Theory of Computation

Course note based on *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd edition, authored by Martin Davis, Ron Sigal, and Elaine J. Weyuker.

course note prepared by

Tyng–Ruey Chuang

Week 4, Spring 2010

About This Course Note

- It is prepared for the course *Theory of Computation* taught at the National Taiwan University in Spring 2010.
- It follows very closely the book *Computability, Complexity, and Languages: Fundamentals of Theoretical Computer Science*, 2nd edition, by Martin Davis, Ron Sigal, and Elaine J. Weyuker. Morgan Kaufmann Publishers. ISBN: 0-12-206382-1.
- It is available from Tyng-Ruey Chuang's web site:

  `http://www.iis.sinica.edu.tw/~trc/`

1 A Universal Program (4)

1.1 Coding Programs by Numbers (4.1)

Coding Programs by Numbers

For each program $\mathscr{P}$ in language $\mathscr{S}$, we will devise a method

- to associate a unique number, $\#(\mathscr{P})$, to the program $\mathscr{P}$, and
- to retrieve a program from its number.

In addition, for each number $n \in N$, we will retrieve from n a program.

Arranging Variables and Labels

- The variables are arranged in the following order

$$Y, X_1, Z_1, X_2, Z_2, X_3, Z_3, \ldots$$

- The labels are arranged in the following order

$$A_1, B_1, C_1, D_1, E_1, A_2, B_2, C_2, D_2, E_2, A_3, \ldots$$

- $\#(V)$ is the position of variable V in the ordering. So is $\#(L)$ for label L.
- Thus, $\#(X_2) = 4,\ \#(Z_1) = \#(Z) = 3,\ \#(E) = 5,\ \#(B_2) = 7,\ \ldots$.

Coding Instructions by Numbers

Let I be an instruction of language $\mathscr{S}$. We write

$$\#(I) = \langle a, \langle b, c\rangle\rangle$$

where

1. if I is unlabeled, then $a = 0$; if I is labeled L, then $a = \#(L)$;
2. if variable V is mentioned in I, then $c = \#(V) - 1$;
3. if the statement in I is

$$V \leftarrow V \quad \text{or} \quad V \leftarrow V + 1 \quad \text{or} \quad V \leftarrow V - 1$$

   then $b = 0$ or 1 or 2, respectively;

4. if the statement in I is

   IF $V \neq 0$ GOTO L'

   then $b = \#(L') + 2$.

Coding Instructions by Numbers, Examples

- The number of the unlabeled instruction

  $X \leftarrow X + 1$

  is

$$\langle 0, \langle 1, 1\rangle\rangle = \langle 0, 5\rangle = 10.$$

- The number of the labeled instruction

  $[A]\ X \leftarrow X + 1$

  is

$$\langle 1, \langle 1, 1\rangle\rangle = \langle 1, 5\rangle = 21.$$

Retrieving The Instruction from A Number

For any given number q, there is a unique instruction I with $\#(I) = q$. How?

- First we compute $l(q)$. If $l(q) = 0$, I is unlabeled; otherwise I has the $l(q)$th label L in our list.
- Then we compute $i = r(r(q)) + 1$ to locate the ith variable V in our list as the variable mentioned in I.
- Then the statement in I will be

$$\begin{array}{ll} V \leftarrow V & \text{if } l(r(q)) = 0 \\ V \leftarrow V + 1 & \text{if } l(r(q)) = 1 \\ V \leftarrow V - 1 & \text{if } l(r(q)) = 2 \\ \text{IF } V \neq 0 \text{ GOTO } L' & \text{if } j = l(r(q)) - 2 > 0 \end{array}$$

  and L' is the jth label in the list.

Coding Programs by Numbers, Finally

Let a program $\mathscr{P}$ consists of the instructions $I_1, I_2, \ldots, I_k$. Then we set

$$\#(\mathscr{P}) = [\#(I_1), \#(I_2), \ldots, \#(I_k)] - 1$$

We call $\#(\mathscr{P})$ the number of program $\mathscr{P}$. Note that the empty program has number 0.

