Lecture Notes for CS 2110
Introduction to Theory of Computation

Robert Daley
Department of Computer Science
University of Pittsburgh
Pittsburgh, PA 15260

Next Up Previous Contents Index

Next: Forward

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: Contents **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** Lecture Notes for CS 2110 Introduction to Theory

Forward

These notes have been compiled over the course of more than twenty years and have been greatly influenced by the treatments of the subject given by Michael Machtey and Paul Young in *An Introduction to the Genereal Theory of Algorithms* and to a lesser extent by Walter Brainerd and Lawrence Landweber in *Theory of Computation*. Unfortunately both these books have been out of print for many years. In addition, these notes have benefited from my conversations with colleagues especially John Case on the subject of the Recursion Theorem.

Rather than packaging these notes as a commercial product (i.e., book), I am making them available via the World Wide Web (initially to Pitt students and after suitable debugging eventually to everyone).

Next Up Previous Contents Index

Next: Contents **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** Lecture Notes for CS 2110 Introduction to Theory

Bob Daley
2001-11-28

Contents

Next Up Previous Index

Next: 1. Introduction **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** Forward

Bob Daley

2001-11-28

Next Up Previous Contents Index
Next: 1.1 Preliminaries **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** Contents

1. Introduction

- **Goal**

  To learn the fundamental properties and limitations of computability (i.e., the ability to solve problems by computational means)
- **Major Milestones**
  - **Invariance**
    in formal descriptions of computable functions -- Church's Thesis
  - **Undecidability**
    by computer programs of any dynamic (i.e., behavioral) properties of computer programs based on their text
- **Major Topics**
  - **Models**
    of computable functions
  - **Decidable vs undecidable**
    properties
  - **Feasible vs infeasible**
    problems -- **P** $\stackrel{?}{=}$ **NP**
  - **Formal Languages**
    (i.e., languages whose sentences can be parsed by computer programs)

Next Up Previous Contents Index
Next: 1.1 Preliminaries **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** Contents

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 1.2 Representation of Objects **Up:** 1. Introduction **Previous:** 1. Introduction

1.1 Preliminaries

We will study a variety of computing devices. Conceptually we depict them as being ``black boxes" of the form

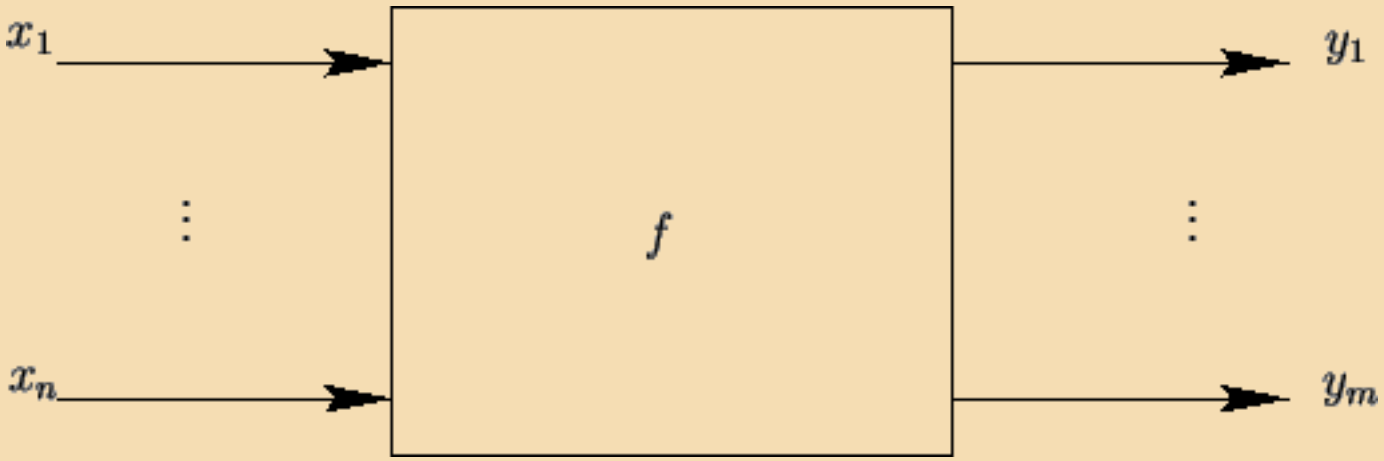
Figure 1.1: Black box computing device

where x is an input object of type X (i.e., $x \in X$) and y is an output object of type Y. Thus, at this level the device computes a function $f : X \longrightarrow Y$ defined by $f(x) = y$.

- For some computing devices the function f will be a *partial function* which means that for some inputs x the function is not defined (i.e., produces no output). In this case we write $f(x) \uparrow$. Similarly, we write $f(x) \downarrow$ whenever f on input x is defined.
- The set of all inputs on which the function f is defined is called its *domain* (denoted by **dom** f), and is given by **dom** $f = \{x : f(x) \downarrow\}$.
- Also, the *range* of a function f (denoted by **ran** f), and is given by **ran** $f = \{y : \exists x \in$ **dom** $f, y = f(x)\}$.

We will also be interested in computing devices which have multiple inputs and outputs, i.e., which can be depicted as follows:

Figure 1.2: Multiple input-output computing device



where $x_1,..., x_n$ are objects of type $X_1,..., X_n$ (i.e., $x_1 \in X_1,..., x_n \in X_n$), and $y_1,..., y_m$ are objects of type $Y_1,..., Y_m$. Thus, the device computes a function

$$f : X_1 \times \cdots \times X_n \longrightarrow Y_1 \times \cdots \times Y_m$$

defined by $f(x_1,..., x_n) = (y_1,..., y_m)$. Here we use $X_1 \times \cdots \times X_n$ to denote the cartesian product, i.e.,

$$X_1 \times \cdots \times X_n = \{(x_1,..., x_n) : x_1 \in X_1,..., x_n \in X_n\}.$$

- We also use X^n to denote the cartesian product when $X_1 = X_2 = \cdots = X_n = X$.
- Of course, since $X_1 \times \cdots \times X_n$ is just some set X and $Y_1 \times \cdots \times Y_m$ is some set Y, the situation with multiple inputs and outputs can be viewed as a more detailed description of a single input-output device where the inputs are n-tuples of elements and the outputs are m-tuples of elements.
- We use $\vec{x}_i^{\,n}$ to denote $x_i, x_{i+1},..., x_n$ where $i \leq n$, and $\vec{x}^{\,n}$ to denote $\vec{x}_1^{\,n}$ (i.e., $x_1,..., x_n$).

Besides viewing computing devices as mechanisms for *computing functions* we are also interested in them as mechanisms for *computing sets*.

- Given a set X the *characteristic function* of X (denoted by χ_X) is given by

$$\chi_X(x) = \begin{cases} 1, & \text{if } x \in X \\ 0, & \text{if } x \notin X. \end{cases}$$

- A computing device (which computes the function *f*) can ``compute" a set *X* in 3 different ways:
  1. it can compute the characteristic function χ_X of the set *X*, i.e., *f* = χ_X .
  2. its domain is equal to *X*, i.e., *X* = **dom** *f*. In this case we say that the device is an *acceptor* (or a *recognizer*) for the set *X*.
  3. its range is equal to *X*, i.e., *X* = **ran** *f*. In this case we say that the device is a *generator* for the set *X*.

Next Up Previous Contents Index

Next: 1.2 Representation of Objects **Up:** 1. Introduction **Previous:** 1. Introduction

Bob Daley

2001-11-28

1.2 Representation of Objects

We use $\mathbb{N}$ to denote the set {0, 1, 2,...} of *Natural Numbers*, and we use $\mathbb{B}$ for the set {0, 1} of *Binary Digits*. We are most interested in functions over $\mathbb{N}$, but in reality numbers are abstract objects and not concrete objects. Therefore it will be necessary to deal with representations of the natural numbers by means of strings over some alphabet.

- An *alphabet* Σ is any finite set of symbols $\{\sigma_1, ..., \sigma_n\}$. The symbols themselves will be unimportant, so we will use 1 for σ_1, ..., and n for σ_n, and denote by Σ_n the set {1,..., n}.
- A *word* over the alphabet Σ is any finite string $a_1 \cdots a_j$ of symbols from Σ (i.e., $x = a_1 \cdots a_j$). We denote by Σ^* the set of *all words* over the alphabet Σ.
- The *length* of a word $x = a_1 \cdots a_j$ (denoted by $| x |$) is the number j of symbols contained in x.
- The *null* or *empty* word (denoted by ε) is the (unique) word of length 0.
- Given two words $x = a_1 \cdots a_j$ and $y = b_1 \cdots b_k$, the *concatenation* of x and y (denoted by $x \cdot y$) is the word $a_1 \cdots a_j b_1 \cdots b_k$. Clearly, $| x \cdot y | = | x | + | y |$. We will often omit the $\cdot$ symbol in the concatenation of x and y and simply write xy.
- The word x is called an *initial segment* (or *prefix*) of the word y if there is some word z such that $y = x \cdot z$.
- For any symbol $a \in \Sigma$, we use a^m to denote the word of length m consisting of m a's.
- We often refer to a set of strings over an alphabet Σ as a *language*.
- We extend concatenation to sets of strings over an alphabet Σ as follows:
  If $X, Y \subseteq \Sigma^*$, then

$$X \cdot Y = \{x \cdot y : x \in X \quad \text{and} \quad y \in Y\}$$

$$X^{(0)} = \{\varepsilon\}$$

$$X^{(n+1)} = X^{(n)} \cdot X, \qquad \text{for} \quad n \geq 0$$

$$X^* = \bigcup_{n=0}^{\infty} X^{(n)}$$

$$X^+ = \bigcup_{n=1}^{\infty} X^{(n)}$$

Thus $X^{(n)}$is the set of all ``words" of length $n$over the ``alphabet" X.

Next Up Previous Contents Index

Next: 1.3 Codings for the Natural Numbers **Up:** 1. Introduction **Previous:** 1.1 Preliminaries

Bob Daley

2001-11-28

1.3 Codings for the Natural Numbers

We will introduce a correspondence between the natural numbers and strings over Σ_n^* which is different from the usual number systems such as binary and decimal representations.

Table 1.1: Codings for the Nautral Numbers

$\mathbb{N}$	$\mathbb{B}^*$	Σ_2^*	Σ_4^*
0	0	ε	ε
1	1	1	1
2	10	2	2
3	11	11	3
4	100	12	4
5	101	21	11
6	110	22	12
7	111	111	13
8	1000	112	14
9	1001	121	21
10	1010	122	22
11	1011	211	23

The codings via Σ_2^* and Σ_4^* are one-to-one and onto. The coding via $\mathbb{B}^*$ is not -- 010 = 10.

- **The function** $\kappa_n : \mathbb{N} \longrightarrow \Sigma_n^*$,

  providing the one-to-one and onto map is defined inductively as follows:

$$\kappa_n(0) = \varepsilon$$

Next, suppose that $\kappa_n(x) = d_1 \cdots d_j$, and let $k \leq j$ be the *greatest* integer such that $d_k \neq n$ (so $k = 0$ if $d_1 = \cdots = d_j = n$). Then,

$$\kappa_n(x+1) = \begin{cases} d_1 \cdots d_{k-1}(d_k+1)1^{j-k}, & \text{if } k > 0, \\ 1^{j+1}, & \text{otherwise.} \end{cases}$$

- **The function** $\nu_n : \Sigma_n^* \longrightarrow \mathbb{N}$,

  which is the inverse for κ_n is defined as follows:

  Let $x \in \Sigma_n^*$ be the string $a_j \cdots a_1 a_0$. Then,

$$\begin{aligned} \nu_n(x) &= \nu_n(a_j \cdots a_1 a_0) \\ &= \sum_{i=0}^{j} a_i \times n^i \\ &= a_j \times n^j + \cdots + a_1 \times n + a_0 \end{aligned}$$

- **Observe that**

  $\nu_n(x \cdot y) = \nu_n(x) \times n^{|y|} + \nu_n(y)$,

  e.g., $16 = 3 \times 2^2 + 4 = \nu_2(11 \cdot 12) = \nu_2(1112)$.

Next Up Previous Contents Index

Next: 1.4 Inductive Definition and Proofs **Up:** 1. Introduction **Previous:** 1.2 Representation of Objects

Bob Daley
2001-11-28

1.4 Inductive Definition and Proofs

An inductive definition over the natural numbers $\mathbb{N}$ usually takes the form:

$$f(0, y) = g(y)$$
$$f(n + 1, y) = h(n, y, f(n, y))$$

where g and h are previously defined.

- **Example of inductive definition**

$$y^0 = 1$$
$$y^{n+1} = y^n \times y$$

so that $g(y) = 1$ and $h(x, y, z) = z \times y$.

- **Definitions involving ``...'' are usually inductive.**

- **Example of ... definition**

$$\sum_{i=0}^{n} a_i = a_0 + a_1 + \cdots + a_n$$

The inductive equivalent is:

$$\sum_{i=0}^{0} a_i = a_0$$

$$\sum_{i=0}^{n+1} a_i = \sum_{i=0}^{n} a_i + a_{n+1}$$

so that $g(y) = a_0$ and $h(x, y, z) = z + a_{x+1}$.

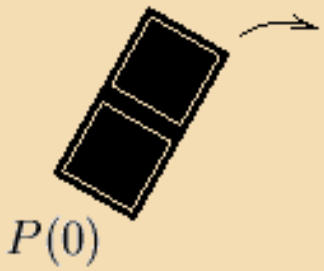
Most ``recursive'' procedures are really just inductive definitions.

Induction Principle I: For any proposition P over $\mathbb{N}$, if
1) $P(0)$ is true, and
2) $\forall n, P(n) \rightarrow P(n+1)$ is true,
then $\forall n, P(n)$ is true.

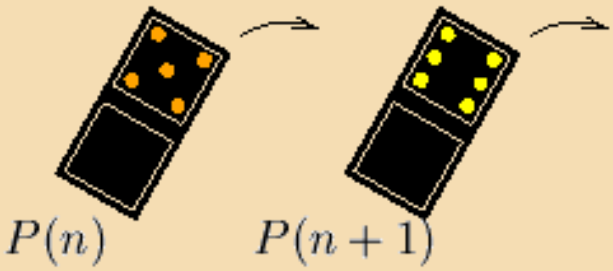
1) is called the *Basis Step*
2) is called the *Induction Step*

The validity of this principle follows by a ``Dominoe Principle''

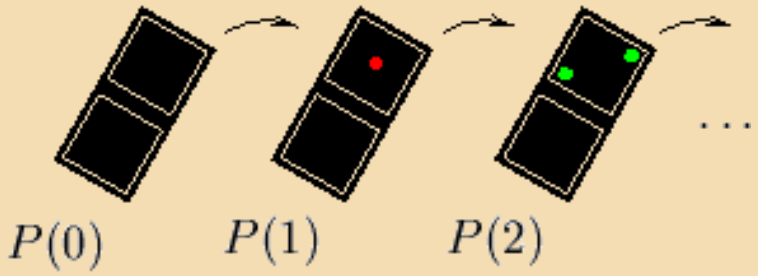
- ***P*(0) means ``0 falls'':**



- ***P*(*n*) $\rightarrow$ *P*(*n* + 1) means ``if *n* falls, then *n* + 1 falls'':**



Combining these two parts, we see that ``all dominoes fall'':



- **Example of inductive proof**

Let $P(n)$: $\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}$.

- **Basis Step:**

  Show $P(0)$ is true

$$\sum_{i=0}^{0} i = 0 = \frac{0 \times (0+1)}{2}$$

- **Induction Step:**

  Let n be arbitrary and assume $P(n)$ is true. This assumption is called the *Induction Hypothesis*, viz. that

$$\sum_{i=0}^{n} i = \frac{n \times (n+1)}{2}$$

Then,

$$\sum_{i=0}^{n+1} i = \sum_{i=0}^{n} i + (n+1)$$

$$= \frac{n \times (n+1)}{2} + (n+1)$$

$$= \frac{n \times (n+1) + 2 \times (n+1)}{2}$$

$$= \frac{(n+1) \times (n+2)}{2}$$

Note: Line 1 uses the inductive definition of $\sum$ (here $a_i = i$).

Line 2 uses the Induction Hypothesis; and

Line 4 is $P(n+1)$, so we have shown $P(n) \rightarrow P(n+1)$.

By reasoning similar to that for Induction Principle I, we also have

Induction Principle II: For any proposition P over the positive integers, if
1) $P(0)$ is true, and
2) $\forall n, (\forall i < n+1, P(i)) \rightarrow P(n+1)$ is true,
then $\forall n, P(n)$ is true.

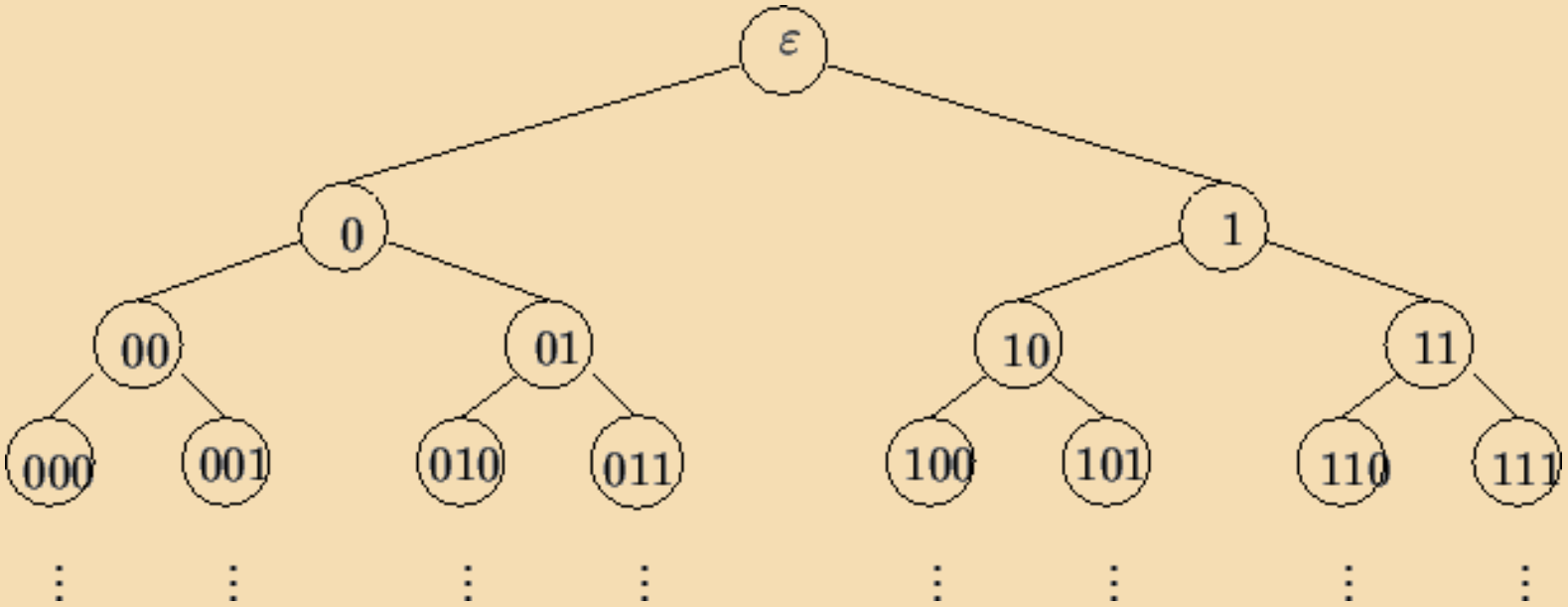
Here 2) means ``If 0, 1, 2,..., $n$ falls, then $n+1$ falls". Note that `` $\forall i < n+1, P(i)$" is really shorthand for `` $\forall i, i < n+1 \rightarrow P(i)$".

Induction Principle II is needed for inductive definitions like the one for the fibonacci numbers:

$$f(0) = 0$$
$$f(1) = 1$$
$$f(n+1) = f(n) + f(n-1)$$

However, some domains of interest **do not** have such a ``linear" structure as the natural numbers. For example, the set $\mathbb{B}^*$ has a ``tree" structure:

Figure 1.3: Structure of $\mathbb{B}^*$



Thus each word $x \in \mathbb{B}^*$ has *two* successors: $x \cdot 0$ and $x \cdot 1$.

- **Example of inductive definition over Σ^***

The reversal function ρ such that $\rho(a_1 \cdots a_n) = a_n \cdots a_1$ is defined inductively by:

$$\rho(\varepsilon) = \varepsilon$$

$$\rho(x \cdot a) = a \cdot \rho(x)$$

Thus, we see that inductive defintions over Σ^* have the general form:

$$f(\varepsilon, y) = g(y)$$

$$f(x \cdot a, y) = h_a(x, y, f(x, y)), \qquad \text{for each} \quad a \in \Sigma$$

Principle III: For any proposition P over Σ^*, if

1) $P(\varepsilon)$ is true, and

2) $\forall x \in \Sigma^*$,($\forall a \in \Sigma$, $P(x) \rightarrow P(x \cdot a)$) is true,

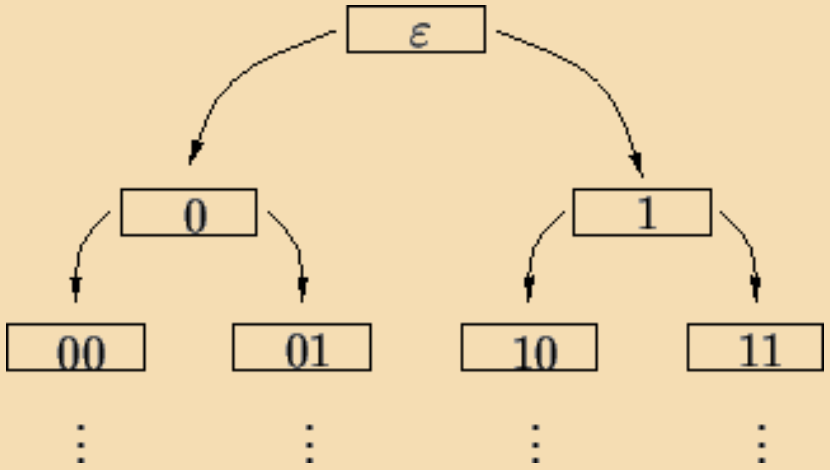
then $\forall x \in \Sigma^*$, $P(x)$ is true.

The validity of this principle also follows from a ``Dominoe Principle"

- **$P(\varepsilon)$ means `` ε falls'':**
- **$P(x) \rightarrow P(x \cdot a)$ means ``if x falls, then $x \cdot a$ falls'':**

These combined yield ``all dominoes fall" when they are arranged according to the structure of Σ^*.

Figure 1.4: Top view



- **Example of inductive proof over Σ^***

  Let $P(x) \equiv \forall a \in \Sigma, \rho(a \cdot x) = \rho(x) \cdot a$

- **Basis Step:**

  Show $P(\varepsilon)$ is true

$$\rho(a \cdot \varepsilon) = \rho(a) = \rho(\varepsilon \cdot a) = a \cdot \rho(\varepsilon) = a \cdot \varepsilon = a = \varepsilon \cdot a = \rho(\varepsilon) \cdot a$$

- **Induction Step:**

  Let x be arbitrary and assume $P(x)$ is true, so the *Induction Hypothesis* is

$$\forall a \in \Sigma, \rho(a \cdot x) = \rho(x) \cdot a$$

Then, for any $a, b \in \Sigma$

$$\begin{aligned}
\rho(a \cdot (x \cdot b)) &= \rho((a \cdot x) \cdot b) \\
&= b \cdot \rho(a \cdot x) \\
&= b \cdot (\rho(x) \cdot a) \\
&= (b \cdot \rho(x)) \cdot a \\
&= \rho(x \cdot b) \cdot a
\end{aligned}$$

So $P(x \cdot a)$ is true. Therefore, $\forall x \in \Sigma^*, \forall a \in \Sigma, \rho(a \cdot x) = \rho(x) \cdot a$.

Note: Lines 1 and 4 use the associativity of $\cdot$;

Lines 2 and 5 use the definition of ρ;

and Line 3 use the Induction Hypothesis.

Next Up Previous Contents Index

Next: 2. Models of Computation **Up:** 1. Introduction **Previous:** 1.3 Codings for the Natural Numbers

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 2.1 Memoryless Computing Devices **Up:** Lecture Notes for CS 2110 Introduction to Theory
Previous: 1.4 Inductive Definition and Proofs

2. Models of Computation

- **Memoryless Computing Devices**
  - **Boolean functions and Expressions**
  - **Digital Circuits**
  - **Propositional Logic**
- **Finite Memory Computing Devices**
  - **Finite state machines**
  - **Regular expressions**
- **Unbounded Memory Devices**
  - **Loop programs**
  - **(Partial) recursive functions**
  - **Random access machines**
  - **First-order number theory**
- **Other Aspects**
  - **Non-deterministic devices**
  - **Probabilistic devices**

Next Up Previous Contents Index

Next: 2.1 Memoryless Computing Devices **Up:** Lecture Notes for CS 2110 Introduction to Theory

Previous: 1.4 Inductive Definition and Proofs

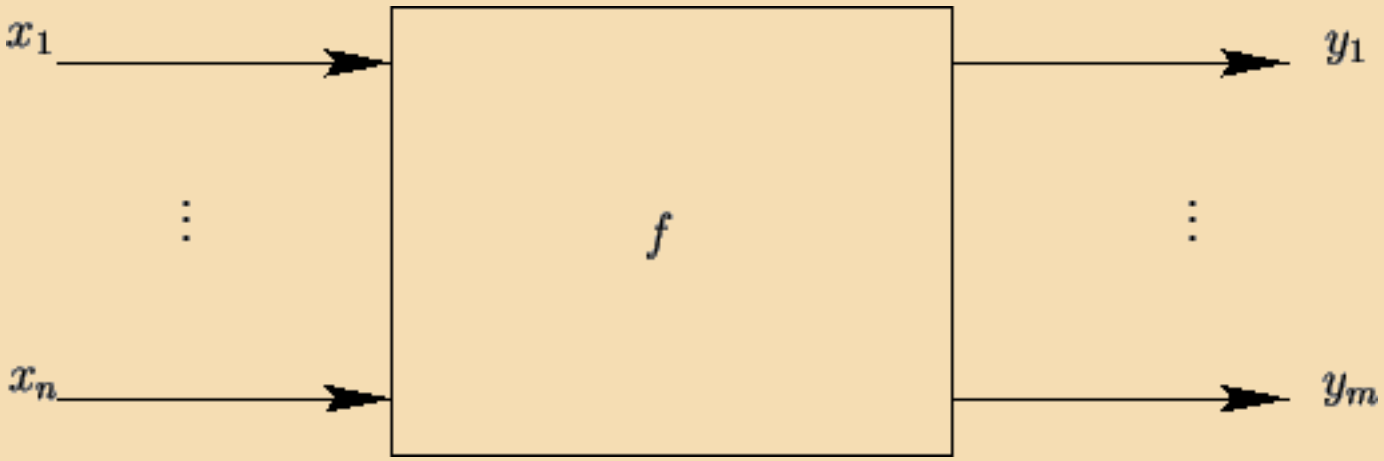
Bob Daley

2001-11-28

2.1 Memoryless Computing Devices

A *boolean function* is any function $f: \mathbb{B}^n \longrightarrow \mathbb{B}^m$, and thus has the schematic form

Figure 2.1: Multiple input-output computing device



We will be concerned here primarily with the case where $m = 1$. Since $\mathbb{B}$ has finite cardinality, the domain of f is finite, and f can be represented by means of a finite table with 2^n entries.

Example 2.1

Table 2.1: Example boolean function

x_1	x_2	x_3	f
0	0	0	1
0	0	1	0
0	1	0	0
0	1	1	0
1	0	0	0
1	0	1	1

1	1	0	1
1	1	1	1

It is also possible to represent a boolean function by means of a *boolean expression.* A boolean expression consists of *boolean variables* (x_1, x_2,...), *boolean constants* (0 and 1), and *boolean operations* ($\neg$, $\vee$, and $\wedge$), and is defined inductively as follows:

1. Any boolean variable x_1, x_2,... and any boolean constant 0, 1 is a boolean expression;
2. If e_1 and e_2 are boolean expressions, then so are ($\neg e_1$), ($e_1 \vee e_2$), and ($e_1 \wedge e_2$).

The operations $\neg$, $\vee$, $\wedge$ are defined by the table:

Table 2.2: Boolean operations

x_1	x_2	$\neg x_1$	$x_1 \vee x_2$	$x_1 \wedge x_2$
0	0	1	0	0
0	1		1	0
1	0	0	1	0
1	1		1	1

so that $\neg$, $\vee$, $\wedge$ represent boolean functions. In general, every boolean expression with n variables represents some boolean function $f: \mathbb{B}^n \longrightarrow \mathbb{B}$.

Conversely, we have

Theorem 2.1 Every boolean function $f: \mathbb{B}^n \longrightarrow \mathbb{B}$ is represented by some boolean expression with n variables.

Example 2.2

The function given in Example 2.1 above can be represented by the boolean expression

$$(\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2),$$

i.e.,

$$f(x_1, x_2, x_3) = (\neg x_1 \wedge \neg x_2 \wedge \neg x_3) \vee (x_1 \wedge \neg x_2 \wedge x_3) \vee (x_1 \wedge x_2).$$

Terminology:

- A *literal* is either a variable (e.g., x_j) or its negation (e.g., $\neg x_j$).
- A *term* is a conjunction (i.e., $e_1 \wedge \cdots \wedge e_k$) of literals $e_1,\ldots, e_k$.
- A *clause* is a disjunction (i.e., $e_1 \vee \cdots \vee e_k$) of literals $e_1,\ldots, e_k$.
- A boolean expression is a *DNF* (*disjunctive normal form*) expression if it is a disjunction of terms.
- A *monomial* is a one-term *DNF* expression.
- A boolean expression is a *CNF* (*conjunctive normal form*) expression if it is a conjunction of clauses.

The previous theorem is proved by constructing a *DNF* expresssion for any given boolean function.

Next Up Previous Contents Index

Next: 2.2 Digital Circuits **Up:** 2. Models of Computation **Previous:** 2. Models of Computation

Bob Daley

2001-11-28

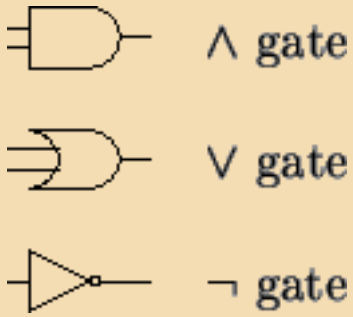
Next Up Previous Contents Index

Next: 2.3 Propositional Logic **Up:** 2. Models of Computation **Previous:** 2.1 Memoryless Computing Devices

2.2 Digital Circuits

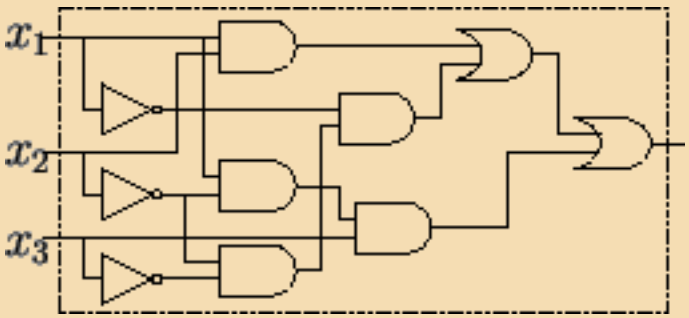
We can ``implement" boolean functions using digital logic circuits consisting of ``gates" which compute the operations $\neg$, $\vee$, and $\wedge$, and which are depicted as follows:

Figure 2.2: Digital logic gates



Example 2.3 (circuit for function of Example 2.1)

Figure 2.3: Digital logic circuit



Next Up Previous Contents Index

Next: 2.3 Propositional Logic **Up:** 2. Models of Computation **Previous:** 2.1 Memoryless Computing Devices

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 2.4 Finite Memory Devices **Up:** 2. Models of Computation **Previous:** 2.2 Digital Circuits

2.3 Propositional Logic

If we interpret the boolean value 0 as ``FALSE'' ( **F**) and the boolean value 1 as ``TRUE'' (**T**), then the boolean operations become ``logical operations'' which are defined by the following ``truth tables'':

Table 2.3: Logical operations

x_1	x_2	$\neg x_1$	$x_1 \vee x_2$	$x_1 \wedge x_2$
F	*F*	*T*	*F*	*F*
F	*T*		*T*	*F*
T	*F*	*F*	*T*	*F*
T	*T*		*T*	*T*

Then the boolean variables become ``logical variables'', which take on values from the set $V = \{\mathbf{T},\mathbf{F}\}$. Analagously, boolean expressions become ``logical expressions'' (or ``propositional sentences''), and are useful in describing concepts.

Example 2.4 Suppose $x_1, x_2, x_3, x_4, x_5, x_6$ are propositional variables which are interpreted as follows:

x_1 -- "is a large mammal"

x_2 -- "lives in water"

x_3 -- "has claws"

x_4 -- "has stripes"

x_5 -- "hibernates"

x_6 -- "has mane"

Then the propositional statement $x_1 \wedge \neg x_2 \wedge (x_4 \vee x_5 \vee x_6)$ defines a concept for a class of animals which inclues lions and tigers and bears!

Next Up Previous Contents Index

Next: 2.4 Finite Memory Devices **Up:** 2. Models of Computation **Previous:** 2.2 Digital Circuits

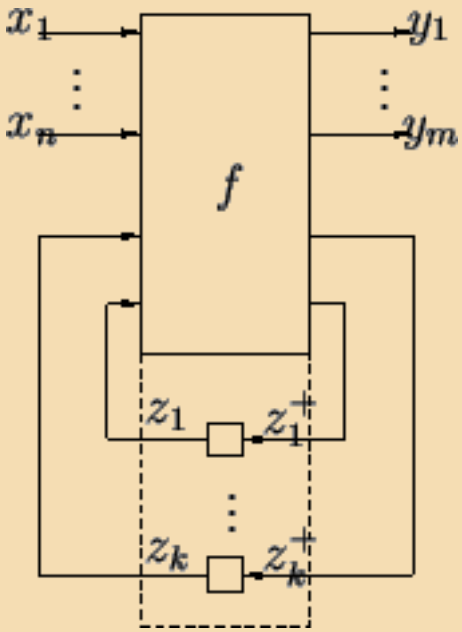
Bob Daley

2001-11-28

2.4 Finite Memory Devices

We construct finite memory devices be adding a finite number of memory cells (``flip-flops''), which can store a single bit (0 or 1), to a logical circuit as depicted below:

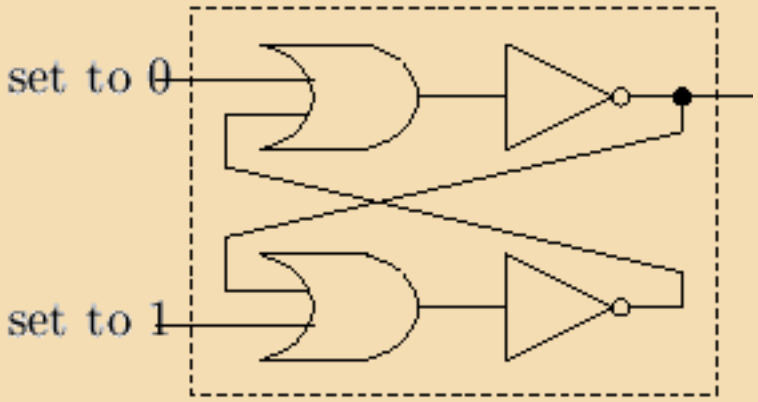
Figure 2.4: Finite memory device



Here, z_i is the current contents of memory cell *i*, and z_i^+ is the contents of that memory cell at the next unit of time (i.e., clock cycle).

Of course, memory cells themselves can be realized by digital circuits, e.g., the following cicuit realizes a flip-flop:

Figure 2.5: Flip Flop



The device operates as follows: At each time step, the current input values $x_1,..., x_n$ are combined with the current memory values $z_1,..., z_k$ to produce via the logical circuit the output values $y_1,..., y_m$ and memory values $z_1^+,..., z_k^+$ for the *next* time cycle. Then, the device uses the next input combination of $x_1,..., x_n$ and $z_1,..., z_k$ (i.e., the previously calculated $z_1^+,..., z_k^+$) to compute the next output $y_1,..., y_m$ and the next memory contents $z_1^+,..., z_k^+$, and so on.

Of course, at the beginning of the computation there must be some initial memory values. In this way we see that such a device transforms a *string* of inputs (i.e., a word over $\mathbb{B}^*$) into a *string* of outputs.

A device that has *k* memory cells will have 2^k combinations of memory values or *states* . Of course, depending on the circuitry, not all combinations will be realizable, so the device may have fewer actual states.

We formalize matters as follows:

- We regard the pattern of bits $x_1,..., x_n$ as encoding the letters of some *input alphabet* Σ, and similarly $y_1,..., y_m$ as encoding the letters of some *output alphabet* Γ.
- We let *Q* denote the set of possible states (i.e., legal combinations of $z_1,..., z_k$).

As indicated above Σ, Γ, and *Q* need not have cardinality that is a power of 2.

- Since the output ($y_1,..., y_m$) depends on the input ($x_1,..., x_n$) and the current memory state ($z_1,... z_k$), we have an *output function* $\lambda : Q \times \Sigma \longrightarrow \Gamma$.
- Similarly, since the next memory state ($z_1^+,...z_k^+$) depends on the input and the current memory

state, we have a *state transition function* $\delta : Q \times \Sigma \longrightarrow Q$.

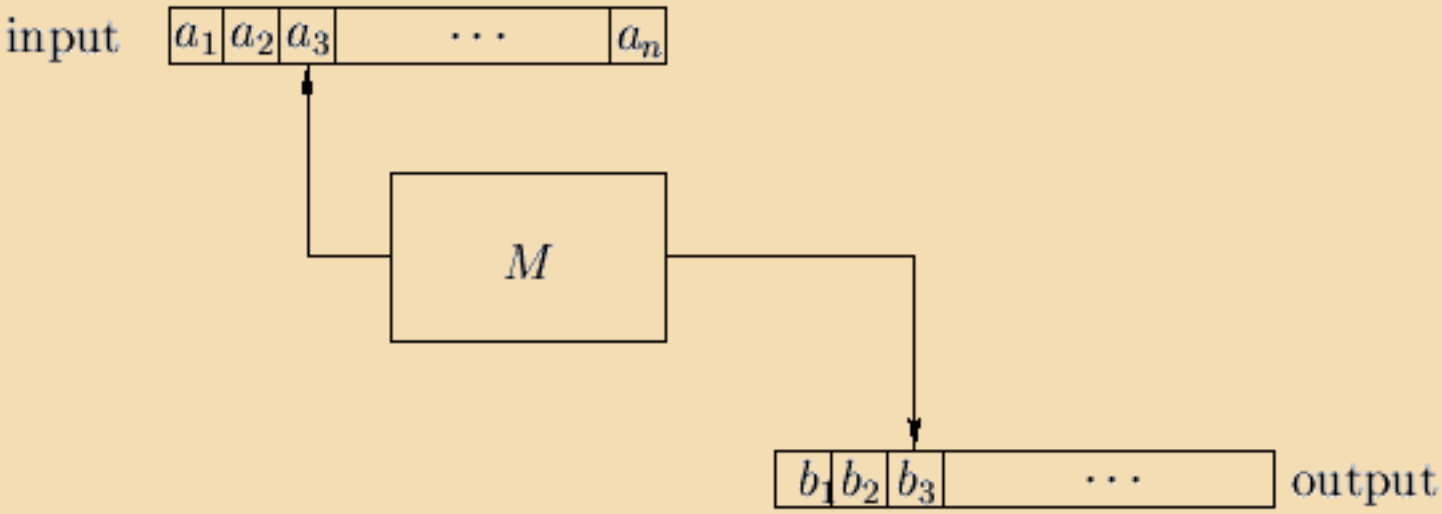
- When the device begins its computation on a given input its memory will be in some *initial state* q_0.

Therefore, such a device can be abbreviated as a tuple

$$M = \langle \Sigma, \Gamma, Q, \delta, \lambda, q_0 \rangle .$$

We depict *M* schematically as follows:

Figure 2.6: Schematic for Finite State Automaton



While this model of a finite memory device clearly models the computation of functions $f : \Sigma^* \longrightarrow \Gamma^*$ with finite memory, we need only consider a restricted form which are acceptors for languages over Σ^* (i.e., subsets of strings from Σ^*). In this restricted model we replace the output function λ by a set of specially designated states $F \subseteq Q$ called *final states* . The purpose of *F* is to indicate which input words are accepted by the device.

Definition 2.1 A *deterministic finite state automation (DFA)* is a 5-tuple

$$M = \langle \Sigma, Q, \delta, q_0, F \rangle,$$

where Σ is the *input alphabet*, Q is the finite set of *states*, q_0 is the *initial state*, $F \subseteq Q$ is the set of *final states*, and $\delta : Q \times \Sigma \longrightarrow Q$ is the *state transition function*.

- We say that an input $x = a_1 \cdots a_j$ is *accepted* by the *DFA* $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ if there is a sequence of states $p_1,..., p_{j+1}$ such that p_1 is the initial state q_0 and $p_{j+1} \in F$ and for each $i \leq j$, $\delta(p_i, a_i) = p_{i+1}$.
- We say that a language $X \subseteq \Sigma^*$ is *accepted* by the *DFA* M if and only if every word $x \in X$ is accepted by M.

Next Up Previous Contents Index

Next: 2.5 Regular Languages **Up:** 2. Models of Computation **Previous:** 2.3 Propositional Logic

Bob Daley

2001-11-28

Next Up Previous Contents Index
Next: 3. Loop Programs **Up:** 2. Models of Computation **Previous:** 2.4 Finite Memory Devices

2.5 Regular Languages

The class of *regular languages* over Σ^* is defined by induction as follows:

1. the sets $\emptyset$, $\{\varepsilon\}$, and $\{a\}$ for each $a \in \Sigma$ are regular languages;
2. if R_1 and R_2 are regular languages, then so are $R_1 \cup R_2$, $R_1 \cdot R_2$, and R_1^*.

In other words, the class of regular languages is the smallest class of subsets of Σ^* containing $\emptyset$, $\{\varepsilon\}$, and $\{a\}$ for each $a \in \Sigma$, and closed under the operations of set union, set concatenation, and *.

We define the class of *regular expressions* for denoting regular sets by induction as follows:

1. $\emptyset$, ε, and a are regular expressions for $\emptyset$, $\{\varepsilon\}$, and $\{a\}$, respectively;
2. if r_1 and r_2 are regular expressions for the regular sets R_1 and R_2, then $(r_1 \cup r_2)$, $(r_1 \cdot r_2)$, and (r_1^*) are regular expressions for $R_1 \cup R_2$, $R_1 \cdot R_2$, and R_1^*, respectively.

Theorem 2.2 Every regular language is accepted by some deterministc finite automaton, and conversely every language accepted by some deterministic finite automaton is a regular language.

Next Up Previous Contents Index
Next: 3. Loop Programs **Up:** 2. Models of Computation **Previous:** 2.4 Finite Memory Devices

Bob Daley
2001-11-28

3. Loop Programs

The programming language *LOOP* over Σ_k^* consists of:

- **Program Variables:**

  $\mathbf{X}_1, \mathbf{X}_2, \mathbf{X}_3, \cdots$ (also **U**,**V**,**W**,**Y**,**Z** with subscripts)

- **Elementary Statements:**

  - **Input Statements:**

    INPUT($\mathbf{X}_1, \ldots, \mathbf{X}_n$)

  - **Output Statements:**

    OUTPUT($\mathbf{Y}_1$)

  - **Assignment Statements:**

    - $\mathbf{X}_1 \leftarrow \mathbf{0}$
    - $\mathbf{X}_1 \leftarrow \mathbf{X}_1 + \mathbf{1}$
    - $\mathbf{X}_1 \leftarrow \mathbf{Y}_1$

- **Control Structures:**

  - **For Statements:**

FOR X_1 **TIMES DO**

.
.
.

ENDFOR

- **Until Statements:**

UNTIL X_1 **TRUE DO**

.
.
.

ENDUNTIL

Bob Daley

2001-11-28

3.1 Semantics of *LOOP* Programs

$[X_1]$ denotes the contents of the variable $\mathbf{X}_1$.

