Lecture 4: The Primitive Recursive Functions

Michael Beeson

Primitive recursive functions

f is **primitive recursive** if and only if it is primitive recursive by one of the following schemata, in which $\mathbf{x}$ denotes a list of one or more variables, x denotes a single variable, and all functions have number of arguments implicitly indicated. On this slide are four not-so-important parts of the definition:

(i) The constant functions (of any number of arguments) are primitive recursive.

(ii) The successor function $f(x) = x'$, the next integer after x, is primitive recursive.

(iii) The projection function $I_{n,i}(x_1, \ldots, x_n) = x_i$ is primitive recursive.

(iv) (Generalized composition). If g and $h_1, \ldots h_n$ are primitive recursive, and

$$f(\mathbf{x}) = g(h_1(\mathbf{x}), \ldots, h_n(\mathbf{x}))$$

then f is also primitive recursive.

The important part

(v) (Primitive recursion) If g and h are primitive recursive, and

$$f(\mathbf{x}, 0) = h(x)$$

$$f(\mathbf{x}, y') = g(\mathbf{x}, y, f(\mathbf{x}, y))$$

then f is also primitive recursive. The case when $\mathbf{x}$ does not occur is also allowed, and since we said $\mathbf{x}$ represents "one or more variables", it has to be listed separately:

$$f(0) = c$$

$$f(y') = g(y, f(y))$$

Rephrasing the definition

The class of primitive recursive functions is the least class of functions (of one or several natural-number arguments) containing the constant, successor, and projection functions, closed under generalized composition, and closed under primitive recursion. Compare this definition to the one in Kleene, p. 220, where the "least class" part is made more explicit.

History

The definition goes back at least to Skolem (1908). Scholars argue about the history, primarily because the method of definition by primitive recursion predates the actual definition of the class of primitive recursive functions.

Hilbert definitely understood the class of primitive recursive functions in the 1920s, as it occurs explicitly in his paper about "the infinite."

Examples of primitive recursive functions

One can easily show that the following functions are primitive recursive:

$$\begin{aligned} f(x,y) &= x+y \\ f(x,y) &= x \cdot y \\ f(x,y) &= x^y \\ f(x,y) &= x! \end{aligned}$$

At this point we introduce the notation $1 = 0'$ and $2 = 1' = 0''$, and so on. We can then use the primitive recursion equations to calculate that $2 + 2 = 4$. This familiar equation does have some computational content!

Predecessor, min, and max

One introduces the predecessor function by

$$\mathrm{pd}(0) = 0$$

$$\mathrm{pd}(x+1) = x$$

and "cut-off subtraction" by

$$x \dot{-} 0 = x$$

$$x \dot{-} (y+1) = \mathrm{pd}(x - y)$$

Then one has

$$\min(a,b) = b \dot{-} (b \dot{-} a)$$

$$\min(a,b,c) = \min(\min(a,b),c)$$

and so on for the minimum of more arguments;

$$\max(a,b) = (a+b) - \min(a,b)$$

$$\max(a,b,c) = \max(\max(a,b),c)$$

and so on for more arguments.

Sign and distance

$$|a - b| = (a \dot{-} b) + (b \dot{-} a).$$

The following functions can also easily be shown to be primitive recursive:
The "sign of a" is 1 if a is positive and 0 when $a = 0$. This is written $\mathrm{sg}(a)$. The function that is instead 0 when a is positive and 1 when a is negative is written $\overline{\mathrm{sg}}(a)$.

Division and mod

These are primitive recursive:

- The remainder of a on division by b, written by Kleene as $\mathrm{rm}(a, b)$, by mathematicians as $a \mod b$, and in 2014 by programmers as $a\%b$,
- cutoff division

$$\lfloor a/b \rfloor$$

Bounded sum and product

The following two equations lead from a primitive recursive g to a primitive recursive f:

$$f(\mathbf{x}, y) = \sum_{z<y} g(\mathbf{x}, z)$$

$$f(\mathbf{x}, y) = \prod_{z<y} g(\mathbf{x}, z)$$

The n-th prime number

$$f(n) = p_n, \qquad \text{the } n\text{-th prime number}$$

This requires a bound for "the next prime number". *Hint*: use the factorial function.

