Lecture 16:
A Minimal Programming Language

Lecture Outline

- A Minimal Programming Language: **MIN**
- Partial Recursive Functions are **MIN**-computable
- **MIN**-computable Functions are Partial Recursive

Reading

Chapter 4.3-4.4 of Brookshear

N.J. Cutland *Computability: An Introduction to Recursive Function Theory*, Cambridge University Press, Cambridge, 1980. Chapters 1-2.

A Minimal Programming Language: MIN

- How can we know if our favourite programming language is incapable of computing the solutions to certain problems ?

  How can we tell if one programming language is superior to another in terms of what is can compute ?

- In order to study these problems we can define a minimal programming language and explore its computing power.

  Any language which contains at least the constructs of our minimal language must then be capable of computing at least as much.

- We shall develop a minimal language which we we call **MIN** and then show it is capable of computing any partial recursive function.

  It then follows, if we accept the Church-Turing thesis, that it, and consequently any language at least powerful, is capable of expressing the solution to any algorithmically solvable problem.

A Minimal Programming Language: MIN (cont)

- Since **MIN** is to compute partial recursive functions, it requires only one data type: nonnegative integers.

- We define identifiers to be alphanumeric strings, starting with a letter.

  Therefore, we need no type declarations (since there is only data type and since constants can easily be distinguished from identifiers).

- **MIN** contains just three statement types:

  1. `incr varname;`

     This increments the value named by the identifier `varname`.

  2. `decr varname;`

     This decrements the value named by the identifier `varname`.

  3. `while varname` $\neq$ `0 do;`

     ⋮

     `end;`

     This repeats the statements occurring between the `while` and the `end` until the value named by the identifier `varname` is 0.

A Minimal Programming Language: MIN (cont)

- Using these basic statement types we can immediately define more powerful statement types that make programming in **MIN** much easier.

  These more powerful statement types we view as analogous to macros in many 'real' programming languages – statements that a preprocessor converts into their more verbose equivalents.

- There are two additional statement types:

  1. `clear varname;`

     This is an abbreviation of

     ```
     while varname ≠ 0 do;
        decr varname;
     end;
     ```

     This statement type has the effect of assigning 0 to `varname`.

  2. `varname1 ← varname2;`

     This is an abbreviation of

     ```
     clear aux;
     clear varname1;
     while varname2 ≠ 0 do;
        incr aux;
        decr varname2;
     end;
     while aux ≠ 0 do;
        incr varname1;
        incr varname2;
        decr aux;
     end;
     ```

     This has the effect of nondestructively assigning the value of variable `varname2` to variable `varname1`.

     It achieves this by transferring `varname2`'s value to the auxiliary variable `aux`, then using transferring the value from `aux` to each of `varname1` and `varname2`.

A Minimal Programming Language: MIN – Example

Write a **MIN** program that computes $f : \mathbf{N} \to \mathbf{N}$ defined by

$$f(x) = \begin{cases} 1 & \text{if } x \text{ is even} \\ 0 & \text{otherwise} \end{cases}$$

Assume x is made available in `x1` and that $f(x)$ is left in `z1`.

```
aux ← x1;
clear z1;
clear even;
clear tmp;
incr even;
while aux ≠ 0 do;
   decr aux;
   incr tmp ;
   while even ≠ 0 do;
      decr even;
      decr tmp;
   end;
   while tmp ≠ 0 do;
      decr tmp;
      incr even;
   end;
end;
while even ≠ 0 do;
   decr even;
   incr z1;
end;
```

`even` is always 1 if the main loop has been executed an even number of times; 0 if it has been executed an odd number of times.

Partial Recursive Functions are MIN-computable

- We want to show that any partial recursive function $f : \mathbf{N}^m \to \mathbf{N}^n$ is **MIN**-computable.

- We start with establishing the convention that the input m-tuples will be held in the variables `x1,x2, ..., xm` and the output will be produced in variables `z1, z2, ..., zn`.

- First we construct programs for the three initial functions:

  1. ζ is computed by `clear z1;`
  2. σ is computed by

     ```
     z1 ← x1;
     incr z1;
     ```

  3. π_j^m is computed by `z1 ← xj;`

- Next we need to show that programs can be created that compute partial recursive functions which are built up using the operations of combination, composition, primitive recursion, and minimalization from the initial functions.

  There are four cases to consider:

  1. **Combination**. We assume that we have **MIN** programs F and G that compute the partial recursive functions $f : \mathbf{N}^k \to \mathbf{N}^m$ and $g : \mathbf{N}^k \to \mathbf{N}^n$ respectively.
     To compute $f \times g$ we concatenate program G to the end of program F
     (a) modifying G to assign its output to `zm+1, ..., zm+n`
     (b) modifying F, if necessary, to insure it does not destroy its input values before G begins.
     (c) renaming any common auxiliary variables in F and G so that they have different names.

Partial Recursive Functions are MIN-computable (cont)

2. **Composition**. Again we assume that we have **MIN** programs F and G that compute the partial recursive functions $f : \mathbf{N}^k \to \mathbf{N}^m$ and $g : \mathbf{N}^m \to \mathbf{N}^n$ respectively.

   To compute $g \circ f$ we concatenate program G to the end of program F

   (a) modifying the output variables of F to be identical to the input variables of G

   (b) renaming any common auxiliary variables in F and G so that they have different names.