- **logical values:**
  - ***FALSE***
    is zero
  - ***TRUE***
    is any non-zero value
- **INPUT(**$\mathbf{X}_1, \ldots, \mathbf{X}_n$**)**

  -- input $[X_1]$,...,$[X_n]$
- **OUTPUT(**$\mathbf{Y}_1$**)**

  -- output $[Y_1]$
- $\mathbf{X}_1 \leftarrow \mathbf{0}$

  -- replace $[X_1]$ with ε
- $\mathbf{X}_1 \leftarrow \mathbf{X}_1 + \mathbf{1}$

  -- replace $[X_1]$ with x, where $\nu_k(x) = \nu_k([X_1]) + 1$
- $\mathbf{X}_1 \leftarrow \mathbf{Y}_1$

  -- replace $[X_1]$ with $[Y_1]$

- **For Statement:**

  FOR $\mathbf{X}_1$ **TIMES DO**

body 

ENDFOR

-- repeat body of loop $\nu_k([X_1])$ times

- **Until Statement:**

UNTIL $\mathbf{X_1}$ **TRUE DO**

body 

ENDUNTIL

-- repeat body of loop until $[X_1] \neq \varepsilon$

Definition 3.1 A *LOOP*-program over Σ_k^* is a sequence of *LOOP* statements S_1,..., S_n such that

1. S_1 is an input statement
2. S_n is an output statement
3. and none of S_2, ..., S_{n-1} are input or output statements.

Definition 3.2 A *LOOP*-program P over Σ_k^* *computes* the (partial) function $f : (\Sigma_k^*)^n \longrightarrow \Sigma_k^*$ if and only if

1. the input statement of P has n variables;
2.

for all $x_1,..., x_n$, when P is executed † with $x_1,..., x_n$ as its input,

(a)

P halts if and only if $f(x_1,..., x_n)$ ↓,

(b)

if P halts, then P outputs $f(x_1,..., x_n)$.

† Execution of a *LOOP* program involves:

1. initially all variables have value 0
2. statements are executed according to the ``obvious'' semantics in the ``obvious'' order.

♠ Observe that the choice of alphabet Σ_k enters into consideration only through I/O and the ``internal representation'' or ``semantics'' of the program. We could have taken as our primitive operation $\mathbf{X}_1 \cdot \mathbf{a}$ (for each $a \in \Sigma_k$ instead of $\mathbf{X}_1 + \mathbf{1}$ and then the choice of Σ_k would have been much more evident.

Example 3.1 The following program computes the function $f(x) = x \dot{-} 1$, where the operation $\dot{-}$ (called ``monus'') is defined by:

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

INPUT($\mathbf{X}_1$**)**

FOR $\mathbf{X}_1$ **TIMES DO**

$\mathbf{Z}_1 \leftarrow \mathbf{Y}_1$

$\mathbf{Y}_1 \leftarrow \mathbf{Y}_1 + \mathbf{1}$

ENDFOR
OUTPUT($\mathbf{Z}_1$)

Notation 3.3 Let *P* be a *LOOP*-program with input statement **INPUT**($\mathbf{X}_1, \ldots, \mathbf{X}_n$) and output statement **OUTPUT**($\mathbf{Y}_1$). We denote by P^- the result of removing from *P* its input and output statements, and we denote by $\mathbf{U}_1 \leftarrow \mathbf{P}(\mathbf{V}_1, \ldots, \mathbf{V}_n)$ the sequence of statements:

$\mathbf{X}_1 \leftarrow \mathbf{V}_1$

.
.
.

$\mathbf{X}_n \leftarrow \mathbf{V}_n$

$\mathbf{P}^-$

$\mathbf{U}_1 \leftarrow \mathbf{Y}_1$

We can implement other control structures using **FOR** and **UNTIL** loops. First, we need a program *BLV* for the function *blv* (``boolean / logical value'') define by:

$$blv(x) = \begin{cases} 1, & \text{if } x > 0, \\ 0, & \text{otherwise.} \end{cases}$$

and we need a program *NEG* for the function *neg* (``logical negation'') defined by:

$$neg(x) = \begin{cases} 0, & \text{if } x > 0, \\ 1, & \text{otherwise.} \end{cases}$$

The program *BLV* is given by

INPUT($\mathbf{X_1}$ **)**

$\mathbf{Z_1} \leftarrow 0$

FOR $\mathbf{X_1}$ **TIMES DO**

$\mathbf{Z_1} \leftarrow 0$

$\mathbf{Z_1} \leftarrow \mathbf{Z_1} + \mathbf{1}$

ENDFOR
OUTPUT($\mathbf{Z_1}$**)**

and the program *NEG* is given by:

INPUT($\mathbf{X_1}$ **)**

$\mathbf{X_1} \leftarrow \mathbf{BLV}(\mathbf{X_1})$

$\mathbf{Z_2} \leftarrow \mathbf{Z_2} + \mathbf{1}$

FOR $\mathbf{X_1}$ **TIMES DO**

$\mathbf{Z_2} \leftarrow 0$

ENDFOR
OUTPUT($\mathbf{Z_2}$**)**

Then the if-then-else control structure, that takes the form

IF $\mathbf{X_1}$ **TRUE THEN**

S_1

ELSE

S_2

ENDIF

where S_1 and S_2 stand for *lists* of statements, can be implemented by:

$\mathbf{X_2 \leftarrow BLV(X_1)}$

FOR $\mathbf{X_2}$ **TIMES DO**

S_1

ENDFOR

$\mathbf{X_2 \leftarrow NEG(X_2)}$

FOR $\mathbf{X_2}$ **TIMES DO**

S_2

ENDFOR

Next Up Previous Contents Index

Next: 3.2 Other Aspects **Up:** 3. Loop Programs **Previous:** 3. Loop Programs

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 3.3 Complexity of LOOP Programs **Up:** 3. Loop Programs **Previous:** 3.1 Semantics of LOOP Programs

3.2 Other Aspects

- We can construct *non-deterministic LOOP* programs by adding statements of the form

  SELECT($\mathbf{X}_1$ **)**

  which assigns either a 0 or a 1 non-deterministically to the variable $\mathbf{X}_1$.
- We can construct *probabilistic LOOP* programs by adding statements of the form

  PRASSIGN($\mathbf{X}_1$ **)**

  which assigns either a 0 or a 1 probabilistically with probability $\frac{1}{2}$ to the variable $\mathbf{X}_1$.

We distinguish between deterministic, non-deterministic, and probabilistic *LOOP* programs by using the notation *DLOOP*, *NLOOP*, and *PLOOP*, respectively.

Next Up Previous Contents Index

Next: 3.3 Complexity of LOOP Programs **Up:** 3. Loop Programs **Previous:** 3.1 Semantics of LOOP Programs

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 4. Primitive Recursive Functions **Up:** 3. Loop Programs **Previous:** 3.2 Other Aspects

3.3 Complexity of LOOP Programs

Definition 3.4 If P is a deterministic *LOOP* program (a program without **SELECT** or **PRASSIGN** statements) over Σ_k^* with input variables $\mathbf{X}_1, \ldots, \mathbf{X}_n$ and all variables included in $\mathbf{X}_1, \ldots, \mathbf{X}_r$, then we define the following complexity measures for P.

$$DLPtime_P(\vec{x}^n) = \begin{cases} \sum_{i=1}^{n} \mid x_i \mid + \#\text{ of stmts of } P \text{ executed on input } \vec{x}^n, \\ \text{if } P \text{ halts on it,} \\ \uparrow, \text{ otherwise.} \end{cases}$$

$$DLPspace_P(\vec{x}^n) = \begin{cases} \max \sum_{i=1}^{r} \mid \mathbf{X}_i^t \mid, \forall t \leq DLPtime_P(\vec{x}^n), \text{ if } P \text{ halts,} \\ \uparrow, \text{ otherwise.} \end{cases}$$

where $\mathbf{X}_i^t$ denotes the contents of register $\mathbf{X}_i$ at step t of the computation of P on input $\vec{x}^n$.

Next Up Previous Contents Index

Next: 4. Primitive Recursive Functions **Up:** 3. Loop Programs **Previous:** 3.2 Other Aspects

Bob Daley

2001-11-28

4. Primitive Recursive Functions

The class of *primitive recursive functions* is defined inductively as follows:

- **Base functions:**
  - **Null function:**

    $N(x) = 0$, for any $x \in \mathbb{N}$
  - **Successor function:**

    $S(x) = x + 1$, for any $x \in \mathbb{N}$
  - **Projection functions:**

    $P_j^n(\vec{x}^n) = x_j$, for any $1 \leq j \leq n$, and any $\vec{x}^n \in \mathbb{N}^n$
- **Operations:**
  - **Substitution:**

    Given integers m and n, and functions $g : \mathbb{N}^m \longrightarrow \mathbb{N}$, and $h_1,..., h_m$, where $h_j : \mathbb{N}^n \longrightarrow \mathbb{N}$, then $f : \mathbb{N}^n \longrightarrow \mathbb{N}$ is defined from $g, h_1,..., h_m$ via *substitution* if for any $\vec{x}^n \in \mathbb{N}^n$,

    $$f(\vec{x}^n) = g(h_1(\vec{x}^n),..., h_m(\vec{x}^n)).$$
  - **Primitive recursion:**

Given an integer n, and functions $g : \mathbb{N}^{n-1} \longrightarrow \mathbb{N}$, and $h : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$, then $f : \mathbb{N}^{n} \longrightarrow \mathbb{N}$ is defined from g and h via *primitive recursion* if for any $y \in \mathbb{N}$ and any $\vec{x}_{2^n} \in \mathbb{N}^{n-1}$,

$$f(0, \vec{x}_{2^n}) = g(\vec{x}_{2^n})$$

$$f(y+1, \vec{x}_{2^n}) = h(y, f(y, \vec{x}_{2^n}), \vec{x}_{2^n}).$$

Definition 4.1 A function $f : \mathbb{N}^{n} \longrightarrow \mathbb{N}$ is *primitive recursive* if it can be obtained from the base functions (null, successor, and projections) by finitely many applications of the operations of substitution and primitive recursion.

- Thus, the class of primitive recursive functions is the smallest class containing the base functions and closed under the operations of substitution and primitive recursion.
- If in the definition of primitive recursion $n = 1$, then the schema takes the form:

$$f(0) = c$$
$$f(y+1) = h(y, f(y))$$

for some constant $c$and some function h.

- We could have defined the primitive recursive functions over Σ_k^* instead of $\mathbb{N}$ by replacing S with k successors $S_a(y) = y \cdot a$ for each $a \in \Sigma_k$; and by replacing primitive recursion over $\mathbb{N}$ with primitive recursion over Σ_k^* which takes the form:

$$f(\varepsilon, \vec{x}_{2^n}) = g(\vec{x}_{2^n})$$

$$f(y \cdot a, \vec{x}_{2^n}) = h_a(y, f(y, \vec{x}_{2^n}), \vec{x}_{2^n}) \qquad \forall a \in \Sigma_k$$

- Addition is primitive recursive as seen by the following application of the operation of primitive recursion:

$$0 + x = x$$
$$y + 1 + x = (y + x) + 1$$

Actually, the formal definition takes the form (where $add(y, x) = y + x$)

$$add(0, x) = P_1^1(x)$$
$$add(y + 1, x) = S(P_2^3(y, add(y, x), x))$$

- We can then define multiplication ($mult(y, x) = y \times x$) using primitive recursion applied to the null function and addition:

$$mult(0, x) = N(x)$$
$$mult(y + 1, x) = add(P_2^3(y, mult(y, x), x), P_3^3(y, mult(y, x), x))$$

or less formally,

$$0 \times x = 0$$
$$y + 1 \times x = (y \times x) + x$$

- Sometimes, as is the case with addition and multiplication, it is more natural or convenient to allow the recursive definition to occur over a variable other than the first variable. This is permissable since we can use the projection functions to rearrange the variables in any order we wish. For example, we can define the function

$$add'(x, y) = add(P_2^2(x, y), P_1^2(x, y))$$
$$= add(y, x)$$

so that in effect we have:

$$x + 0 = x$$
$$x + y + 1 = (x + y) + 1$$

- The function *blv* is also primitive recursive:

$$blv(0) = 0$$
$$blv(y + 1) = 1$$

Or, formally

$$blv(0) = 0$$
$$blv(y + 1) = S(N(P_1^2(y, blv(y))))$$

- Similarly, the function *neg* is primitive recursive

$$neg(0) = 1$$
$$neg(y + 1) = 0$$

Proposition 4.1 Every primitive recursive function is a total function, i.e., defined on all natural numbers.

Proof: The proof is by induction on the definition of a primitive recursive function *f*. Clearly, all the base functions are total functions. Next, if *f* is defined by substitution from *g* and h_1,..., h_m, then *f* is total whenever *g* and h_1,..., h_m are total.

Suppose *f* is defined by primitive recursion from *g* and *h*, and suppose by induction hypothesis that *g* and *h* are total functions. We prove by induction that for every $y \in \mathbb{N}$, $f(y, \vec{x}_{2^n}) \downarrow$. First, $f(0, \vec{x}_{2^n}) \downarrow$, since $f(0, \vec{x}_{2^n}) = g(\vec{x}_{2^n})$ and *g* is total. Next, assuming that $f(y, \vec{x}_{2^n}) \downarrow$, we see that $f(y + 1, \vec{x}_{2^n}) \downarrow$, since $f(y + 1, \vec{x}_{2^n}) = h(y, f(y, \vec{x}_{2^n}), \vec{x}_{2^n})$ and *h* is total.

Next Up Previous Contents Index

Next: 4.1 Primitive Recursive Expressibility **Up:** Lecture Notes for CS 2110 Introduction to Theory

Previous: 3.3 Complexity of LOOP Programs

Bob Daley

2001-11-28

4.1 Primitive Recursive Expressibility

An *n*-ary predicate on $\mathbb{N}$ is a subset of $\mathbb{N}^n$

$$P(\vec{x}^n) \text{ is TRUE} \iff \vec{x}^n \in P$$

i.e., $P = \{\vec{x}^n : P(\vec{x}^n) \text{is TRUE}\}$.

Thus sets and predicates are interchangeable. The characteristic function of a predicate P is the function χ_P defined by

$$\chi_P(\vec{x}^n) = \begin{cases} 1, & \text{if } \vec{x}^n \in P \\ 0, & \text{otherwise.} \end{cases}$$

Definition 4.2 A predicate P is *primitive recursive* if and only if χ_P is primitive recursive.

Conversely, given any 0 - 1 valued function f, we can associate with a predicate P_f and a set S_f defined by

$$P_f(\vec{x}^n) \text{ is TRUE} \iff f(\vec{x}^n) = 1$$

$$S_f = \{\vec{x}^n : f(\vec{x}^n) = 1\}$$

Proposition 4.2 If P and Q are primitive recursive predicates with the same number of variables, then so are $\neg P$, $P \wedge Q$, and $P \vee Q$.

Proof: The characteristic functions of these predicates are given in terms of χ_P and χ_Q as follows:

$$\chi_{\neg P}(\vec{x}^{\,n}) = neg(\chi_P(\vec{x}^{\,n}))$$

$$\chi_{P \wedge Q}(\vec{x}^{\,n}) = \chi_P(\vec{x}^{\,n}) \times \chi_Q(\vec{x}^{\,n})$$

$$\chi_{P \vee Q}(\vec{x}^{\,n}) = blv(\chi_P(\vec{x}^{\,n}) + \chi_Q(\vec{x}^{\,n}))$$

Proposition 4.3 If $P_1, \ldots, P_m$ are pairwise disjoint primitive recursive predicates over $\mathbb{N}^n$ and $f_1, \ldots, f_{m+1}$ are primitive recursive functions over $\mathbb{N}^n$, then so is the function $g : \mathbb{N}^n \longrightarrow \mathbb{N}$ defined by

$$g(\vec{x}^{\,n}) = \begin{cases} f_1(\vec{x}^{\,n}), & \text{if } P_1(\vec{x}^{\,n}), \\ \quad\vdots & \\ f_m(\vec{x}^{\,n}), & \text{if } P_m(\vec{x}^{\,n}), \\ f_{m+1}(\vec{x}^{\,n}), & \text{otherwise.} \end{cases}$$

Proof:

$$g(\vec{x}^{\,n}) = (f_1(\vec{x}^{\,n}) \times \chi_{P_1}(\vec{x}^{\,n})) + \cdots + (f_m(\vec{x}^{\,n}) \times \chi_{P_m}(\vec{x}^{\,n}))$$

$$+ (f_{m+1}(\vec{x}^{\,n}) \times \chi_{\neg(P_1 \vee \cdots \vee P_m)}(\vec{x}^{\,n}))$$

Definition 4.3 (Bounded Quantifiers) If $P(y, \vec{z}^{\,n})$ is a $n + 1$-ary predicate, then we define the $n + 1$-ary predicates $\exists y \leq x P(y, \vec{z}^{\,n})$ and $\forall y \leq x P(y, \vec{z}^{\,n})$ as follows:

$$\exists y \leq x P(y, \vec{z}^{\,n}) \iff \text{there is some } \; y \leq x \; \text{ such that } \; P(y, \vec{z}^{\,n})$$

$$\forall y \leq x P(y, \vec{z}^{\,n}) \iff \text{for all } \; y \leq x, P(y, \vec{z}^{\,n})$$

We abbreviate $\exists y \leq x P(y, \vec{z}^{\,n})$ by $\exists y \leq x P$ and $\forall y \leq x P(y, \vec{z}^{\,n})$ by $\forall y \leq x P$.

Proposition 4.4 If P is a primitive recursive predicate, then so are $\exists y \leq x P$ and $\forall y \leq x P$.

Proof: We show only that $\chi_{\exists y \leq x P(y, \vec{z}^{\,n})}$ is primitive recursive, since $\forall y \leq x P(y, \vec{z}^{\,n}) = \neg \exists y \leq x \neg P(y, \vec{z}^{\,n})$.

$\chi_{\exists y \leq x P}$ (which we abbreviate by $\chi_{\exists \leq P}$) is defined as follows:

$$\chi_{\exists \leq P}(0, \vec{z}^{\,n}) = \chi_P(0, \vec{z}^{\,n})$$

$$\chi_{\exists \le P}(x+1, \vec{z}^{\,n}) = \chi_{\exists y \le x P}(x, \vec{z}^{\,n}) \vee \chi_P(x+1, \vec{z}^{\,n})$$

$$= blv(add(P_2^{n+2}(x, \chi_{\exists y \le x P}(x, \vec{z}^{\,n}), \vec{z}^{\,n}),$$

$$\chi_P(S(P_1^{n+2}(x, \chi_{\exists y \le x P}(x, \vec{z}^{\,n}), \vec{z}^{\,n})),$$

$$P_3^{n+2}(x, \chi_{\exists y \le x P}(x, \vec{z}^{\,n}), \vec{z}^{\,n}),\ldots,$$

$$P_{n+2}^{n+2}(x, \chi_{\exists y \le x P}(x, \vec{z}^{\,n}), \vec{z}^{\,n})))$$

Next Up Previous Contents Index

Next: 4.2 Equivalence between models **Up:** 4. Primitive Recursive Functions **Previous:** 4. Primitive Recursive Functions

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 4.3 Primitive Recursive Expressibility (Revisited) **Up:** 4. Primitive Recursive Functions
Previous: 4.1 Primitive Recursive Expressibility

4.2 Equivalence between models

In order to compare primitive recursive functions with functions computed by *LOOP* programs over Σ_k^* we need to interpret functions computed by such programs as functions over $\mathbb{N}$.

Definition 4.4 Let *P* be a *LOOP* program over Σ_k^* and let $f_P : (\Sigma_k^*)^n \longrightarrow \Sigma_k^*$ be the function computed by *P*. The we say that *P computes the numer-theoretic function f* : $\mathbb{N}^n \longrightarrow \mathbb{N}$, where

$$f(\vec{x}^n) = \nu_k(f_P(\kappa_k(x_1),\ldots,\kappa_k(x_n)))$$

Theorem 4.5 Every primitive recursive function is computed by some *LOOP* program which contains no **UNTIL** loops.

Proof: We prove this by induction on the number of operations used in the definition of the given primitive recursive function *f*.

- **Induction basis:**
  Base functions
  - **Case 1:**
    The null function *N* is computed by the program

    INPUT($\mathbf{X}_1$ **)**

    $\mathbf{X}_1 \leftarrow \mathbf{0}$

    OUTPUT($\mathbf{X}_1$ **)**

  - **Case 2:**
    The successor function *S* is computed by the program

INPUT($\mathbf{X}_1$ **)**

$\mathbf{X}_1 \leftarrow \mathbf{X}_1 + \mathbf{1}$

OUTPUT($\mathbf{X}_1$ **)**

- **Case 3:**

  The projection function P_j^n is computed by the program

  INPUT($\mathbf{X}_1, \ldots, \mathbf{X}_n$ **)**

  OUTPUT($\mathbf{X}_j$ **)**

- **Induction step:**

  Operations

  - **Case 1:**

    Suppose

$$f(\vec{x}^{\,n}) = g(h_1(\vec{x}^{\,n}), \ldots, h_m(\vec{x}^{\,n})).$$

and let P, Q_1,..., $Q_m$be *LOOP*programs (without **UNTIL**loops) for g, h_1,..., h_m, respectively. The following program computes f, where $\mathbf{Z}_1$,...,$\mathbf{Z}_m$,$\mathbf{Y}_1$,...,$\mathbf{Y}_n$and $\mathbf{W}_1$are new program variables which do not occur in any of P, Q_1,..., Q_m.

INPUT($\mathbf{Y}_1$,...,$\mathbf{Y}_n$**)**

$\mathbf{Z}_1 \leftarrow \mathbf{Q}_1(\mathbf{Y}_1,\ldots,\mathbf{Y}_n)$

.

.

.

$\mathbf{Z}_m \leftarrow \mathbf{Q}_m(\mathbf{Y}_1,\ldots,\mathbf{Y}_n)$

$\mathbf{W}_1 \leftarrow \mathbf{P}(\mathbf{Z}_1, \ldots, \mathbf{Z}_m)$

OUTPUT($\mathbf{W}_1$)

- **Case 2:**

  Suppose

$$f(0, \vec{x}_{2^n}) = g(\vec{x}_{2^n})$$

$$f(y+1, \vec{x}_{2^n}) = h(y, f(y, \vec{x}_{2^n}), \vec{x}_{2^n}).$$

for $y \in \mathbb{N}$ and $\vec{x}_{2^n} \in \mathbb{N}^{n-1}$, and suppose *P*and *Q*are *LOOP*programs (without **UNTIL**loops) for *g*and *h*, respectively. The following program computes *f*, where $\mathbf{Y}_1,...,\mathbf{Y}_n,\mathbf{Z}_1$, and $\mathbf{W}_1$are new program variables not occurring in *P*or *Q*.

INPUT($\mathbf{Y}_1,...,\mathbf{Y}_n$)

$\mathbf{Z}_1 \leftarrow \mathbf{P}(\mathbf{Y}_2, \ldots, \mathbf{Y}_n)$

$\mathbf{W}_1 \leftarrow 0$

FOR $\mathbf{Y}_1$ TIMES DO

$\mathbf{Z}_1 \leftarrow \mathbf{Q}(\mathbf{W}_1,\mathbf{Z}_1,\mathbf{Y}_2,...,\mathbf{Y}_n)$

$\mathbf{W}_1 \leftarrow \mathbf{W}_1 + \mathbf{1}$

ENDFOR

OUTPUT($\mathbf{Z}_1$)

♠ The above proof is really an *informal* proof, since we haven't proved formally that the programs are correct. We do that now.

- **Induction basis:**

  Base functions

  - **Case 1:**

INPUT($\mathbf{X}_1$ **)**

$\mathbf{X}_1 \leftarrow \mathbf{0}$

OUTPUT($\mathbf{X}_1$ **)**

The output of this program is always ε, and since $\nu_k(\varepsilon) = 0$, this program correctly computes the null function *N*.

- **Case 2:**

INPUT($\mathbf{X}_1$ **)**

$\mathbf{X}_1 \leftarrow \mathbf{X}_1 + \mathbf{1}$

OUTPUT($\mathbf{X}_1$ **)**

Let $x \in \mathbb{N}$ be the input to *S*, then the input to this program is $\kappa_k(x)$, and the output is that string $[\mathbf{X}_1]$ such that

$$\nu_k([\mathbf{X}_1]) = \nu_k(\kappa_k(x)) + 1 = x + 1 = S(x)$$

- **Case 3:**

INPUT($\mathbf{X}_1, \ldots, \mathbf{X}_n$ **)**

OUTPUT($\mathbf{X}_j$ **)**

Given input $\vec{x}^{\,n} \in \mathbb{N}^n$ to P_j^n, the output of this program is $\kappa_k(x_j)$, and since $\nu_k(\kappa_k(x_j)) = x_j = P_j^n(\vec{x}^{\,n})$, the program is correct.

- **Induction step:**

Operations

- **Case 1:**

INPUT($\mathbf{Y}_1,...,\mathbf{Y}_n$)

$\mathbf{Z}_1 \leftarrow \mathbf{Q}_1(\mathbf{Y}_1,...,\mathbf{Y}_n)$

.

.

.

$\mathbf{Z}_m \leftarrow \mathbf{Q}_m(\mathbf{Y}_1,...,\mathbf{Y}_n)$

$\mathbf{W}_1 \leftarrow \mathbf{P}(\mathbf{Z}_1, \ldots, \mathbf{Z}_m)$

OUTPUT($\mathbf{W}_1$)

The *Induction Hypothesis* is that

$$g(\vec{y}^{\,m}) = \nu_k(f_P(\kappa_k(y_1),..., \kappa_k(y_m)))$$

and for each $1 \leq j \leq m$

$$h_j(\vec{x}^{\,n}) = \nu_k(f_{Q_j}(\kappa_k(x_1),..., \kappa_k(x_n)))$$

Given inputs $\vec{x}^{\,n} \in \mathbb{N}^n$ to *f*, at the end of this program

$[\mathbf{Z}_j] = f_{Q_j}(\kappa_k(x_1),..., \kappa_k(x_n))$ for all $1 \leq j \leq m$, and hence

$$\nu_k([\mathbf{W}_1]) = \nu_k \circ f_P(f_{Q_1}(\kappa_k(x_1),..., \kappa_k(x_n)),...,$$

$$f_{Q_m}(\kappa_k(x_1),..., \kappa_k(x_n)))$$

$$= \nu_k \circ f_P(\kappa_k \circ \nu_k \circ f_{Q_1}(\kappa_k(x_1),..., \kappa_k(x_n)),...,$$

$$\kappa_k \circ \nu_k \circ f_{Q_m}(\kappa_k(x_1),\ldots,\kappa_k(x_n)))$$

$$= \nu_k \circ f_P(\kappa_k \circ h_1(\vec{x}^{\,n}),\ldots,$$

$$\kappa_k \circ h_m(\vec{x}^{\,n}))$$

$$= g(h_1(\vec{x}^{\,n}),\ldots, h_m(\vec{x}^{\,n}))$$

$$= f(\vec{x}^{\,n})$$

where $\circ$ denotes the operation of *function composition*.

- **Case 2:**
  (Left as an exercise)

Theorem 4.6 Every number-theoretic function computed by a *LOOP* program without **UNTIL** loops is primitive recursive.

Proof: Let *P* be a given *LOOP* program without **UNTIL** loops of the form:

INPUT($\mathbf{X}_1,\ldots,\mathbf{X}_n$)

$\mathbf{P}^-$

OUTPUT($\mathbf{X}_k$)

Let $\mathbf{X}_1,\ldots,\mathbf{X}_r$ be a list of all the variables occurring in *P*, and let $\mathbf{Y}_1,\ldots,\mathbf{Y}_m$ be a list of ``imaginary" loop control variables needed by the internal implementation of **FOR** loops. We define by induction on the number of steps used in the construction of P^- a set of primitive recursive functions $f_{P^-}^{\,j}$ of $r + m$ variables such that if $\vec{x}^{\,r}$ and $\vec{y}^{\,m}$ are the values of the variables $\mathbf{X}_1,\ldots,\mathbf{X}_r$ and $\mathbf{Y}_1,\ldots,$

$\mathbf{Y}_m$ at the beginning of the execution of P^-, then for each $1 \leq j \leq r$, $f_{P^-}^j(\vec{x}^{\,r}, \vec{y}^{\,m})$ is the (numerical) value of the variable $\mathbf{X}_j$ at the end of the execution of P^-, and similarly for each $1 \leq j \leq m$, $f_{P^-}^{r+j}(\vec{x}^{\,r}, \vec{y}^{\,m})$ is the value of the imaginary loop contol variable $\mathbf{Y}_j$ at the end of the execution of P^-. Of course, if P^- doesn't halt (which it will *always* do), then the value of $f_{P^-}^{r+j}(\vec{x}^{\,r}, \vec{y}^{\,m})$ is undefined.

Having defined f_P^j, then the primitive recursive function which P computes is given by

$$f(\vec{x}^{\,n}) = f_{P^-}^k(\vec{x}^{\,n}, 0,..., 0)$$

$$= f_{P^-}^k(P_1^n(\vec{x}^{\,n}),..., P_n^n(\vec{x}^{\,n}), N^n(\vec{x}^{\,n}),..., N^n(\vec{x}^{\,n}))$$

where $N^n(\vec{x}^{\,n}) = N(P_1^n(\vec{x}^{\,n})) = 0$.

- **Induction basis:**
  - **Case 1:**

    P^- is $\mathbf{X}_i \leftarrow \mathbf{0}$. Then,

    $$f_{P^-}^i(\vec{x}^{\,r}, \vec{y}^{\,m}) = N^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})$$

    and for all $j \neq i$,

    $$f_{P^-}^j(\vec{x}^{\,r}, \vec{y}^{\,m}) = P_j^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})$$

- **Case 2:**

  P^- is $\mathbf{X}_i \leftarrow \mathbf{X}_i + \mathbf{1}$. Then,

$$f_{P^-}{}^{i}(\vec{x}^{\,r}, \vec{y}^{\,m}) = S(P_i^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m}))$$

  and for all $j \neq i$,

$$f_{P^-}{}^{j}(\vec{x}^{\,r}, \vec{y}^{\,m}) = P_j^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})$$

- **Case 3:**

  P^- is $\mathbf{X}_i \leftarrow \mathbf{X}_t$. Then,

$$f_{P^-}{}^{i}(\vec{x}^{\,r}, \vec{y}^{\,m}) = P_t^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})$$

  and for all $j \neq i$,

$$f_{P^-}{}^{j}(\vec{x}^{\,r}, \vec{y}^{\,m}) = P_j^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})$$

- **Induction step:**

  - **Case 1:**

    P^- is of the form:

    P_1

    P_2

where, of course P_1 and P_2 are lists of *LOOP* statements which do not include any I/O statements (or **UNTIL** loops). Then,

$$f_{P^-}^{j}(\vec{x}^{\,r}, \vec{y}^{\,m}) = f_{P_2}^{j}(f_{P_1}^{1}(\vec{x}^{\,r}, \vec{y}^{\,m}),..., f_{P_1}^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})).$$

- **Case 2:**

  P^- is of the form:

  FOR X_i **TIMES DO**

  Q

  ENDFOR

Suppose that this is the t^{th} **FOR** loop thus far encountered in the construction of P^-. We first define via primitive recursion a set of primitive recursive functions g_Q^j of $r+m$ arguments such that if $\vec{x}^{\,r}, \vec{y}^{\,m}$ are the values of the variables before entering this **FOR** loop, then $g_Q^j(\vec{x}^{\,r}, y_1,..., y_t,..., y_m)$ is the value of the j^{th} variable after y_t consecutive executions of the loop body Q. First,

$$g_Q^{r+t}(\vec{x}^{\,r}, \vec{y}^{\,m}) = P_{r+t}^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m}).$$

Next, for $j \neq r+t$,

$$g_Q^j(\vec{x}^{\,r}, y_1,..., 0,..., y_m) = P_j^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m})$$

$$g_Q^j(\vec{x}^{\,r}, y_1,..., y_t + 1,..., y_m) = f_Q^j(g_Q^1(\vec{x}^{\,r}, y_1,..., y_t,..., y_m),...,$$

$$= \quad g_Q^{r+m}(\vec{x}^{\,r}, y_1, \ldots, y_t, \ldots, y_m)).$$

Then, the primitive recursive function f_P^j is defined by

$$f_P^j(\vec{x}^{\,r}, \vec{y}^{\,m}) = g_Q^j(\vec{x}^{\,r}, y_1, \ldots, P_i^{r+m}(\vec{x}^{\,r}, \vec{y}^{\,m}), \ldots, y_m).$$

♠ Technically, the ``recursive" definition of $g_Q^j$ is **not** primitive recursive, since for each $j$, the definition of $g_Q^j$ depends on $g_Q^1, \ldots, g_Q^{r+m}$, i.e., on *all* $g_Q^j$. This is an example of the ``simultaneous inductive definition" of a set of functions. We will show that this form of recursion as well as other general forms of recursion are all constructible from primitive recursive functions.

Next Up Previous Contents Index

Next: 4.3 Primitive Recursive Expressibility (Revisited) **Up:** 4. Primitive Recursive Functions
Previous: 4.1 Primitive Recursive Expressibility

Bob Daley
2001-11-28

4.3 Primitive Recursive Expressibility (Revisited)

Definition 4.5 (Bounded Minimization) The function $f: \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ is obtained from the predicate P of $n+1$ arguments by *bounded minimization* if for all $x, \vec{z}^{\,n}$

$$f(x, \vec{z}^{\,n}) = \begin{cases} m, \text{ where } m \text{ is the least number } \leq x \text{ such that } P(m, \vec{z}^{\,n}) \\ x+1, \text{ otherwise.} \end{cases}$$

We use $f(x, \vec{z}^{\,n}) = \min y \leq x[P(y, \vec{z}^{\,n})]$ to denote that f is obtained from P via bounded minimization.

Proposition 4.7 If P is a primitive recursive predicate, then so is any function f obtained from P via bounded minimization.

Proof: If $f(x, \vec{z}^{\,n}) = \min y \leq x[P(y, \vec{z}^{\,n})]$, then we define f by induction as follows:

$$f(0, \vec{z}^{\,n}) = 1 \dot{-} \chi_P(0, \vec{z}^{\,n})$$

$$f(x+1, \vec{z}^{\,n}) = \begin{cases} f(x, \vec{z}^{\,n}), \text{ if } \exists y \leq x[P(y, \vec{z}^{\,n})] \\ x+1, \text{ otherwise if } P(x+1, \vec{z}^{\,n}) \\ x+2, \text{ otherwise.} \end{cases}$$

Proposition 4.8 Integer division is primitive recursive.

Proof:

$$x/y = \min z \leq x[(z + 1) \times y > x].$$

Next Up Previous Contents Index

Next: 4.4 General Recursion **Up:** 4. Primitive Recursive Functions **Previous:** 4.2 Equivalence between models

Bob Daley
2001-11-28

4.4 General Recursion

Definition 4.6 Given functions $g : \mathbb{N}^n \longrightarrow \mathbb{N}$, $h : \mathbb{N}^{n+2} \longrightarrow \mathbb{N}$, and a total function $r : \mathbb{N} \longrightarrow \mathbb{N}$ such that $r(0) = 0$ and $r(x) < x$ for all $x > 0$, then $f : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ is defined from g, h and r via *recursion*, if for any $\vec{x}^n \in \mathbb{N}^n$

$$f(0, \vec{x}^n) = g(\vec{x}^n)$$

$$f(y, \vec{x}^n) = h(y, f(r(y), \vec{x}^n), \vec{x}^n) \qquad \text{for any} \quad y > 0.$$

Proposition 4.9 If f is defined by recursion from (primitive / partial) recursive functions g, h, and r, then f is (primitive / partial) recursive.

Proof: First define the function r^* by

$$r^*(0, x) = x$$
$$r^*(y + 1, x) = r(r^*(y, x))$$

and the function q by

$$q(x) = \min y \leq x[r^*(y, x) = 0]$$

The value $q(y)$ specifies the number of steps in the building-up process for $f(y, \vec{x}^n)$.

Since r is total (primitive) recursive and $r(x) < x$ for any $x > 0$, we see that r^* and q are also total (primitive) recursive. Also, $q(x) = 0 \iff x = 0$. Next define the (primitive / partial) recursive function

H as follows:

$$H(0, y, z, \vec{x}^{\,n}) = z$$

$$H(m+1, y, z, \vec{x}^{\,n}) = h(r^*(q(y) \dot{-} (m+1), y), H(m, y, z, \vec{x}^{\,n}), \vec{x}^{\,n})$$

We prove by induction for all $m \leq q(y)$ that

$$H(m, y, g(\vec{x}^{\,n}), \vec{x}^{\,n}) = f(r^*(q(y) \dot{-} m, y), \vec{x}^{\,n})$$

from which it follows

$$f(y, \vec{x}^{\,n}) = H(q(y), y, g(\vec{x}^{\,n}), \vec{x}^{\,n})$$

so that f is (primitive / partial) recursive.

- **Induction basis:**

$$H(0, y, g(\vec{x}^{\,n}), \vec{x}^{\,n}) = g(\vec{x}^{\,n})$$

$$= f(0, \vec{x}^{\,n}) = f(r^*(q(y), y), \vec{x}^{\,n})$$

- **Induction step:**

  Suppose that $H(m, y, g(\vec{x}^{\,n}), \vec{x}^{\,n}) = f(r^*(q(y) \dot{-} m, y), \vec{x}^{\,n})$, then

$$H(m+1, y, g(\vec{x}^{\,n}), \vec{x}^{\,n}) = h(r^*(q(y) \dot{-} (m+1), y), H(m, y, g(\vec{x}^{\,n}), \vec{x}^{\,n}), \vec{x}^{\,n})$$

$$= h(r^*(q(y) \dot{-} (m+1), y), f(r^*(q(y) \dot{-} m, y), \vec{x}^{\,n}), \vec{x}^{\,n})$$

$$= f(r^*(q(y) \dot{-} (m+1), y), \vec{x}^{\,n})$$

Next Up Previous Contents Index

Next: 4.5 String Operations **Up:** 4. Primitive Recursive Functions **Previous:** 4.3 Primitive Recursive Expressibility (Revisited)

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 4.6 Coding of Tuples **Up:** 4. Primitive Recursive Functions **Previous:** 4.4 General Recursion

4.5 String Operations

Fix an alphabet $\Sigma_k = \{1,..., k\}$. We adopt the convention of using *u*, *v*, and *w* to denote strings over Σ_k^*. We define the following elementary string functions:

Suppose $w = a_n \cdots a_1 a_0 \in \Sigma_k^*$, then

$$end_k(w) = a_0$$
$$rsf_k(w) = a_n \cdots a_1$$

Proposition 4.10 The functions end_k and rsf_k are primitive recursive in the sense that $\nu_k \circ end_k \circ \kappa_k$ and $\nu_k \circ rsf_k \circ \kappa_k$ are primitive recursive.

Proof: The functions are defined as follows:

$$rsf_k(x) = (x \dot{-} 1)/k$$

$$end_k(x) = x \dot{-} (rsf_k(x) \times k)$$

To see that these are correct, observe that if $w = \kappa_k(x) = a_n \cdots a_1 a_0$, then $x \dot{-} 1 = a_n \times k^n + \cdots + a_1 \times k +$ $(a_0 - 1)$, where $0 \leq a_0 - 1 < k$, so that $rsf_k(x) = a_n \times k^{n-1} + \cdots + a_2 \times k + a_1$ as required. Given the correctness of rsf_k, the correctness of end_k is immediate.

Proposition 4.11 The string functions $|w|_k$ and $u \cdot v$ (i.e., length and concatenation) are primitive recursive.

Proof: String length over Σ_k^* is defined by

$$|x|_k = \min y \leq x[rsf_k^*(x, y) = 0],$$

where rsf_k^* is defined by

$$rsf_k^*(x, 0) = x$$
$$rsf_k^*(x, y + 1) = rsf_k(rsf_k^*(x, y)).$$

and concatenation over Σ_k^* is defined by

$$x \cdot y = x \times k^{|y|_k} + y.$$

Proposition 4.12 The following string predicates and functions are primitive recursive.

$occ_k(u, w) \equiv$ the string u occurs in the string w.

$pre_k(u, w) =$ the prefix of the first occurrence of u in w.

$suf_k(u, w) =$ the suffix of the first occurrence of u in w.

$rep_k(u, v, w) =$ the result of replacing the first occurrence of u in w by v.

For pre_k and suf_k we require that if u does not occur in w, then the value of the function is $w + 1$.

Proof:

$$occ_k(x, z) \equiv \exists_{y_1 \le z}\, \exists_{y_2 \le z}\, [z = y_1 \cdot x \cdot y_2].$$

$$pre_k(x, z) = \min y_1 \le z\ \exists_{y_2 \le z}\, [z = y_1 \cdot x \cdot y_2].$$

$$suf_k(x, z) = \min y_2 \le z\ [z = pre_k(x, z) \cdot x \cdot y_2].$$

$$rep_k(x, y, z) = pre_k(x, z) \cdot y \cdot suf_k(x, z).$$

Corollary 4.13 If g and h_a for each $a \in \Sigma_k$ are (primitive / partial) recursive functions, then so is the function f defined by

$$f(\varepsilon, \vec{x}^{\,n}) = g(\vec{x}^{\,n})$$

$$f(y \cdot a, \vec{x}^{\,n}) = h_a(y, f(y, \vec{x}^{\,n}), \vec{x}^{\,n}), \qquad \text{for each} \quad a \in \Sigma_k$$

Proof: Define the (primitive / partial) recursive function H by

$$H(y, z, \vec{x}^{\,n}) = \begin{cases} h_1(rsf_k(y), z, \vec{x}^{\,n}), & \text{if } end_k(y) = 1 \\ \quad\vdots & \\ h_k(rsf_k(y), z, \vec{x}^{\,n}), & \text{if } end_k(y) = k \end{cases}$$

Then

$$f(y \cdot a, \vec{x}^{\,n}) = h_a(y, f(y, \vec{x}^{\,n}), \vec{x}^{\,n})$$

$$= H(y \cdot a, f(rsf_k(y \cdot a), \vec{x}^{\,n}), \vec{x}^{\,n})$$

so that f is defined by recursion from g, H, and rsf_k. But, clearly $rsf_k(x) < x$ for all $x > 0$, so that the result follows from Proposition 4.9.

Exercise 4.1 Show that if g is a primitive recursive function and P is a primititve recursive predicate, then are the following are also primitive recursive functions and predicates.

$$\min y \leq g(x)\ [P(y, \vec{z}^{\,n})]$$

$$\max y \leq g(x)\ [P(y, \vec{z}^{\,n})]$$

$$\forall y \leq g(x)\ [P(y, \vec{z}^{\,n})]$$

$$\exists y \leq g(x)\ [P(y, \vec{z}^{\,n})]$$

Exercise 4.2 Show that the following are primitive recursive:

$$x \leq y$$

$$x = y$$

$$x \neq y$$

$$x^y$$

Next Up Previous Contents Index

Next: 4.6 Coding of Tuples **Up:** 4. Primitive Recursive Functions **Previous:** 4.4 General Recursion

Bob Daley

2001-11-28

4.6 Coding of Tuples

As an application of the above we show how to code *n*-tuples of integers in a primitive recursive fashion. We simply view $x_1,..., x_n$ as a string over Σ_{k+1} consisting of *n* strings over Σ_k separated by the symbol ``,'' (which is the k+1st symbol of Σ_{k+1} and as such does not belong to Σ_k. Thus, the function of *n* arguments which produces this string, which we denote by $\langle x_1,..., x_n \rangle_n$, is primitive recursive via

$$\langle x_1,..., x_n \rangle_n = x_1 \cdot , \cdot \ldots \cdot , \cdot x_n.$$

Note that $\langle x_1,..., x_n \rangle_n$ is simply some primitive recursive function of *n* arguments which we could have denoted by $f_n^{\langle\,\rangle}(x_1,..., x_n)$. Next the projection functions Π_j^n for each $1 \leq j \leq n$ are defined by

$$\Pi_j^n(x) = x_j, \qquad \text{where} \quad x = \langle x_1,..., x_n \rangle_n .$$

In order to see that Π_n^j is primitive recursive, we must first define some useful primitive recursive functions. We use ',' (instead of k + 1) when we wish to refer to the special separation symbol ``,''.