Bits and left-shift

The following functions are useful in connection with representing strings in terms of ascii codes:

$$\mathrm{size}(n) = \text{ the number of bits in (the binary representation of) } n$$

$$\mathrm{Bit}(i, n) = \text{ the power of } 2^i \text{ in the binary representation of } n$$

$$x << n = \ x \text{ left-shifted by } n$$

$$x >> n = \ x \text{ right-shifted by } n$$

Primitive Recursive Predicates

The *characteristic function* of a predicate $P(\mathbf{x})$ is the function whose value is 1 if $P(x)$ and 0 otherwise. The *representing function* of P is similar, but has value 0 if $P(x)$ and 1 otherwise. A predicate is primitive recursive by definition, if and only if its representing function is primitive recursive. The primitive recursive predicates are closed under the logical connectives and under bounded quantification, by which we mean

$$P(y, \mathbf{x}) \leftrightarrow \forall z < y \, Q(z, y, \mathbf{x})$$

and similarly with $\exists$ instead of $\forall$.

Divisibility

The predicates $n \mid m$ ("n divides m") is primitive recursive.

Sequence numbers

Finite sequences of integers were coded by Gödel using unique factorization. We define $\langle a, b, c \rangle$ to be $2^{a+1}3^{b+1}5^{c+1}$. For longer sequences we use more primes than the first three. A sequence can then be decoded by extracting the powers of primes in its factorization. The function $\ell h(x)$ ("length of x") is the largest n such that p_n divides x. $(x)_y$ is defined to be one less than the exponent of p_y in the factorization of x, if p_y divides x, else 0. All these functions are primitive recursive.

Fibonacci numbers

These are given by

$$F(0) = F(1) = 1$$

$$F(n+1) = F(n) + F(n-1)$$

This recursion is not a primitive recursion, yet the function $F(n)$ can be defined more elaborately in a primitive recursive way, for example by defining

$$G(n) = \langle F(n), F(n+1) \rangle$$

by primitive recursion and then defining $F(n) = (G(n))_0$.

It's the 21st century now

In these lectures, we shall sometimes use a different method of coding finite sequences of integers, one more in accordance with the times, and one that does not involve exponential increases in length due to the use of powers of primes. Namely, to code the sequence a, b, c, we write out the binary representations of a, b, and c, as ordinary strings of digits, separated by commas, and interpret that string as an integer. For example, the sequence $4, 5, 6$ is represented as follow: 4 is 100 in binary, 5 is 101, and 6 is 110. But we use 8 bytes for each digit, padding on the left. So $(4, 5, 6)$ is computed from the string `"00000100,00000101,00000110"`. The length of this string is 26 characters. Note that this string contains three different symbols, `'0'`, `'1'`, and `','`. As ascii codes, those are 48, 49, and 44 respectively. This string of 26 characters is now regarded as the binary representation of an integer. That integer is, by definition, $(4, 5, 6)$. It has $8 \cdot 26 = 208$ bits.

Decoding a sequence

A consequence of this definition is that the number (a, b, c) can be uniquely decoded; no two tuples will yield the same number under this encoding. To decode a sequence number N, we first find the string whose characters are the 8-bit pieces of N (padding N to a multiple of 8 bits by zeroes on the left if necessary). Then we identify the occurrences of 44 (ascii code of comma) in this string; that determines the elements of the sequence. These must be composed of the digits 0 and 1 only, or N does not code a sequence. Then the strings between commas can be decoded into integers. Note that if we used the bits of a, b, and c directly, instead of the ascii codes of their digits, we might get confused by "accidental" occurrences of the ascii code of comma among the bits of a, b, and c; but using ascii codes of digits prevents that confusion. This would have been more confusing than powers of primes in 1936, but today it is straightforward and offers some technical advantages. It also offers one technical disadvantage: it is harder to formalize its basic properties in Peano arithmetic. We will therefore make use of both methods of coding sequences.

