

# CS20a: Models (Nov 5, 2002)

- Turing Machines
  - *Church's thesis: TM are a general model of computation*
  - *But (almost) everything is undecidable*
- Today: alternative models
  - *Primitive recursive functions*
  - *Total recursive functions*
  - *Partial recursive functions*



# Primitive recursion

Functions  $\mathbb{N}$  tuples  $\rightarrow \mathbb{N}$  tuples

The following functions are primitive recursive:

- $s(x) = x + 1$  (the successor function)
- $z(x) = 0$  (the zero function)
- $\pi_i^m(x_1, \dots, x_m) = x_i$  (projection)
- For any  $f : \mathbb{N}^m \rightarrow \mathbb{N}^m$ , and  $g_1, \dots, g_m : \mathbb{N}^k \rightarrow \mathbb{N}$ , the *composition*  $f(g_1(\bar{x}), \dots, g_m(\bar{x}))$  is primitive recursive



# Primitive recursion (conventional form)

Given  $h : \mathbb{N}^{n-1} \rightarrow \mathbb{N}^m$ , and  $g : \mathbb{N}^{n+m} \rightarrow \mathbb{N}^m$ , define  $f : \mathbb{N}^n \rightarrow \mathbb{N}^m$  as follows:

- $f(0, \bar{x}) = h(\bar{x})$
- $f(s(y), \bar{x}) = g(y, \bar{x}, f(y, \bar{x}))$



# Arithmetic

## Addition +

$$x + y \equiv x + \underbrace{1 + \dots + 1}_{y \text{ times}}$$

---

$$\begin{aligned} x + 0 &= x \\ x + s(y) &= s(x + y) \end{aligned}$$

## Multiplication \*

$$xy = \underbrace{x + \dots + x}_{y \text{ times}}$$

---

$$\begin{aligned} x0 &= 0 \\ x s(y) &= xy + x \end{aligned}$$



# Exponentiation

## Exponentiation

$$\begin{array}{l} x^y = \underbrace{x \cdot x \cdot \dots \cdot x}_{y \text{ times}} \\ \hline x^0 = 1 \\ x^{s(y)} = x^y \cdot x \end{array}$$

## Hyperexponentiation

$$\begin{array}{l} x \uparrow y \equiv \underbrace{x^{x^{\cdot^{\cdot^{\cdot^x}}}}}_{y \text{ times}} \\ \hline x \uparrow 0 = 1 \\ x \uparrow s(y) = x^{x \uparrow y} \end{array}$$



# Hyper-exponentiation

## Hyper-hyper-exponentiation

$$x \uparrow\uparrow y \equiv \underbrace{x \uparrow x \uparrow \cdots \uparrow x}_{y \text{ times}}$$

---

$$x \uparrow\uparrow 0 = 1$$

$$x \uparrow\uparrow s(y) = x \uparrow (x \uparrow\uparrow y)$$



# Ackerman's function

$$\begin{array}{lcl} A(0, x, y) & = & s(y) \\ A(1, x, y) & = & x + y \\ A(2, x, y) & = & xy \end{array} \quad \left| \quad \begin{array}{lcl} A(0, x, y) & = & s(y) \\ A(1, x, 0) & = & x \\ A(2, x, 0) & = & 0 \\ A(n, x, 0) & = & 1 (n \geq 3) \end{array}\right.$$

$$A(s(n), x, s(y)) = A(n, x, A(s(n), x, y))$$

**Theorem** For any fixed  $n$ ,  $A(n, x, y)$  is primitive recursive.



# Subtraction

**predecessor**

$$\begin{aligned}p(0) &= 0 \\ p(s(y)) &= y\end{aligned}$$

**subtraction**

$$x \dot{-} y \begin{cases} x - y & \text{if } y \leq x \\ 0 & \text{otherwise} \end{cases}$$

Algorithm: iterate predecessor on  $x$ ,  $y$  times



# Equality

## equality

$$\begin{aligned} |x - y| &= (x \dot{-} y) + (y \dot{-} x) \\ eq(x, y) &= \begin{cases} 0 & \text{if } x \neq y \\ 1 & \text{if } x = y \end{cases} \\ &= 1 \dot{-} |x - y| \end{aligned}$$