- The function $noc_k(j, x)$, which gives the number of occurrences of the symbol j in the string x over Σ_k.

$$noc_k(j, \varepsilon) = 0$$

$$noc_k(j, x \cdot a) = \begin{cases} 0, & \text{if } j > k \\ noc_k(j, x), & \text{if } a \neq j \\ noc_k(j, x) + 1, & \text{if } a = j. \end{cases}$$

- Then, the predicate $tup(n, x)$, which specifies whether or not x codes an n-tuple, is given by

$$tup(n, x) \equiv noc_{k+1}(',', x) = n \dot{-} 1$$

- Next, we define the function $prt_k(j, x, n)$, which gives the part in the string x over Σ_k between the n^{th} and the $n+1^{st}$ occurrence of the symbol j,

$$prt_k(j, x, n) = \begin{cases} \max\ z \leq x\ \exists y_1 \leq x\ \exists y_2 \leq x\ [x = y_1 \cdot z \cdot y_2 \\ \qquad \wedge noc_k(j, y_1) = n \wedge \neg occ_k(j, z)], \\ \qquad \text{if } noc_k(j, x) \geq n \\ x + 1, \text{ otherwise.} \end{cases}$$

Observe, that if x has n occurrences of j, then $prt_k(j, x, n)$ gives the part of x between the n^{th} occurrence of j in x and the end of x.

- Next, we define the primitive recursive *uniform projection function* as follows:

$$\Pi(n, j, x) = \begin{cases} prt_{k+1}(',', x, j \dot{-} 1), & \text{if } tup(n, x) \wedge 1 \leq j \leq n \\ x + 1, & \text{otherwise.} \end{cases}$$

Finally, the projections Π^n_j are defined by

$$\Pi^n_j(x) = \Pi(n, j, x).$$

Thus $\langle \cdot \rangle_n$ together with $\Pi^n_1, ..., \Pi^n_n$ establish a one-to-one correspondence between all n-tuples of natural numbers and all strings over Σ^*_{k+1} with n- 1 occurrences of `` ,''. Furthermore, the uniform projection function Π allows for the decoding of every natural number as a unique tuple of natural numbers.

As another application of coding we at last show that it is possible to define several functions simultaneously by induction.

Proposition 4.14 Let $g_1, ..., g_m$ and $h_1, ..., h_m$ be (primitive / partial) recursive functions. Then the functions $f_1, ..., f_m$ defined by

$$f_i(0, \vec{x}^{\,n}) = g_i(\vec{x}^{\,n})$$

$$f_i(y + 1, \vec{x}^{\,n}) = h_i(y, f_1(y, \vec{x}^{\,n}), ..., f_m(y, \vec{x}^{\,n}), \vec{x}^{\,n})$$

for each $1 \leq i \leq m$, are also (primitive / partial) recursive.

Proof: Define G and H by

$$G(\vec{x}^{\,n}) = \langle g_1(\vec{x}^{\,n}), ..., g_m(\vec{x}^{\,n}) \rangle_m$$

$$H(y, z, \vec{x}^{\,n}) = \langle h_1(y, \Pi^m_1(z), ..., \Pi^m_m(z), \vec{x}^{\,n}), ...,$$

$$h_m(y, \Pi^m_1(z), ..., \Pi^m_m(z), \vec{x}^{\,n}) \rangle_m$$

and then the function F by

$$F(0, \vec{x}^{\,n}) = G(\vec{x}^{\,n})$$

$$F(y+1, \vec{x}^{\,n}) = H(y, F(y, \vec{x}^{\,n}), \vec{x}^{\,n})$$

Clearly, G, H and hence F are (primitive / partial) recursive. We first show by induction that

$$F(y, \vec{x}^{\,n}) = \langle f_1(y, \vec{x}^{\,n}), \ldots, f_m(y, \vec{x}^{\,n}) \rangle_m .$$

- **Induction basis:**

$$F(0, \vec{x}^{\,n}) = G(\vec{x}^{\,n})$$

$$= \langle g_1(\vec{x}^{\,n}), \ldots, g_m(\vec{x}^{\,n}) \rangle_m$$

$$= \langle f_1(0, \vec{x}^{\,n}), \ldots, f_m(0, \vec{x}^{\,n}) \rangle_m$$

- **Induction step:**

Assume that $F(y, \vec{x}^{\,n}) = \langle f_1(y, \vec{x}^{\,n}), \ldots, f_m(y, \vec{x}^{\,n}) \rangle_m$. Then,

$$F(y+1, \vec{x}^{\,n}) = H(y, F(y, \vec{x}^{\,n}), \vec{x}^{\,n})$$

$$= H(y, \langle f_1(y, \vec{x}^{\,n}), \ldots, f_m(y, \vec{x}^{\,n}) \rangle_m, \vec{x}^{\,n})$$

$$= \langle h_1(y, f_1(y, \vec{x}^{\,n}),..., f_m(y, \vec{x}^{\,n}), \vec{x}^{\,n}),...,$$

$$h_m(y, f_1(y, \vec{x}^{\,n}),..., f_m(y, \vec{x}^{\,n}), \vec{x}^{\,n}), \rangle_m$$

$$= \langle f_1(y + 1, \vec{x}^{\,n}),..., f_m(y + 1, \vec{x}^{\,n}) \rangle_m .$$

Therefore, we see that $f_i(y, \vec{x}^{\,n}) = \Pi_i^m(F(y, \vec{x}^{\,n}))$, and so $f_1,..., f_m$ are each (primitive / partial) recursive.

- We now see *in retrospect* that in the proof of Theorem 4.6 the definition of the functions g_Q^j are *legitimate* primitive recursive definitions.
- We can now see that it suffices to consider only (primitive / partial) recursive functions of one variable. Suppose f is a (primitive / partial) recursive function of n variables and let f^1 be the (primitive / partial) recursive function defined by $f^1(x) = f(\Pi_1^n(x),..., \Pi_n^n(x))$. Then, for any input $x_1,..., x_n \in \mathbb{N}^n$ to f, we see that

  $$f(x_1,..., x_n) = f^1(\langle x_1,..., x_n \rangle_n).$$

  Therefore, every (primitive / partial) recursive function of n variables can be replaced by a (primitive / partial) recursive function of one variable whose input is $\langle x_1,..., x_n \rangle_n$ instead of $x_1,..., x_n$. Furthermore, we can easily implement (primitive / partial) recursive functions with outputs by ``interpreting" outputs as tuples.

Next Up Previous Contents Index

Next: 5. Diagonalization Arguments **Up:** 4. Primitive Recursive Functions **Previous:** 4.5 String Operations

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 6. Partial Recursive Functions **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 4.6 Coding of Tuples

5. Diagonalization Arguments

♠ Observe that there clearly are *LOOP* programs which compute non-total functions:

INPUT($\mathbf{X}_1$**)**

UNTIL $\mathbf{Y}_1$ **TRUE DO**

ENDUNTIL
OUTPUT($\mathbf{Y}_1$**)**

Thus, because of the foregoing we must add some operation which can transform total functions into non-total functions to our set of primitive recursive functions in order to capture all the functions computed by *LOOP* programs. In fact, we now give an argument which shows that all models of computability must include some non-total functions.

Proverb 5.1 To define something (e.g., a function) which does **not** have a specified property, make it **different** from all those things (i.e., functions) which **do** have that property.

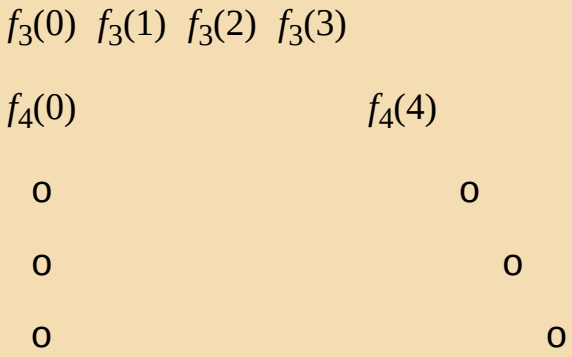
Arguments that use this proverb are called *diagonalization* arguments.

Example 5.1 There exist uncountably many total functions from $\mathbb{N}$ to $\mathbb{N}$.

Proof: Let $f_0, f_1, \ldots$ be some list of the countably many functions from $\mathbb{N}$ to $\mathbb{N}$. Consider the following tableau:

Table 5.1: Diagonalization Construction

$f_0(0)$	$f_0(1)$	$f_0(2)$	$f_0(3)$	$f_0(4)$	o o o
$f_1(0)$	$f_1(1)$	$f_1(2)$	$f_1(3)$		
$f_2(0)$	$f_2(1)$	$f_2(2)$	$f_2(3)$		



Then the function $f^{o}(n) = f_n(n) + 1$ is clearly different from each function on the list. Moereover, since each function on the list is total, so is f^{o}.

♠ Many arguments by contradiction are in fact diagonalization arguments in disguise.

Example 5.2 The cardinality of the power set 2^X of any set X is greater than the cardinality of X itself.

Proof: We denote the cardinality of a set Y by $\#Y$. Clearly, $\#X \leq \#2^X$, since we can define a function $h : X \longrightarrow 2^X$ by $h(x) = \{x\}$. Now suppose $g : X \longrightarrow 2^X$ is any function. We show that g cannot be *onto* (so 2^X must have *more* elements than X). Define

$$X^d = \{x \in X : x \notin g(x)\}.$$

If g is onto, then there is some element $y \in X$ such that $g(y) = X^d$. Consider the question whether $y \in X^d$:

$$y \in X^d \Longrightarrow y \notin g(y) \Longrightarrow y \notin X^d \qquad \text{contradiction!}$$

$$y \notin X^d \Longrightarrow y \in g(y) \Longrightarrow y \in X^d \qquad \text{contradiction!}$$

Therefore, no $g : X \longrightarrow 2^X$ can be onto. We can also give an explicit diagonalization argument as follows: Let $f_k = \chi_{g(k)}$, so

$$f_k(x) = \begin{cases} 1, & \text{if } x \in g(k) \\ 0, & \text{if } x \notin g(k) \end{cases}$$

Then, define

$$f^o(x) = \begin{cases} 1, & \text{if } f_x(x) = 0 \\ 0, & \text{if } f_x(x) = 1 \end{cases}$$

Then, f^o is a total 0 - 1 valued function on X, i.e., it is the characteristic function of some subset X^o of X. But, $X^o \in 2^X$ and X^o is different from each set $g(k)$, so g cannot be onto. Observe!

$$\begin{aligned} X^o &= \{x : f^o(x) = 1\} \\ &= \{x : f_x(x) = 0\} \\ &= \{x : x \notin g(x)\} \\ &= X^d \end{aligned}$$

- Under the assumption that the class of effectively computable functions should be countable and that programs for them should be effectively listable, we can show that the effectively computable functions *must* contain some non-total functions, i.e., functions which are undefined for some inputs.

  In the proof above that there are uncountably many total functions from $\mathbb{N}$ to $\mathbb{N}$ if we let f_n be the function computed by the n^{th} program in the effective listing of programs for computable

functions, we see that if all f_n are total, then so is f^o.

But, f^o is also effectively computable (intuitively) since on input n we simply find the n^{th} program; run it on input n; and then add 1 to the result.

Thus the list cannot contain *all* the effectively computable functions, which contradicts our assumption. Thus, the list must contain some non-total function.

- This argument also shows that there cannot exist any effective listing of *all* and *only* the total computable functions.

Next Up Previous Contents Index

Next: 6. Partial Recursive Functions **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 4.6 Coding of Tuples

Bob Daley

2001-11-28

6. Partial Recursive Functions

Notation 6.1 We use $\phi, \psi, \theta, \ldots$ to denote (possible) partial functions.

We use *f*, *g*, *h*,... to denote total functions.

$\phi(x)\downarrow$ means that $\phi(x)$ is defined (convergent), i.e., $x \in$ **dom** ϕ.

$\phi(x)\uparrow$ means that $\phi(x)$ is undefined (divergent), i.e., $x \notin$ **dom** ϕ.

$\phi = \psi$ means that for all *x* either both $\phi(x)\uparrow$ and $\psi(x)\uparrow$, or $\phi(x)\downarrow$ and $\psi(x)\downarrow$ and $\phi(x) = \psi(x)$.

Definition 6.2 The function $\phi : \mathbb{N}^n \longrightarrow \mathbb{N}$ is obtained from the function $\psi : \mathbb{N}^{n+1} \longrightarrow \mathbb{N}$ via *minimization* if for all $\vec{x}^n \in \mathbb{N}^n$

$$\phi(\vec{x}^n) = \begin{cases} m, \text{ where } m \text{ is the least number such that for all} \\ \qquad 0 \leq k < m, \psi(k, \vec{x}^n) = 0 \text{ and } \psi(m, \vec{x}^n) \neq 0 \\ \uparrow, \text{ otherwise.} \end{cases}$$

$\phi(\vec{x}^n) = \min y[\psi(y, \vec{x}^n) \neq 0]$ denotes that ϕ is obtained from ψ via minimization.

- The intuitive basis for minimization is that of an unbounded search for the first *y* satisfying the property that $\psi(y, \vec{x}^n) \neq 0$. In this regard, since ψ may be a *non-total* function, we must be sure that $\psi(k, \vec{x}^n)\downarrow$ for all $0 \leq k < m$ before testing $\psi(m, \vec{x}^n)$.

- If P is a predicate, then min $yP(y, \vec{x}^{\,n})$ means min $y[\chi_P(y, \vec{x}^{\,n}) \neq 0]$.

Definition 6.3 A function is *partial recursive* if it can be obtained from the base functions (null, successor, projections) by finitely many applications of the operations of substitution, primitive recursion, and minimization.

- A partial recursive function which is total is called *total recursive*.
- A predicate P is a *recursive predicate* if χ_P is a total recursive function.

Theorem 6.1 Every partial recursive function is computed by a *LOOP* program.

Proof: We need only add to Theorem 4.5 an additional case in the induction step dealing with the operation of minimization.

- **Case 3:**
  Suppose

$$\phi(\vec{x}^{\,n}) = \min y[\psi(y, \vec{x}^{\,n}) \neq 0]$$

and let Pbe a *LOOP*program for ψ and let $\mathbf{Y}_1,...,\mathbf{Y}_n,\mathbf{Z}_1,\mathbf{W}_1$be new program variables which do not occur in P. The program for $\phi$is given by:

INPUT($\mathbf{Y}_1,...,\mathbf{Y}_n$)
$\mathbf{W}_1 \leftarrow \mathbf{P}(\mathbf{Z}_1, \mathbf{Y}_1, \ldots, \mathbf{Y}_n)$
UNTIL $\mathbf{W}_1$ **TRUE DO**
  $\mathbf{Z}_1 \leftarrow \mathbf{Z}_1 + \mathbf{1}$
  $\mathbf{W}_1 \leftarrow \mathbf{P}(\mathbf{Z}_1, \mathbf{Y}_1, \ldots, \mathbf{Y}_n)$
ENDUNTIL
OUTPUT($\mathbf{Z}_1$)

Theorem 6.2 Every number-theoretic function computed by a *LOOP* program is partial recursive.

Proof: We need only add to the proof of Theorem 4.6 an additional case in the induction step dealing with **UNTIL** loops:

- **Case 3:**

  Suppose P^- is of the form

  UNTIL $\mathbf{X}_i$ **TRUE DO**

  Q

  ENDUNTIL

Let this be the t^{th}loop (of any kind), and let $\mathbf{Y}_t$be an imaginary program variable which will be used to count the number of times through the **UNTIL**loop, and let g_Q^jbe the set of functions defined previously in the proof of Theorem 4.6. Define,

$$h(\vec{x}^{\,r}, \vec{y}^{\,m}) = \min y_t \, [g_Q^i(\vec{x}^{\,r}, y_1, \ldots, y_t, \ldots, y_m) \neq 0].$$

Then,

$$f_P^j(\vec{x}^{\,r}, \vec{y}^{\,m}) = g_Q^j(\vec{x}^{\,r}, y_1, \ldots, h(\vec{x}^{\,r}, \vec{y}^{\,m}), \ldots, y_m).$$

Theorem 6.3 Fix some alphabet Σ_k. The class of number-theoretic functions computed by *LOOP* programs over Σ_k^* is identical to the class of partial recursive functions.

♠ Observe, that there is an *effective* (i.e., computable) procedure which given a *LOOP* program over Σ_k^* constructs (an expression for) the partial recursive function which computes it. Conversely, there is also an *effective* procedure which given a partial recursive function constructs a *LOOP* program over Σ_k^* which computes it.

♠ Observe also that for any *LOOP* program text, the partial recursive function which computes it is *independent* of the alphabet Σ_k which is used to specify its semantics.

♠ Observe further, however, that the complexity of a *LOOP* program *does depend* on the alphabet Σ_k, since it depends on the *length* of the internal and I/O representation used. Specifically, since for any $k > 1$, $|\kappa_k(x)| = \lceil \log_k x \rceil$, where $\lceil y \rceil$ denotes the least integer $\geq y$, but $|\kappa_1(x)| = x$, we see that between any two alphabets of more than one symbol, the respective complexity measures are related by a constant factor, whereas between Σ_1^* and any other alphabet consisting of more than one symbol the difference in complexity can be exponential.

Next Up Previous Contents Index

Next: 7. Random Access Machines **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 5. Diagonalization Arguments

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 7.1 Parsing RAM Programs **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 6. Partial Recursive Functions

7. Random Access Machines

A random access machine is an *idealized* computer with a random access memory consisting of a finite number of *idealized* registers (i.e., they can hold any sized number) R_1, R_2,... whose contents are strings over some alphabet Σ_k, and which has a finite set of *machine instructions*. The set of machine instructions are as follows:

Table 7.1:RAM Machine Instructions

Machine	Assembly	Effect
Instruction	Language	
1/m/j;	**jmp**$_1$ **R**$_m$/*j*	
.	.	**jmp**$_a$: if *a* is the leftmost
.	.	symbol of **R**$_m$, then GoTo
.	.	line *j* of the program
k/m/j;	**jmp**$_k$ **R**$_m$/*j*	
k + 1/m;	**suc**$_1$ **R**$_m$	
.	.	**suc**$_a$: concatenate
.	.	an *a* to the right
.	.	end of **R**$_m$
2k/m;	**suc**$_k$ **R**$_m$	
2k + 1/m;	**inp R**$_m$	input a value into **R**$_m$
2k + 2/m;	**out R**$_m$	output a value from **R**$_m$

2k + 3/m;	**lsf $\mathbf{R}_m$**	delete leftmost symbol of $\mathbf{R}_m$

Definition 7.1 A *RAM*-program over Σ_k^* is a sequence of *RAM* statements $S_1,..., S_n$ such that for some $1 < m < n$,

1. $S_1,..., S_m$ are the only input statements
2. S_n is the only output statement
3. and no conditional jump statement in $S_{m+1}, ..., S_{n-1}$ can cause a jump to any line $\leq m$.

Definition 7.2 A *RAM*-program P over Σ_k^* *computes* the (partial) function $f : (\Sigma_k^*)^n \longrightarrow \Sigma_k^*$ if and only if

1. there are n input statements of P;
2. for all $x_1,..., x_n$, when P is executed † with $x_1,..., x_n$ as its input,
   (a) P halts if and only if $f(x_1,..., x_n) \downarrow$,
   (b) if P halts, then P outputs $f(x_1,..., x_n)$.

† Execution of a *RAM* program involves:

1. initially all registers have value 0
2. statements are executed according to the ``obvious'' semantics in the ``obvious'' order.

Proposition 7.1 Every function computed by a *LOOP* program is also computed by a *RAM* program.

Proof: (Left as an exercise)

Next Up Previous Contents Index

Next: 7.1 Parsing RAM Programs **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 6. Partial Recursive Functions

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 7.2 Simulation of RAM Programs **Up:** 7. Random Access Machines **Previous:** 7. Random Access Machines

7.1 Parsing RAM Programs

Every *RAM* program over Σ_k is a string over the alphabet $\Sigma_k \cup \{/,;\} \approx \Sigma_{k+2}$, i.e., the / and ; are the k+1st and k+2nd letters of Σ_{k+2}, respectively. We will use '/' and ';' to denote (the codes of) these special symbols. We now show that given any natural number, regarding it as a string over Σ_{k+2}, there is a primitive recursive function which ``parses" that number.

- First suppose that x codes a *RAM* instruction (minus the ``;"). We define primitive recursive functions *opc*, *reg*, *gto*, which produce, respectively, the opcode part of x, the register named in x, and the goto part of x (if x codes a conditional jump instruction).

$$opc(x) = pre_{k+2}('/', x)$$

$$reg(x) = \begin{cases} suf_{k+2}('/', x), & \text{if } noc_{k+2}('/', x) = 1 \\ prt_{k+2}('/', x, 1), & \text{if } noc_{k+2}('/', x) = 2 \\ 0, & \text{otherwise.} \end{cases}$$

$$gto(x) = \begin{cases} prt_{k+2}('/', x, 2), & \text{if } noc_{k+2}('/', x) = 2 \\ 0, & \text{otherwise.} \end{cases}$$

- Next, we define a primitive recursive predicate *ins*(x) which determines whether x codes a legal instruction (minus the ``;"):

$$ins(x) \equiv (\neg occ_{k+2}(';', x)) \wedge (opc(x) \leq 2k + 3)$$
$$\wedge (opc(x) > 0) \wedge (reg(x) > 0)$$
$$\wedge (opc(x) \leq k \implies noc_{k+2}('/', x) = 2)$$
$$\wedge (opc(x) > k \implies noc_{k+2}('/', x) = 1)$$

- Suppose now that x codes a *RAM* program. We define primitive recursive functions $lng(x)$ and $lne(j, x)$ which give, respectively, the number of lines of x and the j^{th} line (ie., instruction) of x:

$$lng(x) = noc_{k+2}(';', x)$$
$$lne(j, x) = prt_{k+2}(';', x, j \dot{-} 1)$$

- Next, define primitive recursive programs $nrg(x)$ and $mxr(x)$ which give, respectively, the number of arguments of program x (i.e., the number of input statements), and the maximum number of any register used in x.

$$nrg(x) = \min m \leq lng(x)[\forall j \leq m[j > 0 \implies opc(lne(j, x)) = 2k + 1]$$
$$\wedge \forall j \leq lng(x)[j > m \implies opc(lne(j, x)) \neq 2k + 1]]$$
$$mxr(x) = \min y \leq x \quad \forall j \leq lng(x)[j > 0 \implies reg(lne(j, x)) \leq y]$$

- Then, we define the primitive recursive predicate $prg(x)$ which specifies whether or not x codes a legal program:

$$prg(x) \equiv \forall j \leq lng(x)[j > 0 \implies ins(lne(j, x))] \wedge (nrg(x) > 0)$$
$$\wedge (nrg(x) < lng(x)) \wedge (opc(lne(lng(x), x) = 2k + 2)$$
$$\wedge \forall j \leq lng(x) \dot{-} 1 [opc(lne(j, x)) \neq 2k + 2]$$

$$\wedge\ \forall j \leq lng(x)[j > 0\ \wedge\ opc(lne(j, x)) \leq k \Longrightarrow$$

$$nrg(x) < gto(lne(j, x)) \leq lng(x)]$$

Next Up Previous Contents Index

Next: 7.2 Simulation of RAM Programs **Up:** 7. Random Access Machines **Previous:** 7. Random Access Machines

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 7.3 Index Theorem **Up:** 7. Random Access Machines **Previous:** 7.1 Parsing RAM Programs

7.2 Simulation of RAM Programs

We now show how to simulate the execution of a *RAM* program (coded by) p over Σ_k^* on inputs (coded by) $y = \langle x_1,..., x_n \rangle_n$. Thus, p is a string over Σ_{k+2} and y is a string over Σ_{k+1}. In order to do this we need at each step to record the ``state'' of the program execution, which will be given by the pair $\langle j, z \rangle_2$, where j is the current line number, and z codes the current values of the registers used by p (so z will be a $mxr(p)$ tuple).

- First, we need to show that the primitive operations of *RAM* programs are primitve recursive. We define primitive recursive functions $val(p, z, j)$, $lnd(p, z, j)$, $lsf(p, z, j)$, $suc(a, p, z, j)$, and $inp(p, z, j, m)$, which give, respectively, the current value of register j, the leftmost symbol of register j, the result of deleting the leftmost symbol of register j, the result of adding the symbol a to the right end of register j, and the result of copying m into register j:

$$val(p, z, j) = \begin{cases} \Pi(mxr(p), j, z), & \text{if } 0 < j \leq mxr(p) \text{ and } tup(mxr(p), z) \\ 0, & \text{otherwise.} \end{cases}$$

$$lnd(p, z, j) = rsf_{k+1}{}^*(val(p, z, j), \mid val(p, z, j) \mid_{k+1} \dot{-} 1)$$

$$inp(p, z, j, m) = isg(p, z, j) \cdot rep_{k+1}(val(p, z, j), m, suf_{k+1}(isg(p, z, j), z))$$

$$lsf(p, z, j) = inp(p, z, j, suf_{k+1}(lnd(p, a, j), val(p, z, j)))$$

$$suc(a, p, z, j) = inp(p, z, j, val(p, z, j) \cdot a)$$

where $isg(p, z, j) = val(p, z, 1) \cdot ',' \cdot \cdots \cdot ',' \cdot val(p, z, j \dot{-} 1) \cdot ','$.

- We can now simulate the execution of *RAM* programs. We define two primitive recursive functions $nxl(p, y, z, j)$, which gives the next line of program p on input y to be executed given that the current register values are z and the current line is j; and $nxv(p, y, z, j)$, which gives the next values of the registers for program p on input y given that the current register values are z and the current line is j:

$$nxl(p, y, z, j) = \begin{cases} gto(lne(j,p)), \text{ if } \exists i \leq k \quad [opc(lne(j,p)) = i \\ \qquad \text{and } lnd(p, z, reg(lne(j,p))) = i] \\ j + 1, \text{ otherwise} \end{cases}$$

$$nxv(p, y, z, j) = \begin{cases} suc(a, p, z, reg(lne(j,p))), \text{ if } opc(lne(j,p)) = k + a \\ lsf(p, z, reg(lne(j,p))), \text{ if } opc(lne(j,p)) = 2k + 3 \\ inp(p, z, reg(lne(j,p)), \Pi(nrg(p), j, y)), \\ \qquad \text{if } opc(lne(j,p)) = 2k + 1 \\ z, \text{ otherwise.} \end{cases}$$

- Now we define the primitive recursive function $sim(p, y, m)$, which gives the pair $\langle j, z \rangle_2$ which codes the current line and the current register values after m steps of the computation of p on input y:

$$sim(p, y, 0) = \langle 1, zro(mxr(p)) \rangle_2$$

$$sim(p, y, m + 1) = \langle nxl(p, y, \Pi_2^2(sim(p, y, m)), \Pi_1^2(sim(p, y, m))),$$
$$nxv(p, y, \Pi_2^2(sim(p, y, m)), \Pi_1^2(sim(p, y, m))) \rangle_2$$

  where $zro(n) = \langle 0,..., 0 \rangle_n$.

- Next, we define the *partial* recursive function $stp(p, y)$, which gives the number of steps in the computation of p on input y *if* p halts on input y:

$$stp(p, y) = \min t[\Pi_1^2(sim(p, y, t)) = lng(p)]$$

- Now, we can define the ``universal" partial recursive function $\phi_{unv}(p, y)$, which gives the result, if any, of the computation of p on input y:

$$\phi_{unv}(p, y) = \begin{cases} \Pi(mxr(p), reg(lne(lng(p), p)), \Pi_2^2(sim(p, y, stp(p, y)))), \\ \qquad \text{if } prg(p) \\ \uparrow, \text{ otherwise.} \end{cases}$$

♠ Observe that if p does not code a legal program then $\phi_{unv}(p, y)$ is undefined for all y. We define an *indexing* or *Gödel numbering* $\{\phi_i\}$ of the *RAM* computable functions (of one argument) by letting ϕ_i denote the partial recursive function computed by the *RAM* program (with code) i. Observe that since every partial recursive function is computable by a *LOOP* program, and hence in turn by a *RAM* program, every partial recursive function is included in the list $\{\phi_i\}$. The promised effective translation of *RAM* programs into partial recursive functions is given by the following.

Theorem 7.2 For the indexing $\{\phi_i\}$ given above there is a ``universal" partial recursive function ϕ_{unv} such that for all x and y, $\phi_{unv}(x, y) = \phi_x(y)$.

- This result is not specific to *RAM* programs and partial recursive functions. We could have just as well written a *LOOP* program which transforms partial recursive function definitions into *RAM* programs.
- Since every partial recursive function is computable by a *RAM* program, there exists a *RAM* program P_{unv} which computes the function ϕ_{unv}, i.e., a *RAM* program which interprets (i.e., an ``interpreter" for) other *RAM* programs and simulates their execution.

- Observe that in the process of defining ϕ_{unv} *only one* application of (unbounded) minimization was used. Therefore, every partial recursive function can be computed by a *LOOP* program which uses only one **UNTIL** loop!

♠ The equivalence of *LOOP* computable, *RAM* computable and the class of partial recursive functions gives empirical evidence for *Church's Thesis*, which states that the class of partial recursive functions yield a formalization of our intuitive notion of effectively computable function.

Next Up Previous Contents Index

Next: 7.3 Index Theorem **Up:** 7. Random Access Machines **Previous:** 7.1 Parsing RAM Programs

Bob Daley

2001-11-28

7.3 Index Theorem

Theorem 7.2 shows that we can effectively interpret *RAM* programs. We now show that we can also effectively transform them. In particular, we show

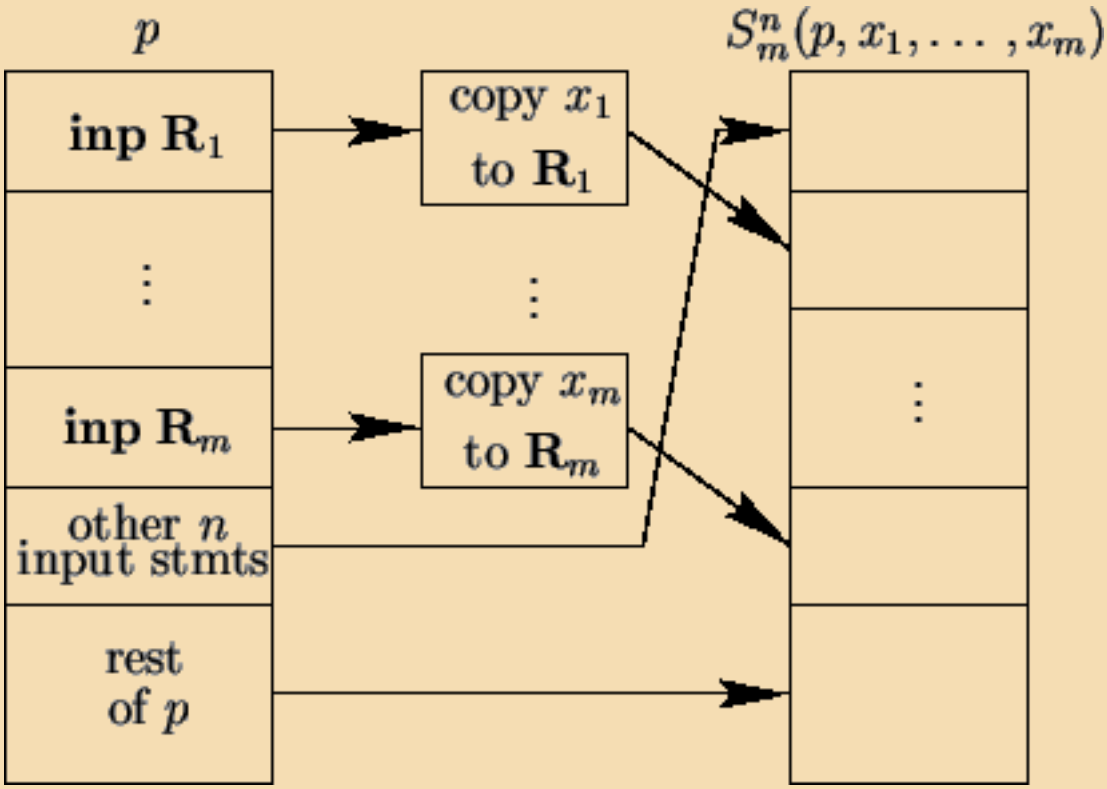
Theorem 7.3 For every $m, n \in \mathbb{N}$, there is a primitive recursive function S_m^n such that for every *RAM* program p of $m + n$ agruments, $S_m^n(p, x_1,..., x_m)$ is a *RAM* program of n arguments such that

$$\phi_{S_m^n(p,x_1,\ldots,x_m)}(y_1,..., y_n) = \phi_p(x_1,..., x_m, y_1,..., y_n)$$

The inutitive meaning of Theorem 7.3 is that given any *RAM* program p of $m + n$ arguments and any set of *fixed values* $x_1,..., x_m$ we can build these as constants into p and construct a program $S_m^n(p, x_1,..., x_n)$ of the remaining n arguments which behaves exactly like p with its first m arguments fixed to be $x_1,..., x_m$. This will allow us to build data into programs. We will suppose that $x_1,..., x_m$ and $y_1,..., y_n$ are coded as tuples and so will denote them by x and y, respectively. Thus, we need to show that $\phi_{S_m^n(p,x)}(y) = \phi_p(x, y)$.

We can imagine the structure of p consisting of m input statements (which will be replaced), followed by the remaining n input statements, followed by the remainder of p.

Figure 7.1: Index Theorem Transformation



Proof: First, the primitive recursive function $isg_{k+1}(p, z, j)$, mentioned previously, is defined by:

$$isg_{k+1}(p, z, j) = \min y_1 \leq z \; \exists y_2 \leq z \; [z = y_1 \cdot y_2 \; \wedge \; noc_{k+1}(',', y_1) = j]$$

Exercise 7.1 Show that if g is a primitive recursive function of $n + 1$ arguments, then the function defined by

$$f(x, \vec{y}^{\,n}) = \bigodot_{j=1}^{x} g(j, \vec{y}^{\,n})$$

$$= g(1, \vec{y}^{\,n}) \cdots g(x, \vec{y}^{\,n})$$

is primitive recursive. Observe that $f(0, \vec{y}^{\,n})$ should have the value 0.

One key part of the required transformation is to replace an input statement by a block of statements

which assign a specified fixed value z to the variable $\mathbf{R}_m$ of the assignment statement. More precisely, we need to replace the statement

inp$\mathbf{R}_m$

by the block of statements

suc$_{a_1}\mathbf{R}_m$

.

.

.

suc$_{a_n}\mathbf{R}_m$

where the specified value $z = a_1 \cdots a_n$. This replacement is effected by the primitive recursive function $rcp(z, m)$, which is defined by

$$rcp(z, m) = \bigodot_{j=1}^{|z|_k+1} (k + smb(z, j)) \cdot '/' \cdot m \cdot ';'$$

where $smb(z, j)$ is the primitive recursive function which gives the j^{th} symbol (from the left) of the string z.

Then, the block of such copy statements for the m-tuple x is given by

$$cpb(p, m, x) = \bigodot_{j=1}^{m} rcp(\prod(m, j, x), reg(lne(j, p)))$$

Next, we need to adjust the goto parts of the rest of the program in order to account for the change in the number of lines. The function $adl(z, r, s)$ adjusts the goto part of the instruction coded by z by + r if $r > 0$, and by - s if $r = 0$, and is defined by

$$adl(z, r, s) = \begin{cases} opc(z) \cdot^{\prime} /^{\prime} \cdot reg(z) \cdot^{\prime} /^{\prime} \cdot (gto(z) + r), \\ \qquad \text{if } opc(z) \leq k \text{ and } r > 0 \\ opc(z) \cdot^{\prime} /^{\prime} \cdot reg(z) \cdot^{\prime} /^{\prime} \cdot (gto(z) \dot{-} s), \\ \qquad \text{if } opc(z) \leq k \text{ and } r = 0 \\ z, \text{ otherwise.} \end{cases}$$

and the result of adjusting all the lines of a program p is given by

$$adp(p, r, s) = \bigodot_{j=1}^{lng(p)} adl(lne(j, p), r, s) \cdot ';'$$

Finally, we can express the definition of the transfoirmation S_m^n by

$$\begin{aligned} S_m^n(p, x) = {} & suf_{k+2}(isg_{k+2}(p, m, ';'), isg_{k+2}(p, m + n, ';')) \\ & \cdot cpb(p, m, x) \\ & \cdot adp(suf_{k+2}(isg_{k+2}(p, m + n, ';'), p), |\, x \,|_{k+2} + 1 \dot{-} (2 \times m), \\ & \qquad (2 \times m) \dot{-} 1 \dot{-} |\, x \,|_{k+2}) \end{aligned}$$

The number $|\, x \,|_{k+2} + 1 \dot{-} (2 \times m)$ arises from the fact that the net increase in length due to the copy block is equal to the length of x minus the m lines which are replaced. Observe, that it is possible for this number to be negative (e.g., when each element of the m-tuple x is a 0).

Next Up Previous Contents Index

Next: 7.4 Other Aspects **Up:** 7. Random Access Machines **Previous:** 7.2 Simulation of RAM Programs

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 7.5 Complexity of RAM Programs **Up:** 7. Random Access Machines **Previous:** 7.3 Index Theorem

7.4 Other Aspects

- We can construct *non-deterministic RAM* programs by adding instructions of the form

  2k + 4/j$_1$/j$_2$; **njp** j_1 **or** j_2

  which non-deterministically selects one of two lines (j_1 or j_2) to jump to.
- We can construct *probabilistic RAM* programs by adding instructions of the form

  2k + 5/j$_1$/j$_2$; **pjp** j_1 **or** j_2

  which selects with probability $\frac{1}{2}$ one of two lines (j_1 or j_2) to jump to.

We distinguish between deterministic, non-deterministic, and probabilistic *RAM* programs by using the notation *DRAM*, *NRAM*, and *PRAM*, respectively.

Next Up Previous Contents Index

Next: 7.5 Complexity of RAM Programs **Up:** 7. Random Access Machines **Previous:** 7.3 Index Theorem

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 8. Acceptable Programming Systems **Up:** 7. Random Access Machines **Previous:** 7.4 Other Aspects

7.5 Complexity of RAM Programs

Definition 7.3 If P is a deterministic *RAM* program (a *DRAM* program) over Σ_k^* with n inputs and which uses only registers $\mathbf{R}_1,...,\mathbf{R}_r$, then we define the following complexity measures for P.

$$DRMtime_P(\vec{x}^n) = \begin{cases} \sum_{i=1}^{n} \mid x_i \mid + \#\text{ of stmts of } P \text{ executed on input } \vec{x}^n, \\ \text{if } P \text{ halts on it,} \\ \uparrow, \text{ otherwise.} \end{cases}$$

$$DRMspace_P(\vec{x}^n) = \begin{cases} \max \sum_{i=1}^{r} \mid \mathbf{R}_i^t \mid, \forall t \leq DRMtime_P(\vec{x}^n), \text{ if } P \text{ halts,} \\ \uparrow, \text{ otherwise.} \end{cases}$$

where $\mathbf{R}_i^t$ denotes the contents of register $\mathbf{R}_i$ at step t of the computation of P on input $\vec{x}^n$.

Proposition 7.4 The following predicates are primitive recursive:

$$Q_{DRMtime}(p, \vec{x}^n, y) \equiv [DRMtime_p(\vec{x}^n) \leq y]$$

$$Q_{DRMspace}(p, \vec{x}^n, y) \equiv [DRMspace_p(\vec{x}^n) \leq y]$$

Proof: For time complexity, we have

$$Q_{DRMtime}(p, \vec{x}^{\,n}, y) \equiv \exists_{z \leq} (y \dot{-} \sum_{i=1}^{n} |\, x_i \,|_k)[\, \Pi_1^2(sim(p, \langle \vec{x}^{\,n} \rangle_n, z)) = lng(p)].$$

For space complexity, observe that given a fixed amount of space, it is possible for a computation to enter an infinite computation loop within that amount of space. In this case, since the space complexity is still undefined, the predicate $Q_{DRMspace}$ must respond with **False**. Moreover, if the program is ever in the situation where it is about to execute an instruction with a current memory contents that is *identical* to an instruction and memory contents combination that it encountered *earlier*, then clearly it is in such an infinite loop. Thus, the number of distinct instruction-memory combinations is an upper bound on the number of steps a program can execute in an *a priori* given amount of space before it is certain to be in an infinite loop. Given this analysis we now define

$$Q_{DRMspace}(p, \vec{x}^{\,n}, y) \equiv \exists_{z \leq} lng(p) \times k^y \quad [\, \Pi_1^2(sim(p, \langle \vec{x}^{\,n} \rangle_n, z)) = lng(p)$$

$$\wedge \forall_{z_1 \leq} z \quad [\, |\, \Pi_2^2(sim(p, \langle \vec{x}^{\,n} \rangle_n, z_1))\, |_{k+1} + 1 \dot{-} mxr(p) \leq y]],$$

where the term $1 \dot{-} mxr(p)$ is (minus) the number of commas in the internal representation of the contents of the registers of p.

Proposition 7.5 For each *DRAM* program p there exist constants c_1, c_2 such that

$$DRMtime_p(\vec{x}^{\,n}) \leq c_1^{c_2} \times DRMspace_p(\vec{x}^{\,n})$$

$$DRMspace_p(\vec{x}^{\,n}) \leq DRMtime_p(\vec{x}^{\,n})$$

Proof: The first inequality follows from the analysis given preceding the primitive recursive definition of $Q_{DRMspace}$ in Proposition 7.4. The second inequality follows since the only *RAM* instructions which can

increase the space beyond that occupied by the input is the $\mathbf{suc}_a$ instruction, which can only increase it by one symbol.

Proposition 7.6 Let p be any *DLOOP* program over Σ_k^*, and let p' be the equivalent *DRAM* program over Σ_k^* constructed in Proposition 7.1. There exist constants c_1, c_2, and c_3 such that

$$DRMtime_{p'}(\vec{x}^{\,n}) \leq c_1 \times (DLPtime_p(\vec{x}^{\,n}))^{c_2}$$

$$DRMspace_{p'}(\vec{x}^{\,n}) \leq c_3 \times DLPspace_p(\vec{x}^{\,n})$$

Next Up Previous Contents Index

Next: 8. Acceptable Programming Systems **Up:** 7. Random Access Machines **Previous:** 7.4 Other Aspects

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 8.1 General Computational Complexity **Up:** Lecture Notes for CS 2110 Introduction to Theory
Previous: 7.5 Complexity of RAM Programs

8. Acceptable Programming Systems

We now wish to examine the properties of computable functions without getting bogged down with the details of any of the particular models which we have heretofore studied. Therefore, we generalize our notion of (standard) model of computable function.

Definition 8.1 A *programming system* is a listing $\phi_0, \phi_1, \ldots$ (denoted by $\{\phi_i\}$) which includes *all* of the partial recursive functions (of one variable) over $\mathbb{N}$. A *acceptable programming system* is a programming system $\{\phi_i\}$ for which

1. there exists a *universal program unv* such that $\phi_{unv}(i, x) = \phi_i(x)$ for all i and x; and
2. there is a total recursive *S-m-n function* S_m^n such that $\phi_{S_m^n(i,x)}(y) = \phi_i(x, y)$ for all i, m-tuples x, and n-tuples y.

We will abbreviate S_m^n by S whenever it is clear how many arguments it takes.

Theorem 8.1 Let $\{\phi_i\}$ be any acceptable programming system, and let $\{\psi_i\}$ be any programming system. Then, $\{\psi_i\}$ is acceptable if and only if there exist total recursive functions f and g such that for all i, $\phi_{f(i)} = \psi_i$ and $\psi_{g(i)} = \phi_i$.

Proof: Since $\{\phi_i\}$ is acceptable, there exist partial recursive ϕ_{unv} and total recursive S such that

$$\phi_{unv}(i, x) = \phi_i(x)$$

$$\phi_{S(i,x)}(y) = \phi_i(x, y).$$

- **Case ($\Longrightarrow$):**

  Since { ψ_i } is also acceptable, there exist partial recursive ψ_{unv} and total recursive S' such that

$$\psi_{unv}(i, x) = \psi_i(x)$$

$$\psi_{S'(i,x)}(y) = \psi_i(x, y).$$

Now, since { ϕ_i } is a listing of *all* partial recursive functions, there is an index (i.e., program code) e such that $\phi_e = \psi_{unv}$. Then, we define

$$f(i) = S(e, i)$$

so that

$$\psi_i(x) = \psi_{unv}(i, x) = \phi_e(i, x) = \phi_{S(e,i)}(x) = \phi_{f(i)}(x).$$

Similarly, there exists an index e' such that $\psi_{e'} = \phi_{unv}$, and we define $g(i) = S'(e', i)$ so that $\phi_i(x) = \psi_{g(i)}(x)$.

