- The conversion is given on preceding slides.
- Claim: \( t(n, \text{acc}) = h(n, x_1, x_2, \ldots, x_k) \) provided acc is initialized with \( g(x_1, x_2, \ldots, x_k) \).
- Proof is by induction on n.
  - Show that both
    - \( t(n, \text{acc}) = f(X, n-1, f(X, n-2, \ldots, f(X, 0, g(X))\ldots)) \)
    - \( h(n, X) = f(X, n-1, f(X, n-2, \ldots, f(X, 0, g(X))\ldots)) \)
Totality Theorem

- Every primitive recursive function is total.
- Two levels of induction are involved:
  - For each use of the primitive-recursion pattern, there is an induction to show that $h$ is defined for all $n$, assuming that $f$ and $g$ are total.
  - There is induction on the number of uses of the induction rules in defining the top level function.
Computability Theorem

• Every primitive-recursive function is computable by a Turing machine.

• This is more-or-less obvious, but it can be shown in significant detail by showing how a Turing machine can be constructed by composing functions using the basis functions and induction rules.

• Please consult the text for details.
Diagonalization Theorem

• There is a total recursive function that is not primitive recursive.
• Proof: Using the computability theorem, we know that we can enumerate the primitive recursive functions of one argument:
  \[ p_0, p_1, p_2, \ldots \]
  They are just a subsequence of the ones computed by Turing machines in the ordering of all Turing machines.
  Then define \( q(x) = p_x(x) + 1 \). This function is clearly total, since each \( p_x \) is, but \( q \) cannot appear in the list.
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, using mult as a base.
• When does this stop?
• We quickly reach functions that have very large values for small arguments.
• It is possible to diagonalize over this hierarchy.
The Ackermann Hierarchy

- $A_0(m) = S(m)$
- $A_{n+1}(0) = A_n(1)$
- $A_{n+1}(m+1) = A_n(A_{n+1}(m))$

- Each function in the list: $A_0, A_1, A_2, \ldots$ is clearly primitive-recursive.
- Define $A(n, m) = A_n(m)$.
- It can be proved that $A$ grows faster than any single primitive-recursive function, hence is not primitive-recursive itself.
Recursive Functions

- This is a family of partial functions, also known as “the partial recursive functions”.

- We have used that term to describe the partial computable functions, and the definitions turn out to be equivalent.
\( \square \)-Recursive Functions

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

\[
h(x_1, x_2, \ldots x_{k-1}) =
\]

\[
\square x_k [f(x_1, x_2, \ldots x_k) = 0]
\]

read “the least value of \( x_k \) such that \( f(x_1, x_2, \ldots x_k) = 0 \).
Recursive Functions

- The definition of a function using the operator really only makes sense if the function $f$ is **total**.

- Totality cannot be established syntactically (why?).

- We will adopt the convention that the values of for which $f$ is computed are given sequentially, and that if $f(x_1, x_2, \ldots x_k)$ is divergent for any value before reaching a value $x_k$ such that $f(x_1, x_2, \ldots x_k) = 0$, then $h(x_1, x_2, \ldots, x_{k-1})$ also diverges for the given arguments.
Computability Theorem for m-Recursive Functions

- We can extend to the partial recursive functions the proof that the primitive recursive functions are Turing computable.
- Consult the text for details.
Converse of the Computability Theorem

• Every Turing computable partial function is computable by a $\mu$-recursive partial function.

• Moreover, the $\mu$ operator needs to be used only once to achieve any partial-recursive function.
Importance of the Computability Therem and its Converse

• Turing-computable partial functions and \( \mu \)-recursive partial functions are established as being the same thing.

• One was defined using strings, the other using numbers.
Establishing the Converse

• The converse shows that any Turing-computable partial function is a $\mu$-recursive partial 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 TM equivalents

- \( R(x) \) is the configuration resulting after 1 step from \( x \).

- \( T(i, x) \) is the configuration resulting from configuration \( x \) after \( i \) steps.

- \( P(x) \) indicates whether or not a configuration is halting (0 or 1).
Recursive TM equivalents

- Halting in \( i \) steps is expressed by:

\[
\forall i \ [\text{P}(T(i, x_0)) = 0]
\]

- The halting configuration is:

\[
T(\forall i \ [\text{P}(T(i, x_0)) = 0], x_0)
\]
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 
  \((x_1, x_2, \ldots, x_k)\)
  can be encoded as a **single** natural number:
  \[p_1^{x_1}p_2^{x_2}\ldots p_k^{x_k}\]
Universal $\mu$-Recursive Functions

- Most results for Turing machines have parallels for the $\mu$-Recursive Functions.
- The $\mu$-recursive functions are programs that can be coded and enumerated just like Turing machines can:
  
  \[ \mu^k_0, \mu^k_1, \mu^k_2, \mu^k_3, \ldots \]

  are the k-ary $\mu$-recursive functions for any fixed $k$. 
Kleene’s Normal Form Theorem

- For each $k > 1$, there exists a 1-ary primitive recursive function $U$ and a $(k+2)$-ary primitive recursive predicate $T_k$ such that
  
  - $\Box^k_n(x_1, x_2, \ldots, x_k) \iff (\exists z) T(n, x_1, x_2, \ldots, x_k, z)$
  
  - $\Box^k_n(x_1, x_2, \ldots, x_k) = U(\exists z [T(n, x_1, x_2, \ldots, x_k, z) = 0])$

- 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 and the number of steps.
Universal $\square$-Recursive Functions

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

$$\square(n, x_1, x_2, \ldots, x_k) = \square^k_n(x_1, x_2, \ldots, x_k)$$

- $\square$ is a universal function for $k$ arguments.
Recursive and R.E. Sets

- Languages are now sets of numbers.
- A set is recursive if its characteristic function is total recursive.
- A set is recursively-enumerable if there is a total recursive function that enumerates it.
- Equivalently, a set is recursively-enumerable if it is the domain of some partial recursive function.
Halting and Divergence

- $j(x)\in$ is used to mean that $j$ has a value for argument $x$.
- $j(x)\uparrow$ is used to mean that $j$ diverges on argument $x$.
- The set $\{j \mid j(j)\uparrow\}$ is not recursively-enumerable: this is the **halting problem**.
- The set $\{j \mid j(j)\in\}$ is recursively-enumerable, but not recursive.
- The set $\{j \mid j(0)\uparrow\}$ is not R.E. either, analogous to the blank-tape halting problem.
Problem: Exhibit a total function that is not computable.

- Define

\[ f(j) = \begin{cases} 
\neg j(j) + 1 & \text{if } \neg j(j) \\
0 & \text{otherwise} 
\end{cases} \]

\( f \) is evidently total. \( f \) is not computable, since if it were, say \( \neg k \), then \( f(k) \) would give a contradiction.
Hilbert’s Tenth Problem

- This is an unsolvable problem of practical interest.
  - Give an algorithm that will determine whether a multi-variate polynomial equation has an integer solution.
    - Example of an equation: \( x^2 + 3y^3 + 13 = 0 \)
- The problem of whether this is possible was posed by Hilbert in 1900.
- This problem was not proved unsolvable until 1970, when it was established that every recursively-enumerable set of k-tuples is the set of solutions of some such equation.