CS311 Computational Structures

Other models of Computation

Lecture 13

Andrew Black
Andrew Tolmach

Portland State University

What is Computation?

- What does “computable” mean?
  - A computer can calculate it?
  - There is some (formally described execution) process and a (formally described) set of instructions — an algorithm — that describes how to get the answer

- Examples:
    - Generating strings from a grammars:
        - derivation is the process; the algorithm is encoded in the rules of the grammar
    - Accepting a string in a state machine
        -
    - Executing an ML program
        -
- These are all *Models of Computation*

The *Power* of a Model

- We know that some models of computation are more powerful than others:
  - CFG are more powerful than Regular Grammars
  - DFAs have the same power as NFAs
  - Turing machines are more powerful than PDAs
- Is there a “most powerful model”

Turing's Thesis

- Our intuitive notion of "computation" is precisely captured by the formal device known as a Turing Machine
- There is no model of computation more powerful than a Turing machine

Steam-powered Turing machine

Sieg Hall, 1987

Recall: Church-Turing Thesis

- Thesis: the problems that can be decided by an algorithm are exactly those that can be decided by a Turing machine.
- This cannot be proved; it is essentially a **definition** of the word “algorithm.”
- But there’s lots of evidence that our intuitive notion of algorithm is equivalent to a TM, and no convincing counter-examples have been found yet.

What about Alonzo Church?

- The Turing thesis is usually called the Church-Turing Thesis, in honor of Alonzo Church (1903–1995)
  - Working with his students (J. Barkley Rosser, Steven C. Kleene, and Alan M. Turing) Church established the equivalence of the Lambda calculus, recursive function theory, and Turing machines
  - They all capture the notion of computability

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - Grammars
  - Partial Recursive Functions
  - Lambda calculus
  - Markov Algorithms
  - Post Algorithms
  - Post Canonical Systems
  - **Simple programming language** with while loops
- All have been shown equivalent to Turing machines by simulation proofs

Simple (Hein p 776)

A *Simple* program is defined as follows:

1. V, an infinite set of variables that take values in Nat_0, and are initially 0
2. S, statements, which are either
   2.1. While statements: **while** V≠0 **do** S **od**
   2.2. Assignments: V := 0, V_1 := succ(V_2), V_1 := pred(V_2), or
   2.3. a sequence of statements separated by ;
3. S is a simple program

Portland State University

- Note that pred(0) = 0, to ensure that we stay in $\mathbb{N}_0$
- Can we compute anything interesting with this language?
  - yes!

Macro Statement	*Simple Code for Macro*
$X := Y$	$X := \text{succ}(Y); X := \text{pred}(X)$
$X := 3$	$X := 0; X := \text{succ}(X);$ $X := \text{succ}(X); X := \text{succ}(X)$
$X := X + Y$	$I := Y;$ **while** $I \neq 0$ **do** $X := \text{succ}(X); I := \text{pred}(I)$ **od**
Loop Forever	$X := 0; X := \text{succ}(X);$ **while** $X \neq 0$ **do** $Y := 0$ **od**
repeat S **until** $X = 0$	S; **while** $X \neq 0$ **do** S **od**

Figure 13.2 Some simple macros.

- Let's try: x−y, x<y, **while** x<y **do** S **od**

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - Grammars
  - Partial Recursive Functions
  - Lambda calculus
  - **Markov Algorithms**
  - Post Algorithms
  - Post Canonical Systems
  - Simple programming language with while loops
- All have been shown equivalent to Turing machines by simulation proofs

Markov Algorithms

- A Markov Algorithm over an alphabet A is a finite ordered sequence of productions $x \to y$, where $x, y \in A^*$. Some productions may be "Halt" productions.
  - e.g. $abc \to b$
    $ba \to x$ (halt)
- Execution proceeds as follows:

1. Let the input string be w
2. The productions are scanned in sequence, looking for a production x → y where x is a substring of w
3. the left-most x in w is replaced by y
4. If the production is a halt production, we halt
5. If no matching production is found, the process halts
6. If a replacement was made, we repeat from step 2.