- **Case ($\Longleftarrow$):**

  Suppose f and g are total recursive functions such that

$$\phi_{f(i)} = \psi_i \qquad \text{and} \qquad \psi_{g(i)} = \phi_i$$

Then, we can define the universal function for $\{ \psi_i \}$ by

$$\psi_{unv}(i, x) = \phi_{unv}(f(i), x) = \phi_{f(i)}(x) = \psi_i(x).$$

Finally, we define the function $S'$for $\{ \psi_i \}$ by

$$S'(i, x) = g(S(f(i), x))$$

so that

$$\psi_{S'(i,x)}(y) = \psi_{g(S(f(i),x))}(y) = \phi_{S(f(i),x)}(y) = \phi_{f(i)}(x, y) = \psi_i(x, y).$$

Definition 8.2 A *program transformation* is any total recursive function whose domain and range are programs (i.e., indices) for partial recursive functions.

Observe that this definition is vacuous in the sense that every number can be interpreted as a program. However, it is useful for its intensional aspect.

Next Up Previous Contents Index

Next: 8.1 General Computational Complexity **Up:** Lecture Notes for CS 2110 Introduction to Theory
Previous: 7.5 Complexity of RAM Programs

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 8.2 Algorithmically Unsolvable Problems **Up:** 8. Acceptable Programming Systems **Previous:** 8. Acceptable Programming Systems

8.1 General Computational Complexity

One of the most important behavioral aspects of a computation is the complexity of the computation, i. e., the amount of computation resources used during that computation. It will play a key role in many of the proofs which follow, so we now define a general notion of computational complexity which is suitable for our generalized model of computability.

Definition 8.3 Let $\{ \phi_i \}$ be any acceptable programming system. A listing of partial functions $\{ \Phi_i \}$ is a *computational complexity measure* for $\{ \phi_i \}$ if it satisfies:

1. **dom** ϕ_i = **dom** Φ_i, i.e., for all i, x, $\phi_i(x) \downarrow \iff \Phi_i(x) \downarrow$;
2. $\Phi_i(x) \leq y$ is a recursive predicate in i, x, and y.

Clearly, the complexity measures defined for *LOOP* and *RAM* programs satisfy the first condition of a general computational complexity measre. It is also clear from Proposition 7.4 (and its analog for *LOOP* programs) that the second condition is satisfied by these complexity measures as well.

Proposition 8.2 If $\{ \Phi_i \}$ is a computational complexity measure for $\{ \phi_i \}$, then Φ_i is a partial recursive function for each i.

Proof:

$$\Phi_i(x) = \min y[\Phi_i(x) \leq y].$$

Proposition 8.3 There is a program transformation τ such that $\phi_{\tau(i)} = \Phi_i$.

Proof: Define

$$h(i, x) = \min y[\Phi_i(x) \leq y]$$
$$= \Phi_i(x)$$

Let e be a program for h, i.e., $\phi_e(i, x) = h(i, x)$, then by the S-m-n function which exists for $\{\phi_i\}$,

$$\phi_{S(e,i)}(x) = \phi_e(i, x) = h(i, x) = \Phi_i(x)$$

Therefore, we define $\tau(i) = S(e, i)$, so that $\phi_{\tau(i)} = \Phi_i$.

♠ Nearly all program transformations which we will encounter will be defined in this way using the S-m-n function.

Theorem 8.4 (Recursive Relatedness of Complexity Measures) Let $\{\phi_i\}$ and $\{\psi_i\}$ be acceptable programming systems, and let g be a total recursive function such that $\phi_i = \psi_{g(i)}$ for all i. Let $\{\Phi_i\}$ and $\{\Psi_i\}$ be computational complexity measures for $\{\phi_i\}$ and $\{\psi_i\}$, respectively. Then, there exists a total recursive function r such that for all i and for all $x \geq i$,

$$\Phi_i(x) \leq r(x, \Psi_{g(i)}(x)) \qquad \text{and}$$
$$\Psi_{g(i)}(x) \leq r(x, \Phi_i(x))$$

Proof: Define the total recursive function r as follows:

$$r(x, z) = \max j \leq x \{ \Phi_j(x), \Psi_{g(j)}(x) : \Phi_j(x) \leq z \quad \text{or} \quad \Psi_{g(j)}(x) \leq z \}$$

Then, for all $x \geq i$,

$$\begin{aligned} r(x, \Psi_{g(i)}(x)) &= \max j \leq x \{ \Phi_j(x), \Psi_{g(j)}(x) : \Phi_j(x) \leq \Psi_{g(i)}(x) \\ &\qquad \text{or} \quad \Psi_{g(j)}(x) \leq \Psi_{g(i)}(x) \} \\ &\geq \max \{ \Phi_i(x), \Psi_{g(i)}(x) \} \\ &\geq \Phi_i(x) \end{aligned}$$

Similarly, $r(x, \Phi_i(x)) \geq \Psi_{g(i)}(x)$, for all $x \geq i$.

We fix some arbitrary acceptable programming system $\{ \phi_i \}$ and computational complexity measure $\{ \Phi_i \}$ for it which we will use from now on.

Proposition 8.5 There is a program transformation g such that for all x, **ran** ϕ_x = **dom** $\phi_{g(x)}$.

Proof: Define the partial recursive function ψ by,

$$\psi(x, y) = \min z[\Phi_x(\Pi_1^2(z)) \leq \Pi_2^2(z) \text{ and } \phi_x(\Pi_1^2(z)) = y]$$

First, observe the ψ is indeed partial recursive, since we can write $\phi_{unv}(\tau(x), y)$ for $\Phi_x(y)$ (where τ is the program transformation of Proposition 8.3), and $\phi_{unv}(x, y)$ for $\phi_x(y)$. Next we have

$$\psi(x, y)\downarrow \iff \exists \langle z_1, z_2 \rangle_2 \, [\Phi_x(z_1) \leq z_2 \text{ and } \phi_x(z_1) = y]$$
$$\iff y \in \textbf{ran } \phi_x$$

Let i be a program for ψ, so $\phi_i = \psi$, and define $g(x) = S(i, x)$, so that $\phi_{g(x)}(y) = \psi(x, y)$. Then,

$$y \in \textbf{dom } \phi_{g(x)} \iff \phi_{g(x)}(y)\downarrow \iff y \in \textbf{ran } \phi_x.$$

Therefore, **ran** ϕ_x = **dom** $\phi_{g(x)}$.

Proposition 8.6 There is a program transformation h such that for all x, **ran** $\phi_{h(x)}$ = **dom** ϕ_x.

Proof: Define the partial recursive function ψ by

$$\psi(x, 0) = \Pi_1^2(\min z[\Phi_x(\Pi_1^2(z)) \leq \Pi_2^2(z)])$$

$$\psi(x, y+1) = \begin{cases} \psi(x, y), & \text{if } \Phi_x(\Pi_1^2(y+1)) > \Pi_2^2(y+1) \\ \Pi_1^2(y+1), & \text{otherwise.} \end{cases}$$

Let *i* be such that $\phi_i = \psi$, and define $h(x) = S(i, x)$, so that $\phi_{h(x)}(y) = \psi(x, y)$. We consider two cases:

- **Case 1:**

  dom $\phi_x = \emptyset$.

  In this case, since **dom** Φ_x = **dom** ϕ_x, we see that $\psi(x, y) \uparrow$ for all y, so that **ran** $\phi_{h(x)}$ = $\emptyset$ = **dom** ϕ_x.

- **Case 2:**

  dom $\phi_x \neq \emptyset$.

  Observe first that since **dom** $\phi_x \neq \emptyset$, the function ψ must be total recursive. For each $y \in$ **dom** ϕ_x, clearly $\psi(x, \langle y, \Phi_x(y) \rangle_2) = y$, so that **dom** $\phi_x \subseteq$ **ran** $\phi_{h(x)}$. On the other hand, if $\psi(x, y) = z \neq \psi(x, y - 1)$ (including the case $y = 0$), then we have that $\Phi_x(z) \leq \Pi_2^2(y)$, so that $\phi_x(z) \downarrow$ and **ran** $\phi_{h(x)} \subseteq$ **dom** ϕ_x.

Notation 8.4 For any predicate *P*, we write $\overset{\infty}{\exists} x\ P(x)$ (or $P(x)$ **i.o.**) if there exist *infinitely many* numbers *x* for which $P(x)$ is true. We also write $\overset{\infty}{\forall} x\ P(x)$ (or $P(x)$ **a.e.**) if for all *but finitely many* numbers *x* $P(x)$ is true. The expressions **i.o.** and **a.e.** are abbreviations for ``infinitely often" and ``almost everywhere", respectively.

Theorem 8.7 For any total recursive function *t* there exists a total recursive function *f* such that if $\phi_i = f$, then for all $x \geq i$, $\Phi_i(x) > t(x)$.

Proof: Proof is by diagonalization using Proverb 5.1. Define the total recursive function

$$f(x) = \max\{\phi_j(x) + 1 : j \leq x \text{ and } \Phi_j(x) \leq t(x)\}.$$

Thus, if $\phi_i = f$ and $x \geq i$, then $\Phi_i(x) > t(x)$, since otherwise we would have

$$\begin{aligned}\phi_i(x) = f(x) &= \max\{\phi_j(x) + 1 : j \leq x \wedge \Phi_j(x) \leq t(x)\} \\ &\geq \phi_i(x) + 1.\end{aligned}$$

Thus, we see that there are functions which are functions which are **a.e.** difficult to compute with respect to any given complexity measure. We observe that we cannot improve this result to *everywhere* difficult to compute, since we can always ``speed-up" the computation of any function on finitely many of its inputs by building in a table with the corresponding outputs and then computing the function on those inputs by table lookup.

Next Up Previous Contents Index

Next: 8.2 Algorithmically Unsolvable Problems **Up:** 8. Acceptable Programming Systems **Previous:** 8. Acceptable Programming Systems

Bob Daley

2001-11-28

8.2 Algorithmically Unsolvable Problems

Theorem 8.8 (Unsolvability of the Halting Problem) The function f such that for all x and y,

$$f(x,y) = \begin{cases} 1, & \text{if } \phi_x(y)\downarrow \\ 0, & \text{if } \phi_x(y)\uparrow . \end{cases}$$

is *not* recursive.

Proof: Define the total function $g(x) = f(x,x)$, and the partial function ψ by

$$\psi(x) = \begin{cases} 0, & \text{if } g(x) = 0 \\ \uparrow, & \text{if } g(x) = 1. \end{cases}$$

If ψ is partial recursive, then there is a program i such that $\psi = \phi_i$, but then

$$\psi(i) = \phi_i(i) = 0 \iff g(i) = 0 \iff \phi_i(i)\uparrow$$

which is a contradiction. Therefore, ψ cannot be partial recursive, so that g and hence f cannot be total recursive.

♠ The following set $\mathbb{H}$ (referred to as the ``Halting Problem'') plays an important role in undecibability results:

$$\mathbb{H} = \{x : \phi_x(x) \downarrow\}.$$

Corollary 8.9 The set $\mathbb{H}$ (and its complement $\overline{\mathbb{H}}$) is not recursive.

We will be able to show that there are many such problems which are algorithmically unsolvable. One of the major techniques is to reduce one problem to another, i.e., to show that if one problem were solvable then the other would also be solvable.

Definition 8.5 Let $X, Y \subseteq \mathbb{N}$. We say that X is *many-one reducible* to Y (denoted by $X \leq_m Y$), if there is a total recursive function f such that for all x, $x \in X \iff f(x) \in Y$. We write $X \equiv_m Y$ whenever $X \leq_m Y$ and $Y \leq_m X$.

Proposition 8.10 If Y is a recursive set and $X \leq_m Y$, then X is also recursive.

Proof: Let f be a total recursive function such that $x \in X \iff f(x) \in Y$. Then, the characteristic function of X is given by $\chi_X = \chi_Y \circ f$, i.e.,

$$\chi_X(x) = 1 \iff \chi_Y(f(x)) = 1.$$

Proposition 8.11 The following sets are not recursive:

$$\mathbb{FIN} = \{x : \textbf{dom } \phi_x \text{ is finite}\}$$

$$\mathbb{TOT} = \{x : \phi_x \text{ is total}\}$$

Proof: Define the partial recursive function ψ by

$$\psi(x, y) = \phi_{unv}(x, x) \dot{-} \phi_{unv}(x, x).$$

Then,

$$\psi(x, y) = \begin{cases} 0, & \text{if } \phi_x(x) \downarrow \text{ (i.e., } x \in \mathbb{H}) \\ \uparrow, & \text{if } \phi_x(x) \uparrow \text{ (i.e., } x \notin \mathbb{H}) \end{cases}$$

Let i be such that $\phi_i = \psi$ and define the total recursive function f by $f(x) = S(i, x)$, so that $\phi_{f(x)}(y) = \psi(x, y)$. Let ω be the everywhere undefined partial recursive function. Clearly,

$$\phi_{f(x)} = \begin{cases} N, & \text{if } x \in \mathbb{H} \\ \omega, & \text{if } x \notin \mathbb{H}. \end{cases}$$

Therefore, $x \in \mathbb{H} \iff f(x) \in \mathbb{TOT}$, hence $\mathbb{H} \leq_m \mathbb{TOT}$. By Proposition 8.10 if $\mathbb{TOT}$ were a recursive set, then so would $\mathbb{H}$ be recursive, contradicting Corollary 8.9. Similarly, $x \notin \mathbb{H} \iff f(x) \in \mathbb{FIN}$, and $\overline{\mathbb{H}} \leq_m \mathbb{FIN}$, so if $\mathbb{FIN}$ were recursive so would $\overline{\mathbb{H}}$ be, again a contradiction.

Definition 8.6 For any class C of partial recursive functions, we define the set of programs $\mathbf{P}_C$ (called an *index set*) for these functions by

$$\mathbf{P}_C = \{x : \phi_x \in C\}.$$

Theorem 8.12 (Rice's Theorem) $\mathbf{P}_C$ is recursive if and only if either $\mathbf{P}_C = \emptyset$ of $\mathbf{P}_C = \mathbb{N}$.

Proof: Clearly, $\emptyset$ and $\mathbb{N}$ are recursive sets. So, suppose that $\mathbf{P}_C \neq \emptyset$ and $\mathbf{P}_C \neq \mathbb{N}$. Let ω be the everywhere undefined partial recursive function, and assume without loss of generality that $\omega \in C$. Since $\mathbf{P}_C \neq \mathbb{N}$, there is some partial recursive function ψ such that $\psi \notin C$. Let i be such that $\phi_i = \psi$, and define the partial recursive function

$$\theta(x, y) = \phi_{unv}(i, y) + (\phi_{unv}(x, x) \dot{-} \phi_{unv}(x, x)).$$

Then,

$$\theta(x, y) = \begin{cases} \psi(y), & \text{if } x \in \mathbb{H} \\ \uparrow, & \text{if } x \notin \mathbb{H}. \end{cases}$$

Let j be such that $\phi_j = \theta$, and define the program transformation f by $f(x) = S(j, x)$, so that $\phi_{f(x)}(y) = \theta(x, y)$. Then,

$$\phi_{f(x)} = \begin{cases} \psi, & \text{if } x \in \mathbb{H} \\ \omega, & \text{if } x \notin \mathbb{H}. \end{cases}$$

Therefore, $x \in \overline{\mathbb{H}} \iff f(x) \in \mathbf{P}_C$, so that $\overline{\mathbb{H}} \leq_m \mathbf{P}_C$, and so $\mathbf{P}_C$ cannot be recursive.

♠ Rice's Theorem says in essence that there are no *non-trivial* apsects of the behavior of a program which are algorithmically determinable given only the text of the program. By *trivial* we mean that either *no* programs have that behavior or *all* programs have that behavior. As such, Rice's Theorem represents an extremely severe limitation on the power of algorithms.

Next Up Previous Contents Index

Next: 9. Recursively Enumerable Sets **Up:** 8. Acceptable Programming Systems **Previous:** 8.1 General Computational Complexity

Bob Daley
2001-11-28

Next Up Previous Contents Index

Next: 10. Recursion Theorem **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 8.2 Algorithmically Unsolvable Problems

9. Recursively Enumerable Sets

Definition 9.1 A set X is *recursively enumerable* (or **r.e.**) if and only if $X =$ **ran** ϕ, for some partial recursive function ϕ.

By Propositions 8.5 and 8.6 we have

Corollary 9.1 A set X is recursively enumerable if and only if $X =$ **dom** ϕ, for some partial recursive function ϕ.

Corollary 9.2 A set X is recursively enumerable if and only if either $X = \emptyset$ or $X =$ **ran** f for some total recursive function f.

♠ Thus, we see that the class of sets *generated* by partial recursive functions is *identical* to the class of sets *accepted* by partial recursive functions.

Proposition 9.3 A set is recursive if and only if both it and its complement are recursively enumerable.

Proof: Since $\emptyset$ is clearly recursive and **r.e.**, it suffices to consider only non-empty sets.

- **($\Longrightarrow$):**
  Since the recursive sets are closed under complementation, it suffices to show that every non-empty recursive set is recursively enumerable. Let X be recursive and let $y \in X$. Then, X is enumerated by the function

$$f(x) = \begin{cases} x, & \text{if } \chi_X(x) = 1 \\ y, & \text{if } \chi_X(x) = 0. \end{cases}$$

($\Longleftarrow$):

Suppose X is non-empty and enumerated by the total recursive function f and that $\overline{X}$ is non-empty and enumerated by the total recursive function g. Then,

$$\chi_X(x) = \begin{cases} 1, & \text{if } f(\min\, y[f(y) = x \text{ or } g(y) = x]) = x \\ 0, & \text{otherwise.} \end{cases}$$

Proposition 9.4 a) $\mathbb{H}$ is recursively enumerable.

b) $\overline{\mathbb{H}}$ is not recursively enumerable.

Proof: $\mathbb{H}$ = **dom** ψ, where $\psi(x) = \phi_x(x)$. Since $\mathbb{H}$ is **r.e.**, if $\overline{\mathbb{H}}$ were **r.e.**, then by Proposition 9.3 $\mathbb{H}$ would be recursive, which would contradict Corollary 8.9.

Proposition 9.5 If Y is recursively enumerable and $X \leq_m Y$, then X is recursively enumerable.

Proof: Let Y = **dom** ψ, for some partial recursive function ψ, and let f be a total recursive function such that $x \in X \iff f(x) \in Y$. Define $\phi = \psi \circ f$, so that

$$\phi(x)\downarrow \iff \psi(f(x))\downarrow \iff f(x) \in Y \iff x \in X.$$

Hence, X is **r.e.**

Definition 9.2 A set Z is called *complete* for the class of recursively enumerable sets *with respect to the reducibility* $\leq_m$ (called *many-one complete*) if and only if Z is **r.e.** and for all **r.e.** sets X, $X \leq_m Z$.

Proposition 9.6 $\mathbb{H}$ is complete for the class of recursively enumerable sets with respect to $\leq_m$.

Proof: Clearly, $\mathbb{H}$ is **r.e.**. Now, let X be any **r.e.** set and let x be such that $X =$ **dom** ϕ_x. Define the program transformation f by

$$\phi_{f(i,j)}(z) = \phi_i(j).$$

Then,

$$\begin{aligned} y \in X &\iff \phi_x(y)\downarrow \iff \phi_{f(x,y)}(z)\downarrow \text{ for all } z \\ &\iff \phi_{f(x,y)}(z)\downarrow \text{ for some } z \\ &\iff \phi_{f(x,y)}(f(x,y))\downarrow \end{aligned}$$

Define $g(y) = f(x, y)$. Then, $y \in X \iff g(y) \in \mathbb{H}$, so $X \leq_m \mathbb{H}$.

Proposition 9.7 The set $\mathbb{FIN}$ is not recursively enumerable.

Proof: Let the partial recursive ψ and total recursive f be as defined in Proposition 8.11. Then,

$$\textbf{dom}\ \phi_{f(x)} = \begin{cases} \mathbb{N}, & \text{if } x \in \mathbb{H} \\ \emptyset, & \text{if } x \notin \mathbb{H}. \end{cases}$$

Therefore, $x \in \overline{\mathbb{H}} \iff f(x) \in \mathbb{FIN}$, so $\overline{\mathbb{H}} \leq_m \mathbb{FIN}$, and by Proposition 9.5, if $\mathbb{FIN}$ were **r.e.**, then so would $\overline{\mathbb{H}}$ be, which contradicts Proposition 9.4.

Definition 9.3 A function is called *finite* if and only if it has a finite domain.

- Thus, if C is the class of all *finite functions*, then $\mathbf{P}_C = \mathbb{FIN}$.
- We can effectively enumerate the class of finite functions as follows: Since each finite function f consists of only finitely many pairs $(x_1, y_1),\ldots, (x_n, y_n)$, we can code f by $\langle \langle x_1, y_1 \rangle_2, \ldots, \langle x_n, y_n \rangle_2 \rangle_n$. Next, we define the recursive function ψ by

$$\psi(z, x) = \begin{cases} y_j, & \text{if } z = \langle \langle x_1, y_1 \rangle_2, \ldots, \langle x_n, y_n \rangle_2 \rangle_n \\ & \quad \text{and } x = x_j \text{ and } 1 \leq j \leq n \\ \uparrow, & \text{otherwise.} \end{cases}$$

  Let i be such that $\phi_i = \psi$, and let $\psi_z = \phi_{S(i,z)}$. Then, for any finite function f with code z, $\psi(z, x) = f(x)$, and hence $\psi_z = f$. Also, if z does not code any finite function, then $\psi_z = \omega$, the everywhere undefined partial recursive function (which is a finite function). Thus, $\{\psi_z\}$ is an effective enumeration of the class of all finite functions.
- We fix $\{\psi_i\}$ as the above effective enumeration of the class of all finite functions.

♠ Observe that there is a *very important* distinction to be made between effectively enumerating a class *C* of *functions*, and effectively enumerating the class $\mathbf{P}_C$ of all *programs* for those functions. To enumerate *C* we need only enumerate *one program* for each function in *C*. Thus, the effective enumerability of the class of all finite functions does *not* contradict Proposition 8.11.

We now consider two lemmas which are very useful for demonstrating that a set $\mathbf{P}_C$ is not **r.e.**, where *C* is a class of partial recursive functions.

Lemma 9.8 (Closure Under Finite Subfunctions) If $\mathbf{P}_C$ is **r.e.** and $\phi \in C$, then there is some finite function $\psi_z \in C$ such that $\psi_z \subseteq \phi$.

Proof: Let $\mathbf{P}_C$ be **r.e.**, let $\phi \in C$, and define a program transformation *g* such that

$$\phi_{g(x)}(y) = \begin{cases} \phi(y), & \text{if } \Phi_x(x) > y \\ \uparrow, & \text{if } \Phi_x(x) \leq y. \end{cases}$$

Suppose ϕ has no finite subfunctions which also belong to *C*. If $x \notin \mathbb{H}$, then $\phi_{g(x)} = \phi$, so $g(x) \in \mathbf{P}_C$. If $x \in \mathbb{H}$, then $\phi_{g(x)}$ is a finite subfunction of ϕ (since $\phi_{g(x)}(y) \uparrow$ for all $y \geq \Phi_x(x)$), so $g(x) \notin \mathbf{P}_C$. Thus, $x \in \overline{\mathbb{H}} \iff g(x) \in \mathbf{P}_C$, and hence $\overline{\mathbb{H}} \leq_m \mathbf{P}_C$. But then, since $\mathbf{P}_C$ is **r.e.**, $\overline{\mathbb{H}}$ is **r.e.**, which is a contradiction.

Therefore, ϕ must contain some finite subfunction $\psi_z \subseteq \phi$, which also belongs to *C*.

Corollary 9.9 The set $\mathbb{TOT}$ is not recursively enumerable.

Lemma 9.10 (Closure Under Superfunctions) If $\mathbf{P}_C$ is recursively enumerable and $\phi \in C$, then for any partial recursive function ψ if $\phi \subseteq \psi$, then $\psi \in C$.

Proof: Let $\mathbf{P}_C$ be **r.e.** and let $\phi_i \in C$, and suppose that ψ is a partial recursive function for which $\phi_i \subseteq \psi$. Define the partial recursive function θ by

$$\theta(x, y) = \min z[\Phi_x(x) \leq z \text{ or } \Phi_i(y) \leq z]$$
$$= \min\{\Phi_x(x), \Phi_i(y)\}.$$

Define the program transformation h such that

$$\phi_{h(x)}(y) = \begin{cases} \phi_i(y), & \text{if } \theta(x,y)\downarrow \text{ and } \Phi_x(x) > \theta(x,y) \\ \psi(y), & \text{if } \theta(x,y)\downarrow \text{ and } \Phi_x(x) \leq \theta(x,y) \\ \uparrow, & \text{if } \theta(x,y)\uparrow . \end{cases}$$

Assume that $\psi \notin C$. We claim that

$$\phi_{h(x)} = \begin{cases} \phi_i, & \text{if } x \notin \mathbb{H} \\ \psi, & \text{if } x \in \mathbb{H}. \end{cases}$$

Presuming the claim is true, we have $x \in \overline{\mathbb{H}} \iff h(x) \in \mathbf{P}_C$, so that $\overline{\mathbb{H}} \leq_m \mathbf{P}_C$, so $\mathbf{P}_C$ can't be **r.e.**, which is a contradiction.

To show that claim first suppose that $x \in \overline{\mathbb{H}}$, then $\Phi_x(x) \uparrow$.

if $\phi_i(y) \downarrow$, then $\theta(x, y) \downarrow$ and $\Phi_x(x) > \theta(x, y)$;

if $\phi_i(y) \uparrow$, then $\theta(x, y) \uparrow$,

so that in either case $\phi_{h(x)}(y) = \phi_i(y)$.

Suppose on the other hand that $x \in \mathbb{H}$, then $\Phi_x(x) \downarrow$ and $\theta(x, y) \downarrow$.

if $\phi_i(y) \downarrow$, then since $\phi_i \subseteq \psi$, $\psi(y) \downarrow = \phi_i(y)$, so in either case $\phi_{h(x)}(y) = \psi(y)$;

if $\phi_i(y) \uparrow$, then $\Phi_x(x) \leq \theta(x, y)$, so $\phi_{h(x)}(y) = \psi(y)$.

Thus, the claim and hence the lemma is proved.

Corollary 9.11 The set $\overline{\mathbb{TOT}} = \{x : \phi_x \text{ is not total}\}$, the complement of $\mathbb{TOT}$, is not recursively enumerable.

Proof: Any total function extends the everywhere undefined function ω, which belongs to $\overline{\mathbb{TOT}}$.

Theorem 9.12 (Rice's Theorem for R.E. Sets) Let C be any class of partial recursive functions. Then, $\mathbf{P}_C = \{x : \phi_x \in C\}$ is recursively enumerable if and only if there is some **r.e.** set Z such that for all x, $\phi_x \in C \iff \exists z \in Z, \psi_z \subseteq \phi_x$.

Proof:

($\Longleftarrow$):

Let Z be a (non-empty) **r.e.** set such that

$$\mathbf{P}_C = \{x : \psi_z \subseteq \phi_x \text{ for some } z \in Z\}.$$

Define the partial recursive function θ by

$$\theta(w) = \begin{cases} \Pi_1^2(w), & \text{if } \psi_{f(\Pi_2^2(w))} \subseteq \phi_{\Pi_1^2(w)} \\ \uparrow, & \text{otherwise.} \end{cases}$$

where the total recursive function $f$is such that Z= **ran** f. Observe, that $\psi_j \subseteq \phi_i$, if true, will eventually be discovered, since it entails checking that ϕ_i has the proper outputs on a finite set of inputs. Let $x \in \mathbf{P}_C$. Then, $\psi_z \subseteq \phi_x$ for some $z \in Z$. Let $y$be such that $f(y) = z$, so that $\theta(\langle x, y \rangle_2) = x$. Also, if $\theta(\langle x, y \rangle_2) \downarrow$, then $\theta(\langle x, y \rangle_2) = x$and $\psi_{f(y)} \subseteq \phi_x$, so $x \in \mathbf{P}_C$. Therefore, $\mathbf{P}_C = \mathbf{ran}\ \theta$.

($\Longrightarrow$):

Let i be such that $\mathbf{P}_C = \mathbf{dom}\ \phi_i$, and let g be a total recursive function such that $\phi_{g(z)} = \psi_z$ for all z. Define the recursively enumerable set Z by Z = **dom** ($\phi_i \circ g$), so that

$$z \in Z \iff \phi_i(g(z)) \downarrow \iff g(z) \in \mathbf{P}_C \iff \psi_z \in C.$$

Suppose $\phi_x \in C$. Then by Lemma 9.8there is some $z \in Z$such that $\psi_z \subseteq \phi_x$. On the other hand, suppose $\psi_z \subseteq \phi_x$ for some $z \in Z$. Then, $\psi_z \in C$and by Lemma 9.10 $\phi_x \in C$.

Next Up Previous Contents Index

Next: 10. Recursion Theorem **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 8.2 Algorithmically Unsolvable Problems

Bob Daley

2001-11-28

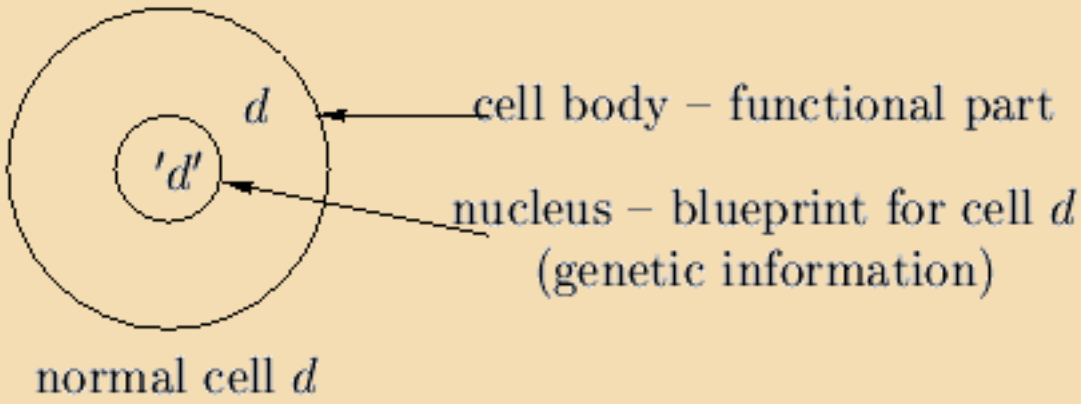
10. Recursion Theorem

- **Special Case:**

  There exists a program e such that $\phi_e(x) = e$.

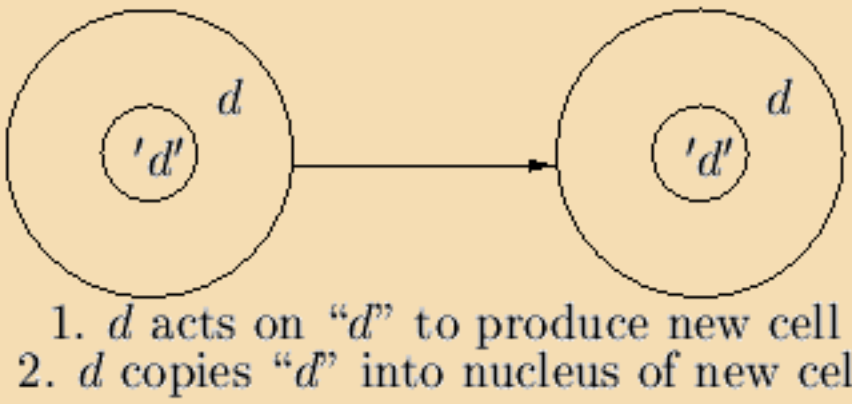
- **Cell Analogy:**

Figure 10.1: Cell Analogy



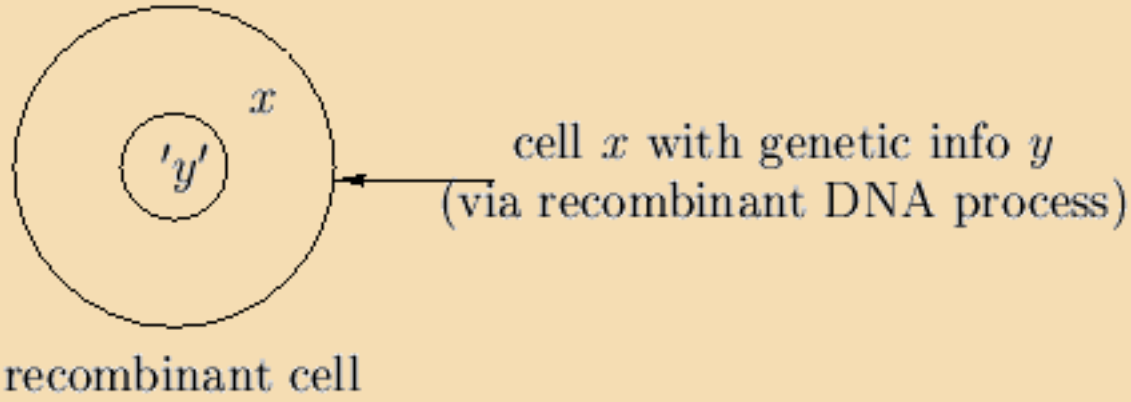
- **Replication Process:**

Figure 10.2: Replication Process



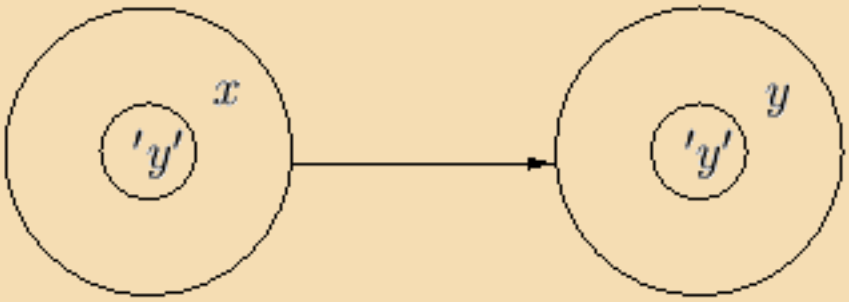
- **General Cell:**

Figure 10.3: General Cell



- **General Replication:**

Figure 10.4: General Replication



Observation 10.1

Cell x with genetic information y
$\approx$ ``program'' $x$ with ``data'' y
$\approx$ $S(x, y)$

Mimicing replication in a general cell we find that for some program x (which does only replication)

$$\phi_x(y, z) = S(y, y)$$

so that by the S-m-n function,

$$\phi_{S(x,y)}(z) = S(y, y).$$

Now, let $y = x$, so

$$\phi_{S(x,x)}(z) = S(x, x).$$

Finally, letting $e = S(x, x)$, we have for all z

$$\phi_e(z) = e.$$

♠ The program e is program x with data x.

Theorem 10.1 (General Form of Recursion Theorem) For every partial recursive function $\psi : \mathbb{N}^2 \longrightarrow \mathbb{N}$ there is a program e such that for all x, $\psi(e, x) = \phi_e(x)$.

Proof: Let i be a program such that

$$\phi_i(y, x) = \psi(S(y, y), x).$$

Then, $\phi_{S(i,y)}(x) = \psi(S(y, y), x)$. Let $y = i$ and $e = S(i, i)$, then we have

$$\phi_e(x) = \phi_{S(i,i)}(x) = \psi(S(i, i), x) = \psi(e, x).$$

Theorem 10.2 (Fixed-Point Form of Recursion Theorem) For every program transformation $f: \mathbb{N} \longrightarrow \mathbb{N}$ there is a program e such that $\phi_{f(e)} = \phi_e$.

Proof: Let $f: \mathbb{N} \longrightarrow \mathbb{N}$ be a total recursive function and define the partial recursive function ψ by

$$\psi(y, x) = \phi_{f(y)}(x).$$

Then, by Theorem 10.1 there exists a program e such that $\phi_e(x) = \psi(e, x) = \phi_{f(e)}(x)$.

Proposition 10.3 For every program transformation $f: \mathbb{N}^3 \longrightarrow \mathbb{N}$, there exists a program transformation $g: \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that $\phi_{f(i,j,g(i,j))} = \phi_{g(i,j)}$ for all i and j.

Proof: Define the partial recursive function ψ by

$$\psi(y, i, j, x) = \phi_{f(i,j,S(y,i,j))}(x).$$

By Theorem 10.1 there exists a program e such that $\phi_e(i, j, x) = \psi(e, i, j, x)$. Let $g(i, j) = S(e, i, j)$. Then,

$$\begin{aligned}\phi_{g(i,j)}(x) = \phi_{S(e,i,j)}(x) = \phi_e(i, j, x) &= \psi(e, i, j, x) \\ &= \phi_{f(i,j,S(e,i,j))}(x) \\ &= \phi_{f(i,j,g(i,j))}(x)\end{aligned}$$

As a consequence of the General Form of the Recursion Theorem we will, whenever we need to, assume that programs which we construct have copies of themselves built into them.

Next Up Previous Contents Index

Next: 10.1 Applications of the Recursion Theorem **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 9. Recursively Enumerable Sets

Bob Daley

2001-11-28

10.1 Applications of the Recursion Theorem

Corollary 10.4 The set $\mathbb{TOT}$ is not recursively enumerable.

Proof: Suppose that $\mathbb{TOT}$ is **r.e.**, and let f be a total recursive function such that **ran** f = $\mathbb{TOT}$.

Define the partial recursive function ψ by

$$\psi(x, y) = \begin{cases} 0, & \text{if } \forall z \leq y \ f(z) \neq x \\ \uparrow, & \text{if } \exists z \leq y \ f(z) = x. \end{cases}$$

By the Recursion Theorem there is a program e such that $\psi(e, y) = \phi_e(y)$, so that

$$\phi_e(y) = \begin{cases} 0, & \text{if } \forall z \leq y \ f(z) \neq e \\ \uparrow, & \text{if } \exists z \leq y \ f(z) = e. \end{cases}$$

Suppose that ϕ_e is total. Then, $e \in \mathbb{TOT}$ and so $e \in$ **ran** f, but by the definition of ψ, we see that $\forall z \, f(z) \neq e$, which is a contradiction. On the other hand, suppose that ϕ_e is not total. Then, $e \notin$ **ran** f, but again from the definition of ψ we see that $\exists z \, f(z) = e$, which again is a contradiction. Therefore, no such function f can exist.

Proposition 10.5 (Inefficiency Lemma) There exists a program transformation $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such

that

$$\textbf{dom}\ \phi_{g(i,j)} = \textbf{dom}\ \phi_i \cap \textbf{dom}\ \phi_j$$

and

$$\forall_x \in \textbf{dom}\ \phi_{g(i,j)}\,[\,\phi_{g(i,j)}(x) = \phi_i(x) \wedge \Phi_{g(i,j)}(x) > \phi_j(x)].$$

Proof: Define the program transformation $f : \mathbb{N}^3 \longrightarrow \mathbb{N}$ by

$$\phi_{f(i,j,k)}(x) = \begin{cases} \phi_k(x) + 1, & \text{if } \Phi_k(x) \leq \phi_j(x) \\ \phi_i(x), & \text{otherwise.} \end{cases}$$

By Proposition 10.3 there exists a program transformation $g : \mathbb{N}^2 \longrightarrow \mathbb{N}$ such that $\phi_{g(i,j)} = \phi_{f(i,j,g(i,j))}$. Then, we have

$$\phi_{g(i,j)}(x) = \begin{cases} \phi_{g(i,j)}(x) + 1, & \text{if } \Phi_{g(i,j)}(x) \leq \phi_j(x) \\ \phi_i(x), & \text{otherwise.} \end{cases}$$

Therefore, if $\phi_{g(i,j)}(x)\downarrow$, then $\phi_{g(i,j)}(x) = \phi_i(x)$ and $\Phi_{g(i,j)}(x) > \phi_j(x)$.

- 10.1.1 Machine Learning

- 10.1.2 Speed-Up Theorem

Next Up Previous Contents Index

Next: 10.1.1 Machine Learning **Up:** 10. Recursion Theorem **Previous:** 10. Recursion Theorem

Bob Daley

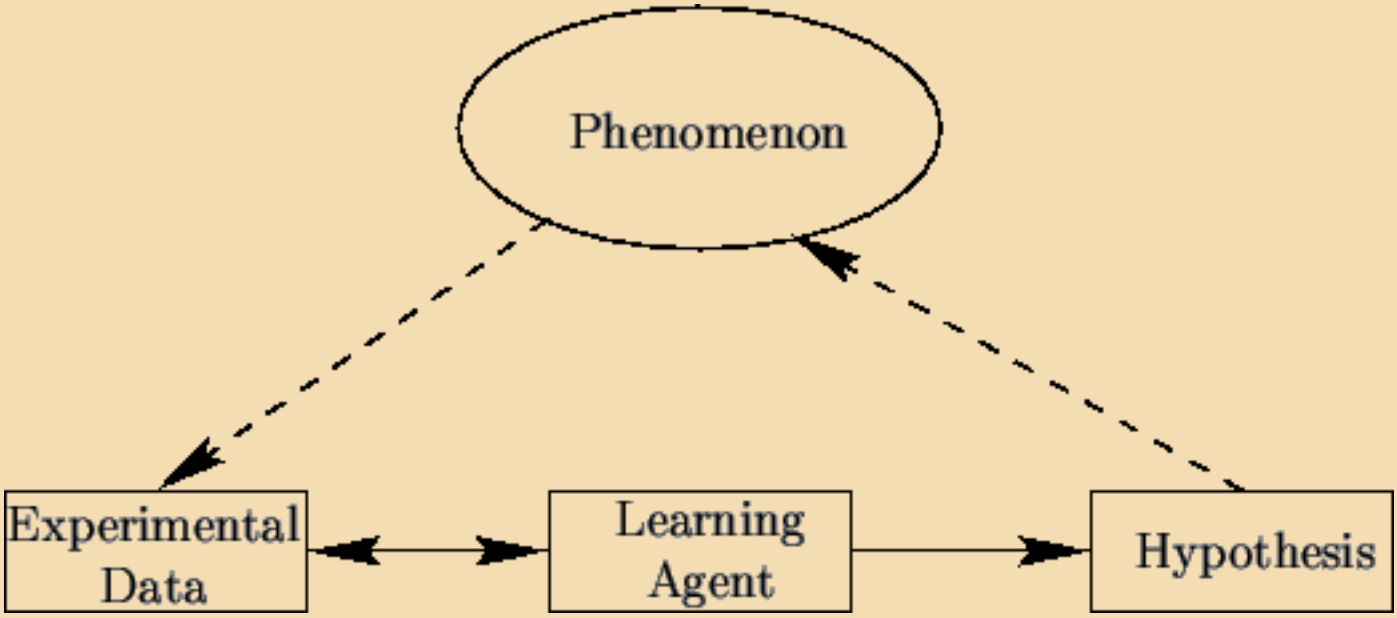
2001-11-28

Next Up Previous Contents Index

Next: 10.1.2 Speed-Up Theorem **Up:** 10.1 Applications of the Recursion Theorem **Previous:** 10.1 Applications of the Recursion Theorem

10.1.1 Machine Learning

Figure 10.5: Learning By Example Scenario



- We view a learning algorithm (or inductive inference machine) as a total recursive function M which takes as input a finite portion of the graph of some (total recursive) function f and produces as output (the code of) some program p which is its conjecture for f.
- We say that M learns a function f *in the limit* if eventually it converges to a fixed conjecture which is a correct program for f.

- We now formalize these notions: If $\{x_i\}$ denotes a sequence of numbers $x_0, x_1, \ldots$, then

$$\lim_{n \to \infty} x_n = x$$

means that $\exists m \, \forall n \geq m \; x_n = x$ (or equivalently $\overset{\infty}{\forall} n \; x_n = x$). In this case we say that $\{x_i\}$ *converges in the limit to* x.

Given a total function $f: \mathbb{N} \longrightarrow \mathbb{N}$, we denote by $f \mid n$, the finite subfunction of f consisting of f restricted to the set {0, 1, 2,..., n}, and code it by $\langle f(0),..., f(n) \rangle_n$.

Definition 10.1 We say that a total recursive function M is a *total learner* if M conjectures only programs for *total* functions, i.e., $\forall x\ \phi_{M(x)}$ is a total recursive function.

Definition 10.2 We say that the total learner M *learns* a function f *syntactically in the limit* (written $f \in SYN\langle \mathbf{t} \rangle[M]$) if and only if the sequence of conjectures $p_n = M(f \mid n)$ by M on f converges to a correct program p for f, i.e.,

- **(Convergence Criterion)**
  $\lim_{n\to\infty} p_n = p$, and
- **(Correctness Criterion)**
  $\phi_p = f$.

We denote by **R** the class of all total recursive functions.

We denote by $SYN\langle \mathbf{t} \rangle$ the class of sets of functions which can be learned with respect to $SYN\langle \mathbf{t} \rangle$-type learning:

$$SYN\langle \mathbf{t} \rangle = \{S \subseteq \mathbf{R} : \exists M\ S \subseteq SYN\langle \mathbf{t} \rangle[M]\}.$$

Theorem 10.6 $\mathbf{R} \notin SYN\langle \mathbf{t} \rangle$.

