# Lecture 36

# Other Formalisms

In this lecture and the next we take a brief look at some of the other traditional formalisms that are computationally equivalent to Turing machines. Each one of these formalisms embodies a notion of *computation* in one form or another, and each can simulate the others. In addition to Turing machines, we'll consider

- Post systems;
- type 0 grammars;
- $\mu$-recursive functions ($\mu$ = "mu", Greek for m);
- $\lambda$-calculus ($\lambda$ = "lambda", Greek for l);
- combinatory logic; and
- **while** programs.

## Post Systems

By the 1920s, mathematicians had realized that much of formal logic was just symbol manipulation and strongly related to emerging notions of computability. Emil Post came up with a general formalism, now called *Post systems*, for talking about rearranging strings of symbols. A Post system

D. C. Kozen, *Automata and Computability*
© Springer Science+Business Media, Inc. 1997

consists of disjoint finite sets $N$ and $\Sigma$ of *nonterminal* and *terminal symbols*, respectively, a special *start symbol* $S \in N$, a set of *variables* $X_0, X_1, \ldots$ ranging over $(N \cup \Sigma)^*$, and a finite set of *productions* of the form

$$x_0 X_1 x_1 X_2 x_2 X_3 \cdots X_n x_n \rightarrow y_0 Y_1 y_1 Y_2 y_2 Y_3 \cdots Y_m y_m,$$

where the $x_i$ and $y_j$ are strings in $(N \cup \Sigma)^*$, and each $Y_j$ is some $X_i$ that occurs on the left-hand side. If a string in $(N \cup \Sigma)^*$ matches the left-hand side for some assignment of strings to the variables $X_i$, then that string can be rewritten as specified by the right-hand side. A string $x \in \Sigma^*$ is *generated* by the system if $x$ can be derived from $S$ by a finite sequence of such rewriting steps.

Post systems and Turing machines are equivalent in computational power. Any Post system can be simulated by a TM that writes the start symbol on a track of its tape, then does the pattern matching and string rewriting according to the productions of the Post system in all possible ways, accepting if its input $x$ is ever generated. Conversely, given any TM $M$, a Post system $P$ can be designed that mimics the action of $M$. The sentential forms of $P$ encode configurations of $M$.

One of Post's main theorems was that Post systems in which all productions are of the more restricted form

$$xX \rightarrow Xy$$

are just as powerful as general Post systems. Productions of this form say, "Take the string $x$ off the front of the sentential form if it's there, and put $y$ on the back." If you did Miscellaneous Exercise 99 on queue machines, you may have already recognized that this is essentially the same result.

## Type 0 Grammars

*Grammars* are a restricted class of Post systems that arose in formal language theory. There is a natural hierarchy of grammars, called the *Chomsky hierarchy*, which classifies grammars into four types named 0, 1, 2, and 3. The type 2 and type 3 grammars are just the context-free and right-linear grammars, respectively, which we have already seen. A more general class of grammars, called the *type 0* or *unrestricted* grammars, are much like CFGs, except that productions may be of the more general form

$$\alpha \rightarrow \beta, \tag{36.1}$$

where $\alpha$ and $\beta$ are any strings of terminals and nonterminals whatsoever. A type 0 grammar consists of a finite set of such productions. If the left-hand side of a production matches a substring of a sentential form, then the substring can be replaced by the right-hand side of the production. A

string $x$ of terminal symbols is *generated* by $G$, that is, $x \in L(G)$, if $x$ can be derived from the start symbol $S$ by some finite number of such applications; in symbols, $S \xrightarrow[G]{*} x$.

Type 0 grammars are a special case of Post systems: the grammar production (36.1) corresponds to the Post production

$$X\alpha Y \to X\beta Y.$$

Type 0 grammars are the most powerful grammars in the Chomsky hierarchy of grammars and generate exactly the r.e. sets. One can easily build a Turing machine to simulate a given type 0 grammar. The machine saves its input $x$ on a track of its tape. It then writes the start symbol $S$ of the grammar on another track and applies productions nondeterministically, accepting if its input string $x$ is ever generated.