- Note that a production $\varepsilon \rightarrow a$ inserts a at the start of the string.
- What does this Markov algorithm do?

$$aba \rightarrow b$$
$$ba \rightarrow b$$
$$b \rightarrow a$$

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - (Type 0 *a.k.a.* Unrestricted) **Grammars**
  - Partial Recursive Functions
  - Lambda calculus
  - Markov Algorithms
  - Post Algorithms
  - Post Canonical Systems, etc. etc. etc.
- All have been shown equivalent to Turing machines by simulation proofs

Grammars

- We can extend the notion of context-free grammars to a more general mechanism
- An (unrestricted) grammar $G = (V,\Sigma,R,S)$ is just like a CFG except that rules in R can take the more general form $\alpha \to \beta$ where α,β are **arbitrary** strings of terminals and variables (α must contain a variable).
- If $\alpha \to \beta$ then $u\alpha v \Rightarrow u\beta v$ ("yields") in one step
- Define $\Rightarrow^*$ ("derives") as reflexive transitive closure of $\Rightarrow$.

Example: counting

- Grammar generating {w ∈ {a,b,c}*| w has equal numbers of a's, b's, and c's }
- G = ({S,A,B,C},{a,b,c},R,S) where R is

$S \rightarrow \varepsilon$

$S \rightarrow ABCS$

$AB \rightarrow BA$ $AC \rightarrow CA$ $BC \rightarrow CB$

$BA \rightarrow AB$ $CA \rightarrow AC$ $CB \rightarrow BC$

$A \rightarrow a$ $B \rightarrow b$ $C \rightarrow c$

try generating ccbaba

Example: $\{a^{2^n}, n \geq 0\}$

- Here's a set of grammar rules

1. S → a
2. S→ ACaB
3. Ca → aaC
4. CB → DB
5. CB → E
6. aD → Da
7. AD →AC
8. aE → Ea
9. AE → ε

try generating 2^3 a's

(Unrestricted) Grammars and Turing machines have equivalent power

- For any grammar G we can find a TM M such that L(M) = L(G).
- For any TM M, we can find a grammar G such that L(G) = L(M).

From Grammar to TM (1)

- For any grammar G we can find a TM M such that L(M) = L(G).
- Use a non-deterministic 2-tape TM
  - First tape holds input.
  - Second tape holds a non-deterministically generated string of symbols derivable in G
- Initialize second tape to start symbol S

From Grammar to TM (2)

- Machine M repeatedly does the following:
  - Nondeterministically move to some position *i* in the active part of the second tape.
  - Nondeterministically select a rule $\alpha \rightarrow \beta$
  - If α matches the tape contents starting at *i*, rewrite the tape replacing α with β
  - If the string on tape 2 matches the input on tape 1, accept; otherwise loop.
- Easy to see that M accepts exactly the strings derivable in G.

From TM to Grammar

- For any TM M, we can find a grammar G such that L(G) = L(M).
- Key idea: represent each TM configuration *C* as a string [*C*] and construct grammar rules such that:
  C_1 yields C_2 in the TM iff
  [C_1] yields [C_2] in the grammar.
- As usual, we rely on the fact that only a finite portion of the TM tape is in use at any time.

Simulating machine transitions

- Given $M = (Q,\Sigma,\Gamma,\delta,q_0,\sqcup,F)$, define $G = (V,\Sigma,\mathbf{R},S)$ as follows:
  - $V = Q \cup \Gamma \cup \{[,]\}$ S doesn't matter
  - If $\delta(q,a) = (p,b,R)$ then **R** contains $qa \rightarrow bp$
  - If $\delta(q,\sqcup) = (p,b,R)$ then **R** also contains $q] \rightarrow bp]$
  - If $\delta(q,a) = (p,b,L)$ then **R** contains $cqa \rightarrow pcb$ $(\forall c \in \Gamma)$ and $[qa \rightarrow [p\sqcup b$
  - If $\delta(q,\sqcup) = (p,b,L)$ then **R** also contains $cq] \rightarrow pcb]$ $(\forall c \in \Gamma)$ and $[q] \rightarrow [p\sqcup b]$

Full set of grammar rules (1)

1. From start symbol S, generate a random string <w> where $w \in \Sigma^*$ and <,> are variables $\notin \Gamma$
   - $S \rightarrow <S_1>$
   - $S_1 \rightarrow xS_1$ for each $x \in \Sigma$
   - $S_1 \rightarrow \varepsilon$
2. Convert <w> to $w[q_0w]$

Full set of grammar rules (2)

3. Simulate computation between [and]
   - See previous slides
4. If the string [z] contains a state in F, erase [z] leaving the string w
   - $xq_a \to q_a$ and $q_a x \to q_a$ for each $x \in \Gamma$ and $q_a \in F$
   - $[q_a] \to \varepsilon$ for each q_a in F

- Putting all rules together, we get grammar where $S \Rightarrow^* w$ iff $q_0 w \Rightarrow^* \ldots q_a \ldots$ in TM