Proof: Given any M we can define via the Recursion Theorem a function $\phi_e \in \mathbf{R}$ such that $\phi_e \notin SYN\langle \mathbf{t} \rangle[M]$ by

$$\phi_e(0) = e$$

$$\phi_e(x+1) = 1 + \phi_{M(\phi_e|x)}(x+1).$$

Observe that since for all y $\phi_{M(y)} \in \mathbf{R}$, the function $\phi_e \in \mathbf{R}$. Let $p_n = M(\phi_e \mid n)$. Suppose now that there is some program p such that $p = \lim_{n\to\infty} p_n$, and let m be so large that $\forall n \geq m\ p_n = p$. Then,

$$\phi_e(m+1) = 1 + \phi_{M(\phi_e|m)}(m+1) = 1 + \phi_p(m+1),$$

so that M cannot converge in the limit to a correct program for ϕ_e.

Observe that although M has the index e available to it, it can't produce e as its answer, since in general e might not compute a total function.

Next Up Previous Contents Index

Next: 10.1.2 Speed-Up Theorem **Up:** 10.1 Applications of the Recursion Theorem **Previous:** 10.1 Applications of the Recursion Theorem

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 11. Non-Deterministic Computations **Up:** 10.1 Applications of the Recursion Theorem **Previous:** 10.1.1 Machine Learning

10.1.2 Speed-Up Theorem

From the definition of *DLPtime* that it is possible to ``speed-up" (i.e., reduce) the computation time and space for any recursive function be choosing a program over a larger alphabet. Here, we imagine that we have an acceptable programming system consisting of *LOOP* programs (or *RAM* programs) over all possible alphabets, where the alphabet on which a program is based is included in its coding. We have also observed that any program for a recursive function can be sped-up on finitely many of its inputs by building a finite table for those inputs into that program.

Definition 10.3 Let h be a total recursive function. A program i is called *h-optimal* for a partial recursive function f if and only if

$$\phi_i = f$$

and

$$\forall_j \ \phi_j = f \Longrightarrow \overset{\infty}{\forall}_x \ \Phi_i(x) \leq h(\Phi_j(x)).$$

Thus, modulo h, the program i is as fast as any program for f.

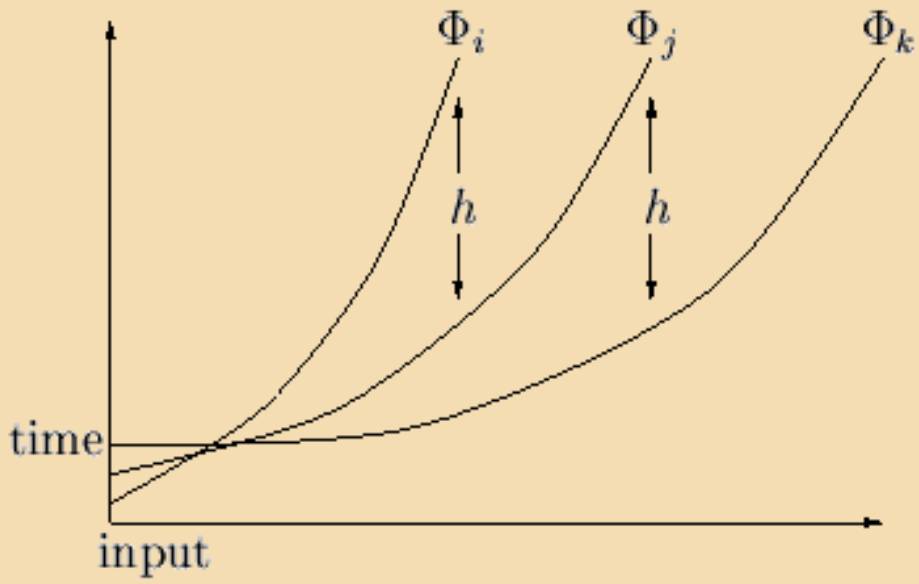
Question 10.1 Does there exist a total recursive function h such that every partial recursive function has an h-optimal program?

Theorem 10.7 (Speed-Up) For every total recursive function h there exists a total recursive function f such that

$$\forall i \ \phi_i = f \Longrightarrow \exists_j \ \phi_j = f \text{ and } \overset{\infty}{\forall}_x \ \Phi_i(x) > h(\Phi_j(x))$$

Corollary 10.8 For every total recursive function *h* there exists a total recursive function *f* that has *no h*-optimal program.

Figure 10.7: Repeated Speed-Up

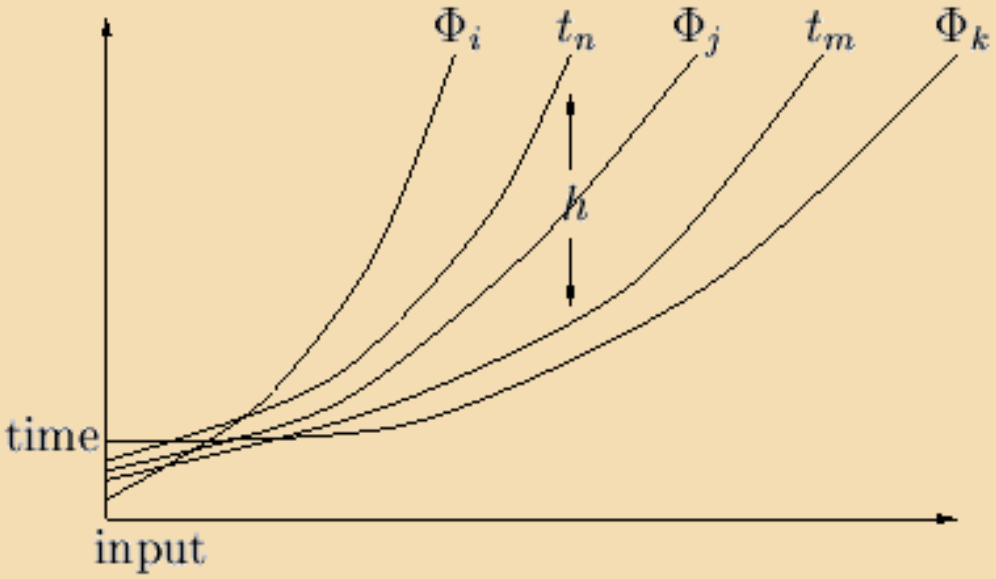


Definition 10.4 A *complexity sequence* for a total recursive function *f* is a set of total functions $\{t_i\}$ such that

1. $\forall n \exists j \ \phi_j = f$ and $\Phi_j \leq t_n$ **a.e.**
2. $\forall j \ \phi_j = f \Longrightarrow \exists m \ t_m \leq \Phi_j$ **a.e.**

Thus a complexity sequence $\{t_i\}$ is *cofinal* with $\{ \Phi_i : \phi_i = f\}$.

Figure 10.7: Complexity Sequence



If we can construct a complexity sequence $\{t_i\}$ for a function f such that $h \circ t_{n+1} \leq t_n$ **a.e.**, then f has h-speed-up:

$$\phi_i = f \Longrightarrow^{(2)} \exists_m\, t_m \leq \Phi_i \text{ **a.e.**}$$

$$\Longrightarrow^{(1)} \exists_j\, \phi_j = f \text{ and } \Phi_j \leq t_{m+1} \text{ **a.e.**}$$

$$\Longrightarrow \exists_j\, \phi_j = f \text{ and } h \circ \Phi_j \leq h \circ t_{m+1} \leq t_m \leq \Phi_i \text{ **a.e.**}$$

Proof: (of Speed-Up Theorem)

- **Construction of *f*:**

  The construction of f is a modification of the standard diagonalization argument (see Theorem 8.7), but is biased against smaller programs (which have less information content). We can assume without loss of generality that h is strictly increasing, i.e., $h(x) > x$.

$$\phi_{\sigma(i,u,v)}(x) = \begin{cases} y_j, \text{ if } v = \langle\langle x_1, y_1\rangle_2, \ldots, \langle x_n, y_n\rangle_2\rangle_n \\ \qquad \text{and } x = x_j \text{ and } 1 \le j \le n \\ 1 + \max\{\phi_j(x) : u \le j \le x \text{ and } \Phi_j(x) \le \Phi_i(x-j) \text{ and} \\ \qquad (*)\ [\forall y\ u \le y \le x \wedge j \le y \Longrightarrow \Phi_j(y) > \Phi_i(y-j)]\}, \\ \qquad \text{otherwise.} \end{cases}$$

Define, $f = \phi_{\sigma(i,0,0)}$, where the function ϕ_i is yet to be determined.

♠ Observe, by (*) that ϕ_j can affect *only one* argument x in the definition of f.

For the present we will assume that ϕ_i is total, so that f is also total. We first have that

$$\phi_j = f \Longrightarrow \forall x \ge j\ \Phi_j(x) > \Phi_i(x - j) \qquad (10.1)$$

This is so because if $x \ge j$ and $\Phi_j \le \Phi_i(x - j)$, then since $(u = v = 0)$, $f(x) \ge 1 + \phi_j(x)$.

Construction of Table *v*:

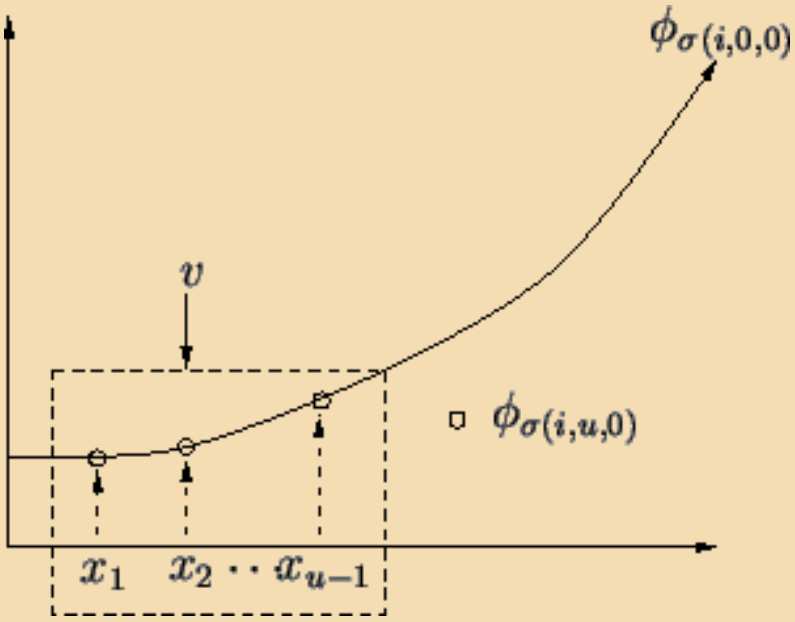
Next,

$$\forall u \exists v\ \phi_{\sigma(i,u,v)} = f. \qquad (10.2)$$

By our previous observation, we have for all u and v,

$$\phi_{\sigma(i,u,v)} = \phi_{\sigma(i,0,0)} \qquad \textbf{a.e.}$$

Figure 10.8: Speed-Up Table



The required ``table" *v* is given by

$$v = \langle \langle x_1, f(x_1) \rangle_2, ..., \langle x_{u-1}, f(x_{u-1}) \rangle_2 \rangle_u ,$$

where for each j, $1 \leq j<u$, x_j was the only value affected by diagonalization against ϕ_j.

Construction of function *r*:

Next, there exists a total recursive function r such that

$$r(x) > x$$

and

$$\forall_i \forall_u \forall_v \overset{\infty}{\forall}_x \ \Phi_{\sigma(i,u,v)}(x) \leq r(\max\{\Phi_i(y) : 0 \leq y \leq x - u\}). \tag{10.3}$$

We define

$$g(i, u, v, x, z) = \begin{cases} \Phi_{\sigma(i,u,v)}(x), \text{ if } z \geq \max\{\Phi_i(x-j) : u \leq j \leq x\} \\ \qquad (\equiv z \geq \max\{\Phi_i(y) : 0 \leq y \leq x-u\}) \\ 0, \text{ otherwise.} \end{cases}$$

Then define

$$r(z) = \max\{g(i, u, v, x, z), z + 1 : i, u, v, x \leq z\}.$$

Then, for all $x \geq u$,

$$r(\max\{\Phi_i(y) : 0 \leq y \leq x - u\}) \geq g(i, u, v, x, \max\{\Phi_i(y) : 0 \leq y \leq x - u\})$$
$$\geq \Phi_{\sigma(i,u,v)}(x)$$

Observe, that in the definition of r the maximum is taken over all $i \leq z$, which may include programs i for non-total functions. Observe also that r is strictly increasing.

- **Construction of complexity sequence $\{t_n\}$:**

  We now construct the complexity sequence $\{t_n\}$. Define,

$$t_n(x) = \Phi_i(x - n).$$

Suppose $\phi_j = f$, then by (10.1) for $x \geq j$ we have $\Phi_j(x) > \Phi_i(x - j) = t_j(x)$. Thus, condition (2) in the definition of a complexity sequence is satisfied.

Suppose ϕ_i is such that

$$\Phi_i(x+1) \geq h(r(\Phi_i(x))), \tag{10.4}$$

so that $\Phi_i(x+1) > \Phi_i(x)$, since $h$and $r$are strictly increasing functions.

Then, using (10.3), that Φ_i is increasing, and (10.4) we have,

$$\begin{aligned} h(\Phi_{\sigma(i,u,v)}(x)) &\leq h(r(\max\{\Phi_i(y) : 0 \leq y \leq x-u\})) \\ &\leq h(r(\Phi_i(x-u))) \\ &\leq \Phi_i(x-u+1) = \Phi_i(x-(u-1)) = t_{u-1}(x). \end{aligned}$$

Therefore, given n, by (10.2) there exists a j($j = \sigma(i, n+1, v)$, for an appropriate v) such that

$$\phi_j = f \qquad \text{and} \qquad \Phi_j \leq h \circ \Phi_j \leq t_n \qquad \textbf{a.e.}$$

so that condition (1) in the definition of a complexity sequence is satisfied. Moreover,

$$\begin{aligned} t_m(x) = \Phi_i(x-m) &\geq h(r(\Phi_i(x-(m+1)))) \\ &\geq h(\Phi_i(x-(m+1))) \\ &\geq (h \circ t_{m+1})(x) \end{aligned}$$

Thus, $\{t_n\}$ is the desired complexity sequence.

- **Construction of an approporiate ϕ_i:**

  Define

$$\psi(i, x) = \begin{cases} 0, & \text{if } x = 0 \\ \phi_i(x) + 1, & \text{if } \Phi_i(x) \leq \max\{\Phi_i(x-1), h(r(\Phi_i(x-1)))\} \\ x, & \text{otherwise.} \end{cases}$$

By the Recursion Theorem there exists an i_0 such that $\psi(i_0, x) = \phi_{i_0}(x)$ for all x. Then, clearly $\Phi_{i_0}(x) > h(r(\Phi_{i_0}(x-1)))$. Observe, also that ϕ_{i_0} is total, which can be proved by induction on the domain of ϕ_{i_0}.

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 11.1 Complexity of Non-Deterministic Programs **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 10.1.2 Speed-Up Theorem

11. Non-Deterministic Computations

- Recall that non-deterministic *LOOP* programs (i.e., *NLOOP* programs) are obtained by adding the following *SELECT* statement:

  SELECT($\mathbf{X}_1$ **)**

  which assigns either a 0 or a 1 non-deterministically to the variable $\mathbf{X}_1$.
- Similarly, non-deterministic *RAM* programs (i.e., *NRAM* programs) are obtained by adding the following *JUMP* instruction:

  2k + 4/j$_1$/j$_2$; **njp** j_1 **or** j_2

  which non-deterministically selects one of two lines ($j_1$or j_2) to jump to.
- We will investigate non-deterministic computations by means of *NRAM* programs, but we could could equally use *NLOOP* programs.

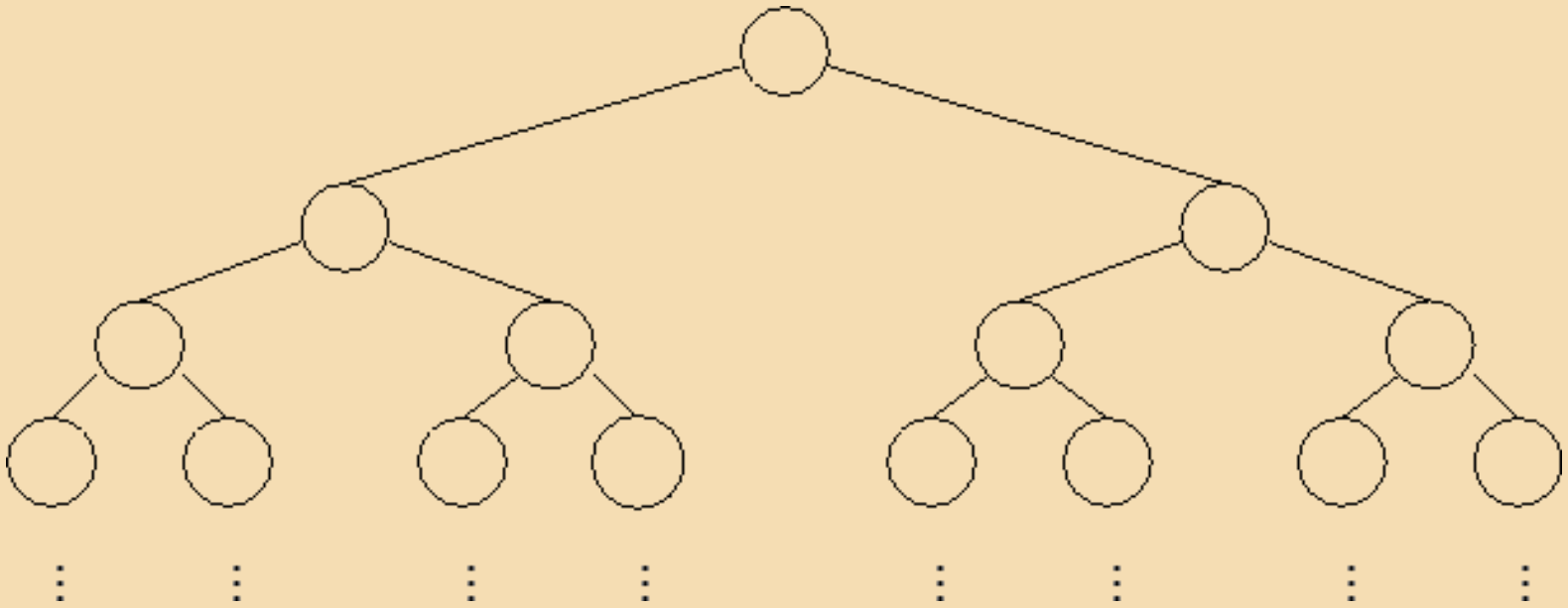
Definition 11.1 Given an *NRAM* program P and an input x, an *accepting computation* of P on x is any legal sequence of instruction executions of P for which that last instruction executed is the output instruction of P, i. e., for which P halts.

Definition 11.2 We say that the *NRAM* program P *accepts* the input x if and only if there exists some accepting computation of P on x. We define the *set accepted by* the *NRAM* program P by

$$L_P = \{x \ : \ P \text{ accepts } x\}.$$

Thus, a non-deterministic computation has the following tree-like structure, where each node of the tree represents a non-deterministic branch point (i.e., execution of a **njp** instruction).

Figure 11.1: Non-Deterministic Computation



Instead of viewing the execution of a **njp** instruction as a non-deterministic selection of a branch point, we can imagine that it corresponds to a bifurcation of a process which is executing the program and which creates two child processes each of which branches to one of the two possible branch points, and such that when a child process halts, it will cause its parent process to halt, etc. Thus, and alternate (and perhaps more realistic) view of non-determinism is as unbounded parallelism.

Theorem 11.1 Every set accepted by a *NRAM* program can be accepted by a *DRAM* program.

Proof: It suffices to show that every set accepted by an *NRAM* program is the domain of some partial recursive function. Recall in the construction of the universal partial recursive function for *DRAM* programs we defined primitive recursive functions *nxl* and *nxv*, which *computed* the next line and next register contents during the simulation. However, since the program which we now wish to simulate is non-deterministic, it is no longer the case that the next line to be executed is determined by (i.e., is a function of) the current line and current contents. Instead, we now construct a primitive recursive predicate *Nxl* which decides whether or not a given line *can leagally be* the next line. Moreover, since the sequence of computation steps is no longer determined by the program and input, we will define a predicate *Acc* that will decide whether or not a given sequence of program states represents an accepting computation.

First we need to provide some parsing predicates which allow us to parse *NRAM* programs:

$$go1(x) = \begin{cases} prt_{k+2}('/', x, 1), \text{ if } opc(x) = 2k+4 \\ 0, \text{ otherwise.} \end{cases}$$

$$go2(x) = \begin{cases} prt_{k+2}('/', x, 2), \text{ if } opc(x) = 2k+4 \\ 0, \text{ otherwise.} \end{cases}$$

which produce the two branch points of the **njp** instruction coded by x. We also need to define the predicates $Ins(x)$ and $Prg(x)$ which decide whether or not x codes a legal instuction and program respectively. Then, we define the primitive recursive predicate

$$Nxl(p, y, z, j, r) \equiv (opc(lne(j, p)) \neq 2k + 4 \implies r = nxl(p, y, z, j)) \wedge$$
$$(opc(lne(j, p)) = 2k + 4 \implies$$
$$(r = go1(lne(j, p)) \vee r = go2(lne(j, p))))$$

Next, let w code for a comma separated sequence $s_0, s_1, ..., s_n$ of numbers each of which is interpreted as a pair representing a state (line number, register contents) of the program p during its execution.

Then, the predicate $Acc(p, y, w)$, where p codes the program and y codes the input, is defined by:

$$Acc(p, y, w) \equiv (lin(w, 0) = 1 \wedge con(w, 0) = zro(mxr(p))) \wedge$$
$$(lin(w, noc_{k+1}(',', w)) = lng(p)) \wedge$$
$$\forall j < noc_{k+1}(',', w)[Nxl(p, y, con(w, j), lin(w, j), lin(w, j + 1)) \wedge$$
$$nxv(p, y, con(w, j), lin(w, j)) = con(w, j + 1)]$$

where $lin(w, j)$ and $con(w, j)$ give the line number and register contents for the j^{th} state in w and are defined by

$$lin(w, j) = \Pi_1^2(prt_{k+1}(',', w, j))$$
$$con(w, j) = \Pi_2^2(prt_{k+1}(',', w, j)).$$

Finally, we see that

$$L_p = \mathbf{dom}\ \phi$$

where the partial recursive function ϕ is defined by

$$\phi(y) = \min w[Acc(p, y, w)].$$

Next Up Previous Contents Index

Next: 11.1 Complexity of Non-Deterministic Programs **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 10.1.2 Speed-Up Theorem

Bob Daley

2001-11-28

11.1 Complexity of Non-Deterministic Programs

Definition 11.3 Let P be a *NRAM* program over Σ_k^* (where $k > 1$), then we define the following complexity measures for P.

$$NRAMtime_P(x) = \begin{cases} \textbf{minimum over all accepting computations} \\ \quad |x| + \#\text{ of stmts of } P \text{ executed on input } x, \\ \quad \text{if } P \text{ halts on it,} \\ \uparrow, \text{ otherwise.} \end{cases}$$

$$NRAMspace_P(x) = \begin{cases} \textbf{minimum over all accepting computations} \\ \quad \max \sum_{i=1}^{r} |\mathbf{R}_i^t|, \forall t \leq DRMtime_P(x), \\ \quad \text{if } P \text{ halts,} \\ \uparrow, \text{ otherwise.} \end{cases}$$

where $\mathbf{R}_i^t$ denotes the contents of register $\mathbf{R}_i$ at step t of the computation of P on input x.

Thus, for non-deterministic computations the complexity is defined in terms of the most efficient (with respect to time or space) accepting computation. The rationale for this is that since the program is allowed to ``guess" an accepting computation it might as well be allowed to guess the most efficient accepting computation. Observe that the most space efficient accepting computation need not be the most time efficient one, and vice versa.

Definition 11.4 We define the following (deterministic) complexity classes:

$$DPTIME = \{L : \exists \text{ } DRAM \text{ program } P \text{ and a polynomial function } t \text{ such that}$$

$$P \text{ computes } \chi_L \text{ and } \forall x \quad DRAMtime_P(x) \leq t(|x|)\}.$$

$$DPSPACE = \{L : \exists \text{ } DRAM \text{ program } P \text{ and a polynomial function } t \text{ such that}$$

$$P \text{ computes } \chi_L \text{ and } \forall x \quad DRAMspace_P(x) \leq t(|x|)\}.$$

- Aliases for *DPTIME* is $\mathbb{P}$, and for *DPSPACE* is $\mathbb{PSPACE}$.
- The definitions of *DPTIME* and *DPSPACE* are independent of the (standard) model of computation used (see Proposition 7.6).

Definition 11.5 We define the following (non-deterministic) complexity classes:

$$NPTIME = \{L : \exists \text{ } NRAM \text{ program } P \text{ and a polynomial function } t \text{ such that}$$

$$P \text{ accepts } L \text{ and } \forall x \in L \quad NRAMtime_P(x) \leq t(|x|)\}.$$

$$NPSPACE = \{L : \exists \text{ } NRAM \text{ program } P \text{ and a polynomial function } t \text{ such that}$$

$$P \text{ accepts } L \text{ and } \forall x \in L \quad NRAMspace_P(x) \leq t(|x|)\}.$$

- Aliases for *NPTIME* is $\mathbb{NP}$, and for *NPSPACE* is $\mathbb{NSPACE}$.

Next Up Previous Contents Index

Next: 11.2 NP-Completeness **Up:** 11. Non-Deterministic Computations **Previous:** 11. Non-Deterministic Computations

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 11.3 Polynomial Time Reducibility **Up:** 11. Non-Deterministic Computations **Previous:** 11.1 Complexity of Non-Deterministic Programs

11.2 NP-Completeness

Definition 11.6 We say that a function f is *polynomial-time computable* if and only if there is some *DRAM* program P and a polynomial function t such that P computes the function f and $DRAMtime_P(x) \leq t(|x|)$. We say that the set Y is *polynomial-time reducible* to the set X (written $Y \leq_p X$) if and only if there exists a polynomial-time computable function f such that $y \in Y \iff f(y) \in X$.

♠ Observe if f is computable in polynomial time t, then $|f(x)| \leq t(|x|)$.

Definition 11.7 A set X is called *NP-complete* if and only if X is complete for $\mathbb{NP}$ with respect to $\leq_p$, i.e., $X \in \mathbb{NP}$ and $Y \leq_p X$ for all $Y \in \mathbb{NP}$.

Definition 11.8 A propositional formula B is called *satisfiable* if and only if there exists some assignment of truth values to its variables which makes the value of B true.

Example 11.1 Let $B = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$. Then B is satisfiable via the assignment $x_1 = \mathbf{T}$, $x_2 = \mathbf{F}$, $x_3 = \mathbf{T}$.

Definition 11.9 *SAT* is the set of all satisfiable propositional formulas in conjunctive normal form.

Proposition 11.2 $SAT \in \mathbb{NP}$.

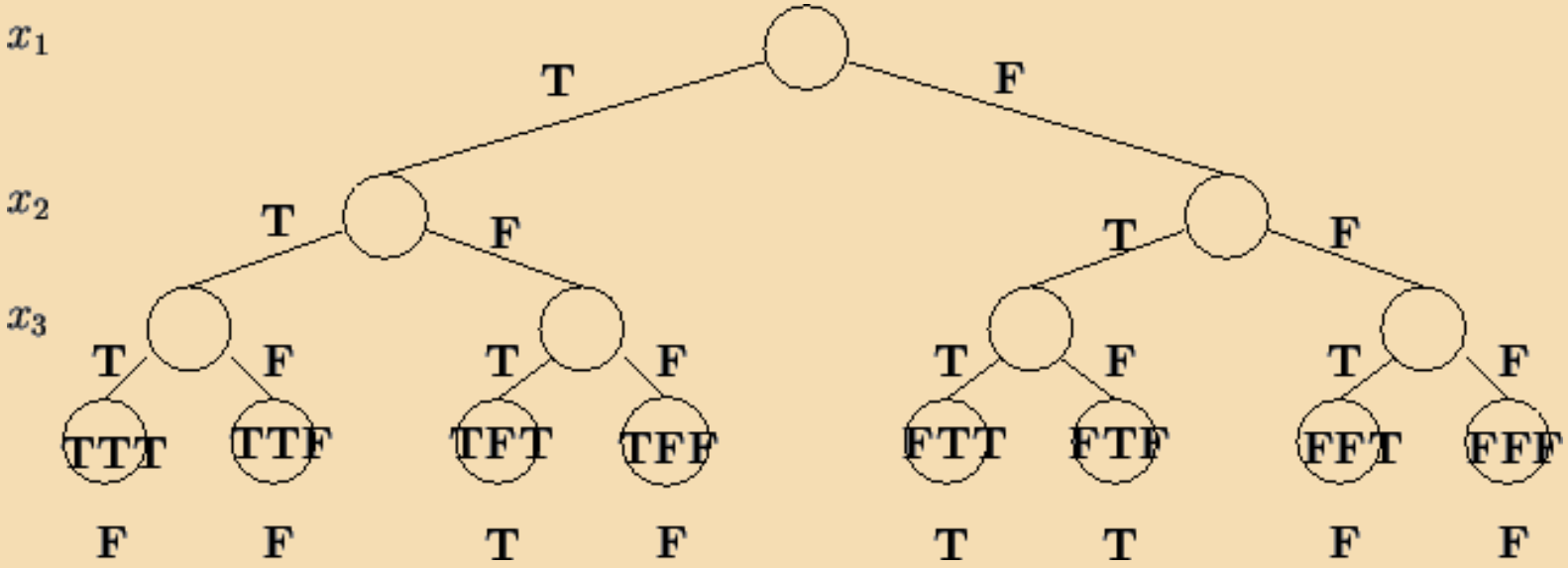
Proof: A non-deterministic algorithm works as follows:
Given a propositional formula B with variables $x_1,\ldots, x_n$, it:

- guesses (correctly, if possible) a satisfying truth assignment to $x_1,\ldots, x_n$;
- verifies that the chosen assignment to $x_1,\ldots, x_n$ makes the value of B true, and if so, accepts.

Thus, if $B \in SAT$, then there is some assignment to $x_1,\ldots, x_n$, so the algorithm will ``guess' it and so will accept. If

$B \notin SAT$, then no guess will make the value of B true, so the algorithm does not accept.

Figure 11.2: Non-Deterministic Computation for $B = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$



Theorem 11.3 *SAT* is *NP*-complete.

Proof: It suffices to show that every set $X \in \mathbb{NP}$ is polynomial-time reducible to *SAT*. Let $X \in \mathbb{NP}$ and let P be a *NRAM* program over Σ_k^* which accepts X in polynomial time p, i.e., $x \in X \iff NRAMtime_P(x) \leq p(|x|)$. We construct *for each* x a propositional formula B_x in conjunctive normal form such that $x \in X \iff B_x$ is satisfiable.

The propositional formula B_x must be satisfiable if and only if there is an accepting computation of P on input x, so we will need to describe *NRAM* computation by means of propositional variables and formulas. Let m be the number of lines of P and let w be the maximum number register named in P. Let $x = a_1 \cdots a_n$, so $|x| = n$. We will represent the contents of each register by a string of length $p(n)$ over Σ_{k+1}, where we use the k+1st symbol as a blank $\sqcup$ to pad (to the right) the actual contents so the representation is *exactly* of length $p(n)$. Length $p(n)$ suffices since we can add at most one symbol per time step to the contents of any register and since the length of the input is included in the computation time.

We first introduce polynomially many propositional variables as follows:

Table 11.1: Variables for B_x

Variable	Intended meaning
$SMB[t : i : r : s]$	At time t the symbol in position i of register r is s
$LIN[t : j]$	At time t the current line number is j

We also introduce notation for the various symbols which occur in any given line j (depending on the type of instruction).

Table 11.2: Constants for B_x

Constant	Description	Instruction Type
r_j	Register named in line j	All, except **njp**
s_j	Symbol named in line j	**jmp** and **suc**
g_j	Goto part of line j	**jmp**
g_j^1	First goto part of line j	**njp**
g_j^2	Second goto part of line j	**njp**

- Part of the definition of B_x will be devoted to making sure that the intended interpretation of the above variables is in fact the actual meaning.
- In order to describe in a more readable form the formula B_x, we introduce the following notation. Let $E(z)$ be some propositional formula with variable symbol z, where $E(z)$ is well formed for all $u \leq z \leq v$. Then,

$$\bigvee_{z=u}^{v} E(z) \quad \text{stands for} \quad E(u) \vee E(u+1) \vee \cdots \vee E(v)$$

$$\bigwedge_{z=u}^{v} E(z) \quad \text{stands for} \quad E(u) \wedge E(u+1) \wedge \cdots \wedge E(v)$$

- Observe that if $A_1,\ldots, A_u$ and $B_1,\ldots, B_v$ are literals, then the formula $A_1 \wedge \cdots \wedge A_u \Longrightarrow B_1 \vee \cdots \vee B_v$ is

logically equivalent to $\neg A_1 \vee \cdots \vee \neg A_u \vee B_1 \vee \cdots \vee B_v$, and so is a single disjunction of literals.

- To further enhance the readibility of B_x we will assign types to certain variables and abbreviate quantifiers over these variables as indicated in the following table.

Table 11.3: Quantifiers for B_x

Var.	Type	Range	$\exists$	$\forall$
t	time	$0 \leq t \leq p(n)$	$\bigvee_{t=0}^{p(n)}$	$\bigwedge_{t=0}^{p(n)}$
i	positions	$1 \leq i \leq p(n)$	$\bigvee_{i=1}^{p(n)}$	$\bigwedge_{i=1}^{p(n)}$
r	registers	$1 \leq r \leq w$	$\bigvee_{r=1}^{w}$	$\bigwedge_{r=1}^{w}$
s	symbols	$1 \leq s \leq k+1$	$\bigvee_{s=1}^{k+1}$	$\bigwedge_{s=1}^{k+1}$
j	lines	$1 \leq j \leq m$	$\bigvee_{j=1}^{m}$	$\bigwedge_{j=1}^{m}$

- We will also use the abbreviation $\exists_1 s \ E(s)$ (read ``there exists a unique s such that $E(s)$'') for the expression

$$\exists s \ E(s) \wedge \forall s_1 \bigwedge_{s_2=1, s_2 \neq s_1}^{k+1} (\neg E(s_1) \vee \neg E(s_2))$$

Observe that the above expression is in conjunctive normal form. Similarly, $\exists_1 j \ E(j)$ stands for

$$\exists j \ E(j) \wedge \forall j_1 \bigwedge_{j_2=1, j_2 \neq j_1}^{m} (\neg E(j_1) \vee \neg E(j_2))$$

In this context the meaning of the quantifiers $\forall t<p(n)$, $\forall i>n$, etc., should also be clear.

- The formula B_x consists of the conjunction of several ``subformulas'' B_1,..., B_6 which are defined below, i.e., $B_x = B_1 \wedge \cdots \wedge B_6$.

B_1:

The formula B_1 asserts that at each point in time each bit position of each register contains a unique symbol:

$$\forall_t \ \forall_i \ \forall_r \ \exists_1 s \ SMB[t : i : r : s]$$

B_2:

The formula B_2 asserts that at each point in time there is a unique current line number:

$$\forall_t \ \exists_1 j \ LIN[t : j]$$

B_3:

The formula B_3 asserts that the computation begins correctly:

$$LIN[0 : 1] \wedge \forall_r \ \forall_i \ SMB[0 : i : r : \sqcup]$$

B_4:

The formula B_4 asserts that at some point in time the last line of P is reached:

$$\exists_t \ LIN[t : m]$$

B_5:

The formula B_5, which is a conjunction of subformulas $B_{5.1}$,..., $B_{5.5}$, asserts that at any point in time the next line to be executed is legally reachable from the current line:

$B_{5.1}$:

$$\forall_{t < p(n)}\ LIN[t : m] \implies LIN[t + 1 : m]$$

Each of the following subformulae are included for each line j of the specified type:

- **$B_{5.2}$:**

  for each line j which is **not** a **jmp**, **njp** , or **out** instruction

$$\forall_{t < p(n)}\ LIN[t : j] \implies LIN[t + 1 : j + 1]$$

- **$B_{5.3}$:**

  for each line j which is a **jmp** instruction

$$\forall_{t < p(n)}\ LIN[t : j] \wedge \neg SMB[t : 1 : r_j : s_j] \implies LIN[t + 1 : j + 1]$$

- **$B_{5.4}$:**

  for each line j which is a **jmp** instruction

$$\forall_{t < p(n)}\ LIN[t : j] \wedge SMB[t : 1 : r_j : s_j] \implies LIN[t + 1 : g_j]$$

- **$B_{5.5}$:**

  for each line j which is a **njp** instruction

$$\forall_{t < p(n)}\ LIN[t : j] \implies LIN[t + 1 : g_j{}^1] \vee LIN[t + 1 : g_j{}^2]$$

- **B_6:**

  The formula B_6, which is a conjunction of subformulas $B_{6.1}$,..., $B_{6.9}$, asserts that at any point in time the next register contents are correctly calculated:

  - **$B_{6.1}$:**

    for each line j and for all $r \neq r_j$ or for all r if line j is a **njp** or **jmp** instruction

$$\forall_{t < p(n)}\ \forall_i\ \forall_s\ LIN[t : j] \wedge SMB[t : i : r : s] \implies SMB[t + 1 : i : r : s]$$

- **$B_{6.2}$:**

  for line 0 which is an **inp** instruction and time 1

$$\forall i \leq n \;\; SMB[1 : i : r_0 : a_i]$$

- **$B_{6.3}$:**

  for line 0 which is an **inp** instruction and time 1

$$\forall i > n \;\; SMB[1 : i : r_0 : \sqcup]$$

- **$B_{6.4}$:**

  for each line j which is a **suc** instruction

$$\forall t < p(n) \;\; \forall i \;\; \forall s \leq k \;\; LIN[t : j] \wedge SMB[t : i : r_j : s] \Longrightarrow SMB[t + 1 : i : r_j : s]$$

- **$B_{6.5}$:**

  for each line j which is a **suc** instruction

$$\forall t < p(n) \;\; \forall i < p(n) \;\; LIN[t : j] \wedge \neg SMB[t : i : r_j : \sqcup] \wedge SMB[t : i + 1 : r_j : \sqcup] \Longrightarrow SMB[t + 1 : i + 1 : r_j : s_j]$$

- **$B_{6.6}$:**

  for each line j which is a **suc** instruction

$$\forall t < p(n) \;\; \forall i < p(n) \;\; LIN[t : j] \wedge SMB[t : i : r_j : \sqcup] \wedge SMB[t : i + 1 : r_j : \sqcup] \Longrightarrow SMB[t + 1 : i + 1 : r_j : \sqcup]$$

- **$B_{6.7}$:**

  for each line j which is a **suc** instruction

$$\forall_{t < p(n)}\ LIN[t : j] \wedge SMB[t : 1 : r_j : \sqcup\ ] \Longrightarrow SMB[t + 1 : 1 : r_j : s_j]$$

- **$B_{6.8}$:**

  for each line j which is an **lsf** instruction

$$\forall_{t < p(n)}\ \forall_{i < p(n)}\ \forall_{s}\ LIN[t : j] \wedge SMB[t : i + 1 : r_j : s] \Longrightarrow$$
$$SMB[t + 1 : i : r_j : s]$$

- **$B_{6.9}$:**

  for each line j which is an **lsf** instruction

$$\forall_{t < p(n)}\ LIN[t : j] \Longrightarrow SMB[t + 1 : p(n) : r_j : \sqcup\ ]$$

This completes the construction of the formula B_x. To complete the proof it is necessary to prove by induction on the time t that

1. if B_x is satisfiable, then there exists an accepting computation for P on input x. The accepting computation is constructed from the satisfying assignment to the variables of B_x; and
2. if P accepts input x, then there is a satisfying assignment to the variables of B_x. The satisfying assignment is constructed from the accepting computation of P on input x.

Proposition 11.4 For any set X, if X is *NP*-complete and $X \in \mathbb{P}$, then $\mathbb{P} = \mathbb{NP}$.

Proof: Clearly, $\mathbb{P} \subseteq \mathbb{NP}$. Let X be *NP*-complete and suppose $X \in \mathbb{P}$, so that there exists some *DRAM* program Q which accepts X in polynomial time. Next, let $Y \in \mathbb{NP}$. Then, since $Y \leq_p X$, there is some

polynomial-time computable function f such that $y \in Y \iff f(y) \in X$. Thus, $y \in Y \iff f(y) \in X \iff Q$ accepts $f(y)$. Hence, there is a polynomial time acceptor for Y that, given y, computes $f(y)$ and applies Q to $f(y)$. Therefore, $Y \in \mathbb{P}$, and so $\mathbb{NP} \subseteq \mathbb{P}$.

Proposition 11.5 For any sets X and Y, if X is *NP*-complete and $Y \in \mathbb{NP}$ and $X \leq_p Y$, then Y is also *NP*-complete.

Proof: This follows from the transitivity of the relation $\leq_p$, i.e., from the fact that the composition of two polynomial-time computable functions is polynomial-time computable.

Notation 11.10 Let V be a set of propositional variables. Then we use θ to denote an arbitrary truth assignment to the variables of V, i.e., $\theta : V \longrightarrow \{\mathbf{T},\mathbf{F}\}$. Given any propositional formula B we denote by $Var(B)$ the set of variables occurring in B. If $Var(B) \subseteq V$, then the truth assignment θ above determines uniquely a truth value for B which we denote by $\theta(B)$. In these terms, then, a *CNF* formula $B = C_1 \wedge \cdots \wedge C_n$ is satisfiable if and only if there exists a $\theta : Var(B) \longrightarrow \{\mathbf{T},\mathbf{F}\}$ such that $\theta(C_i) = \mathbf{T}$ for all $1 \leq i \leq n$

The next two results involve specializing the *NP*-completeness of *SAT* to restricted cases of the satisfiability problem which retain the property of being *NP*-complete. In each case beginning with a propositional formula $B = C_1 \wedge \cdots \wedge C_n$ in conjunctive normal form, we construct a new *CNF* formula $\hat{B}$ belonging to the restricted satisfiability class by replacing each clause C_i by a set of clauses $\hat{C}_{i,1},\ldots,\hat{C}_{i,m_i}$, whose variables are those of C_i plus some new variables $\hat{V}_i$ that are used nowhere else and such that

1. for each truth assignment θ to $Var(B)$ for which $\theta(C_i) = \mathbf{T}$, there is an extension of θ to a truth assignment θ_i to $Var(B) \cup \hat{V}_i$ such that $\theta_i(\hat{C}_{i,j}) = \mathbf{T}$ for all $1 \leq j \leq m_i$; and
2. given any truth assignment θ_i to $Var(B) \cup \hat{V}_i$ such that $\theta_i(\hat{C}_{i,j}) = \mathbf{T}$ for all $1 \leq j \leq m_i$, we have θ_i

(C_i) = **T**.

It then follows that $C_1 \wedge \cdots \wedge C_n$ is satisfiable if and only if ($\hat{C}_{1,1} \wedge \cdots \wedge \hat{C}_{1,m_1}$) $\wedge \cdots \wedge$ ($\hat{C}_{n,1} \wedge \cdots \wedge \hat{C}_{n,m_n}$) is satisfiable. Finally, as in the case of the general satisfiability problem it will be easy to see that each of the restricted cases belongs to $\mathbb{NP}$ by guessing an assignment of truth values and then verifying that all the appropriate conditions are satisfied.

Definition 11.11 *3SAT* is the set of all satisfiable propositional formulas in conjunctive normal form which have exactly 3 literals per clause.

Proposition 11.6 *3SAT* is *NP*-complete.