Conversely, type 0 grammars can simulate Turing machines. Intuitively, sentential forms of the grammar encode configurations of the machine, and the productions simulate $\delta$ (Miscellaneous Exercise 104).

Type 1 grammars are the *context-sensitive grammars* (CSGs). These are like type 0 grammars with productions of the form (36.1), except that we impose the extra restriction that $|\alpha| \leq |\beta|$. Context-sensitive grammars are equivalent (except for a trivial glitch involving the null string) to *nondeterministic linear bounded automata* (LBAs), which are TMs that cannot write on the blank portion of the tape to the right of the input string (see Exercise 2 of Homework 8 and Exercise 2 of Homework 12). The bound on the tape in LBAs translates to the restriction $|\alpha| \leq |\beta|$ for CSGs.

## The $\mu$-Recursive Functions

Gödel defined a collection of number-theoretic functions $\mathbb{N}^k \to \mathbb{N}$ that, according to his intuition, represented all the computable functions. His definition was as follows:

(1) *Successor.* The function $\mathrm{s} : \mathbb{N} \to \mathbb{N}$ given by $\mathrm{s}(x) = x+1$ is computable.

(2) *Zero.* The function $\mathbf{z} : \mathbb{N}^0 \to \mathbb{N}$ given by $\mathbf{z}(\,) = 0$ is computable.

(3) *Projections.* The functions $\pi^n_k : \mathbb{N}^n \to \mathbb{N}$ given by $\pi^n_k(x_1, \ldots, x_n) = x_k$, $1 \leq k \leq n$, are computable.

(4) *Composition.* If $f : \mathbb{N}^k \to \mathbb{N}$ and $g_1, \ldots, g_k : \mathbb{N}^n \to \mathbb{N}$ are computable, then so is the function $f \circ (g_1, \ldots, g_k) : \mathbb{N}^n \to \mathbb{N}$ that on input $\overline{x} = x_1, \ldots, x_n$ gives

$$f(g_1(\overline{x}), \ldots, g_k(\overline{x})).$$

(5) *Primitive recursion.* If $h_i : \mathbb{N}^{n-1} \to \mathbb{N}$ and $g_i : \mathbb{N}^{n+k} \to \mathbb{N}$ are computable, $1 \leq i \leq k$, then so are the functions $f_i : \mathbb{N}^n \to \mathbb{N}$, $1 \leq i \leq k$, defined by mutual induction as follows:

$$f_i(0, \overline{x}) \stackrel{\text{def}}{=} h_i(\overline{x}),$$
$$f_i(x+1, \overline{x}) \stackrel{\text{def}}{=} g_i(x, \overline{x}, f_1(x, \overline{x}), \ldots, f_k(x, \overline{x})),$$

where $\overline{x} = x_2, \ldots, x_n$.

(6) *Unbounded minimization.* If $g : \mathbb{N}^{n+1} \to \mathbb{N}$ is computable, then so is the function $f : \mathbb{N}^n \to \mathbb{N}$ that on input $\overline{x} = x_1, \ldots, x_n$ gives the least $y$ such that $g(z, \overline{x})$ is defined for all $z \leq y$ and $g(y, \overline{x}) = 0$ if such a $y$ exists and is undefined otherwise. We denote this by

$$f(\overline{x}) = \mu y.(g(y, \overline{x}) = 0).$$

The functions defined by (1) through (6) are called the *$\mu$-recursive functions*. The functions defined by (1) through (5) only are called the *primitive recursive functions*.

Example 36.1

- The constant functions $\mathbf{const}_n( \ ) = n$ are primitive recursive:

$$\mathbf{const}_n \stackrel{\text{def}}{=} \underbrace{\mathbf{s} \circ \cdots \circ \mathbf{s}}_{n} \circ \mathbf{z}.$$

- Addition is primitive recursive, since we can define

$$\mathbf{add}(0, y) \stackrel{\text{def}}{=} y,$$
$$\mathbf{add}(x+1, y) \stackrel{\text{def}}{=} \mathbf{s}(\mathbf{add}(x, y)).$$