Coding Programs by Numbers, Examples

Consider the following "nowhere defined" program $\mathscr{P}$

[A] $X \leftarrow X + 1$
IF $X \neq 0$ GOTO A

Let I_1 and I_2, respectively, be the first and the second instruction in $\mathscr{P}$, then

$$\begin{aligned} \#(I_1) &= \langle 1, \langle 1, 1 \rangle \rangle = \langle 1, 5 \rangle = 21 \\ \#(I_2) &= \langle 0, \langle 3, 1 \rangle \rangle = \langle 0, 23 \rangle = 46 \end{aligned}$$

Therefore

$$\#(\mathscr{P}) = 2^{21} \cdot 3^{46} - 1$$

Coding Programs by Numbers, Examples

What is the program whose number is 199? We first compute

$$199 + 1 = 200 = 2^3 \cdot 3^0 \cdot 5^2 = [3, 0, 2]$$

Thus, if $\#(\mathscr{P}) = 199$, then $\mathscr{P}$ consists of 3 instructions whose numbers are 3, 0, and 2. As

$$\begin{aligned} 3 &= \langle 2, 0\rangle = \langle 2, \langle 0, 0\rangle\rangle \\ 2 &= \langle 0, 1\rangle = \langle 0, \langle 1, 0\rangle\rangle \end{aligned}$$

We conclude that $\mathscr{P}$ is the following program

$[B]$ $Y \leftarrow Y$
$Y \leftarrow Y$
$Y \leftarrow Y + 1$

This is not a very interesting program, as it just computes $f(x) = 1$.

A Problem with Number 0

- The number of the unlabeled instruction $Y \leftarrow Y$ is

$$\langle 0, \langle 0, 0\rangle\rangle = \langle 0, 0\rangle = 0$$

- By the definition of Gödel number, the number of a program will be unchanged if an unlabeled $Y \leftarrow Y$ is appended to its end. Note that this does not change the output of the program.
- However, we remove even this ambiguity by requiring that *the final instruction in a program is not permitted to be the unlabeled statement* $Y \leftarrow Y$.
- Now, each number determines a unique program (just as each program determines a unique number)!

1.2 The Halting Problem (4.2)

HALT(x, y): A Predicate on Programs and Their Inputs

We define predicate $\text{HALT}(x, y)$ such that

$$\text{HALT}(x, y) \Leftrightarrow \text{program number } y \text{ eventually halts on input } x.$$

Let $\mathscr{P}$ be the program such that $\#(\mathscr{P}) = y$. Then

$$\text{HALT}(x, y) = \begin{cases} 1 & \text{if } \Psi^{(1)}_{\mathscr{P}}(x) \text{ is defined,} \\ 0 & \text{if } \Psi^{(1)}_{\mathscr{P}}(x) \text{ is undefined.} \end{cases}$$

Note that $\text{HALT}(x, y)$ is a total function. But, is $\text{HALT}(x, y)$ computable?

HALT(x, y) Is Not Computable

Theorem 2.1. HALT(x, y) is not a computable predicate. *Proof.* Suppose HALT(x, y) were computable. Then we could construct the following program $\mathscr{P}$:

$[A]$ IF HALT(X, X) GOTO A

It is clear that

$$\Psi^{(1)}_{\mathscr{P}}(x) = \begin{cases} \mathit{undefined} & \text{if HALT}(x, x) \\ 0 & \text{if} \sim \text{HALT}(x, x). \end{cases}$$

Let $\#(\mathscr{P}) = y_0$. Then, for all x,

$$\text{HALT}(x, y_0) \;\Leftrightarrow\; \Psi^{(1)}_{\mathscr{P}}(x) \text{ is defined} \;\Leftrightarrow\; \mathscr{P} \text{ halts on } x \;\Leftrightarrow\; \sim \text{HALT}(x, x)$$

Let $x = y_0$, we arrive at

$$\text{HALT}(y_0, y_0) \quad \Leftrightarrow \quad \sim \text{HALT}(y_0, y_0)$$

which is a contradiction. □

"HALT(x, y) Is Not Computable." ***What's that?***

Let's be precise on what have be proved.

- HALT(x, y) is a predicate on programs in language $\mathscr{S}$. It is a predicate on the computational behavior of the programs, i.e., whether a program y of language $\mathscr{S}$ will halt on input x.
- It is shown *there exists no program in language $\mathscr{S}$* that computes HALT(x, y).
- As HALT(x, y) is a total function, we now have as an example a total function that cannot be expressed as a program in $\mathscr{S}$.
- But can HALT(x, y) be expressed in languages other than $\mathscr{S}$? Will HALT(x, y) become "computable" if other (more powerful) formalisms of computation are used?

