Supplementary Lecture I

While Programs

We can relate the primitive and μ-recursive functions of Gödel to more modern concepts. Consider a simple programming language with variables $\mathbf{Var} = \{x, y, \ldots\}$ ranging over $\mathbb{N}$ containing the following constructs:

(i)	*simple assignments*	$x := 0 \quad x := y + 1 \quad x := y$
(ii)	*sequential composition*	$p\,;q$
(iii)	*conditional*	**if** $x < y$ **then** p **else** q
(iv)	*for loop*	**for** y **do** p
(v)	*while loop*	**while** $x < y$ **do** p

In (iii) and (v), the relation $<$ can be replaced by any one of $>$, $\geq$, $\leq$, $=$, or $\neq$. In (ii) we can parenthesize using **begin**... **end** if necessary.

Programs built inductively from these constructs are called **while** programs. Programs built without the **while** construct (v) are called **for** programs. We will show in Theorem I.1 that **while** programs compute exactly the μ-recursive functions and that **for** programs compute exactly the primitive recursive functions.

The intuitive operation of the **for** loop is as follows: upon entering the loop **for** y **do** p, the current value of variable y is determined, and the program

p is executed that many times. Assignment to the variable y within the body of the loop does not change the number of times the loop is executed, nor does execution of the body of the loop alone decrement y or change its value in any way except by explicit assignment.

The intuitive operation of the **while** loop is as follows: upon entering the loop **while** $x < y$ **do** p, the condition $x < y$ is tested with the current values of the variables x, y. If the condition is false, then the body of the loop is not executed, and control passes through to the statement following the **while** loop. If the condition is true, then the body p of the loop is executed once, and then the procedure is repeated with the new values of x, y. Thus the **while** loop repeatedly tests the condition $x < y$, and if true, executes the body p. The first time that the condition $x < y$ tests false (if ever), the body of the loop is not executed and control passes immediately to the statement following the loop. If the condition always tests true, then the **while** loop never halts, as for example with the program **while** $x = x$ **do** $x := x + 1$.

In the presence of the **while** loop, the **for** loop is redundant: **for** y **do** p is simulated by the **while** program

$$z := 0\,;w := y\,;\textbf{while } z < w \textbf{ do begin } p\,; z := z + 1 \textbf{ end}$$

where z and w are variables not occurring in p. However, note that **for** programs always halt. Thus the only source of potential nontermination is the **while** loop.

Semantics of While Programs

In order to prove the equivalence of **while** programs and the μ-recursive functions, we must give formal semantics for **while** programs.

A *state* or *environment* σ is an assignment of a nonnegative integer to each variable in **Var**; that is, $\sigma : \mathbf{Var} \to \mathbb{N}$. The set of all such environments is denoted **Env**. If a program is started in an initial environment σ, then in the course of execution, the values of variables will be changed, so that if and when the program halts, the final environment will in general be different from σ. We thus interpret programs p as *partial* functions $[\![p]\!] : \mathbf{Env} \to \mathbf{Env}$. The value $[\![p]\!](\sigma)$ is the final environment after executing the program p with initial environment σ, provided p halts. If p does not halt when started in initial environment σ, then $[\![p]\!](\sigma)$ is undefined. Thus $[\![p]\!] : \mathbf{Env} \to \mathbf{Env}$ is a partial function; its domain is the set of σ causing p to halt. Note that whether or not p halts depends on the initial environment; for example, if $\sigma(x) = 0$, then the program **while** $x > 0$ **do** $x := x + 1$ halts on initial environment σ, whereas if $\sigma(x) = 1$, then it does not.

Formally, the meaning $[\![p]\!]$ of a **while** program p is defined inductively as follows. For $\sigma \in \mathbf{Env}$, $x \in \mathbf{Var}$, and $a \in \mathbb{N}$, let $\sigma[x \leftarrow a]$ denote the environment that is identical to σ except for the value of x, which is a. Formally,

$$\sigma[x \leftarrow a](y) \overset{\text{def}}{=} \sigma(y), \quad \text{if } y \text{ is not } x,$$
$$\sigma[x \leftarrow a](x) \overset{\text{def}}{=} a.$$

Let $[\![p]\!]^n$ denote the n-fold composition of the partial function $[\![p]\!]$:

$$[\![p]\!]^n = \underbrace{[\![p]\!] \circ \cdots \circ [\![p]\!]}_{n},$$

where $[\![p]\!]^0$ is the identity function on **Env**. Formally,

$$[\![p]\!]^0(\sigma) \overset{\text{def}}{=} \sigma,$$
$$[\![p]\!]^{n+1}(\sigma) \overset{\text{def}}{=} [\![p]\!]([\![p]\!]^n(\sigma)).$$