Other Notions of Computability

- Many other notions of computability have been proposed, e.g.
  - Grammars
  - Partial **Recursive Functions**
  - Lambda calculus
  - Markov Algorithms
  - Post Algorithms
  - Post Canonical Systems
  - Simple programming language with while loops
- All have been shown equivalent to Turing machines by simulation proofs

Computation using Numerical Functions

- We're used to thinking about computation as something we do with **numbers** (e.g. on the naturals)
- What kinds of functions from numbers to numbers can we actually compute?
- To study this, we make a very careful selection of building blocks

Primitive Recursive Functions

- The primitive recursive functions from $\mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \mathbb{N}$ are those built from these primitives:
  - zero(x) = 0   succ(x) = x+1
  - $\pi_{k,j}(x_1, x_2, \ldots, x_k) = x_j$ for $0 < j \leq k$
- using these mechanisms:
  - Function composition, and
  - Primitive recursion

Function Composition

- Define a new function f in terms of functions h and $g_1, g_2, \ldots, g_m$ as follows:
  - $f(x_1,\ldots x_n) = h(g_1(x_1,\ldots,x_n),\ldots g_m(x_1,\ldots,x_n))$
  - Example: f(x) = x + 3 can be expressed using two compositions as f(x) = succ(succ(succ(x)))

Primitive Recursion

- Primitive recursion defines a new function f in terms of functions h and g as follows:
  - $f(x_1, ..., x_k, 0) = h(x_1,...,x_k)$
  - $f(x_1, ..., x_k, succ(n)) = g(x_1,...,x_k, n, f(x_1,...,x_k,n))$
- Many ordinary functions can be defined using primitive recursion, e.g.
  - $add(x,0) = \pi_{1,1}(x)$
  - $add(x, succ(y)) = succ(\pi_{3,3}(x, y, add(x,y)))$

More P.R. Functions

- For simplicity, we omit projection functions and write 0 for zero(_) and 1 for succ(0)
  - add(x,0) = x   add(x,succ(y)) = succ(add(x,y))
  - mult(x,0) = 0   mult(x,succ(y)) = add(x,mult(x,y))
  - factorial(0) = 1   factorial(succ(n)) = mult(succ(n),factorial(n))
  - exp(n,0) = 1   exp(n, succ(n)) = mult(n,exp(n,m))
  - pred(0) = 0   pred(succ(n)) = n
- Essentially all practically **useful** arithmetic functions are primitive recursive, but...

Portland State University

Ackermann's Function is not Primitive Recursive

- A famous example of a function that is clearly well-defined but not primitive recursive
- $A(m, n) =$ if $m = 0$ then $n + 1$
  else if $n = 0$ then $A(m - 1, 1)$
  else $A(m - 1, A(m, n - 1))$

- This function grows extremely fast!

Values of $A(m, n)$

$m \backslash n$	0	1	2	3	4	n
0	1	2	3	4	5	$n+1$
1	2	3	4	5	6	$n+2=2+(n+3)-3$
2	3	5	7	9	11	$2n+3=2\cdot(n+3)-3$
3	5	13	29	61	125	$2^{(n+3)}-3$
4	13	65533	$2^{65536}-3$	$2^{2^{65536}}-3$	$A(3, A(4, 3))$	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{n+3 \text{ twos}}-3$
5	65533	$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{65536 \text{ twos}}-3$	$A(4, A(5, 1))$	$A(4, A(5, 2))$	$A(4, A(5, 3))$	$A(4, A(5, n-1))$
6	$A(5, 1)$	$A(5, A(6, 0))$	$A(5, A(6, 1))$	$A(5, A(6, 2))$	$A(5, A(6, 3))$	$A(5, A(6, n-1))$

A is *not* primitive recursive

- Ackermann's function grows faster than any primitive recursive function, that is:
  - for *any* primitive recursive function f, there is an n such that

