CS 311: Computational Structures

James Hook and Tim Sheard

November 18, 2013

7 Primitive and Partial Recursive Functions

The primitive and partial recursive functions approach computability from a different point of view than the Turing machine. Much like the regular expressions, we will develop an inductive definition of functions. In this case, it will be an inductive definition of functions from natural numbers to natural numbers.

We will first study the primitive recursive functions. All functions in this class will be obviously *computable*, that is, you could write a program in your favorite programming language that reads the description of a function and computes its value on any input. They will also be *total*, that is, they will be defined on all inputs.

The second class we will study are the partial recursive functions. Like the primitive recursive functions, they will all be computable. However they will not all be total. It will be possible to write functions that cannot be computed on some inputs.

When defining both classes of functions we will pay careful attention to how many arguments each function takes. We will call a function that takes n arguments as an n-place function or a function of *arity* n.

7.1 Terminating Recursion

In homework and exams we have seen the following definition of addition:

$$\begin{aligned} Z + y &= y \\ S(x) + y &= S(x + y) \end{aligned}$$

This is a good recursive definition. It defines + as a total function. For any pair of natural numbers x and y, the computation $x + y$ will terminate. Specifically, it will require x applications of the second rule and 1 application of the first rule.

An example of a bad definition of a function is:

$$f(x) = f(x)$$

This equation does not define the function f. If we treat this as a rewrite rule, where the left hand side is rewritten to the right hand side, we can apply this an

infinite number of times without making any progress toward an answer. While this equation is a bad definition, it is a reasonable property. All functions satisfy this property.

The rule for primitive recursive functions captures a particular style of good definitions on the natural numbers. This style says that any function f defined by two equations:

$$\begin{aligned} f(Z, x_1, \ldots, x_k) &= e_Z \\ f(S(n), x_1, \ldots, x_k) &= e_S \end{aligned}$$

is a good definition provided:

1. The expression e_Z only mentions variables $x_1, \ldots, x_k$ and makes no reference to f.

2. The expression e_S only mentions variables $n, x_1, \ldots, x_k$, and there is at most one recursive call to f which is exactly of the form $f(n, x_1, \ldots, x_k)$.

The primitive recursion rule enforces this style of definition by requiring that the expressions e_Z and e_S be characterized by two functions, h and g. The user no longer has the freedom to write down the recursive calls. Instead they just provide these functions, and they are automatically plugged into the following definition schema:

$$\begin{aligned} f(Z, x_1, \ldots, x_k) &= h(x_1, \ldots, x_k) \\ f(S(n), x_1, \ldots, x_k) &= g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{aligned}$$

Returning to the definition of $+$, the first equation needs to satisfy $h_+(x_1) = x_1$. So h_+ must be the identity function. The second equation needs to satisfy

$$g_+(n, n + x_1, x_1) = S(n + x_1)$$

This yields the function:

$$g_+(a, b, c) = S(b)$$

In this manner we will say that $+$ is defined by primitive recursion using these two functions h_+ and g_+. Note that $+$ is a 2-place function, h_+ is a 1-place function, and g_+ is a 3-place function. This pattern will generalize. To define a k-place function f we will need a $k-1$ place function h and a $k+1$-place function g.

7.2 Primitive Recursive Functions

There are five rules for defining the primitive recursive functions. The first three define a set of basic primitive functions. The last two rules explain how to build functions by combining other functions.

I. Zero For every arity k there is a constant function Z satisfying $Z(x_1, \ldots, x_k) = 0$.

II. Successor There is a 1-place function, S, that computes the successor. That is, S satisfies $S(x) = x + 1$.

III. Projection For every arity k there are k projection functions, $P_1, \ldots, P_k$ satisfying: $P_i(x_1, \ldots, x_i, \ldots x_k) = x_i$.

IV. Composition (or Substitution) An arity k function, f, can be combined with k functions of arity l, $g_1, \ldots, g_k$ to produce a new arity l function, $C\ f\ [g_1, \ldots, g_k]$, satisfying:

$$C\ f\ [g_1, \ldots, g_k](x_1, \ldots, x_l) = f(g_1(x_1, \ldots, x_l), \ldots, g_k(x_1, \ldots, x_l))$$

V. Primitive Recursion Given a k place function h and a $k+2$ place function g, the $k+1$ place function f defined by primitive recursion, written PR h g, satisfies:

$$\begin{aligned} f(0, x_1, \ldots, x_k) &= h(x_1, \ldots, x_k) \\ f(n+1, x_1, \ldots, x_k) &= g(n, f(n, x_1, \ldots, x_k), x_1, \ldots, x_k) \end{aligned}$$

7.3 Discussion and Exercises

Note that the official definitions of the functions do not have variable names. They are almost like an assembly language for functions.