Proof: Clearly, *3SAT* $\in \mathbb{NP}$. Let $B = C_1 \wedge \cdots \wedge C_n$ be a propositional formula in conjunctive normal form. For each clause C_i containing k literals, where $k \neq 3$, we replace C_i with a set of clauses $\hat{C}_{i,1},\ldots,\hat{C}_{i,m_i}$, that contain new variables in addition to those of C_i such that C_i will be satisfiable by a truth assignment θ if an only if all of $\hat{C}_{i,1},\ldots,\hat{C}_{i,m_i}$ are satisfiable by a truth assignment θ_i extending θ. The proof is broken naturally into the following three cases:

- **Case 1:**
  $C_i = w$, for some literal w. Define

$$\hat{C}_{i,1} = (w \vee \hat{z}_1 \vee \hat{z}_2)$$

$$\hat{C}_{i,2} = (w \vee \neg \hat{z}_1 \vee \hat{z}_2)$$

$$\hat{C}_{i,3} = (w \vee \hat{z}_1 \vee \neg \hat{z}_2)$$

$$\hat{C}_{i,4} = (w \vee \neg \hat{z}_1 \vee \neg \hat{z}_2)$$

where $\hat{z}_1$ and $\hat{z}_2$ are new propositional variables.

- **Case 2:**

$C_i = (w_1 \vee w_2)$, for literals w_1, w_2. Define

$$\hat{C}_{i,1} = (w_1 \vee w_2 \vee \hat{z}_1)$$

$$\hat{C}_{i,2} = (w_1 \vee w_2 \vee \neg \hat{z}_1)$$

where $\hat{z}_1$ is a new propositional variable.

- **Case 3:**

  $C_i = (w_1 \vee w_2 \vee \cdots \vee w_k)$, for literals $w_1,\ldots, w_k$, where $k > 3$. Define

$$\hat{C}_{i,1} = (w_1 \vee w_2 \vee \hat{z}_1)$$

and for $1 < j \leq k-3$, a clause asserting `` $\hat{z}_{j-1} \longrightarrow w_{j+1} \vee \hat{z}_j$"

$$\hat{C}_{i,j} = (\neg \hat{z}_{j-1} \vee w_{j+1} \vee \hat{z}_j)$$

and finally a clause asserting `` $\hat{z}_{k-3} \longrightarrow w_{k-1} \vee w_k$"

$$\hat{C}_{i,k-2} = (\neg \hat{z}_{k-3} \vee w_{k-1} \vee w_k)$$

In Cases 1 and 2, given a truth assignment θ to the variables of C_i such that $\theta(C_i) = \mathbf{T}$, any extension θ_i of θ will work, since all combinations of the new variables $\hat{z}_j$ are included. In Case 3, we extend θ to θ_i by assigning truth values to the variables $\hat{z}_1,\ldots, \hat{z}_{k-3}$ in order as follows:

$$\hat{z}_1 \equiv \neg(w_1 \vee w_2)$$

and for $1 < j \leq k - 3$,

$$\hat{z}_j \equiv \neg(w_{j+1} \vee \hat{z}_{j-1}).$$

Conversely, suppose that $\theta_i(\hat{C}_{i,j}) = \mathbf{T}$ for all $1 \leq j \leq m_i$. Then, in Cases 1 and 2, since $\hat{C}_{i,j}$ is of the form $C_i \vee \hat{C}_j$, where $\hat{C}_j$ contains only new variables, there is some j such that $\theta_i(\hat{C}_j) = \mathbf{F}$, so $\theta_i(C_i) = \mathbf{T}$. In Case 3, we suppose that $\theta_i(w_j) = \mathbf{F}$ for all $1 \leq j < k$, and show by induction that $\theta_i(\hat{z}_j) = \mathbf{T}$ for all $1 \leq j \leq k - 3$, and hence $\theta_i(w_k) = \mathbf{T}$, so $\theta_i(C_i) = \mathbf{T}$.

Definition 11.12 Let *(1/3)SAT* be the set of all satisfiable propositional formulas with three literals per clause for which there is a satisfying assignment which makes *exactly one literal per clause* true.

Proposition 11.7 *(1/3)SAT* is *NP*-complete.

Proof: Clearly *(1/3)SAT* $\in \mathbb{NP}$. Let $B = C_1 \wedge \cdots \wedge C_n$ be a propositional formula in conjunctive normal form. We construct a new CNF propositional formula $\hat{B}$ by replacing each clause C_i of B by a conjunction of clauses $\hat{C}_{i,1} \wedge \cdots \wedge \hat{C}_{i,9}$. If $C_i = (x \vee y \vee z)$, where x, y, z are the three literals of C_i, then $\hat{C}_{i,1} \wedge \cdots \wedge \hat{C}_{i,9}$ contain *new* variables (not used elsewhere) $\hat{x}_a, \hat{y}_a, \hat{z}_a, \hat{xy}_a, \hat{xz}_a, \hat{yz}_a$ and $\hat{x}_b, \hat{y}_b, \hat{z}_b, \hat{xy}_b, \hat{xz}_b, \hat{yz}_b$ as follows:

$$\hat{C}_{i,1} = (x \vee \hat{x}_a \vee \hat{x}_b)$$

$$\hat{C}_{i,2} = (y \vee \hat{y}_a \vee \hat{y}_b)$$

$$\hat{C}_{i,3} = (z \vee \hat{z}_a \vee \hat{z}_b)$$

$$\hat{C}_{i,4} = (\hat{x}_a \vee \hat{y}_a \vee \hat{xy}_a)$$

$$\hat{C}_{i,5} = (\hat{x}_a \vee \hat{z}_a \vee \hat{xz}_a)$$

$$\hat{C}_{i,6} = (\hat{y}_a \vee \hat{z}_a \vee \hat{yz}_a)$$

$$\hat{C}_{i,7} = (\hat{x}_b \vee \hat{y}_b \vee \hat{xy}_b)$$

$$\hat{C}_{i,8} = (\hat{x}_b \vee \hat{z}_b \vee \hat{xz}_b)$$

$$\hat{C}_{i,9} = (\hat{y}_b \vee \hat{z}_b \vee \hat{yz}_b)$$

Suppose first that θ is a truth assignment to the variables of *B* such that $\theta(C_i)$ = **T**. We will construct a truth assignment θ_i which extends θ such that $\theta_i(\hat{C}_{i,j})$ = **T** for all $1 \leq j \leq 9$, and such that *exactly one literal per clause* is true. We first observe that the clauses $\hat{C}_{i,1} \wedge \cdots \wedge \hat{C}_{i,9}$ are so constructed that the following relationships hold:

$$x \wedge y \equiv \hat{xy}_a \wedge \hat{xy}_b$$

$$x \wedge z \equiv \hat{xz}_a \wedge \hat{xz}_b$$

$$y \wedge z \equiv \hat{yz}_a \wedge \hat{yz}_b$$

We consider three cases:

- **Case 1:**

  $\theta(x) = \theta(y) = \theta(z)$ = **T**:

  Then from the above equivalences the assignment θ_i is unique, and θ_i assigns **T** to

  $\hat{xy}_a, \hat{xy}_b, \hat{xz}_a, \hat{xz}_b, \hat{yz}_a$, and $\hat{yz}_b$,

  and assigns **F**to

  $\hat{x}_a, \hat{x}_b, \hat{y}_a, \hat{y}_b, \hat{z}_a$, and $\hat{z}_b$.

- **Case 2:**

  Exactly two of x, y, z are assigned **T** by θ:

  Suppose for definiteness that *z* is the unique literal such that $\theta(z)$ = **F**. Then, by the above equivalences, θ_i must assign **T** to

  $\hat{xy}_a$ and $\hat{xy}_b$,

and assign **F**to

$\hat{x}_a, \hat{x}_b, \hat{y}_a$, and $\hat{y}_b$.

Next, there are two possible completions of the assignment θ_i. Either θ_i assigns **T**to

$\hat{z}_a, \hat{xz}_b$, and $\hat{yz}_b$

and **F**to

$\hat{z}_b, \hat{xz}_a$, and $\hat{yz}_a$,

or θ_i assigns **T**to

$\hat{z}_b, \hat{xz}_a$, and $\hat{yz}_a$,

and **F**to

$\hat{z}_a, \hat{xz}_b$, and $\hat{yz}_b$.

- **Case 3:**

  Exactly one of x, y, z are assigned **T** by θ:

  Suppose for definiteness that *x* is the unique literal such that $\theta(x)$ = **T**. Clearly, θ_i must assign **F** to $\hat{x}_a$ and $\hat{x}_b$. There are again two possible completions of the assignment θ_i. Either θ_i assigns **T** to

$\hat{y}_a$, $x\hat{z}_a$, $\hat{z}_b$, and $x\hat{y}_b$

and **F**to

$\hat{z}_a$, $x\hat{y}_a$, $y\hat{z}_a$, $\hat{y}_b$, $x\hat{z}_b$, and $y\hat{z}_b$,

or θ_i assigns **T**to

$\hat{y}_b$, $x\hat{z}_b$, $\hat{z}_a$, and $x\hat{y}_a$,

and **F**to

$\hat{z}_b$, $x\hat{y}_b$, $y\hat{z}_b$, $\hat{y}_a$, $x\hat{z}_a$, and $y\hat{z}_a$.

The two possible assignments are summarized in the following tables:

Table 11.4: Alternative 1

x	y	z	$\hat{x}_a$	$\hat{y}_a$	$\hat{z}_a$	$x\hat{y}_a$	$x\hat{z}_a$	$y\hat{z}_a$	$\hat{x}_b$	$\hat{y}_b$	$\hat{z}_b$	$x\hat{y}_b$	$x\hat{z}_b$	$y\hat{z}_b$
T	**T**	**T**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**T**	**T**	**T**
T	**T**	**F**	**F**	**F**	**T**	**T**	**F**	**F**	**F**	**F**	**F**	**T**	**T**	**T**
T	**F**	**T**	**F**	**T**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**T**	**T**	**T**
F	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**F**	**T**	**T**	**T**
T	**F**	**F**	**F**	**T**	**F**	**F**	**T**	**F**	**F**	**F**	**T**	**T**	**F**	**F**
F	**T**	**F**	**F**	**F**	**T**	**T**	**F**	**F**	**T**	**F**	**F**	**F**	**F**	**T**
F	**F**	**T**	**T**	**F**	**F**	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**T**	**F**

Table 11.5: Alternative 2

x	y	z	$\hat{x}_a$	$\hat{y}_a$	$\hat{z}_a$	$x\hat{y}_a$	$x\hat{z}_a$	$y\hat{z}_a$	$\hat{x}_b$	$\hat{y}_b$	$\hat{z}_b$	$x\hat{y}_b$	$x\hat{z}_b$	$y\hat{z}_b$
T	**T**	**T**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**F**	**T**	**T**	**T**
T	**T**	**F**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**F**	**T**	**T**	**F**	**F**

T	**F**	**T**	**F**	**F**	**F**	**T**	**T**	**T**	**F**	**T**	**F**	**F**	**T**	**F**
F	**T**	**T**	**F**	**F**	**F**	**T**	**T**	**T**	**T**	**F**	**F**	**F**	**F**	**T**
T	**F**	**F**	**F**	**F**	**T**	**T**	**F**	**F**	**F**	**T**	**F**	**F**	**T**	**F**
F	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**T**	**F**	**F**	**T**	**T**	**F**	**F**
F	**F**	**T**	**F**	**T**	**F**	**F**	**T**	**F**	**T**	**F**	**F**	**F**	**F**	**T**

Suppose on the other hand that θ_i is a truth asignment such that $\theta_i(\hat{C}_{i,j})$ = **T** for all $1 \leq j \leq 9$, where *exactly one literal per clause* is true under the assignment θ_i. Suppose also that $\theta_i(x)$ = **F** and $\theta_i(y)$ = **F**. We then show that $\theta_i(z)$ = **T**, so that $\theta_i(C_i)$ = **T**. Since $\theta_i(x)$ = **F** and $\theta_i(\hat{C}_{i,1})$ = **T**, we have

$$\theta_i(\hat{x}_a) = \textbf{T} \text{ and } \theta_i(\hat{x}_b) = \textbf{F}$$

or

$$\theta_i(\hat{x}_a) = \textbf{F} \text{ and } \theta_i(\hat{x}_b) = \textbf{T}.$$

Similarly, since $\theta_i(y)$ = **F** and $\theta_i(\hat{C}_{i,2})$ = **T**, we have

$$\theta_i(\hat{y}_a) = \textbf{T} \text{ and } \theta_i(\hat{y}_b) = \textbf{F}$$

or

$$\theta_i(\hat{y}_a) = \textbf{F} \text{ and } \theta_i(\hat{y}_b) = \textbf{T}.$$

Next, since $\theta_i(\hat{C}_{i,4})$ = **T** and $\theta_i(\hat{C}_{i,7})$ = **T**, we have

$$\theta_i(\hat{x}_a) = \textbf{T} \text{ and } \theta_i(\hat{y}_b) = \textbf{T}$$

or

$\theta_i(\hat{x}_b)$ = **T** and $\theta_i(\hat{y}_a)$ = **T**.

In the first case from $\theta_i(\hat{C}_{i,5})$ = **T** we have

$\theta_i(\hat{z}_a)$ = **F**

and from $\theta_i(\hat{C}_{i,9})$ = **T** we have

$\theta_i(\hat{z}_b)$ = **F**.

Similarly, in the second case from $\theta_i(\hat{C}_{i,8})$ = **T** we have

$\theta_i(\hat{z}_b)$ = **F**

and from $\theta_i(\hat{C}_{i,6})$ = **T** we have

$\theta_i(\hat{z}_a)$ = **F**.

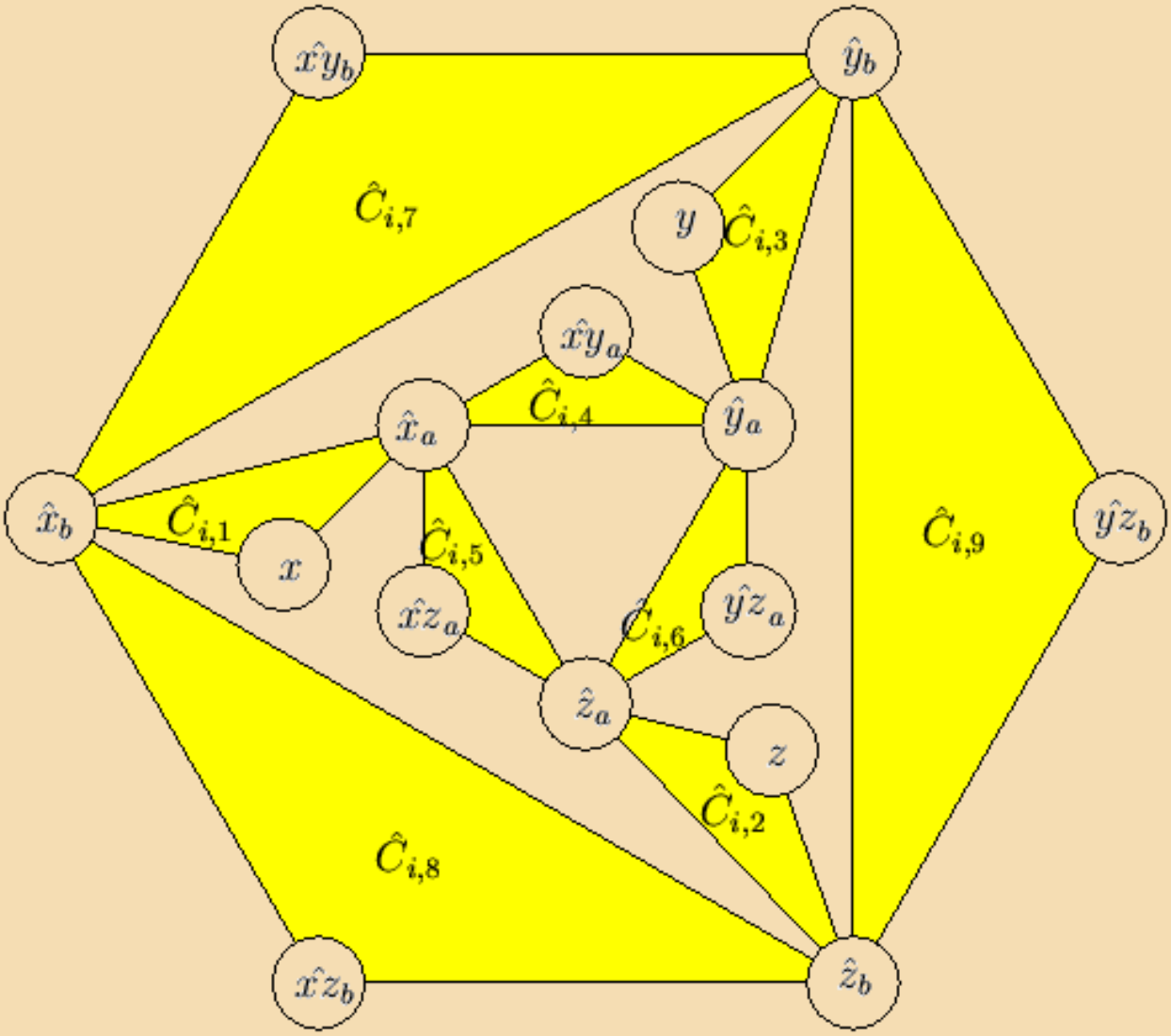
Thus, in either case we have

$\theta_i(\hat{z}_a)$ = **F** and $\theta_i(\hat{z}_b)$ = **F**

so that from $\theta_i(\hat{C}_{i,3})$ = **T** we have $\theta_i(z)$ = **T**.

♠ Choosing of the assignment θ_i in Proposition 11.7 can be viewed as a game on the following graph where one must choose *exactly one node* of each colored triangle. Note that doing so will require choosing exactly one of *x*, *y*, *z*.

Figure 11.3:*(1/3)SAT*Game



Definition 11.13 Let + *(1/3)SAT* denote the set of all satisfiable propositional formulas belonging to *(1/3)SAT* in which there are no negated variables, i.e., all literals are single variables.

Corollary 11.8 + *(1/3)SAT* is *NP*-complete.

Proof: Given a formula $B = C_1 \wedge \cdots \wedge C_n$ we first add two special variables *t* and *f* and the special clause

$$\hat{C}_t = (t \vee f \vee f).$$

Since exactly one literal in each clause must be assigned **T**, we see that any such assignment which makes $\hat{C}_t$ true, must assign **T** to *t* and **F** to *f*. Then for each variable $x \in Var(B)$, we introduce a new variable $\bar{x}$, and the clause

$$\hat{C}_x = (x \vee \bar{x} \vee f).$$

Thus, any appropriate assignment to $\hat{C}_x$ which makes $\hat{C}_x$ true, must assign the opposite truth values to x and $\bar{x}$, so $\bar{x} \equiv \neg x$. Then, we replace each clause C_i by the clause $\hat{C}_i$, where $\hat{C}_i$ is obtained by replacing every negated literal of the form $\neg x$ with the positive literal $\bar{x}$.

Next Up Previous Contents Index

Next: 11.3 Polynomial Time Reducibility **Up:** 11. Non-Deterministic Computations **Previous:** 11.1 Complexity of Non-Deterministic Programs

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 11.4 Finite Automata (Review) **Up:** 11. Non-Deterministic Computations **Previous:** 11.2 NP-Completeness

11.3 Polynomial Time Reducibility

We can now show that many other problems X are *NP*-complete by reducing + *(1/3)SAT* to X and using Proposition 11.5.

Definition 11.14 Let Σ be a finite alphabet and let $V = \{x_i\}$ be a set of symbols which is disjont from Σ. The symbols of V are called *string variables*. A *pattern* π is any non-null string over $\Sigma \cup V$. Let π be a pattern which contains n different variables. Without loss of generality we may assume that the variables of π are $x_1,..., x_n$. Given non-empty strings $s_1,..., s_n \in \Sigma^+$, then $\pi[x_1 \leftarrow s_1,..., x_n \leftarrow s_n]$ is the result of simultaneously substituting s_j for all occurrences of x_j, for all $1 \leq j \leq n$. The *pattern language* L_π generated by π is defined by

$$L_\pi = \{ \pi[x_1 \leftarrow s_1,..., x_n \leftarrow s_n] : s_1,..., s_n \in \Sigma^+ \}.$$

Definition 11.15 Define *PATMEM* as the set of all pairs $\langle \pi, t \rangle$, where π is a pattern and $t \in \Sigma^+$, such that $t \in L_\pi$.

Proposition 11.9 *PATMEM* is *NP*-complete.

Proof: It is easy to see that given π and t a non-deterministic algorithm can simply guess strings $s_1,..., s_n$ wuch that $\pi[x_1 \leftarrow s_1,..., x_n \leftarrow s_n] = t$ and verify this fact in polynomial time, since all s_j must satisfy $|s_j| \leq |t|$. Thus, *PATMEM* $\in \mathbb{NP}$.

To see that *PATMEM* is *NP*-complete we show that + *(1/3)SAT* $\leq_p$ *PATMEM*. Let $B = C_1 \wedge \cdots \wedge C_m$

be a CNF propositional formula, where $C_i = (w_{i,1} \vee w_{i,2} \vee w_{i,3})$, and where $w_{i,j}$ is a positive literal, and let $x_1,\ldots, x_n$ be the variables of B. Let **a**,**b** be two distinct symbols of Σ. We construct a pattern π whose string variables are identical to the propositional variables of B. The pattern π is defined by

$$\pi = \mathbf{a}\,\pi_1\,\mathbf{a}\,\pi_2\,\mathbf{a}\cdots\mathbf{a}\,\pi_m\,\mathbf{a}$$

where for each $1 \leq i \leq m$,

$$\pi_i = w_{i,1}w_{i,2}w_{i,3}.$$

The string t is defined by

$$t = \mathbf{a}t_1\mathbf{a}t_2\mathbf{a}\cdots\mathbf{a}t_m\mathbf{a}$$

where for each $1 \leq i \leq m$,

$$t_i = \mathbf{bbb}.$$

Suppose now that B is satisfiable by a truth assignment θ which assigns **T** to *exactly one literal per clause*. We then construct a string assignment σ to the string variables of π as follows:

$$\sigma(x_j) = \begin{cases} \mathbf{bb}, & \text{if } \theta(x_j) = \mathbf{T} \\ \mathbf{b}, & \text{if } \theta(x_j) = \mathbf{F}. \end{cases}$$

Since θ makes exactly one literal per clause true (and two literals per clause false), σ assigns to π_i the string **bbbb** = t_i. Therefore, $t \in L_\pi$.

Suppose on the other hand that $t \in L_\pi$ and let σ be the corresponding assignment of strings from

Σ^+ to the variables of π. Then, clearly each π_i must generate the string t_i = **bbbb**, so that for each *i* exactly one of $w_{i,1}$, $w_{i,2}$, $w_{i,3}$ is assigned the string **bb** by σ, and the other two are assigned the string **b** by σ. We then construct a truth assignment θ to the propositional variables of *B* as follows:

$$\theta_{(x_j)} = \begin{cases} \mathbf{T}, & \text{if } \sigma(x_j) = \mathbf{bb} \\ \mathbf{F}, & \text{if } \sigma(x_j) = \mathbf{b}. \end{cases}$$

It is clear that θ assigns **T** to exactly one literal per clause of *B*.

Definition 11.16 An instance of the Knapsack Problem (denoted by $\langle s_1,..., s_n;c \rangle$) consists of a set of integers $s_1,..., s_n$, called *sizes*, and an integer *c*, called the *capacity*. An instance of the Knapsack Problem is called *solvable* if and only if there is some set of indices $J \subseteq \{1,..., n\}$ such that $c = \sum_{j \in J} s_j$. We define *KNAPSACK* as the set of all solvable instances of the Knapsack Problem.

Proposition 11.10 *KNAPSACK* is *NP*-complete.

Proof: It is easy to see that *KNAPSACK* $\in \mathbb{NP}$, since a non-deterministic algorithm can, given an instance $\langle s_1,..., s_n;c \rangle$

1. guess a subset $J \subseteq \{1,..., n\}$, and
2. verify that $c = \sum_{j \in J} s_j$.

We show that + *(1/3)SAT* $\leq_p$ *KNAPSACK*. Let $B = C_1 \wedge \cdots \wedge C_m$ be a CNF propositional formula,

where $C_i = (w_{i,1} \vee w_{i,2} \vee w_{i,3})$, and where $w_{i,j}$ is a positive literal, and let $x_1,..., x_n$ be the variables of B. We define an instance $\langle s_1,..., s_n;c \rangle$ of the Knapsack Problem as follows:

For each variable x_j we define a weight

$$s_j = \sum_{i \in I_j} 4^i,$$

where $I_j = \{i : x_j \text{ occurs in } C_i\}$. The knapsack capacity is defined by

$$c = \sum_{i=1}^{m} 4^i.$$

Suppose $\langle s_1,..., s_n;c \rangle \in$ *KNAPSACK*. Let J be such that $c = \sum_{j \in J} s_j$. We define a truth assignment θ to $x_1,..., x_n$ as follows:

$$\theta(x_j) = \begin{cases} \mathbf{T}, & \text{if } j \in J \\ \mathbf{F}, & \text{otherwise.} \end{cases}$$

We first observe that since there are only three literals per clause each ``bit'' 4^i in the capacity c must be generated by some size s_j such that $i \in I_j$. Further, since the coefficient of 4^i is 1 (and not 2 or 3), the assignment θ must assign **T** to exactly one literal of each clause C_i. Thus, $B \in$ *+ (1/3)SAT*.

Suppose on the other hand that $B \in$ *+ (1/3)SAT*. Define

$$J = \{j : \theta(x_j) = \mathbf{T}\}.$$

Then it is easy to see that $\sum_{j \in J} s_j = \sum_{i=1}^{m} 4^i = c$. Thus, $\langle s_1,..., s_n;c \rangle \in$ *KNAPSACK*.

Next Up Previous Contents Index

Next: 11.4 Finite Automata (Review) **Up:** 11. Non-Deterministic Computations **Previous:** 11.2 NP-Completeness

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 11.5 PSPACE Completeness **Up:** 11. Non-Deterministic Computations **Previous:** 11.3 Polynomial Time Reducibility

11.4 Finite Automata (Review)

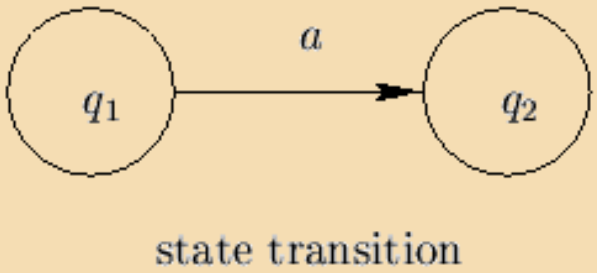
In this section we review the majors results for finite state machines. From Definition 2.1 we see that a deterministic finite automaton (*DFA*) M consists of $\langle \Sigma, Q, \delta, q_0, F \rangle$, where Σ is the input alphabet, Q is the finite set of states, q_0 is the start state, F is the set of final states, and $\delta : Q \times \Sigma \rightarrow Q$ is the state transition function.

♠ Observe that for a *DFA* the state transition function δ *must be defined* for *all* inputs and *all* states.

We depict the internal state transition behavior of M by means of a labelled directed graph G_M as follows:

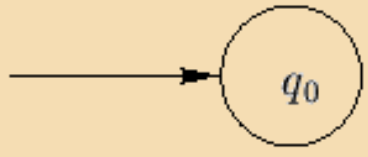
The nodes of G_M are the states of M, and there is a directed edge from q_1 to q_2 labelled a whenever $\delta(q_1, a) = q_2$, and is depicted as:

Figure 11.4: State Transition

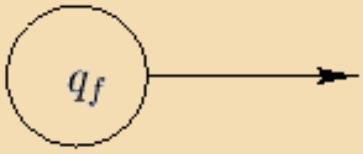


We also depict the initial state q_0 and final states $q_f \in F$ as:

Figure 11.5: Initial and Final States



initial state



final state

Definition 11.17 For any *DFA M* the language L_M accepted by *M* is the set of all input strings $x = a_1 \cdots a_n$ such that there is a path from the initial state q_0 to some final state $q_f \in F$ with label $a_1 \cdots a_n$, i.e.,

Figure 12.11: Accepting Computation Path

Definition 11.18 A *non-deterministic finite state automaton* (*NFA*) *M* consists of $\langle \Sigma, Q, \delta, I, F \rangle$, where Σ is the input alphabet, *Q* is the finite set of states, $I \subseteq Q$ is the set of start states, *F* is the set of final states, and $\delta : Q \times (\Sigma \cup \{\epsilon\}) \rightarrow 2^Q$ is the (non-deterministic) state transition function.

♠ Observe that we allow ϵ-transitions for *NFA*'s.

Definition 11.19 For any *NFA M* the language L_M accepted by *M* is the set of all input strings $x = a_1 \cdots a_n$ such that there is a path from some initial state $q_i \in I$ to some final state $q_f \in F$ with label $a_1 \cdots a_n$.

Theorem 11.11 The class of languages accepted by *NFA*'s is the same as the class of languages accepted by *DFA*'s.

Proof: ($\Longleftarrow$): This is immediate since given a *DFA* $M = \langle \Sigma, Q, \delta, q_0, F \rangle$, we construct an equivalent *NFA* $\hat{M} = \langle \Sigma, Q, \hat{\delta}, I, F \rangle$, where $I = \{q_0\}$ and $\hat{\delta}(q, a) = \{\delta(q, a)\}$.

($\Longrightarrow$): Let $M = \langle \Sigma, Q, \delta, I, F \rangle$ be an *NFA*. We construct an equivalent *DFA* $\hat{M} = \langle \Sigma, \hat{Q}, \hat{\delta}, \hat{q}_0, \hat{F} \rangle$ as follows:

$$\hat{Q} = 2^Q$$

$$\hat{q}_0 = I$$

$$\hat{F} = \{X \subseteq Q : X \cap F \neq \emptyset\}$$

$$\hat{\delta}(X, a) = \bigcup_{q \in X} \delta(q, a)$$

Thus, the states of $\hat{M}$ are subsets of states of *M*. Then one completes the proof by showing that $\hat{M}$ on input *x* enters state $X \in 2^Q$ if and only if *M* on input *x* could enter (via the right choices) each sate $q \in X$.

♠ If the *NFA M* has *n* states, then the equivalent *DFA* $\hat{M}$ has 2^n states.

Proposition 11.12 Every regular language is accepted by some finite state automaton.

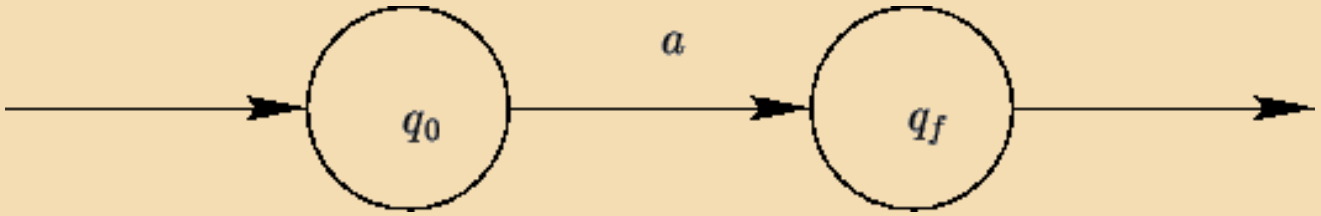
Proof: Let *r* be a regular expression. Then, an *NFA* with ϵ transtions *M* such that $L_M = L_r$ is defined by induction on the length of *r* as follows:

- **Induction Basis:**
  - **Case 1: *r* = $\emptyset$:**

Figure 11.7:NFA for $\emptyset$

- **Case 2: *r* = *a*, where *a* $\in \Sigma \cup \{\epsilon\}$:**

Figure 11.8:NFA for $a \in \Sigma \cup \{\epsilon\}$



- **Induction Step:**

Let

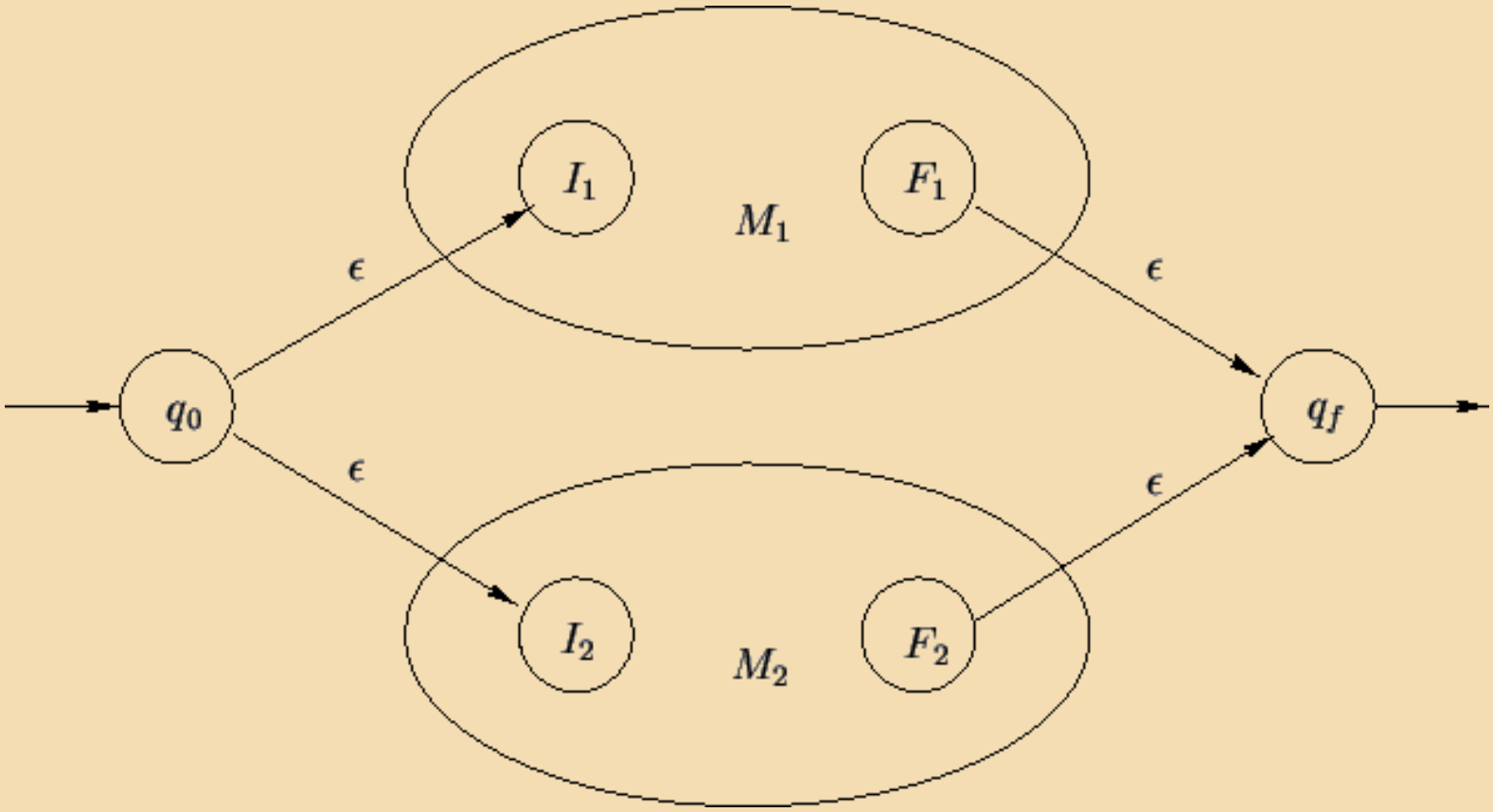
$$M_1 = \langle \Sigma, Q_1, \delta_1, I_1, F_1 \rangle$$

$$M_2 = \langle \Sigma, Q_2, \delta_2, I_2, F_2 \rangle$$

be *NFA*'s such that $L_{M_1} = L_{r_1}$ and $L_{M_2} = L_{r_2}$.

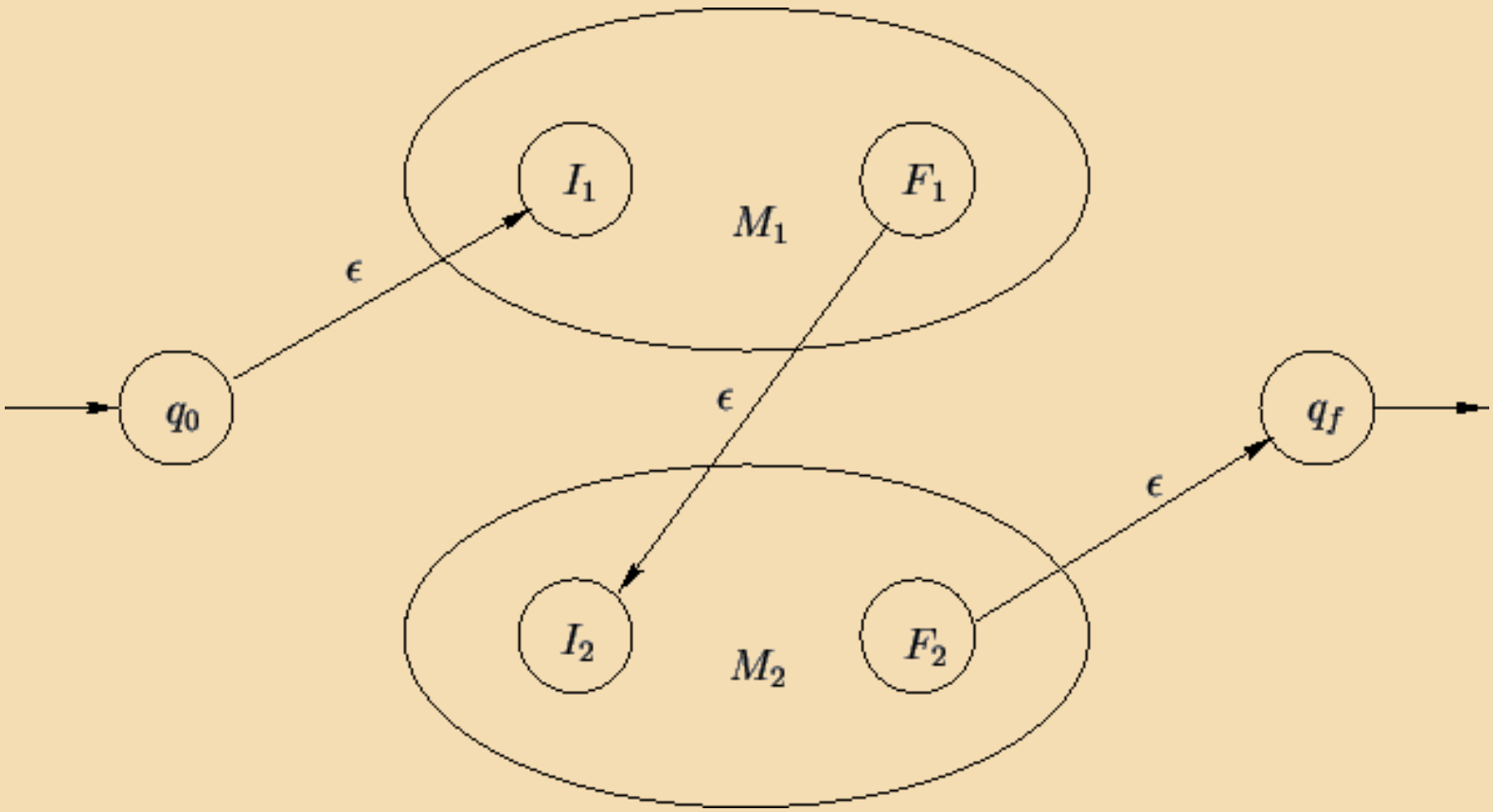
- **Case 1: $r = r_1 \cup r_2$:**

Figure 11.9:NFA for $r_1 \cup r_2$



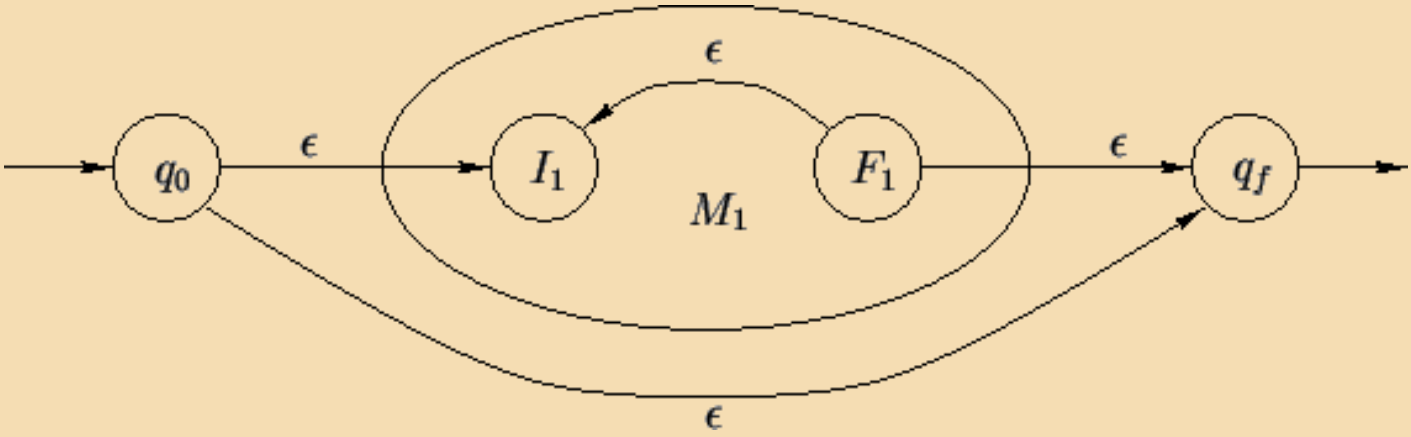
- **Case 2: $r = r_1 \cdot r_2$:**

Figure 11.10: NFA for $r_1 \cdot r_2$



Case 3: $r = r_1^*$:

Figure 11.11: NFA for r_1^*



If the length of the regular expression r is n (excluding parentheses), then the number of states of the equivalent *NFA* is $2n$.

Proposition 11.13 For every *NFA* M there is some regular expression r such that $L_M = L_r$.

Proof: Let $M = \langle \Sigma, Q, \delta, I, F \rangle$. The main idea is to compute the transitive closure of the (labelled) edge relation given by δ in G_M. More precisely, we construct via the standard transitive closure algorithm a regular expression r that describes the set of all labels of accepting paths in G_M. Suppose $Q = \{q_1, ..., q_n\}$. Let r_{ij}^0 be a regular expression which denotes the finite set of labels from q_i to q_j in G_M. Since every finite set of strings is a regular language, such a regular expression clearly exists. Consider the following algorithm from computing the transitive closure:

for $1 \leq i \leq n$ **do**

$$r_{ii}^0 \leftarrow r_{ii}^0 \cup \epsilon$$

endfor

for $1 \leq k \leq n$ **do**

for $1 \leq i, j \leq n$ **do**

$$r_{ij}^{k} \leftarrow r_{ij}^{k-1} \cup r_{ik}^{k-1} \cdot (r_{kk}^{k-1})^{*} \cdot r_{kj}^{k-1}$$

endfor

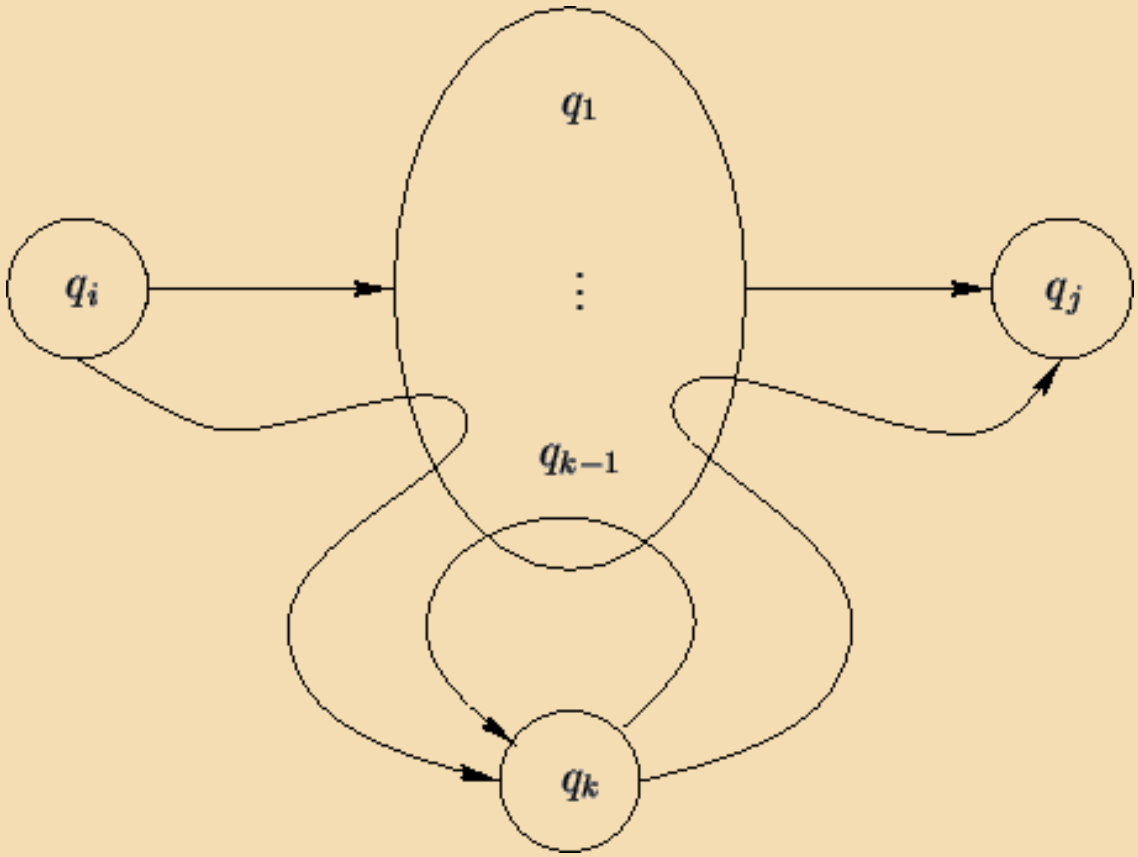
endfor

The required regular expression r is given by

$$r = \bigcup_{q_i \in I, q_j \in F} r_{ij}^{n}.$$

The correctness of the regular expression can be shown by proving by induction on k for $1 \leq k \leq n$ that r_{ij}^{k} describes the set of all labels of paths from q_i to q_j via the intermediate nodes $\{q_1, ..., q_k\}$.