3. **Primitive recursion**. Suppose we have programs G and H which compute the partial recursive functions $g : \mathbf{N}^k \to \mathbf{N}^m$ and $h : \mathbf{N}^{k+m+1} \to \mathbf{N}^m$. Suppose also that the function $f : \mathbf{N}^{k+1} \to \mathbf{N}^m$ is defined by primitive recursion as:

$$\begin{aligned} f(\bar{x}, 0) &= g(\bar{x}) \\ f(\bar{x}, y+1) &= h(\bar{x}, y, f(\bar{x}, y)) \end{aligned}$$

   then f can be calculated by the following program, assuming G and H do not have nasty side effects:

```
G
aux ← xk+1;
clear xk+1;
while aux ≠ 0 do;
   xk+2 ← z1;
   xk+3 ← z2;
   ⋮
   xk+m+1 ← zm;
   H
   incr xk+1;
   decr aux;
end;
```

Partial Recursive Functions are MIN-computable (cont)

4. **Minimalization**. Suppose we have a program G which computes the partial recursive function $g : \mathbf{N}^{n+1} \to \mathbf{N}$. Further, suppose f is defined by minimalization as

$$f(\bar{x}) = \mu y[g(\bar{x}, y) = 0]$$

f can be calculated by the following program:

```
clear xn+1;
G
while z1 ≠ 0 do;
   incr xn+1;
   G
end;
z1 ← xn+1
```

This program computes $g(\bar{x}, 0)$, $g(\bar{x}, 1)$, $\cdots$ until a 0 output (`z1` is reached, at which point it stops.

- Thus, we have shown that **MIN** is capable of computing any partial recursive function, and hence, by the Church-Turing thesis is capable of expressing the solution to any algorithmically solvable problem.

- It follows that all we need in a programming language is the nonnegative integer data type together with the operations of incrementing, decrementing, and `while`-looping: any other features in a programming language are for ease of use and do not effect the computational power of the language.

MIN-computable Functions are Partial Recursive

- Of course we would not expect **MIN** to allow us to compute anything more than partial recursive functions – if it did we would have disproved the Church-Turing thesis !

- We can assure ourselves of this with several observations.

- First observe that any **MIN** program must contain k identifiers where $k > 0$ ($k > 0$ since every program contains at least one statement and every statement type contains an identifier).

  Any **MIN** program P may therefore be viewed as a function from $N^k \rightarrow N^k$ where the input k-tuple is the value of its k identifiers when the program starts and the output k-tuple is the value of the k identifiers when the program stops

  Note: if it does not stop then it is undefined for the given input.

- To show any such function computed by P is partial recursive we perform an induction on the number of statements in P:

  1. Basis step. Suppose P has only one statement. Then P is either an `incr`, `decr` or `while` statement.

     `incr` computes σ and so is partial recursive.

     `decr` computes $pred$ and so is partial recursive.

     ```
     while varname ≠ 0 do;
     end;
     ```

     computes the function

$$f(varname) = \begin{cases} 0 \text{ if } varname = 0 \\ \text{undefined otherwise} \end{cases}$$

MIN-computable Functions are Partial Recursive (cont)

2. Induction step. Suppose P has n statements, $n > 1$, and assume that any program with fewer than n statements computes a partial recursive function.

   There are two cases to consider:

   (a) The program is not one large `while` loop.
   In this case it is the concatenation of two shorter programs, each of which by the hypothesis of induction computes a partial recursive function.
   However the longer program computes the composition of the functions computed by the two shorter functions and hence it too computes a partial recursive function.

   (b) The program is one large `while` loop and so has the form

   ```
   while varname ≠ 0 do;
      B
   end;
   ```

   Since B has fewer than n statements we may assume, by the hypothesis of induction, that it computes a partial recursive function h from $\mathbf{N}^k \rightarrow \mathbf{N}^k$, where its k identifiers form the input and output k-tuples of the function it computes.

MIN-computable Functions are Partial Recursive (cont)

$varname$ must either

i. be one of the identifiers B manipulates or,

ii. if not, B can never terminate once started since it does not alter the loop controlling variable.

Suppose

i. Then $varname$ is the j-th identifier manipulated by B, for some j, $1 \leq j \leq k$.
Define the function $f : \mathbf{N}^{k+1} \rightarrow \mathbf{N}^k$ by primitive recursion as:

$$\begin{aligned} f(\bar{x}, 0) &= ident(\bar{x}) \\ f(\bar{x}, y+1) &= h(f(\bar{x}, y)) \end{aligned}$$

where $ident$ is the identity function, $ident(\bar{x}) = \bar{x}$ (this function can be defined as a combination of projections and so is primitive recursive).

Recalling that h is the partial recursive function computed by B, the value of $f(\bar{x}, y)$ is the k-tuple computed by the loop if B's k input variables are initialised to $\bar{x}$ and the loop is executed y times.

The number of times the body of the `while` loop will actually be performed is (recall j is the loop control variable)

$$\mu y[\pi_j^k \circ f(\bar{x}, y) = 0].$$

Thus the function computed by the `while` structure is:

$$g(\bar{x}) = f(\bar{x}, \mu y[\pi_j^k \circ f(\bar{x}, y) = 0])$$

which is partial recursive.

ii. In this case the whole `while` loop computes the partial recursive function which is the identity function when $varname = 0$ and is undefined for all other inputs.

Summary

- A simple programming language MIN can be defined which consists of
  - the single data type non-negative integers
  - identifiers made up of alphanumeric strings whose first character is alphabetic
  - three statement types – increment, decrement and while-loop

- MIN suffices to compute all the partial recursive functions.

  This is established by showing the three initial primitive recursive functions are MIN-computable and showing the four function-building operations required to build partial recursive functions can be carried out by MIN programs.

- Any function computed by a MIN program (viewed as a mapping from all of its identifiers before it runs to after it halts) is partial recursive.

- MIN shows that any programming language with at least its three simple constructs is as powerful as any conventional computational model (Turing machines, partial recursive functions).