## conditionals

$$\begin{aligned} cond(x, y, z, w) &= \begin{cases} z & \text{if } x = y \\ w & \text{if } x \neq y \end{cases} \\ &= eq(x, y) \cdot z + (1 \dot{-} eq(x, y)) \cdot w \end{aligned}$$



# *For-programs*

- Variables  $x, y, z$  over  $\mathbb{N}$
- Simple assignments:
  - $x \leftarrow y$
  - $x \leftarrow s(y)$
  - $x \leftarrow 0$
- Induction: if  $p, q$  are *for-programs*, so are:
  - $p; q$  (sequential composition)
  - **if**  $x = y$  **then**  $p$  **else**  $q$  (conditional)
  - **for**  $y$  **do**  $p$  **done** (for-loop)



# ***For-loop: for $y$ do $p$ done***

- Execute  $p$ ,  $y$  times
- Assignments to  $y$  in  $p$  do not affect the number of iterations
- **for  $i = 1$  to  $y$  do  $p$  done** is equiv to

```
 $i \leftarrow 0;$   
for  $y$  do  
     $p;$   
     $i \leftarrow s(i)$   
done
```



# *Interpretation of for-programs*

- Some variables are designated as input
- Some are designated as output
- The program defines a function  $N_{input} \rightarrow N_{output}$



# *For programs and primitive recursion*

**Theorem** The for-programs define exactly the primitive-recursive functions.

**Encoding p.r. functions with for-loops.** By induction on the size of the p.r. function.

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



# Encoding p.r. functions as for-loops

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

Assume by induction that we have for-programs for  $g, h$ . Then the translation is:

$f(y, \bar{x}) =$   
     $\bar{z} \leftarrow h(\bar{x});$     ( $\bar{z}$  are new variables)  
     $i \leftarrow 0;$         (assume  $y$  does not appear in  $h, g$ )  
    **for**  $y$  **do**  
         $\bar{z} \leftarrow g(i, \bar{x}, \bar{z});$   
         $i \leftarrow s(i)$   
    **done**



# Encoding for-loops as p.r. functions

Hard part: find a function  $f(y, \bar{x}) =$

```
for y do
  ⋮
  body
  ⋮
done
```

- By induction, we assume that there is a function  $g : \mathbb{N}^m \rightarrow \mathbb{N}^m$  that describes the body
- Define:  $f(s(i), \bar{x}) = g(f(i, \bar{x}))$



# *Partial-recursive functions*



# Properties of p.r. programs

- The programs always terminate
- However, does not include all recursive computations
  - *Can code extremely “large” functions, but some functions are not definable (Ackerman’s)*
  - *We’ll argue later that all tractable computations are primitive recursive*



# *Partial recursive functions (Godel)*

- To capture r.e. computations, we need more
- A partial recursive computation includes the prim-rec computations, plus unbounded minimization



# Unbounded minimization

- Find the least  $y$  s.t.  $g(\bar{x}, y) = 0$

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

where  $g$  is primitive recursive

- Alternatively:

$$f(\bar{x}) = \mu y. (g(\bar{x}, y) = 0 \wedge g(\bar{x}, z) \text{ is defined for } z \leq y)$$

where  $g$  does not have to be primitive recursive



# *While-programs*

The *while* programs are defined inductively:

- All for-programs are while-programs
- Add **while**  $x \neq y$  **do**  $p$  **done**, where  $p$  is a while-program



# Enumerating the partial-recursive funs

Note, there is an injection  $\{0, 1\}^* \rightarrow \mathbb{N}$

1. For a string  $w \in \{0, 1\}^*$ , prepend a 1
2. Consider as a binary number, and subtract 1

$\epsilon$	$\rightarrow$	$1\epsilon$	$\rightarrow$	0
0	$\rightarrow$	10	$\rightarrow$	1
1	$\rightarrow$	11	$\rightarrow$	2
00	$\rightarrow$	100	$\rightarrow$	3



# *Indexing partial recursive funs*

$$\varphi_0, \varphi_1, \dots, \varphi_i, \dots$$

- $i$  is called the *index*
- $\varphi_i$  is the function, a semantic concept

Axiom of extensionality: two functions are equal, if they are equal on all arguments.

$$(\forall x. \varphi_i(x) = \varphi_j(x)) \Rightarrow \varphi_i = \varphi_j$$



# Universal functions

- There is a universal function  $U(\langle i, x \rangle) = \varphi_i(x)$ , where  $U = \varphi_j$  for some  $j$
- There is a while-program interpreter that takes char strings, treats them as an encoding of a while program, and simulates them. Moreover, the interpreter can be written as a while program.