This is a bona fide definition by primitive recursion: in rule (5) above, take $k = 1$, $n = 2$, $h = \pi^1_1$, and $g = \mathbf{s} \circ \pi^3_3$. Then

$$\mathbf{add}(0, y) = h(y) = y,$$
$$\mathbf{add}(x+1, y) = g(x, y, \mathbf{add}(x, y)) = \mathbf{s}(\mathbf{add}(x, y)).$$

- Multiplication is primitive recursive, since

$$\mathbf{mult}(0, y) \stackrel{\text{def}}{=} 0,$$
$$\mathbf{mult}(x+1, y) \stackrel{\text{def}}{=} \mathbf{add}(y, \mathbf{mult}(x, y)).$$

Note how we used the function **add** defined previously. We are allowed to build up primitive recursive functions inductively in this way.

- Exponentiation is primitive recursive, since

$$\mathbf{exp}(x, 0) \stackrel{\text{def}}{=} 1,$$
$$\mathbf{exp}(x, y+1) \stackrel{\text{def}}{=} \mathbf{mult}(x, \mathbf{exp}(x, y)).$$

- The predecessor function

$$x \dot{-} 1 = \begin{cases} x - 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0 \end{cases}$$

is primitive recursive:

$$0 \dot{-} 1 \stackrel{\text{def}}{=} 0,$$
$$(x+1) \dot{-} 1 \stackrel{\text{def}}{=} x.$$

- Proper subtraction

$$x \dot{-} y = \begin{cases} x - y & \text{if } x \geq y, \\ 0 & \text{if } x < y \end{cases}$$

is primitive recursive, and can be defined from predecessor in exactly the same way that addition is defined from successor.

- The sign function is primitive recursive:

$$\begin{aligned} \mathbf{sign}(x) &\stackrel{\text{def}}{=} 1 \dot{-} (1 \dot{-} x) \\ &= \begin{cases} 1 & \text{if } x > 0, \\ 0 & \text{if } x = 0. \end{cases} \end{aligned}$$

- The relations $<$, $\leq$, $>$, $\geq$, $=$, and $\neq$, considered as (0,1)-valued functions, are all primitive recursive; for example,

$$\begin{aligned} \mathbf{compare}_{\leq}(x,y) &\stackrel{\text{def}}{=} 1 \dot{-} \mathbf{sign}(x \dot{-} y) \\ &= \begin{cases} 1 & \text{if } x \leq y, \\ 0 & \text{if } x > y. \end{cases} \end{aligned}$$

- Functions can be defined by cases. For example,

$$g(x,y) = \begin{cases} x+1 & \text{if } 2^x < y, \\ x & \text{if } 2^x \geq y \end{cases}$$

is primitive recursive:

$$g(x,y) \stackrel{\text{def}}{=} \mathbf{compare}_{<}(2^x, y) \cdot (x+1) + \mathbf{compare}_{\geq}(2^x, y) \cdot x.$$

- Inverses of certain functions can be defined. For example, $\lceil \log_2 y \rceil$ is primitive recursive:[1] $\lceil \log_2 y \rceil = f(y,y)$, where

$$f(0,y) \stackrel{\text{def}}{=} 0,$$
$$f(x+1,y) \stackrel{\text{def}}{=} g(f(x,y),y),$$

[1] $\lceil x \rceil$ = least integer not less than $x$; $\log_2$ = base 2 logarithm.

and $g$ is from the previous example. The function $f$ just continues to add 1 to its first argument $x$ until the condition $2^x \geq y$ is satisfied. This must happen for some $x \leq y$. Inverses of other common functions, such as square root, can be defined similarly. □

Observe that all the primitive recursive functions are total, whereas a $\mu$-recursive function may not be. There exist total computable functions that are not primitive recursive; one example is *Ackermann's function*:

$$\begin{aligned} A(0,y) &\stackrel{\text{def}}{=} y+1, \\ A(x+1,0) &\stackrel{\text{def}}{=} A(x,1), \\ A(x+1,y+1) &\stackrel{\text{def}}{=} A(x,A(x+1,y)). \end{aligned} \tag{36.2}$$