Exercise 7.1 *Compute the following:*

1. $P_2(1, 2, 3)$
2. $Z(21, 17, 42)$
3. $S(13)$
4. $(C\ S\ [Z])(1, 2, 3)$
5. $(\text{PR}\ Z\ Z)(3)$
6. $(\text{PR}\ Z\ (C\ S\ (P_1)))(3)$

Exercise 7.2 *Use rules I, II, III, and IV to do the following:*

1. *Use composition, successor, and zero to define the constant function 2.*
2. *What is the identity function?*
3. *Assuming f is a 2-place function build a new 1-place function g that applies f to two copies of its argument. That is $g(x) = f(x, x)$.*
4. *Assuming f is a 2-place function, build a new 5-place function g that applies f to its second and fourth arguments. That is $g(x_1, x_2, x_3, x_4, x_5) = f(x_2, x_4)$.*

5. *Assuming f is a 2-place function build a new 2-place function g that applies f with its arguments reversed. That is $g(x, y) = f(y, x)$.*

6. *Construct the 3-place function g_+ used in the definition of $+$ in the previous section.*

Exercise 7.3 *Complete the definition of $+$ by applying rule V to the appropriate results from the previous exercise.*

Exercise 7.4 *Multiplication can be defined as follows:*

$$\begin{aligned} Z \times y &= Z \\ S(n) \times y &= y + (n \times y) \end{aligned}$$

Construct the Primitive Recursive function that implements this definition of multiplication. You may reuse your definition of addition.

7.4 An Implementation of Primitive Recursion

The primitive recursive functions can be defined in the programming language Haskell as a recursive datatype, `PR`. A simple evaluator is given by the Haskell function `eval`. What follows is a complete Haskell program:

```
data PR =  Z
         | S
         | P Int
         | C PR [PR]
         | PR PR PR

eval :: PR -> [Integer] -> Integer
eval Z _ = 0
eval S [x] = x+1
eval (P n) xs = nth n xs
eval (C f gs) xs = eval f (map (\g -> eval g xs) gs)
eval (PR g h) (0:xs) = eval g xs
eval (PR g h) (x:xs) = eval h ((x-1) : eval (PR g h) ((x-1):xs) : xs)

nth _ [] = error "nth nil"
nth 0 _ = error "nth index"
nth 1 (x:_) = x
nth (n) (_:xs) = nth (n-1) xs
```

This is a simplified version of a file that is posted on the web site as Naturalpr.hs.

The type of primitive recursive functions here is called `PR`. The constructors are `Z`, `S`, `P`, `C`, and `PR`. The constructor `C` takes two arguments, one of type `PR` and the other is a list (indicated by the `[ ]`) or `PR`.

The function `eval` is defined inductively on the type `PR`. It takes a value of type `PR` and a list of Integer's representing the tuple of arguments, and returns an Integer.

The addition function can be defined by the Haskell value:

```
plus = PR (P 1) (C S [P 2])
```

In this case, we can evaluate the sum of 2 and 3 as follows:

```
*Main> eval plus [2,3]
5
```

7.5 Expressive Power

What functions can you write by primitive recursion?

In practice you can write most of the Integer functions that people use in routine mathematics. However, you do not get all integer functions. You do not even get all total computable integer functions.

The class of functions definable with primitive recursion is sufficiently expressive that you can encode Turing machines as numbers, using what is called a *Gödel numbering*. This is essentially a representation of a programming language data type for the formal definition of a Turing machine that allows you to recover the parts of the definition. In particular, you can recover the set of states, the start state, the accept state, the reject state, and the transition function. You can also encode configurations and computation histories as numbers, and you can define a function that can test if they are correct, halting computations. In particular, the function T such that $T(e, x, y)$ is true if y is a halting computation of machine e on input x is a primitive recursive function.

7.6 Partial Recursive functions

To increase the expressive power to include all computable functions it is necessary to overshoot the set of total functions, and enter the domain of partial functions. That is, we will allow functions that are undefined on some values.

In this paradigm the basic way to introduce partiality is to add a search capability that is defined by a potentially non-terminating recursion. Consider the following program:

```
try_from :: (Integer -> Bool) -> Integer -> Integer
try_from test n =
  if test n
     then n
     else try_from test (n + 1)

try :: (Integer -> Bool) -> Integer
try test = try_from test 0
```

This program takes a predicate called `test` and applies it to every natural number starting from 0 and counting up until it finds one that has the expected property.

For example, `try even` returns 0 and `try odd` returns 1.

The unbounded search rule, defined as a sixth rule for defining functions, is given in the next section. It essentially encodes the search strategy of the `try` function in the context of these integer functions.

VI. Unbounded Search Given a $k+1$-place function g, the k-place function μg satisfies exactly $(\mu g)(x_1, \ldots x_k) = n$ if and only if $g(n, x_1, \ldots, x_n) = 0$ and for all $y < n$, $g(y, x_1, \ldots, x_n)$ is defined and $g(y, x_1, \ldots, x_n) \neq 0$.

Problem 7.5 *Modify* `eval` *to accommodate this additional definition scheme. To conform with conventions used later in these notes, I recommend naming the type of partial recursive functions* `MuR` *and naming the new data constructor* `Mu`*. The name* `MuR` *can be pronounced as the "mu-recursive" (or μ-recursive) functions. It is unfortunate that primitive and partial have the same abbreviation.*

With unbounded search it is now possible to define all computable functions. In particular, if you use μ to search through possible computation histories in the T function discussed above you can search the space of all possible computations of a Turing machine e on input x. From the resulting computation history y, which will be returned by μ if such a history exists, you can decode the value computed by the function. In this way it is possible to express any function computed by a Turing machine as a partial recursive function.

For more information about these results you can read about the Kleene T predicate and the Kleene Normal Form theorem. These topics are typically covered in texts on recursion theory. A web search yielded several reasonable articles.