# Gödel numbering

Gödel numbering:

- There is a universal function  $U(\langle i, \mathbf{x} \rangle) = \varphi_i(\mathbf{x})$
- For any  $m, n$  there is a total recursive function  $S_n^m$  (a partial recursive function defined on all inputs), s.t. for any  $i$  and  $x_1, \dots, x_n, y_1, \dots, y_m$

$$\varphi_{S_n^m(i, x_1, \dots, x_n)}(y_1, \dots, y_m) = \varphi_i(x_1, \dots, x_n, y_1, \dots, y_m)$$

An indexing is *acceptable* iff it satisfies these two properties.



# Composition

Composition is effective:

$$(\varphi_i \circ \varphi_j)(x) = \varphi_i(\varphi_j(x))$$

$$\begin{aligned}\varphi_{comp(i,j)} &= \varphi_i(\varphi_j(x)) \\ &= U(i, U(j, x)) \\ &= U \circ (\pi_1^3, U \circ (\pi_2^3, \pi_3^3))(\backslash i, j, x \rangle) \\ &= \varphi_k(\backslash i, j, x \rangle) \quad (\text{for some } k) \\ &= \varphi_{S_2^1(k, i, j)}(x)\end{aligned}$$

$$\text{So } comp(i, j) = S_2^1(k, i, j)$$



# Constant functions

Constant functions:

$$\begin{aligned}\varphi_{const}(\mathbf{x}) &= k \quad (\text{for any } \mathbf{x}) \\ &= \pi_1^2(k, \mathbf{x}) \\ &= \varphi_l(k, \mathbf{x}) \quad (\text{for some } l) \\ &= \varphi_{S_1^1(l, k)}(\mathbf{x})\end{aligned}$$

So take  $const = S_1^1(l, k)$



# Recursion theorem

## Theorem (Fixed-point theorem)

For all total recursive functions  $f : \mathbb{N} \rightarrow \mathbb{N}$ , there is an index  $i$  s.t.  $\varphi_i = \varphi_{f(i)}$ .

## Proof

Take any  $v \in \mathbb{N}$ , and consider a recursive function that does the following on  $x$ :

1. Compute  $\varphi_v(v)$
2. If it halts, apply  $f$  to get  $f(\varphi_v(v))$
3. Use it as an index: compute  $\varphi_{f(\varphi_v(v))}(x)$



## Recursion theorem proof (cont.)

3. Use it as an index: compute  $\varphi_{f(\varphi_v(v))}(x)$
4. There is a total recursive function  $h : \mathbb{N} \rightarrow \mathbb{N}$  s.t.  
 $\varphi_{h(v)}(x) = \varphi_{f(\varphi_v(v))}(x)$
5. Since  $h$  is total, it has an index  $h = \varphi_u$

$$\varphi_{h(u)} = \varphi_{\varphi_u(u)} = \varphi_{f(\varphi_u(u))}$$

6. So  $\varphi_u(u)$  is a fixed-point of  $f$ .



# Rice's theorem

## Theorem (Rice's theorem)

Every non-trivial property of the partial recursive functions is undecidable. That is, there is no total recursive function  $p : \mathbb{N} \rightarrow \{0, 1\}$  s.t.

- $p(k) \neq p(l)$
- For any  $i, j$ , if  $\varphi_i = \varphi_j$ , then  $p(i) = p(j)$

**Proof** Suppose  $p$  exists. Define

$$f(i) = \begin{cases} k & \text{if } p(i) = p(l) \\ l & \text{if } p(i) = p(k) \end{cases}$$

$f$  is total, and it has no fixpoint.