Now define

$$[\![x := 0]\!](\sigma) \overset{\text{def}}{=} \sigma[x \leftarrow 0],$$
$$[\![x := y]\!](\sigma) \overset{\text{def}}{=} \sigma[x \leftarrow \sigma(y)],$$
$$[\![x := y+1]\!](\sigma) \overset{\text{def}}{=} \sigma[x \leftarrow \sigma(y)+1],$$
$$[\![p\,;q]\!](\sigma) \overset{\text{def}}{=} [\![q]\!]([\![p]\!](\sigma)), \quad \text{or in other words,}$$
$$[\![p\,;q]\!] \overset{\text{def}}{=} [\![q]\!] \circ [\![p]\!]$$

(here $[\![q]\!]([\![p]\!](\sigma))$ is undefined if $[\![p]\!](\sigma)$ is undefined),

$$[\![\textbf{if } x < y \textbf{ then } p \textbf{ else } q]\!](\sigma)$$
$$\overset{\text{def}}{=} \begin{cases} [\![p]\!](\sigma) & \text{if } \sigma(x) < \sigma(y), \\ [\![q]\!](\sigma) & \text{otherwise,} \end{cases}$$

$$[\![\textbf{for } y \textbf{ do } p]\!](\sigma)$$
$$\overset{\text{def}}{=} [\![p]\!]^{\sigma(y)}(\sigma),$$

$$[\![\textbf{while } x < y \textbf{ do } p]\!](\sigma)$$
$$\overset{\text{def}}{=} \begin{cases} [\![p]\!]^n(\sigma) & \text{if } n \text{ is the least number such that } [\![p]\!]^n(\sigma) \text{ is defined and } [\![p]\!]^n(\sigma)(x) \geq [\![p]\!]^n(\sigma)(y), \\ \text{undefined} & \text{if no such } n \text{ exists.} \end{cases}$$

We are now ready to give a formal statement of the equivalence of **while** programs and μ-recursive functions.

Theorem I.1 *(i) For every μ- (respectively, primitive) recursive function $f : \mathbb{N}^n \to \mathbb{N}$, there is a* **while** *(respectively,* **for***) program p such that for any* *environment σ, $[\![p]\!](\sigma)$ is defined iff $f(\sigma(x_1),\ldots,\sigma(x_n))$ is defined; and if both are defined, then*

$$[\![p]\!](\sigma)(x_0) = f(\sigma(x_1),\ldots,\sigma(x_n)).$$

(ii) *For every* **while** *(respectively,* **for***) program p with variables $x_1,\ldots,x_n$ only, there are μ- (respectively, primitive) recursive functions $f_i : \mathbb{N}^n \to \mathbb{N}$, $1 \le i \le n$, such that for any environment σ, $[\![p]\!](\sigma)$ is defined iff $f_i(\sigma(x_1),\ldots,\sigma(x_n))$ is defined, $1 \le i \le n$; and if all are defined, then*

$$f_i(\sigma(x_1),\ldots,\sigma(x_n)) = [\![p]\!](\sigma)(x_i), \quad 1 \le i \le n.$$

Once we have stated the theorem, the proof is quite straightforward and proceeds by induction on the structure of the program or μ-recursive function. We argue one case explicitly.

Suppose we are given a function $f : \mathbb{N}^n \to \mathbb{N}$ defined by primitive recursion. For simplicity, assume that the k in the primitive recursive definition of f is 1; then f is defined from $h : \mathbb{N}^{n-1} \to \mathbb{N}$ and $g : \mathbb{N}^{n+1} \to \mathbb{N}$ by

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

where $\overline{x} = x_2,\ldots,x_n$. We wish to give a program p that takes its inputs in variables x_1 and $\overline{x}$, computes f on these values, and leaves its result in x_0. By the induction hypothesis, g and h are computed by programs q and r, respectively. These programs expect their inputs in variables $x_1,\ldots,x_{n+1}$ and $x_2,\ldots,x_n$, respectively, and leave their outputs in x_0. Let $y_1,\ldots,y_n$ be new variables not occurring in either q or r. Then we can take p to be the following program:

```
$y_1 := x_1$ ; ··· ; $y_n := x_n$ ;      /* save values of input variables */
$r$ ;                                    /* set $x_0$ to $h(\overline{x})$ */
$x_1 := 0$ ;                             /* initialize iteration count */
for $y_1$ do                             /* at this point $x_0$ contains $f(x_1,\overline{x})$ */
  begin
    $y_1 := x_1$ ;                       /* save iteration count */
    $x_2 := y_2$ ; ··· ; $x_n := y_n$ ;  /* restore values of other variables */
    $x_{n+1} := x_0$ ;                   /* output from previous iteration */
    $q$ ;                                /* set $x_0$ to $g(x_1,\ldots,x_{n+1})$ */
    $x_1 := y_1 + 1$                     /* increment iteration count */
  end
```

Historical Notes

Gödel originally worked exclusively with the primitive recursive functions. Ackermann's [1] discovery of the non-primitive recursive yet intuitively computable total function (36.2) forced Gödel to rethink the foundations of his system and ultimately to include unbounded minimization, despite the fact that it led to partial functions. As we now know, this is inevitable: no r.e. list of total computable functions could contain all total computable functions, as can be shown by a straightforward diagonalization argument.

The relationship between the primitive recursive functions and **for** programs was observed by Meyer and Ritchie [86].