The Theory of Computation

Bernard M.E. Moret

Addison-Wesley-Longman, 1998

Solution Manual
Chapter 5, Problem 5.6

How to obtain a unique code for a primitive recursive function is clear; the only trick is to add to the code the number of arguments of the function (for reasons that will become clear on verifying the code); the steps thus are

- Zero( ) is coded $\langle 1, 0\rangle_2$.
- Succ( ) is coded $\langle 1, 1\rangle_2$.
- $P_i^j(\,)$ is coded $\langle j, 2, i\rangle_3$.
- If f is a function of n arguments defined by substitution from the functions g (of m arguments) and $\{h_i \mid 1 \leq i \leq m\}$, where g has code c_g and h_i has code c_i for each i, then f is coded by $\langle n, 3, m, c_g, c_1, c_2, \ldots, c_m\rangle_{m+3}$.
- If f is a function of n arguments defined by primitive recursion from the functions g (of $n-1$ arguments) and h (of $n+1$ arguments), where g has code c_g and h has code c_h, then f is coded by $\langle n, 4, c_g, c_h\rangle_3$. (This ignores the small problem when n is 1, which we can easily handle by a separate case.)

Now the enumeration proceeds simply as follows

```
i := 1
repeat
  if is_valid_code(i,Pi1(i)) % # of arguments is unknown
    then print decode(i)
  i := i+1
forever
```

In order for this program to work, we need to define `is_valid_code( )` and `decode( )`. For a code to be valid, we must check that (i) the outermost definition is valid (one of the five above); and (ii) if the outermost definition is of type 3 or 4, the codes of the functions used in the definition are themselves valid. Thus we get something like this (in some convenient pseudocode):

```
is_valid_code(x,n) ==
  % x is the code and n is the expected number of arguments
```

```
if Pi1(x) <> n  % does the # of arguments match expectations?
  then
    false
  else set code Pi2(x) % for clarity
    if (code = 0) or (code = 1))  % Zero() or Succ()
      then true
    else if (Pi1(code) = 2)   % projection, check 1<=i<=j
      then (1 <= Pi2(code) <= Pi1(x))
    else if (Pi1(code) = 3)   % substitution, check pieces
      then set m Pi1(Pi2(code))  % for clarity
           (is_valid_code(Pi_2^(m+1)(Pi2(code)),m) and
            for all i, 1 <= i <= m,
              is_valid_code(Pi_(i+2)^(m+1)(Pi2(code)),n))
    else if (Pi1(code) = 4)   % prim. recursion, check pieces
      then (is_valid_code(Pi1(Pi2(code)),n-1) and
            is_valid_code(Pi2(Pi2(code)),n+1))
      % this again ignores the case n=1
    else false  % not one of the 5 recognized patterns
```

The initial call to the checking routine does not actually check arguments, since it uses as expected number of arguments the number coded in—so that the first matching check is guaranteed to succeed. After that, however, it is critical that the number of arguments be verified at each step. Now the function `decode()` is similar, but in some ways simpler, since it no longer needs to check the syntax, but can assume that the definition is valid.

```
decode(x) ==
  set code Pi2(x)
  print("Code ",code," defines a function of ",Pi1(x)," arguments.")
  case code of
    0: print("The function is Zero( )")
    1: print("The function is Succ( )")
    2: print("The function is the projection of ",Pi2(code))
    3: print("The function is defined by substitution;
       print("the outer function is as follows:")
       decode(Pi_2^(m+1)(Pi2(code)))
       print("and the inner functions are as follows:")
       for i:=1 to Pi1(Pi2(code)) do
          decode(Pi_(i+2)^(m+1)(Pi2(code)))
    4: print("The function is defined by primitive recursion;")
       print("the base function is as follows:")
       decode(Pi1(Pi2(code)))
       print("the recursive step function is as follows:")
       decode(Pi2(Pi2(code)))
  print("End of definition of code ",code)
```