Figure 11.12: Paths from q_i to q_j via $\{q_1, ..., q_k\}$



Theorem 11.14 The class of regular languages is precisely the class of languages accepted by finite state automata.

Proposition 11.15 The class of regular languages is closed under complementation.

Proof: Let L be a regular language and let $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ be a *DFA* such that $L_M = L$. Define

$$\bar{M} = \langle \Sigma, Q, \delta, q_0, Q - F \rangle.$$

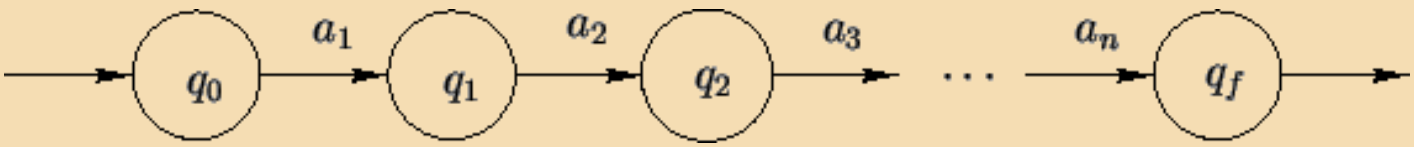
Then, $x \in L\,\bar{M} \iff x \notin L_M$. Thus, $L\,\bar{M} = \Sigma^* - L = \bar{L}$, so $\bar{L}$ is a regular language.

Theorem 11.16 (Pumping Lemma for Regular Languages) For every regular language L there is a positive integer p (called the *pumping length*) such that for all $s \in L$ if $|s| \geq p$, then there exist strings x, y, z such that $s = x \cdot y \cdot z$ and

1. $|y| > 0$,
2. $|xy| \leq p$, and
3. for all $i \geq 0$, $xy^iz \in L$.

Proof: Suppose L is a regular language and that $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ is a *DFA* such that $L_M = L$. Choose $p = \#Q$. Let $s \in L$ be such that $|s| \geq p$. Thus, $s = a_1 \cdots a_n$, where $n \geq p$. Consider the accepting path for s:

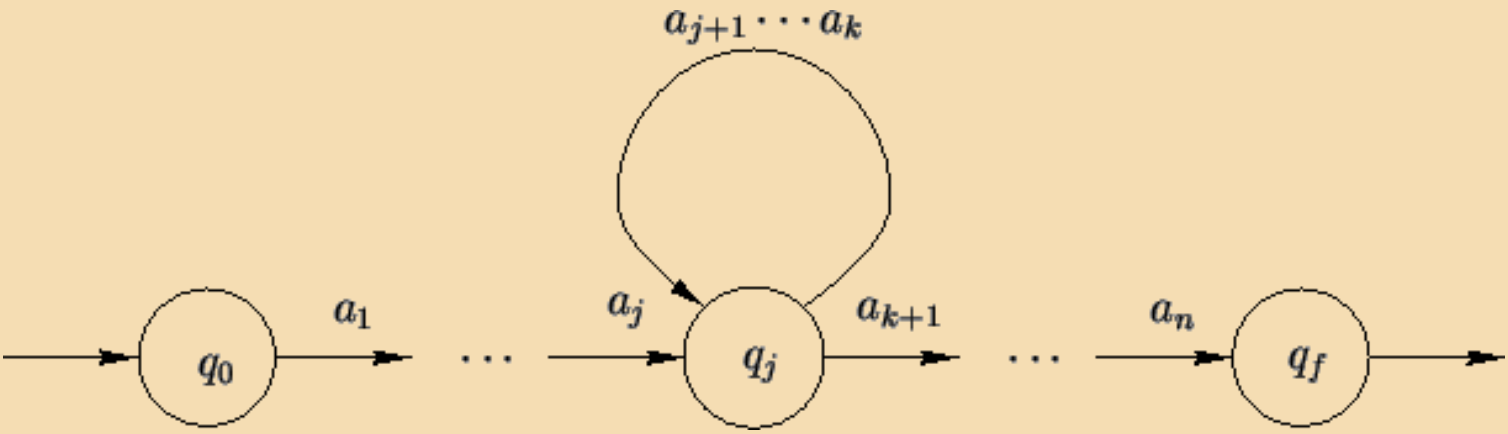
Figure 12.11: Accepting Computation Path



Since there are $n + 1 > p$ states in this accepting path, there must exist two (least) indices $j < k$ such that $q_j = q_k$.

Thus, overlaying q_j and q_k to form a loop we have:

Figure 11.14: Accepting Computation Path with Loop



Choose $x = a_1 \cdots a_j$ (the part *before* the loop), $y = a_{j+1} \cdots a_k$ (the *loop itself*), and $z = a_{k+1} \cdots a_n$ (the part *after* the loop). Since $j < k$, we have $|y| > 0$, and since we chose the least pair $j < k$ such that $q_j = q_k$, we have $|xy| \leq p$. Finally, the path consisting of the part from q_0 to q_j, followed by any number of times (including 0) around the loop, followed by the part from q_k to q_n is an accepting path, i.e., $xy^iz \in L$ for every $i \geq 0$.

Theorem 11.17 For every regular language L there exists a positive integer p such that $L \neq \emptyset$ if and only if $\exists s \in L$ such that $|s| < p$.

Proof: Let L be a regular language and let p be the pumping length as given by the Pumping Lemma above.

- **Case ($\Longleftarrow$):**

  Clearly, if $\exists s \in L$ such that $|s| < p$, then $L \neq \emptyset$.

- **Case ($\Longrightarrow$):**

  Suppose $L \neq \emptyset$, and let $s \in L$. If $|s| \geq p$, then by the Pumping Lemma for regular languages, s can be written as $s = xyz$ where $|y| > 0$, so the string $s_1 = xz \in L$ and $|s_1| < |s|$. By repeating this pruning process (if $|s_1| \geq p$) we must eventually obtain a string $s_1 \in L$ such that $|s_1| < p$.

Next Up Previous Contents Index

Next: 11.5 PSPACE Completeness **Up:** 11. Non-Deterministic Computations **Previous:** 11.3 Polynomial Time Reducibility

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 12. Formal Languages **Up:** 11. Non-Deterministic Computations **Previous:** 11.4 Finite Automata (Review)

11.5 PSPACE Completeness

Definition 11.20 A set X is called *PSPACE-complete* if and only if it is complete for the class $\mathbb{PSPACE}$ with respect to $\leq_p$, i.e., $X \in \mathbb{PSPACE}$ and $Y \leq_p X$ for all $Y \in \mathbb{PSPACE}$.

Definition 11.21 $\overline{\mathbb{RE}} \neq \emptyset$ is the set of all regular expressions r over Σ such that $L_r \neq \Sigma^*$, i.e., $\overline{L_r} \neq \emptyset$.

Theorem 11.18 $\overline{\mathbb{RE}} \neq \emptyset$ is PSPACE-complete.

The theorem follows from the following two propositions.

Proposition 11.19 $\overline{\mathbb{RE}} \neq \emptyset \in \mathbb{PSPACE}$.

Proof: Let r be a regular expression of size n, and let $M = \langle \Sigma, Q, \delta, I, F \rangle$ be an *NFA* with $2n$ states such that $L_M = L_r$. Let $\hat{M} = \langle \Sigma, \hat{Q}, \hat{\delta}, \hat{q}_0, \hat{F} \rangle$ be a *DFA* with 2^{2n} states such that $L\,\hat{M} = \overline{L_M}$ (i.e., the complement of L_M). Then, using Theorem 11.17, we have

$$L_r \neq \Sigma^* \iff L\,\hat{M} \neq \emptyset \iff \exists_z \in L\,\hat{M} \mid z \mid \leq 2^{2n}.$$

We give an algorithm that, when implemented on a *DRAM*, operates in $O(n^2)$ space and that decides whether or not $L\,\hat{M} \neq \emptyset$ by checking all paths in $G\,\hat{M}$ of length $\leq 2^{2n}$ to see if there is an accepting path. Actually, the algorithm cannot store the graph $G\,\hat{M}$ since it is of size *exponential* in n. Therefore, the algorithm will work with G_M instead. Let $Q = \{q_1, \ldots, q_{2n}\}$. The states of $\hat{M}$ will be coded as binary strings of length $2n$ in such a way that for all $X \in \hat{Q}$

$$\text{bit } i \text{ of state } X \text{ is } 1 \iff q_i \in X.$$

By Theorem 11.11,

$$X \in \hat{F} \iff (q \in X \implies q \notin F).$$

The algorithm first constructs the *NFA M* and stores G_M, which requires $O(n^2)$ space, and then executes the following program:

for $X \in \hat{F}$ **do**

    if *Access*(I, X, $2n$) **then**

        output(**true**)

    endif

endfor
output(**false**)

The recursive subroutine *Access*(x_1, x_2, m) is defined by:

input(X_1, X_2, Z)
if $Z = 0$ **then**

    if $X_1 = X_2$ **or** $\exists a \in \Sigma \quad X_2 = \hat{\delta}(X_1, a)$ **then**

        return(**true**)

    else

        return(**false**)

    endif

endif
for $0 \leq X \leq 2^{2n} - 1$ **do**

if *Access*(X_1, X, Z - 1) **and** *Access*(X, X_2, Z - 1) **then**

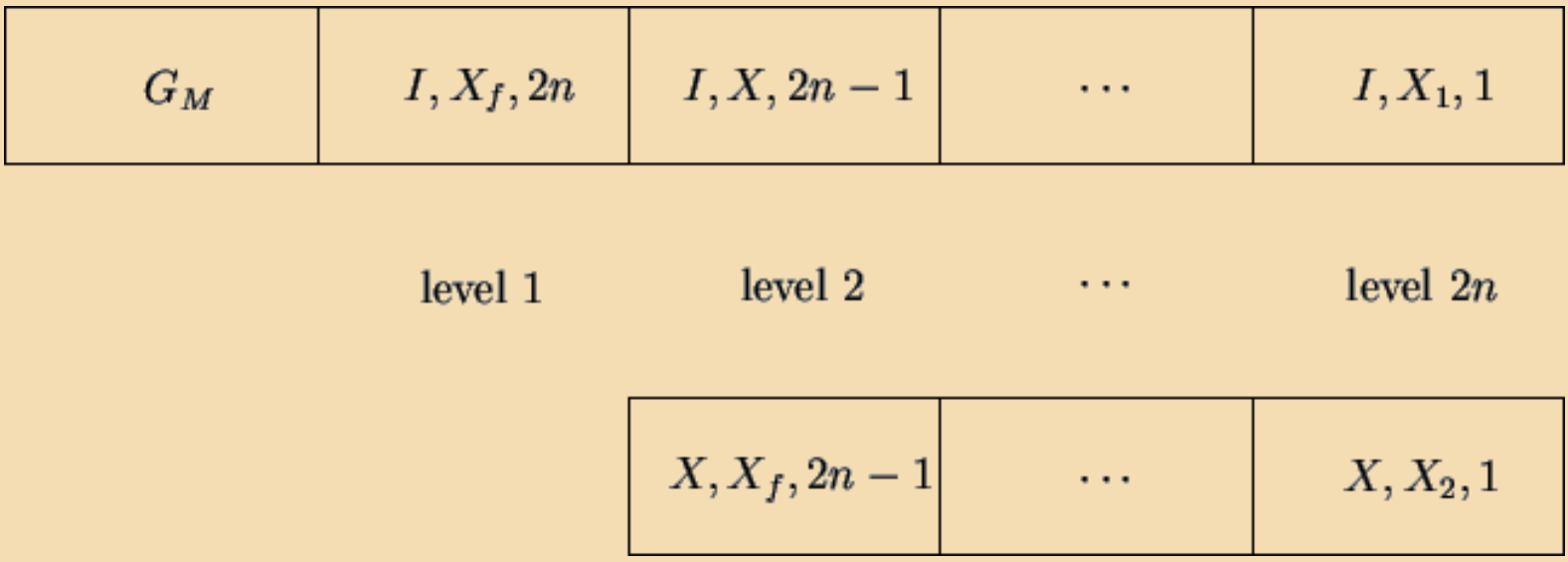
return(**true**)

endif

endfor
return(**false**)

In the subroutine *Access* checking whether or not $X_2 = \hat{\delta}(X_1, a)$ involves checking whether or not $\forall q_2 \in X_2 \exists q_1 \in X_1 \quad q_2 \in \delta(q_1, a)$, which can easily be done by consulting G_M. Since the path length examined doubles with each recursive call (beginning at the lowest level), it is clear that all paths of length $\leq 2^{2n}$ are examined. Each recursive call to *Access*(x_1, x_2, m) requires $O(n)$ space overhead for stacking the arguments x_1, x_2, m ($2n$ space for each of x_1 and x_2, and $\log_2 2n$ for m). The maximum depth of recursion is $2n$, so the total space used by the algorithm is $O(n^2)$.

Figure 11.15: Space Usage for Recursive Algorithm Access



In order to simplify the regular expression constructed in Proposition 11.20 below, we observe that for any *DRAM* program P over Σ_k which uses m registers (with one input instruction) we can construct a *DRAM* program $\hat{P}$ over $\Sigma_{k+1} = \Sigma_k \cup \{,\}$ that uses exactly one register and such that $L\,\hat{P} = L_P$ and for some constant c, $DRAMspace_{\hat{P}}(x) \leq c\ DRAMspace_P(x)$ for all x. The program $\hat{P}$ maintains in its one register a string of the form

$z_1 \cdot, \cdot \cdots \cdot, \cdot z_m \cdot$, where z_1,..., z_m are the current contents of registers R_1,..., R_m of P. It simulates each instruction of P by:

1. exposing the right or left end (depending on the type of instruction) of the register mentioned in the instruction (if any);
2. executing that instruction;
3. returning the contents of its one register to the canonical form which begins with the contents of R_1 at the left.

Proposition 11.20 For any $X \in \mathbb{PSPACE}$, $X \leq_p \overline{\mathbb{RE}} \neq \emptyset$.

Proof: Let $X \in \mathbb{PSPACE}$. Then there is some *DRAM* program P over Σ_k which uses one register and has one input statement, and there is some polynomial function p such that $x \in X \iff DRAMspace_P(x) \leq p(|x|)$. We will construct an alphabet Δ and for each x a regular expression r_x over Δ (in polynomial time) such that

$$x \in X \iff r_x \in \overline{\mathbb{RE}} \neq \emptyset$$

$$\iff \overline{L_{r_x}} \neq \emptyset.$$

In other words,

$$x \notin X \iff L_{r_x} = \Delta^*$$

The construction of r_x will be similar to the construction of B_x in Theorem 11.3 in that we will use regular expressions to describe computations. More precisely, the regular expression over Δ which we construct will describe *non-accepting* computations, i.e., if $x \notin X$, then *every* string in Δ^* represents a non-accepting computation.

As in Theorem 11.3 we let $x = a_1 \cdots a_n$, so $|x| = n$, and let m be the number of lines of P. We use s_j to denote the symbol (if any) mentioned in line j of P, and g_j to denote the goto part (if any) of line j. We will represent the

register contents by a left justified string of length exactly $p(n)$ over Σ_{k+1}, where the k+1st symbol represents a blank (depicted by $\sqcup$). We will encode line numbers by using finitely many special additional symbols $\Gamma = \{b_1,..., b_m\}$ not belonging to Σ_{k+1}. The state of P at any point in time will be represented by the string of length $p(n) + 1$ of the form

$$b_j \cdot z$$

where j is the current line number of P and z represents the current contents of the (only) register of P. Finally, a computation string will be represented by the string

$$y_0 \cdot \cdots \cdot y_t \cdot b_m$$

where t is the number of steps of P on input x, and y_i is the representation of the state of P at the i^{th} step.

The regular expression $r_x = r_1 \cup r_2 \cup r_3 \cup r_4 \cup r_5$, where

1. r_1 describes all strings which don't represent accepting computations because they are syntactically ill-formed;
2. r_2 describes all computation strings which don't start correctly;
3. r_3 describes all computation strings which don't end correctly;
4. r_4 describes all computation strings in which some line number does not follow correctly from the previous state;
5. r_5 describes all computation strings in which the register contents does not follow correctly from the previous state.

We will use the following abbreviations:

- if W is a finite set of symbols $\{w_1,..., w_s\}$, then we use W to stand for the regular expression $w_1 \cup \cdots \cup w_s$.
- if r is a regular expression, then we use r^i for the concatenation of r with itself i times, where $r^0 = \epsilon$.

Define $\Delta = \Sigma_{k+1} \cup \Gamma$.

1.

$r_1 = r_{1,1} \cup \cdots \cup r_{1,6}$, where

$r_{1,1} = \Sigma_{k+1}^*$ (no line number)

$r_{1,2} = \Sigma_{k+1}^* \cdot \Gamma \cdot \Sigma_{k+1}^*$ (only 1 line number)

$r_{1,3} = \Sigma_{k+1} \cdot \Delta^*$ (line number not first)

$r_{1,4} = \Delta^* \cdot \Sigma_{k+1}$ (line number not last)

$r_{1,5} = \Delta^* \cdot \Gamma \cdot \Sigma_{k+1}^{p(n)+1} \cdot \Delta^*$ (contents too long)

$r_{1,6} = r_{1,6,0} \cup \cdots \cup r_{1,6,p(n)-1}$ (contents too short)

where for all $0 \leq j \leq p(n) - 1$

$r_{1,6,j} = \Delta^* \cdot \Gamma \cdot \Sigma_{k+1}^j \cdot \Gamma \cdot \Delta^*$.

2.

$r_2 = r_{2,1} \cup r_{2,2}$, where

$r_{2,1} = (\Gamma - b_1) \cdot \Delta^*$ (wrong initial line number)

$r_{2,2} = b_1 \cdot \Sigma_{k+1}^* \cdot \Sigma_k \cdot \Delta^*$ (initial contents not blank)

3.

$r_3 = \Delta^* \cdot (\Gamma - b_m)$

4.

$r_4 = r_{4,1} \cup \cdots \cup r_{4,m}$, where for all $1 \leq j < m$:

- if j is not a conditional jump instruction

$r_{4,j} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^* \cdot (\Gamma - b_{j+1}) \cdot \Delta^*$

- if j is a conditional jump statement $r_{4,j} = r_{4,j,1} \cup r_{4,j,2}$, where

$$r_{4,j,1} = \Delta^* \cdot b_j \cdot s_j \cdot \Sigma_{k+1}^* \cdot (\Gamma - b_{g_j}) \cdot \Delta^*$$

$$r_{4,j,2} = \Delta^* \cdot b_j \cdot (\Sigma_{k+1} - s_j) \cdot \Sigma_{k+1}^* \cdot (\Gamma - b_{j+1}) \cdot \Delta^*$$

5.

$r_5 = r_{5,1} \cup \cdots \cup r_{5,m}$, where for each $1 \leq j \leq m$:

- if j is the input instruction $r_{5,j} = r_{5,j,1} \cup \cdots \cup r_{5,j,n+1}$, where

  for each $1 \leq i \leq n$

$$r_{5,j,i} = b_1 \cdot \Sigma_{k+1}^* \cdot b_2 \cdot \Sigma_{k+1}^{i-1} \cdot (\Sigma_{k+1} - a_i) \cdot \Delta^*$$

  and

$$r_{5,j,n+1} = b_1 \cdot \Sigma_{k+1}^* \cdot b_2 \cdot \Sigma_{k+1}^n \cdot \Sigma_{k+1}^* \cdot \Sigma_k \cdot \Delta^*$$

- if j is a conditional jump instruction $r_{5,j} = \bigcup_{a \in \Sigma_{k+1}} r_{5,j,a}$, where

$$r_{5,j,a} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^* \cdot a \cdot \Delta^{p(n)} \cdot (\Sigma_{k+1} - a) \cdot \Delta^*$$

- if j is a left shift instruction $r_{5,j} = r_{5,j,0} \cup \bigcup_{a \in \Sigma_{k+1}} r_{5,j,a}$, where

$$r_{5,j,a} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^+ \cdot a \cdot \Delta^{p(n)-1} \cdot (\Sigma_{k+1} - a) \cdot \Delta^*$$

$$r_{5,j,0} = \Delta^* \cdot b_j \cdot \Delta^{2p(n)} \cdot \Sigma_k \cdot \Delta^*$$

- if j is a successor instruction $r_{5,j} = r_{5,j,0} \cup r_{5,j,1} \cup \bigcup_{a \in \Sigma_k} r_{5,j,a}$, where

$$r_{5,j,a} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^* \cdot a \cdot \Delta^{p(n)} \cdot (\Sigma_{k+1} - a) \cdot \Delta^*$$

$$r_{5,j,0} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^* \cdot \sqcup \cdot \sqcup \cdot \Delta^{p(n)} \cdot \Sigma_k \cdot \Delta^*$$

$$r_{5,\,j,\,1} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^* \cdot \Sigma_k \cdot \sqcup \cdot \Delta^{p(n)} \cdot (\Sigma_{k+1} - s_j) \cdot \Delta^*$$

♠ We observe that the alphabet over which the regular expression is defined depends on the program *P*. We can use a fixed alphabet $\Sigma = \{0, 1\}$ by coding the i^{th} symbol of Δ by the string $1 \cdot 0^i$.

Theorem 11.21 $\mathbb{PSPACE} = \mathbb{NSPACE}$.

Proof: Since $\overline{\mathbb{RE}} \neq \emptyset \in \mathbb{PSPACE}$, it suffices to show for every $X \in \mathbb{NSPACE}$ that $X \leq_p \overline{\mathbb{RE}} \neq \emptyset$. As in Proposition 11.20 we may assume that $X = L_P$ for some *NRAM* program *P* over Σ_k with one register and one input statement. Thus, we need only show how to modify the construction of Proposition 11.20 to handle **njp** instructions. If line *j* is an **njp** instruction (with goto parts g_j^1 and g_j^2), then $r_{5,\,j}$ is the same as for comditional jump instructions, and

$$r_{4,\,j} = \Delta^* \cdot b_j \cdot \Sigma_{k+1}^* \cdot (\Gamma - \{b_{g_j^1}, b_{g_j^2}\}) \cdot \Delta^*$$

Theorem 11.21 is usually obtained as a corollary to the following Theorem (known as Savitch's Theorem).

Theorem 11.22 Let *S* be a function satisfying the following conditions:

1. $S(n) \geq \log_2 n$;
2. for some *DRAM* program *P*, $S(|x|) = DRAMspace_P(x)$ for all *x*.

Then, for any *NRAM* program *P* such that $NRAMspace_P(x) \leq S(|x|)$ there is an equivalent *DRAM* program $\hat{P}$ such that $L_P = L\,\hat{P}$ and there is a constant c_1 such that $DRAMspace\,\hat{P}(x) \leq c_1\ (S(|x|))^2$.

Proof: As usual we may assume that *P* is an *NRAM* program over Σ_k with one register and one input statement. Let *P* have *m* lines. Let *x* be some input to *P* with $|x| = n$. As in Theorem 11.20 we represent a state of *P* by a string of length $S(n) + 1$ of the form $b_j \cdot z$, where b_j is a special symbol representing line *j* ($\Gamma = \{b_1, ..., b_m\}$) and *z* is a

string of length $S(n)$ that represents the contents of P's one register. We construct a state transition graph G_P for the computation of P on input x as follows. The nodes of G_P are the strings belonging to $\Gamma \cdot \Sigma_k^{S(n)}$. G_P will be similar to the state transition graph of an *NFA* except that the edges will not be labelled with input symbols, but rather an edge from one state to another will mean that it is possible to go from the first state to the second by executing the current instruction of P with the current register contents.

Then, there is an edge from $b_j \cdot z_1$ to $b_i \cdot z_2$ if and only if

1. line j is an input instruction, $z_1 = \epsilon$, $i = j + 1$, and $z_2 = x$;
2. line j is a conditional jump instruction, z_1 begins with s_j, $i = g_j$, and $z_2 = z_1$.
3. line j is a conditional jump instruction, z_1 does not begin with s_j, $i = j + 1$, and $z_2 = z_1$.
4. line j is a non-deterministic jump instruction, $z_1 = z_2$, and either $i = g_j^1$ or $i = g_j^2$;
5. line j is a successor instruction, $i = j + 1$ and $z_2 = z_1 \cdot s_j$;
6. line j is a left shift instruction, $i = j + 1$, and $z_1 = a \cdot z_2$ for some $a \in \Sigma_k$.

Then initial state of G_M is $b_1 \cdot \epsilon$ and the set of final states is

$$F = \bigcup_{i=0}^{S(n)} b_m \cdot \Sigma_k^i .$$

The rest of the proof proceeds as in the proof of Theorem 11.20. If there is an accepting computation for P on input x, then there must be some path from the initial state to some final state of length $\leq m\ k^{S(n)}$, since otherwise there would be a loop in the non-deterministic computation which could be eliminated. We can rewrite $m\ k^{S(n)} \approx 2^{c\ S(n)}$ for some constant c. We then use the same strategy as in Theorem 11.20 to search by divide-and-conquer the graph G_P for such an accepting computation path, i.e., we execute the following program.

for $X \in F$ **do**

if *Access*($b_1 \cdot \epsilon$, X, $c\ S(n)$) **then**

output(**true**)

endif

endfor
output(**false**)

The recursive soubroutine *Access*(x_1, x_2, m) is defined by:

input(X_1, X_2, Z)
if $Z = 0$ **then**

if $X_1 = X_2$ **or** $(X_1, X_2) \in G_P$ **then**

return(**true**)

else

return(**false**)

endif

endif
for $1 \leq X \leq 2^{c\ S(n)}$ **do**

if *Access*(X_1, X, Z - 1) **and** *Access*(X, X_2, Z - 1) **then**

return(**true**)

endif

endfor
return(**false**)

The algorithm does not store G_P, but rather stores a copy of the program P, that it uses to decide whether of not (X_1, X_2) $\in$ G_P, for any states X_1 and X_2. This can be done without using very much space. Again, as in the proof of Theorem 11.20 an analysis of the space required to store the recursive subroutine calls to *Access* shows that the

total space used is bounded by $c_1 \ (S(n))^2$, for some constant c_1.

Next Up Previous Contents Index

Next: 12. Formal Languages **Up:** 11. Non-Deterministic Computations **Previous:** 11.4 Finite Automata (Review)

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 12.1 Grammars **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 11.5 PSPACE Completeness

12. Formal Languages

Next Up Previous Contents Index

Next: 12.1 Grammars **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 11.5 PSPACE Completeness

Bob Daley
2001-11-28

12.1 Grammars

Example 12.1 (English fragment)

⟨sentence⟩ = ⟨noun phrase⟩ ⟨verb phrase⟩ ⟨noun phrase⟩

⟨noun phrase⟩ = ⟨noun⟩ | ⟨adjective⟩ ⟨noun phrase⟩

⟨verb phrase⟩ = ⟨verb⟩ | ⟨adverb⟩ ⟨verb phrase⟩

⟨adjective⟩ = big | small | black | white | ...

⟨adverb⟩ = slowly | quickly | secretly | ...

⟨noun⟩ = boy | dog | cat | girl | ...

⟨verb⟩ = likes | hates | hits | desires | ...

This fragment generates (or derives) the following:

big black dog hits small boy
small cat secretly desires big black cat

But it also generates:

big small ⟨noun phrase⟩ ⟨verb⟩ black cat

The former are called *sentences* and the latter are called *sentential forms*.

Definition 12.1 A *grammar* G is denoted by $\langle \Sigma, V, R, S \rangle$, where

- Σ is a finite set of symbols called *terminals*;

- V is a finite set of symbols disjoint from Σ called *variables* (or *non-terminals*);
- R is a finite set of *productions* (or *rewrite rules*) of the form $x \rightarrow y$, where $x, y \in (\Sigma \cup V)^*$ and $x \neq \epsilon$;
- $S \in V$ is a special symbol called the *start symbol* (or *axiom*).

Definition 12.2 If $x \rightarrow y$ is a production of the grammar G and $w, z \in (\Sigma \cup V)^*$, then we say that *wxz directly derives wyz* in G (written $wxz \Rightarrow wyz$). Also, we say that x_1 *derives* x_n in G (written $x_1 \Rightarrow^* x_n$) if and only if there exist strings $x_2,..., x_{n-1} \in (\Sigma \cup V)^*$ such that $x_1 \Rightarrow x_2$, $x_2 \Rightarrow x_3,..., x_{n-1} \Rightarrow x_n$.

Definition 12.3 The language generated by the grammar G is defined by $L_G = \{x \in \Sigma^* : S \Rightarrow^* x\}$.

Proposition 12.1 For every grammar G, $x \Rightarrow y$ is a primitive recursive predicate.

Proof: Let $G = \langle \Sigma, V, R, S \rangle$, where $R = \{r_1,..., r_n\}$, and $r_i = x_i \rightarrow y_i$ for each $1 \leq i \leq n$. Then,

$$x \Rightarrow y \equiv D_{r_1}(x, y) \vee \cdots \vee D_{r_n}(x, y),$$

where

$$D_{r_i} \equiv \exists_{u \leq x} \exists_{v \leq x} \; x = u \cdot x_i \cdot v \wedge y = u \cdot y_i \cdot v.$$

Theorem 12.2 For every grammar G the language L_G is recursively enumerable.

Proof: We first code derivations $x_1 \Rightarrow x_2$, $x_2 \Rightarrow x_3,..., x_{n-1} \Rightarrow x_n$ by $\langle x_1,..., x_n \rangle$. Then, the partial

recursive function ϕ such that **dom** $\phi = L_G$ is given by

$$\phi(x) = \text{minz}(z = \langle x_1,..., x_n \rangle \text{ and } x_1 = S \text{ and } x_n = x$$
$$\text{and } \forall m < n \; x_m \Rightarrow x_{m+1})$$

Theorem 12.3 For every *NRAM* program P there is a grammar G such that $L_P = L_G$.

Proof: We first observe that since the grammar G must *output* every string that P accepts on *input*, derivations in G will correspond to the *reverse* of accepting computations. Thus, it will not matter whether or not P is deterministic, since even if it were certain instructions result in a *loss of information* (i.e., are not *reversible*). For example, a left shift instruction loses the information regarding the leftmost symbol, so that in reversing such an instruction one must *guess* which symbol was deleted in the actual instruction execution. We will assume that P is an *NRAM* program over Σ_k with exactly one register and one input instruction. Suppose P has m lines. As usual we use s_j, g_j, g_j^1, and g_j^2 to denote the specific items mentioned in the instruction at line j of the program P. The current global state of program P during its execution will be represented by the string $\vdash \cdot b_j \cdot z \cdot \dashv$, where j is the current line number and z is the current register contents.

Then, the grammar G for P is defined as follows:

$$G = \langle \Sigma_k, \Gamma, R, S \rangle,$$

where $\Gamma = \{b_1,..., b_m\} \cup \{\vec{b}_1,..., \vec{b}_m\} \cup \{\overleftarrow{b}_1,..., \overleftarrow{b}_m\} \cup \{S, \vdash, \dashv\}$, and the set R of productions is defined to contain for each line j of P the following rules:

1. if j is a conditional jump instruction, then

$$b_{g_j} \cdot s_j \rightarrow b_j \cdot s_j$$

$$b_{j+1} \cdot a \rightarrow b_j \cdot a \qquad \text{for all } a \in \Sigma_k - \{s_j\}$$

2. if j is a non-deterministic jump instruction, then

$$b_{g_j^1} \rightarrow b_j$$

$$b_{g_j^2} \rightarrow b_j$$

3. if j is a left shift instruction, then

$$b_{j+1} \rightarrow b_j \cdot a \qquad \text{for all } a \in \Sigma_k$$

4. if j is a successor instruction, then

$$b_{j+1} \rightarrow \vec{b}_j$$

$$\vec{b}_j \cdot a \cdot c \rightarrow a \cdot \vec{b}_j \cdot c \qquad \text{for all } a, c \in \Sigma_k$$

$$\vec{b}_j \cdot s_j \cdot \dashv \rightarrow \overleftarrow{b}_j \cdot \dashv$$

$$a \cdot \overleftarrow{b}_j \rightarrow \overleftarrow{b}_j \cdot a \qquad \text{for all } a \in \Sigma_k$$

$$\vdash \cdot \overleftarrow{b}_j \rightarrow \vdash \cdot b_j$$

5.

if j is the input instruction (i.e., $j = 1$), then

$$\vdash \cdot b_2 \rightarrow b_1$$

$$b_1 \cdot a \rightarrow a \cdot b_1 \qquad \text{for all } a \in \Sigma_k$$

$$b_1 \cdot \dashv \rightarrow \epsilon$$

6.

if j is the output instruction (i.e., $j = m$), then

$$S \rightarrow \vdash \cdot b_m \cdot \dashv$$

$$b_m \rightarrow b_m \cdot a \qquad \text{for all } a \in \Sigma_k$$

The way in which the grammar G generates an output x which P accepts is to first generate by the rules for the output instruction the final contents of P's register when it reached the output instruction during some accepting computation. Then it successively reverses each instruction execution during the computation (guessing appropriate values). When it reaches the input instruction (so the contents of the register should be x) it erases all the special symbols $\vdash$, b_1, $\dashv$ leaving the terminal string x.

Theorem 12.4 A set X is recursively enumerable if and only if $X = L_G$ for some grammar G.

Corollary 12.5 A set X is accepted by some *NRAM* program if and only if X is accepted by a *DRAM* program.

♠ Given the equivalences between languages generated by grammars and recursively enumerable sets, because of Rice's Theorem we see that most questions about the properties of languages generated by grammars are algorithmically undecidable.

Next Up Previous Contents Index

Next: 12.2 Chomsky Classification of Languages **Up:** 12. Formal Languages **Previous:** 12. Formal Languages

Bob Daley

2001-11-28

Next Up Previous Contents Index
Next: 12.3 Context Sensitive Languages **Up:** 12. Formal Languages **Previous:** 12.1 Grammars

12.2 Chomsky Classification of Languages

Table 12.1: Chomsky Hierarchy

Name	Productions	Acceptor
grammar	arbitrary	(Non-det.)
		RAM Programs
context-sensitive	$x \rightarrow y$,	Non-det.
(CSG)	with $\mid x \mid \leq \mid y \mid$	Linear Bounded Automata
	-or-	(LBA)
	$wAz \rightarrow wyz$,	
	with $A \in V, y \neq \epsilon$	
context-free	$A \rightarrow y$,	Non-det.
(CFG)	with $A \in V, y \neq \epsilon$	Push Down Automata
		(PDA)
right linear	$A \rightarrow yB$ or $A \rightarrow y$,	(Non-det.)
(RLG)	with $A, B \in V, y \in \Sigma^+$	Finite State Automaton
		(FSA)

Next Up Previous Contents Index
Next: 12.3 Context Sensitive Languages **Up:** 12. Formal Languages **Previous:** 12.1 Grammars
Bob Daley

2001-11-28

12.3 Context Sensitive Languages

Example 12.2 The grammar

$$S \rightarrow aBC$$

$$S \rightarrow SABC$$

$$CA \rightarrow AC$$

$$BA \rightarrow AB$$

$$CB \rightarrow BC$$

$$aA \rightarrow aa$$

$$aB \rightarrow ab$$

$$bB \rightarrow bb$$

$$bC \rightarrow bc$$

$$cC \rightarrow cc$$

generates the language $\{a^n b^n c^n : n \geq 1\}$.

For example, we have

$$S \Rightarrow SABC \Rightarrow aBCABC \Rightarrow aBACBC \Rightarrow aBABCC \Rightarrow aABBCC$$

$$\Rightarrow aaBBCC \Rightarrow aabBCC \Rightarrow aabbCC \Rightarrow aabbcC \Rightarrow aabbcc$$

Next Up Previous Contents Index

Next: 12.4 Linear Bounded Automata **Up:** 12. Formal Languages **Previous:** 12.2 Chomsky Classification of Languages

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: 12.5 Context Free Languages **Up:** 12. Formal Languages **Previous:** 12.3 Context Sensitive Languages

12.4 Linear Bounded Automata

Definition 12.4 A *linear bounded automaton* is an *NRAM* program P that operates in linear space, i.e., for some constant c

$$\forall x \in \textbf{dom } P \quad NRAMspace_P(x) \leq c \mid x \mid .$$

Theorem 12.6 For every context sensitive grammar G, there is a linear bounded automaton P such that $L_P = L_G$

Proof: Given some $x \in L_G$, the *NRAM* program P can ``guess" each step of a derivation of x by starting with S and at each step guessing the next string in the derivation and verifying that it follows by some rule of G from the current string. If the input string x ever appears as the current string in the derivation, then P halts. Since G is context sensitive, each string in the derivation must be of length $\leq \mid x \mid$, and since P needs only a fixed number of strings of this length (x, the current string, and the guessed next string), it operates in linear space.

Theorem 12.7 For every linear bounded automaton P there is a context sensitive grammar G such that $L_G = L_P$.

Proof: Let P be a one register *LBA* over Σ_k such that for some constant c, $\forall x \in \textbf{dom } P$

$NRAMspace_P(x) \leq c \mid x \mid$.

We first replace P be an equivalent *LBA* P_1 over $\Sigma_{(k+1)^c}$ such that

$$\forall x \in \textbf{dom}\ P_1 \quad NRAMspace_{P_1}(x) \leq |x|.$$

P_1 operates by viewing each symbol of $\Sigma_{(k+1)^c}$ as a string of $\leq c$ symbols over $\Sigma_k \cup \{ \sqcup \}$.

Next we replace P_1 by an equivalent by an equivalent *LBA* P_2 over $\Sigma_{(k+1)^c}$, where

$$\forall x \in \textbf{dom}\ P_2 \quad NRAMspace_{P_2}(x) \leq |x|,$$

and the symbol $\sqcup^c$ is used as a special blank symbol and every successor instruction is immediately preceded by a left shift instruction. P_2 simulates P_1 as follows:

1. every time P_1 executes a left shift instruction, P_2 executes the same left shift instruction, but also adds a blank symbol to the right end of P_1's register;
2. every time P_1 executes a successor instruction, P_2
   (a) exposes (on the right) the non-blank right end of the register by rotating (leftwards) the register contents;
   (b) removes a blank symbol from the left end;
   (c) executes the same successor instruction;
   (d) and reshifts (leftward) the register contents so that all the blank symbols are on the right end.

Careful examination of P_2 reveals that it is also the case that every left shift instruction is immediately followed by a successor instruction, so that these instructions always occur in pairs.

We now show how to construct a *CSG G* such that $L_G = L_{P_2}$. For the most part the construction is the same as in Theorem 12.3. Observe first of all that all of the productions are context sensitive *except*

those given for successor instructions (see 4) and the input instruction (see 5). We deal with these two exceptions separately. The only rule in 4) which is *not* context sensitive is the rule

$$\vec{b}_j \cdot s_j \cdot \dashv \rightarrow \overleftarrow{b}_j \cdot \dashv$$

We eliminate this kind of rules by using the fact that in P_2 every successor instruction is immediately preceded by a left shift instruction, and vice versa. Suppose j is a successor instruction so that j - 1 is a left shift instruction, then we replace parts 3) and 4) in the construction in Theorem 12.3 by the following rules:

$$b_{j+1} \rightarrow \vec{b}_j$$

$$\vec{b}_j \cdot a \cdot c \rightarrow a \cdot \vec{b}_j \cdot c \qquad \text{for all } a, c \in \Sigma_k$$

$$\vec{b}_j \cdot s_j \cdot \dashv \rightarrow \overleftarrow{b}_j \cdot a \cdot \dashv \qquad \text{for all } a \in \Sigma_k$$

$$c \cdot \overleftarrow{b}_j \cdot a \rightarrow \overleftarrow{b}_j \cdot a \cdot c \qquad \text{for all } a, c \in \Sigma_k$$

$$\vdash \cdot \overleftarrow{b}_j \rightarrow \vdash \cdot b_{j-1}$$

The non-context sensitive rules for the input instruction are simply intended to remove the special grammatical markers $\vdash$, b_j, $\dashv$ that were introduced by the rules for the output instruction. We can eliminate the necessity of having the special symbols by adding special diacritical marks to all the symbols of Σ_k (thereby increasing the size of our alphabet) which play the same roles as these special symbols. For example, we could replace the first rule of part 6) with the rule

$$S \rightarrow a \dashv \vdash b_m \qquad \text{for all } a \in \Sigma_k$$

and we could replace the second rules of part 6) with the rules

$$c \vdash b_m \rightarrow a \vdash b_m \cdot c \qquad \text{for all } a, c \in \Sigma_k$$

With careful analysis one can eliminate the use of all the special symbols in all the rules, although there are numerous special cases to consider.

Theorem 12.8 Every context sensitive language is a primitive recursive set.

Proof: First of all, a linear bounded automaton can be simulated by a *DRAM* program that recognizes the context sensitive language and that operates in polynomial space, i.e., there is a constant c such that on input x it uses at most $c \mid x \mid^2$ space. But, the latter is a primitive recursive function, and *DRAM* programs which operate within primitive recursive time or space bounds compute primitive recursive functions.

Theorem 12.9 The class of context sensitive languages is closed under intersection.

Proof: Let L_1 and L_2 be two *CSL*'s and let P_1 and P_2 be two one register *LBA*'s such that $L_1 = L_{P_1}$ and $L_2 = L_{P_2}$. Then the following two-register *LBA* P accepts $L_1 \cap L_2$.

inp $\mathbf{R}_1$
``copy $\mathbf{R}_1$ to $\mathbf{R}_2$"
P_1-
``copy $\mathbf{R}_2$ to $\mathbf{R}_1$"
P_2-
out $\mathbf{R}_1$

Theorem 12.10 The Emptiness Problem for context sensitive languages is undecidable.

Proof: We show that if the question $L_P = \emptyset$ for an arbitrary *LBA* P were algorithmically decidable (i.e., $\{P : L_P = \emptyset\}$ were recursive) then the Halting Problem would be algorithmically decidable. Let P be an arbirary *DRAM* program with one register over Σ_k with m lines. Let x be an arbitrary input to P. We represent an accepting computation of P on input x in the usual way by strings of the form

$$y_0 \cdot \cdots \cdot y_t$$

where t is the number of steps of the computation, y_i represents the state of P on input x at the i^{th} step and is of the form

$$b_j \cdot z$$

where j is the line number at step i and z represents the register contents at step i. Further, we may assume that by padding with blanks the lengths of all the y_i are identical.

We construct an *LBA* $\hat{P}_x$ with two registers, that depends on both P and x, and that will accept only valid computation strings of the above form. The *LBA* $\hat{P}_x$ does this by first copying its input from $\mathbf{R}_1$ to $\mathbf{R}_2$ and shifting left to remove the first state from the second copy. After that it removes symbols from $\mathbf{R}_1$ and $\mathbf{R}_2$ until it reaches the end of $\mathbf{R}_2$ and verifies that the input string was a valid computation string by checking that

1. $|y_i| = |y_{i+1}|$ and it must encounter symbols of Γ simultaneously in both $\mathbf{R}_1$ and $\mathbf{R}_2$;
2. y_0 is the initial state;

3.
   y_t is the final state;
4.
   y_{i+1} follows from y_i by a legal instruction execution of P on input x.

Thus, P halts on input x if and only if $\hat{P}_x$ accepts some input.