Indices of primitive recursive functions

We can assign an index to each primitive recursive function as follows. Each index of a primitive recursive function f will have the form $\langle i, n, \ldots \rangle$, where i tells which of the schemata (i)-(v) defines f, and n gives the number of arguments of f. The part of the index indicated by $\ldots$ depends on i, as follows. If $i = 3$, so f is the projection function $I_{n,i}$, then the index of f is $\langle 3, n, i \rangle$. If $i = 4$, so

$$f(\mathbf{x}) = g(h_1(\mathbf{x}, \ldots, h_n(\mathbf{x}))$$

then the index of f is $\langle 4, n, a, b_1, \ldots, b_n \rangle$ where a is the index of g and b_j is the index of h_j. Finally, if $i = 5$ so f is defined by primitive recursion from h and g, then the index of f is $\langle 5, n, a, b \rangle$, where a and b are the indices of h and g.

A non-primitive-recursive function

We can use these indices together with the diagonal method to construct a computable but not primitive recursive function. (Here we take computable in the sense of computable by a computer program.) First we observe that it is computable whether e is an index of a primitive recursive function, or not, and if so, we can computably extract the parts of e and hence the definition of the primitive recursive function f of which f is an index. Say that $e = \langle i, n, \ldots \rangle$. Then given e and x we can compute

$$F(e, x) = f(x)$$

provided e is the index of a primitive recursive function of 1 argument, and $F(e, x) = 0$ otherwise. Then F is total, i.e. defined for all e and x. Now define

$$g(x) = F(x, x) + 1$$

Then g is not primitive recursive, by the diagonal method: for if g has index $e = \langle i, 1, \ldots \rangle$, then $g(e) = F(e, e) = g(e) + 1$, contradiction.

Rosza Péter

Rosza Péter (1936, although that date of publication of her book is much later than the actual work) analyzed the assignment of indices more carefully and showed that it can be done by a "double recursion"; in this way she explicitly gave a class of recursively defined functions that went beyond the primitive recursive functions. There are also "triple recursions" and "k-ary recursions", each allowing more functions to be defined, according to Péter. Then there are functions defined by transfinite recursions on certain orderings of the integers, and theorems relating these different kinds of recursions. It seemed that no matter how many kinds of recursion you defined, there was always a more complicated kind that allowed more functions to be defined. After all, the diagonal method guaranteed it!

Bounded arithmetic formulas and primitive recursion

Theorem

Every bounded arithmetic formula (formula of PA with only bounded quantifiers) defines a primitive recursive predicate.

Remark. The converse, due to Gödel, is also true, but is more difficult. We will prove it later (not today).

Proof. We already proved above that the primitive recursive predicates are closed under the propositional connectives and bounded quantification. It only remains to check the base case, when the formula is atomic. The only predicate in PA is equality, so the atomic formulae have the form $t = s$ for terms t and s. These terms are built up from successor, $+$, and $\cdot$. Hence they are equivalent (equi-satisfied) to polynomial equations $p(x_1, \ldots, x_n) = q(x_1, \ldots, x_n)$, where p and q are polynomials with coefficients in $\mathbb{N}$. The representing function of such a relation is given by $f(\mathbf{x}) = \mathrm{sg}(p(\mathbf{x}) \dot{-} q(\mathbf{x}))$. Since every polynomial is primitive recursive, and cutoff subtraction is primitive recursive, f is also primitive recursive. That completes the proof.

Course of values recursion

See § 46 of Kleene. A corollary of the theorem about course-of-values recursion is the following lemma, which helps us considerably to understand the nature of recursion. If we have a recursion in which the values of the argument at the recursive call decrease, then it turns out to be a primitive recursion. (And of course, the decrease in the arguments guarantees termination, so the function is total.) You will see in the exercises that there are examples of functions defined by more complicated functions, in which the arguments do not necessarily decrease at recursive calls, which turn out not to be primitive recursive even though they do always terminate.

Lemma

Suppose $f(n, \mathbf{x})$ *is defined recursively as a function of values of* $f(m, \mathbf{x})$ *for* $m < n$. *Then* f *is primitive recursive.*

For example, the Fibonacci function is primitive recursive for this reason.