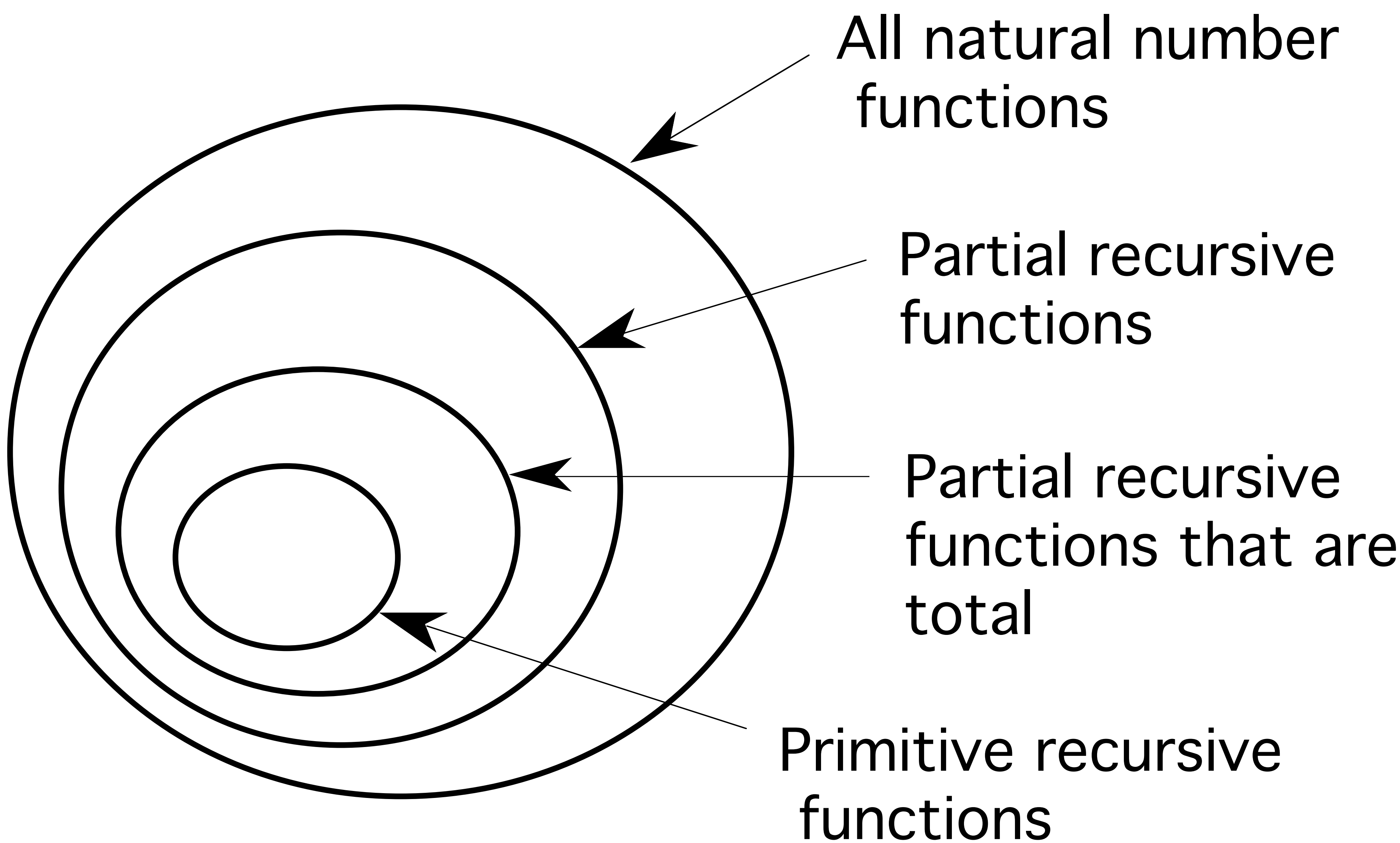
$$A(n, x) > f\, x$$

- So A *can't* be primitive recursive

Partial Recursive Functions

- *A* belongs to class of **partial recursive functions**, a superset of the primitive recursive functions.
- Can be built from primitive recursive operators & new **minimization** operator
  - Let *g* be a (k+1)-argument function.
  - Define $f(x_1,...,x_k)$ as the **smallest** m such that $g(x_1,...,x_k,m) = 0$ (if such an m exists)
    - Otherwise, $f(x_1,...,x_n)$ is undefined
  - We write $f(x_1,...,x_k) = \mu m.[g(x_1,...,x_k,m) = 0]$
    - Example: $\mu m.[mult(n,m) = 0] = zero(\_)$

Hierarchy of Numeric Functions



Turing-computable functions

- To formalize the connection between partial recursive functions and Turing machines, we need to describe how to use TM's to compute functions on $\mathbb{N}$.
- We say a function $f : \mathbb{N} \times \mathbb{N} \times \ldots \times \mathbb{N} \rightarrow \mathbb{N}$ is **Turing-computable** if there exists a TM that, when started in configuration $q_0 1^{n1} \sqcup 1^{n2} \sqcup \ldots \sqcup 1^{nk}$, halts with just $1^{f(n1,n2,\ldots nk)}$ on the tape.
- **Fact: f is Turing-computable iff it is partial recursive.**