Next Up Previous Contents Index

Next: 12.5 Context Free Languages **Up:** 12. Formal Languages **Previous:** 12.3 Context Sensitive Languages

Bob Daley
2001-11-28

12.5 Context Free Languages

Example 12.3 Let the context free grammar G have the following rules:

$$S \rightarrow aAS$$

$$S \rightarrow a$$

$$A \rightarrow SbA$$

$$A \rightarrow ba$$

Then the string $aabbaa \in L_G$ via the derivation

$$\mathbf{S} \Rightarrow aA\mathbf{S} \Rightarrow a\mathbf{A}a \Rightarrow a\mathbf{S}bAa \Rightarrow aab\mathbf{A}a \Rightarrow aabbaa$$

Observe, that the string *aabbaa* can also be derived using the *leftmost derivation*

$$\mathbf{S} \Rightarrow a\mathbf{A}S \Rightarrow a\mathbf{S}bAS \Rightarrow aab\mathbf{A}S \Rightarrow aabba\mathbf{S} \Rightarrow aabbaa$$

Theorem 12.11 For each context free grammar G and each $x \in L_G$ there is a leftmost derivation of x in G.

Definition 12.5 A *derivation tree* for a string w in a context free grammar $G = \langle \Sigma, V, R, S \rangle$ is a tree satisfying:

1.

every vertex has a label, which is a symbol of $V \cup \Sigma$;

2. the label of the root is S;

3. every interior node has a label from V;

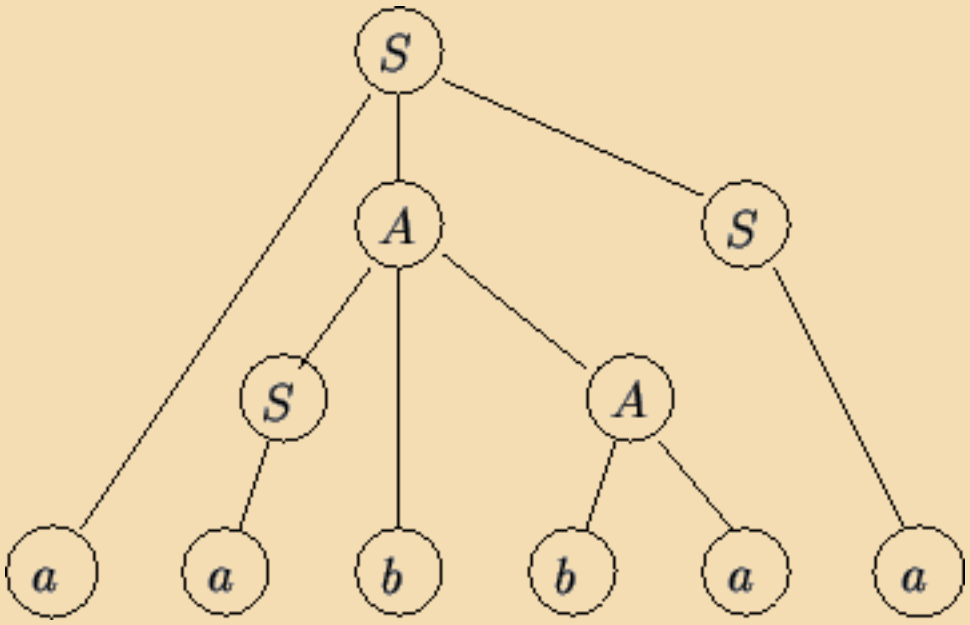
4. if a vertex has a label A and the $X_1,..., X_k$ are the labels of the immediate descendants of the vertex in order from left to right, then the rule $A \rightarrow X_1 \cdots X_k$ must belong to R;

5. w equals the concatenation of the labels of the leaf vertices from left to right.

Theorem 12.12 Let $G = \langle \Sigma, V, R, S \rangle$ be a context free grammar. Then $S \Rightarrow^* x$ if and only if there is a derivation tree in G for x.

Example 12.4 Let G be as in Example 12.3 and let $w = aabbaa$. Then a derivation tree for w in G is:

Figure 12.8: Derivation tree for *aabbaa*



In the above example by inspection we see that the following are also derivation trees in G:

Figure 12.2: Derivation tree for *abaa*

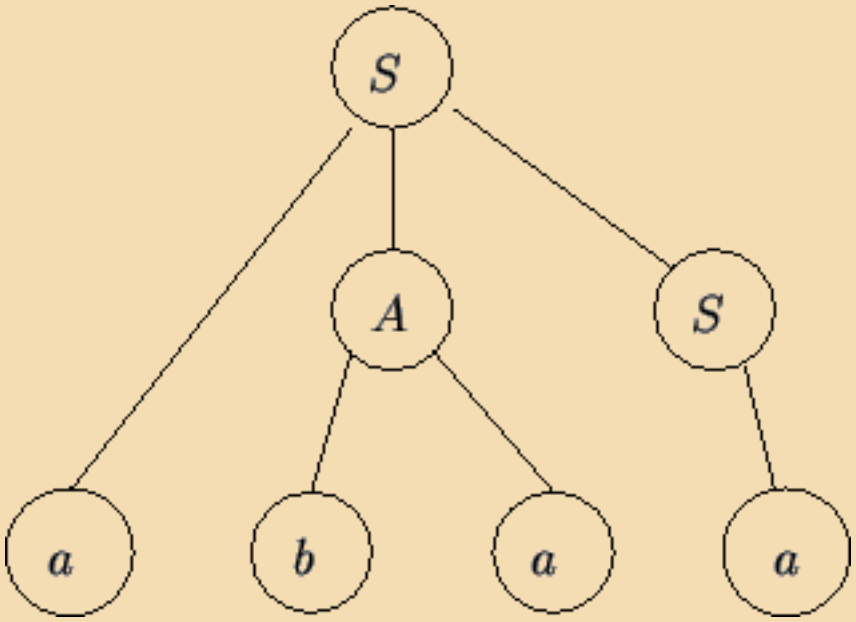
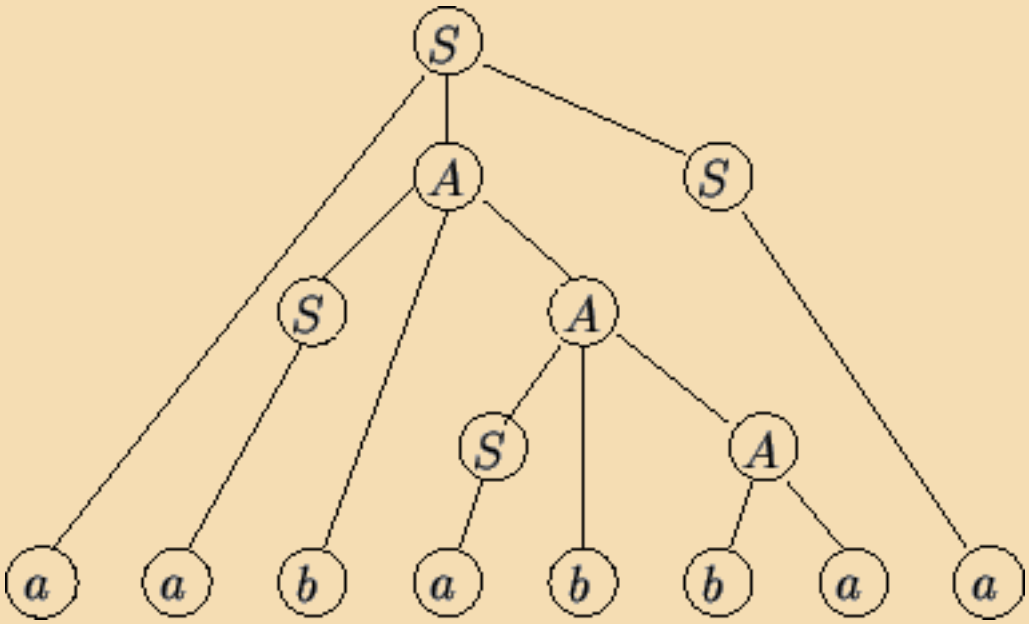
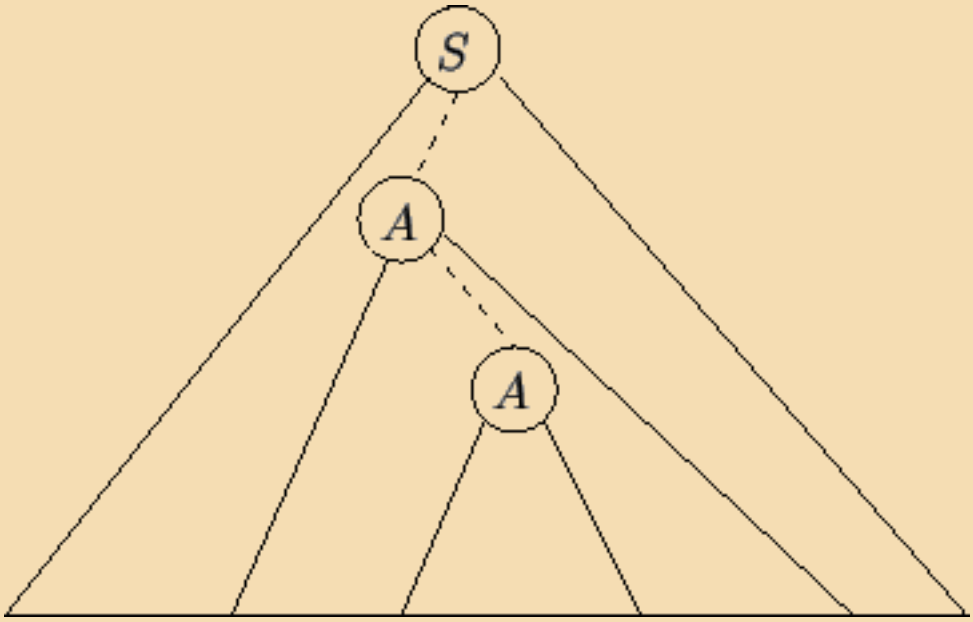


Figure 12.3: Derivation tree for *aababbaa*



Basic Property of Derivation Trees:
Given a derivation tree with repeated non-terminals on some path:

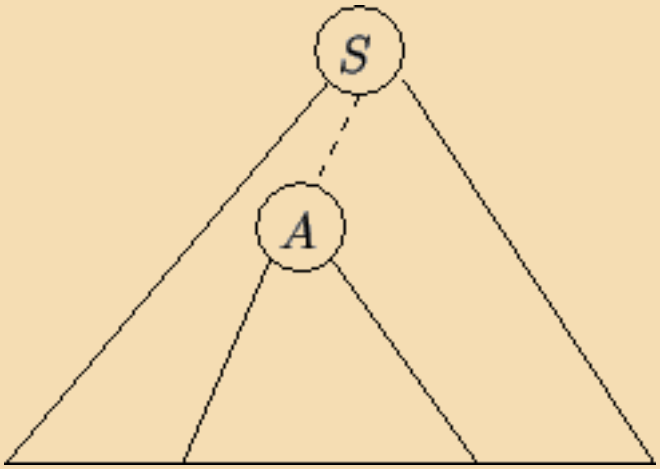
Figure 12.4: Derivation tree for with repeated non-terminal



then the tree can be

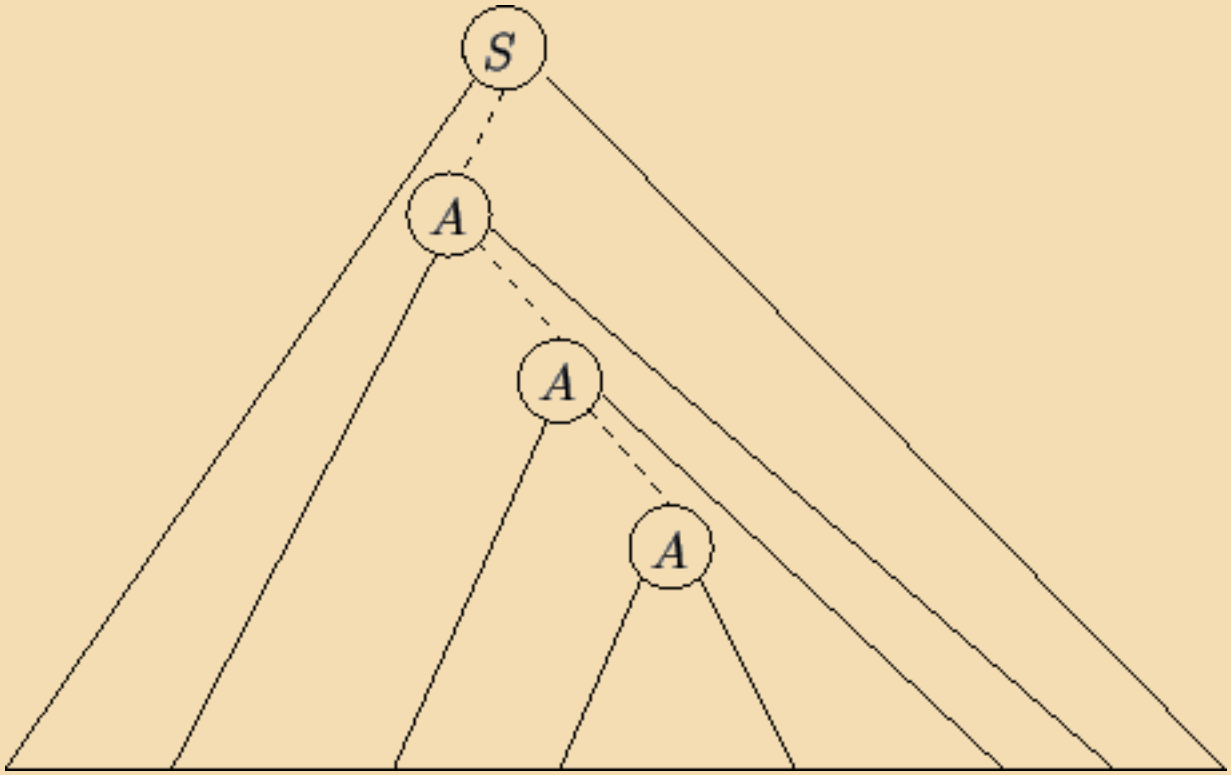
- **Pruned**
  to obtain the tree:

Figure 12.5: Derivation tree after pruning repeated non-terminal



- **Grafted**
  to obtain the tree:

Figure 12.6: Derivation tree after grafting repeated non-terminal



Next Up Previous Contents Index

Next: 12.6 Push Down Automata **Up:** 12. Formal Languages **Previous:** 12.4 Linear Bounded Automata

Bob Daley

2001-11-28

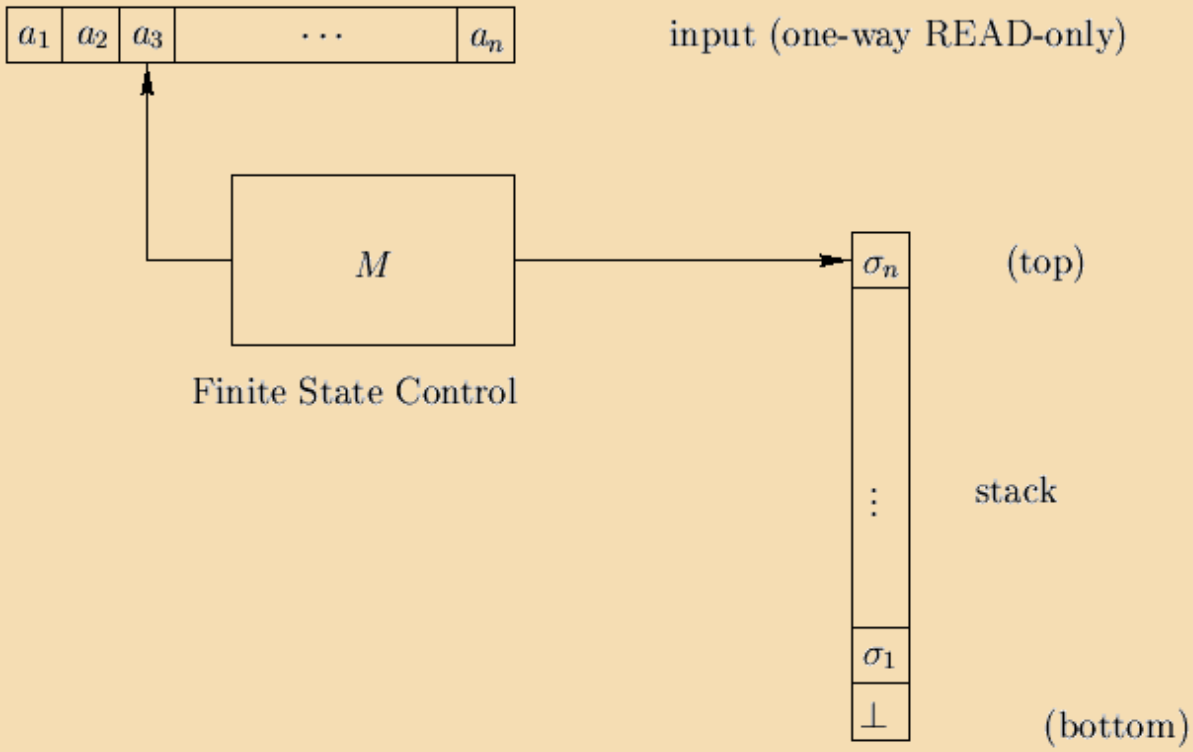
Next Up Previous Contents Index
Next: 12.7 Regular Languages **Up:** 12. Formal Languages **Previous:** 12.5 Context Free Languages

12.6 Push Down Automata

Definition 12.6 A *push down automaton* (PDA) M is a system $\langle \Sigma, Q, \Gamma, \delta, q_0, \perp, F \rangle$, where

1. Σ is a finite set of symbols called the *input alphabet*;
2. Q is a finite set of *states*;
3. Γ is a finite set of symbols called the *stack alphabet*;
4. $q_0 \in Q$ is the *initial state*;
5. $\perp \in \Gamma$ is the *start symbol*;
6. $F \subseteq Q$ is the set of *final states*;
7. δ is a mapping from $Q \times (\Sigma \cup \{\epsilon\}) \times \Gamma$ to finite subsets of $Q \times \Gamma^*$.

Figure 12.7: Schematic for Push Down Automaton



The semantics of the state transition function δ is defined as follows:

For any $q \in Q, a \in \Sigma, A \in \Gamma$

$$\delta(q, a, A) = \{(p_1, \gamma_1),...,(p_m, \gamma_m)\},$$

where for each $1 \leq i \leq m$, $p_i \in Q$ and $\gamma_i \in \Gamma^*$ means that the PDA M is in state q reading input symbol a with the symbol A on top of its stack, can for any $1 \leq i \leq m$ replace A with γ_i, *advance* the input head one symbol to the right, and enter state p_i;

For any $q \in Q, A \in \Gamma$

$$\delta(q, \epsilon, A) = \{(p_1, \gamma_1),...,(p_m, \gamma_m)\},$$

where for each $1 \leq i \leq m$, $p_i \in Q$ and $\gamma_i \in \Gamma^*$ means that the PDA M is in state q, with the symbol A on top of its stack, can for any $1 \leq i \leq m$ replace A with γ_i, and *without advancing its input head* enter state p_i.

Definition 12.7 An *instantaneous description* (ID) for a PDA M is a triple (q, w, γ), where $q \in Q$ (the current state), $w \in \Sigma^*$ (the remaining input string), and $\gamma \in \Gamma^*$ (the current stack contents). We define the relation

$$(q, a \cdot w, \alpha \cdot A) \vdash_M (p, w, \alpha \cdot \beta)$$

where $a \in \Sigma \cup \{\epsilon\}, p, q \in Q, A \in \Gamma$, and $\alpha, \beta \in \Gamma^*$, whenever $(p, \beta) \in \delta(q, a, A)$.

Also, if I and J are ID's then $I \vdash_M^* J$ if and only if there exists a sequence of ID's $I_0,..., I_n$ such that

$$I = I_0 \vdash_M I_1 \cdots I_{n-1} \vdash_M I_n = J$$

Definition 12.8 The language accepted *by empty stack* of a PDA M, denoted by N_M is defined by

$$N_M = \{w : (q_0, w, \perp) \vdash_M^* (p, \epsilon, \epsilon) \text{ for some } p \in Q\}.$$

The language accepted *by final state* of a PDA M, denoted by L_M is defined by

$$L_M = \{w : (q_0, w, \perp) \vdash_M^* (p, \epsilon, \gamma) \text{ for some } p \in F, \gamma \in \Gamma^*\}.$$

Theorem 12.13
For every PDA M_1 there is a PDA M_2 such that $N_{M_1} = L_{M_2}$.

For every PDA M_1 there is a PDA M_2 such that $L_{M_1} = N_{M_2}$.

Example 12.5 Let $M = \langle \{0, 1\}, \{q_1, q_2\}, \{R, B, G\}, \delta, q_1, R, \emptyset \rangle$, where δ is defined by:

$$\delta(q_1, 0, R) = \{(q_1, RB)\}$$

$$\delta(q_1, 0, G) = \{(q_1, GB)\}$$

$$\delta(q_1, 0, B) = \{(q_1, BB), (q_2, \epsilon)\}$$

$$\delta(q_1, 1, R) = \{(q_1, RG)\}$$

$$\delta(q_1, 1, B) = \{(q_1, BG)\}$$

$$\delta(q_1, 1, G) = \{(q_1, GG), (q_2, \epsilon)\}$$

$$\delta(q_1, \epsilon, R) = \{(q_2, \epsilon)\}$$

$$\delta(q_2, 0, B) = \{(q_2, \epsilon)\}$$

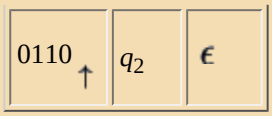
$$\delta(q_2, 1, G) = \{(q_2, \epsilon)\}$$

$$\delta(q_2, \epsilon, R) = \{(q_2, \epsilon)\}$$

Then, $N_M = \{w \cdot \rho(w) : w \in \{0, 1\}^*\}$.

Then on input 0110 the computation proceeds as follows:

input	state	stack
$\underset{\uparrow}{0}110$	q_1	R
$0\underset{\uparrow}{1}10$	q_1	RB
$01\underset{\uparrow}{1}0$	q_2	RBG
$011\underset{\uparrow}{0}$	q_2	RB
$0110\underset{\uparrow}{}$	q_2	R



So the PDA stops and accepts.

Theorem 12.14 For every CFG G there is a PDA M such that $L_G = N_M$.

Proof: Let $G = \langle \Sigma, V, R, S \rangle$ be the given CFG. The define the PDA $M = \langle \Sigma, Q, \Sigma \cup V, \delta, q_0, \perp, \emptyset \rangle$ in such a way that for each w, $w \in L_G$ if and only if M accepts w. The PDA M will proceed by reversing the derivation of w in G based on a derivation tree. We first define a *macro* instruction:

$\Delta(q, \epsilon, \gamma) = \{(p, Z)\}$,

where $q, p \in Q$, $Z \in V$, and $\gamma \in (\Sigma \cup V)^*$, such that M in state q replaces $\gamma = \sigma_1 \ldots \sigma_n$ on the stack (where σ_n is on the top of the stack) by Z.

$$\Delta(q, \epsilon, \gamma) = \{(p, Z)\}:$$

$$\delta(q, \epsilon, \sigma_n) = \{(q.\gamma.n - 1, \epsilon)\}$$

$$\delta(q.\gamma.n - 1, \epsilon, \sigma_{n-1}) = \{(q.\gamma.n - 2, \epsilon)\}$$

$$\vdots$$

$$\delta(q.\gamma.1, \epsilon, \sigma_1) = \{(p, Z)\}$$

The PDA M is then defined by:

$$\delta(q_0, a, Z) = \{(q_0, a \cdot Z)\} \qquad \text{for } Z \in \Gamma$$

$$\Delta(q_0, \epsilon, x) = \{(q_0, A)\} \qquad \text{for } A \to x \in R$$

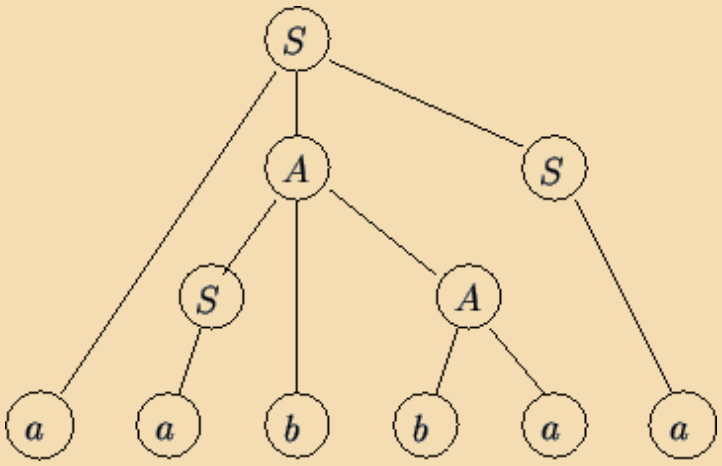
$$\delta(q_0, \epsilon, S) = \{(q_1, \epsilon)\}$$

$$\delta(q_1, \epsilon, \perp) = \{(q_1, \epsilon)\}$$

One then easily shows by induction on the length of the derivation/computation that M accepts w if and only if $w \in L_G$.

Example 12.6 Let G be as in Example 12.3 and let $w = aabbaa$.

Figure 12.8: Derivation tree for *aabbaa*



Then, the computation by M on w as defined in Theorem 12.14 is:

input	state	stack	rule
$\underset{\uparrow}{a}abbaa$	q_0	$\perp$	
$aa\underset{\uparrow}{b}baa$	q_0	$\perp\ aa$	$S \rightarrow a$
$aa\underset{\uparrow}{b}baa$	q_0	$\perp\ aS$	
$aabba\underset{\uparrow}{a}$	q_0	$\perp\ aSbba$	$A \rightarrow ba$
$aabba\underset{\uparrow}{a}$	q_0	$\perp\ aSbA$	$A \rightarrow SbA$
$aabba\underset{\uparrow}{a}$	q_0	$\perp\ aA$	
$aabbaa_{\uparrow}$	q_0	$\perp\ aAa$	$S \rightarrow a$
$aabbaa_{\uparrow}$	q_0	$\perp\ aAS$	$S \rightarrow aAS$
$aabbaa_{\uparrow}$	q_0	$\perp\ S$	
$aabbaa_{\uparrow}$	q_1	$\perp$	
$aabbaa_{\uparrow}$	q_1	ϵ	

Lemma 12.15 For every CFG $G = \langle \Sigma, V, R, S \rangle$ there exists a CFG $\hat{G} = \langle \Sigma, V, \hat{R}, S \rangle$ such that $L_G = L\,\hat{G}$ and $\hat{R}$ contains no rules of the form $A \rightarrow B$ where $A, B \in V$.

Proof: If R contains the rule $A \rightarrow B$, then replace it by the set of rules $\{A \rightarrow x : B \rightarrow x \in R\}$. This replacement occurs one rule at a time, where the rules are ordered according to the lefthand side non-terminal, and rules of the form $A \rightarrow A$ are immediately removed. Clearly, $L_G = L\,\hat{G}$.

Theorem 12.16 (Pumping Lemma) For every CFG G there exists a positive integer p such that for any $z \in L_G$ such that $|z| \geq p$, z can be written as $z = uvwxy$, where $|vwx| \leq p$, $|v| > 0$ or $|x| > 0$, and $uv^iwx^iy \in L_G$ for all $i \geq 0$.

Proof: Let $n = \max\{|x| : A \rightarrow x \in R\}$ and let $k = \#V$. Define $p = n^{k+1}$. Suppose $z \in L_G$ is such that $|z| \geq p$. Let T be a derivation tree for z. Since the maximum length of the righthand side of any rule is $\leq n$, the maximum *branching* of T is also $\leq n$. Therefore, since $|z| \geq n^{k+1}$, there must be some path in the tree T of length $\geq k + 1$ (having $\geq k + 2$ vertices). Furthermore, since there are at most k non-terminals, there must be some path with some repeated non-terminal A on it. Consider the following schematic for the derivation tree T. We can choose the segment vwx of z in such a way that it is derived from the *first* occurrence of a repeated non-terminal A *from the bottom* of the tree. In this way we see that $|vwx| \leq n^{k+1}$. Furthermore, since we can assume that there are no rules of the form $A \rightarrow B$, where $A, B \in V$, we have that either $|v| > 0$ or $|x| > 0$. By the Basic Property of derivation trees repeated grafting (and pruning for the case $i = 0$) yields derivation trees for the strings uv^iwx^iy.

Figure 12.9: Pumping Down

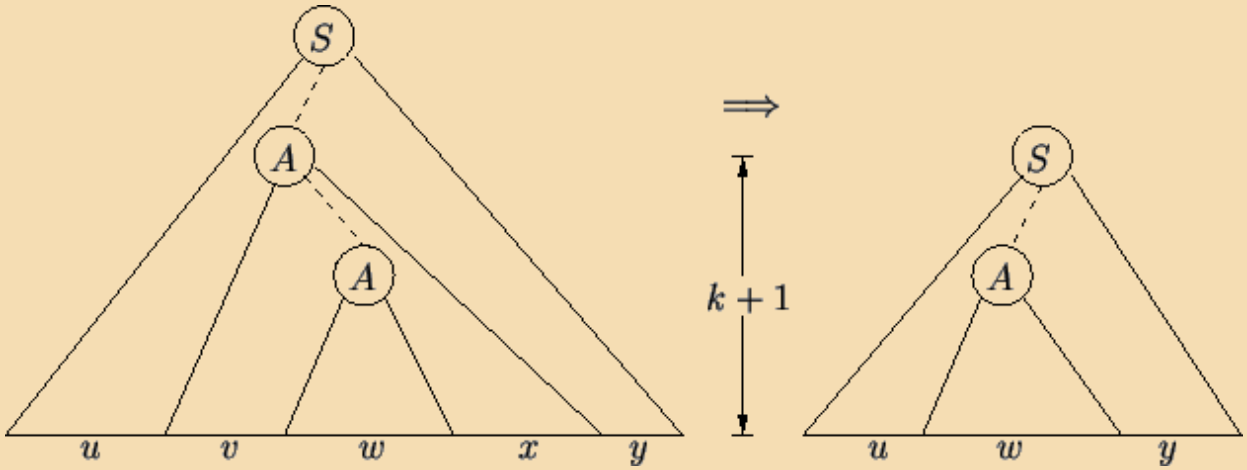
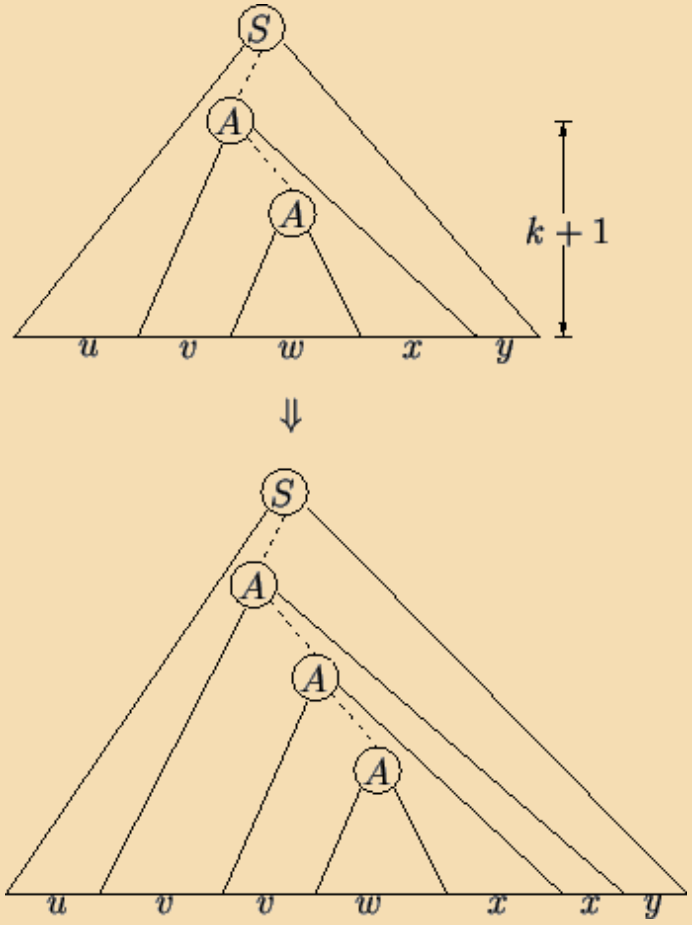


Figure 12.10: Pumping Up



Theorem 12.17 For each CFG G there exist integers p and q such that

1. $L_G \neq \emptyset$ if and only if $\exists z \in L_G \quad |z| < p$
2. L_G is infinite if and only if $\exists z \in L_G \quad p \leq |z| < q$.

Proof: Let $n = \max\{|x| \ : \ A \rightarrow x \in R\}$ and let $k = \#V$. Define $p = n^{k+1}$ and let $q = 2p$.

1. Clearly, if $\exists z \in L_G$ such that $|z| \leq p$, then $L_G \neq \emptyset$. Suppose $L_G \neq \emptyset$ and let $z \in L_G$. If $|z| \geq p$, then by the Pumping Lemma for CFG's, z can be written as $z = uvwxy$ and the string $z_1 = uwy \in L_G$ and either $|v| > 0$ or $|x| > 0$, so $|z_1| < |z|$. By repeating this pruning process (if $|z_1| \geq p$) we must eventually obtain a string $z_1 \in L_G$ such that $|z_1| < p$.
2. Suppose $\exists z \in L_G$ such that $p \leq |z| \leq q$. By the Pumping Lemma for CFG's we have that z can be written as $z = uvwxy$, where $|v| > 0$ or $|x| > 0$, and the string $uv^iwx^iy \in L_G$ for all $i \geq 0$. Clearly, L_G is infinite.

   Suppose L_G is infinite. Then there must exist a string $z \in L_G$ such that $|z| \geq q$. Using the Pumping Lemma again we see

that $z = uvwxy$ and the string $z_1 = uwy \in L_G$ is such that $|z_1| \geq |z| - p > p$ (since $|vwx| \leq p$). By repeating this pruning process (if $|z_1| \geq q$) we must eventually obtain a string $z_1 \in L_G$ such that $p \leq |z_1| < q$.

Example 12.7 Let $L = \{a^n b^n c^n : n \geq 1\}$. Then L is a CSL, but L is *not* a CFL.

Proof: Clearly, L is infinite, so the Pumping Lemma applies to L (assuming that L were a CFL). Let p be the pumping length specified in the Pumping Lemma for L. Let $z = a^p b^p c^p$, so z can be written $z = uvwxy$, where $|v| > 0$ or $|x| > 0$, and $|vwx| \leq p$, and $uv^i wx^i y \in L$ for all $i \geq 0$.

Observe first that v and x can contain *at most one letter*. For example, if $v = ab$, then $v^2 = abab$ and $uv^2wx^2y \notin L$.

Next, we then see that the string uv^2wx^2y cannot have equal numbers of a's, b's and c's, since *at most two* of the letters a, b and c can be *pumped up*.

Proposition 12.18 The class of context free languages is *not* closed under intersection.

Proof: Define the languages

$$L_1 = \{a^n b^m c^m : n, m \geq 1\}$$

and

$$L_2 = \{a^m b^m c^n : n, m \geq 1\}$$

The clearly, $L_1 \cap L_2 = \{a^n b^n c^n : n \geq 1\}$, and so by the previous example is not a CFL. However, L_1 and L_2 are easily seen to be CFL's. The PDA M_1 which accepts L_1 operates as follows:

1. M_1 first scans past the a's checking that there is at least one a;
2. M_1 pushes all b's onto its stack, checking that there is at least one b and that there are no a's mixed in with the b's;
3. M_1 matches c's in the input with b's on the stack, reading a c and popping a b, checking that there are no a's or b's mixed in with the c's;
4. M_1 checks that both the input and the stack are empty simultaneously.

Theorem 12.19 For any PDA M the language N_M is context free.

Proof: Let $M = \langle \Sigma, Q, \Gamma, \delta, q_0, \perp, \emptyset \rangle$ be a given PDA. Define $G = \langle \Sigma, V, R, S \rangle$ as follows:

$V = \{[\mathbf{q, A, p}] : q, p \in Q \text{ and } A \in \Gamma\}$

and R is the set of rules:

1. $S \rightarrow [\mathbf{q}_0, \perp, \mathbf{q}]$,
   for each $q \in Q$;
2. $[\mathbf{q, A, q}_{m+1}] \rightarrow a[\mathbf{q}_1, \mathbf{B}_1, \mathbf{q}_2] \cdots [\mathbf{q}_m, \mathbf{B}_m, \mathbf{q}_{m+1}]$,
   for each $q, q_1, \ldots, q_{m+1} \in Q$, each $a \in \Sigma \cup \{\epsilon\}$, and each $A, B_1, \ldots, B_m \in \Gamma$, where we have that $\delta(q, a, A)$ contains $(q_1, B_1 \cdots B_m)$. (If $m = 0$, then the rule is $[\mathbf{q, A, q}_1] \rightarrow a$).

G is defined in such a way that for any input x, $x \in N_M$ if and only if $x \in L_G$ and a leftmost derivation of x in G corresponds to an accepting computation of x by M. Moreover, $[\mathbf{q, A, p}] \Rightarrow^* x$ if and only if x causes M to erase an A from its stack by some sequence of computation steps beginning in state q and ending in state p.

Example 12.8 Let M be the PDA given in Example 12.5 for the language $L = \{w \cdot \rho(w) : w \in \{0, 1\}^*\}$. The corresponding grammar G is:

$$S \rightarrow [\mathbf{q}_1, \mathbf{R}, \mathbf{q}] \quad \text{for all } q \in Q$$

$$[\mathbf{q}_1, \mathbf{R}, \mathbf{q}] \rightarrow 0[\mathbf{q}_1, \mathbf{B}, \hat{q}][\hat{q}, R, q] \quad \text{for all } q, \hat{q} \in Q$$

$$[\mathbf{q}_1, \mathbf{G}, \mathbf{q}] \rightarrow 0[\mathbf{q}_1, \mathbf{B}, \hat{q}][\hat{q}, G, q] \quad \text{for all } q, \hat{q} \in Q$$

$$[\mathbf{q}_1, \mathbf{B}, \mathbf{q}] \rightarrow 0[\mathbf{q}_1, \mathbf{B}, \hat{q}][\hat{q}, B, q] \quad \text{for all } q, \hat{q} \in Q$$

$$[\mathbf{q}_1, \mathbf{B}, \mathbf{q}_2] \rightarrow 0$$

$$[\mathbf{q}_1, \mathbf{R}, \mathbf{q}] \rightarrow 0[\mathbf{q}_1, \mathbf{G}, \hat{q}][\hat{q}, R, q] \quad \text{for all } q, \hat{q} \in Q$$

$$[\mathbf{q}_1, \mathbf{G}, \mathbf{q}] \rightarrow 0[\mathbf{q}_1, \mathbf{G}, \hat{q}][\hat{q}, G, q] \quad \text{for all } q, \hat{q} \in Q$$

$$[\mathbf{q}_1, \mathbf{B}, \mathbf{q}] \rightarrow 0[\mathbf{q}_1, \mathbf{G}, \hat{q}][\hat{q}, B, q] \quad \text{for all } q, \hat{q} \in Q$$

$$[\mathbf{q}_1, \mathbf{G}, \mathbf{q}_2] \rightarrow 1$$

$$[\mathbf{q}_1, \mathbf{R}, \mathbf{q}_2] \rightarrow \epsilon$$

$[\mathbf{q_2, B, q_2}] \rightarrow 0$

$[\mathbf{q_2, G, q_2}] \rightarrow 1$

$[\mathbf{q_2, R, q_2}] \rightarrow \epsilon$

Consider the input 0110 to *M*:

input	state	stack	string / (rule)
$\underset{\uparrow}{0}110$	q_1	R	$[\mathbf{q_1, R, q_2}]$
			($S \rightarrow [\mathbf{q_1, R, q_2}]$)
$0\underset{\uparrow}{1}10$	q_1	RB	$0[\mathbf{q_1, B, q_2}][\mathbf{q_2, R, q_2}]$
			($[\mathbf{q_1, R, q_2}] \rightarrow 0[\mathbf{q_1, B, q_2}][\mathbf{q_2, R, q_2}]$)
$01\underset{\uparrow}{1}0$	q_1	RBG	$01\mathbf{q_1, G, q_2}][\mathbf{q_2, B, q_2}][\mathbf{q_2, R, q_2}]$
			($[\mathbf{q_1, B, q_2}] \rightarrow 1[\mathbf{q_1, G, q_2}][\mathbf{q_2, B, q_2}]$)
$011\underset{\uparrow}{0}$	q_2	RB	$011[\mathbf{q_2, B, q_2}][\mathbf{q_2, R, q_2}]$
			($[\mathbf{q_1, G, q_2}] \rightarrow 1$)
$0110\underset{\uparrow}{}$	q_2	R	$0110[\mathbf{q_2, R, q_2}]$
			($[\mathbf{q_2, B, q_2}] \rightarrow 0$)
$0110\underset{\uparrow}{}$	q_2	ϵ	0110
			($[\mathbf{q_2, R, q_2}] \rightarrow \epsilon$)

Next Up Previous Contents Index

Next: 12.7 Regular Languages **Up:** 12. Formal Languages **Previous:** 12.5 Context Free Languages

Bob Daley

2001-11-28

12.7 Regular Languages

Theorem 12.20 Given an NFA $M = \langle \Sigma, Q, \delta, q_0, F \rangle$ there is a regular grammar G such that $L_G = L_M$.

Proof: The grammar $G = \langle \Sigma, Q, R, q_0 \rangle$ has the following rules:

1. $q_1 \rightarrow aq_2$, whenever $q_2 \in \delta(q_1, a)$;
2. $q_1 \rightarrow a$, whenever $q_2 \in \delta(q_1, a)$ and $q_2 \in F$.

It is easy to see that each accepting computation path

Figure 12.11: Accepting Computation Path

has the corresponding derivation

$$q_0 \Rightarrow a_1 q_1 \Rightarrow a_1 a_2 q_2 \Rightarrow \cdots \Rightarrow a_1 \cdots a_{n-1} q_{n-1} \Rightarrow a_1 \cdots a_n.$$

Theorem 12.21 For each regular grammar $G = \langle \Sigma, V, R, S \rangle$ there is an NFA M such that $L_M = L_G$.

Proof: The NFA $M = \langle \Sigma, V \cup \{q_f\}, \delta, S, \{q_f\} \rangle$ has its state transition function δ defined in such a way that

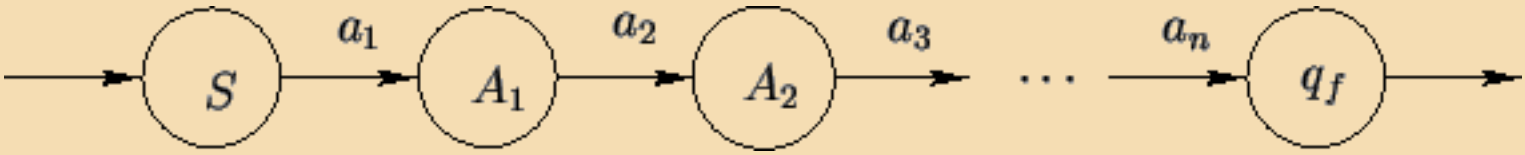
1. $B \in \delta(A, a)$, whenever $A \rightarrow aB \in R$;
2. $q_f \in \delta(A, a)$, whenever $A \rightarrow a \in R$.

Then, for any derivation

$$S \Rightarrow a_1A_1 \Rightarrow a_1a_2A_2 \Rightarrow \cdots \Rightarrow a_1 \cdots a_{n-1}A_n \Rightarrow a_1 \cdots a_n.$$

there is a corresponding computation

Figure 12.12: Accepting Computation Path



Next Up Previous Contents Index

Next: Bibliography **Up:** 12. Formal Languages **Previous:** 12.6 Push Down Automata

Bob Daley

2001-11-28

Next Up Previous Contents Index

Next: Index **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 12.7 Regular Languages

---

Next Up Previous Contents Index

Next: Index **Up:** Lecture Notes for CS 2110 Introduction to Theory **Previous:** 12.7 Regular Languages

Bob Daley

2001-11-28

Next Up Previous Contents

Up: Lecture Notes for CS 2110 Introduction to Theory **Previous:** Bibliography

Index

Next Up Previous Contents

Up: Lecture Notes for CS 2110 Introduction to Theory **Previous:** Bibliography

Bob Daley
2001-11-28