The Unsolvability of Halting Problem

> There is no algorithm that, given a program of $\mathscr{S}$ and an input to the program, can determine whether or not the given program will eventually halt on the given input.

- In this form, the result is called the *unsolvability of halting problem.*
- The statement above is stronger than the statement "*there exists no program in language $\mathscr{S}$ that computes HALT(x, y),*" as an algorithm can refer to a method in any formalism of computation.
- However, language $\mathscr{S}$ can be been shown to be as powerful as any known computational formalism. Therefore, we reason that if no program in $\mathscr{S}$ can solve it, no algorithm can.

Church's Thesis

Any algorithm for computing on numbers can be carried out by a program of $\mathscr{S}$.

- This assertion is called *Church's Thesis*.
- As the word *algorithm* has no general definition separated from a particular language, Church's thesis cannot be proved as a mathematical theorem.
- We will use Church's thesis freely in asserting the nonexistence of algorithms *whenever we have shown that the problem cannot be solved by a program of* $\mathscr{S}$.

Why The Halting Program Is So Hard? (Unsolvable!)

- This shall not be too surprising, as it is easy to construction short programs of $\mathscr{S}$ such that it is very difficult to tell whether they will ever halt.
- Example: Fermat's last theorem.
- Example: Goldbach's conjecture.
- Actually it is always hard to prove whether programs of $\mathscr{S}$ will exhibit specific computational behaviors (which are of sufficient interest).

Fermat's Last Theorem

The equation $x^n + y^n = z^n$ has no solution in positive x, y, z and $n > 2$.

- It is easy to write a program $\mathscr{P}$ of language $\mathscr{S}$ that will search all positive integers x, y, z and numbers $n > 2$ for a solution to the equation $x^n + y^n = z^n$.
- Program $\mathscr{P}$ never halts if only if Fermat's last theorem is true.
- That is, if we can solve the halting problem, then we can easily prove (or dis-prove) the Fermat's last theorem!
- (Fermat's last theorem was finally proved in 1995 by Andrew Wiles with help from Richard Taylor.)

Goldbach's Conjecture

Every even number ≥ 4 is the sum of two prime numbers.

- Check: $4 = 2 + 2,\ 6 = 3 + 3,\ 8 = 3 + 5, \ldots$
- Is there a counterexample?
- Let's write a program $\mathscr{P}$ in $\mathscr{S}$ to search for a counterexample!
- Note that the test that a given even number n is an counterexample only requires checking the primitive recursive predicate:

$$\sim (\exists x)_{\leq n} (\exists y)_{\leq n} [\text{Prime}(x) \ \&\ \text{Prime}(y) \ \&\ x + y = n]$$

- The statement that $\mathscr{P}$ never halts is equivalent to Goldbach's conjecture.
- The conjecture is still open; nobody knows yet whether $\mathscr{P}$ will eventually halt.

1.3 Universality (4.3)

Compute with Numbers of Programs

- Programs taking programs as input: Compilers, interpreters, evaluators, Web browsers, ...
- Can we write a program in language $\mathscr{S}$ to accept the number of another program $\mathscr{P}$, as well as the input x to $\mathscr{P}$, then compute $\Psi^{(1)}_{\mathscr{P}}(x)$ as output?
- Yes, we can! The program above is called a universal program.

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \ldots, x_n, y) = \Psi^{(n)}_{\mathscr{P}}(x_1, \ldots, x_n), \quad \text{where } \#(\mathscr{P}) = y.$$

Theorem 3.1. For each $n > 0$, the function $\Phi^{(n)}(x_1, \ldots, x_n, y)$ is partially computable. □

We shall prove this theorem by showing how to construct, for each $n > 0$, a program $\mathscr{U}_n$ which computes $\Phi^{(n)}$. That is,

$$\Psi^{(n+1)}_{\mathscr{U}_n}(x_1, \ldots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \ldots, x_n, x_{n+1}).$$

The programs $\mathscr{U}_n$ are called universal.

“Computer Organization” of $\mathscr{U}_n$

- Program $\mathscr{U}_n$ accepts $n+1$ input variables of which X_{n+1} is a number of a program $\mathscr{P}$, and $X_1, \ldots, X_n$ are provided to $\mathscr{P}$ as input variables.
- All variables used by $\mathscr{P}$ are arranged in the following order

$$Y, X_1, Z_1, X_2, Z_2, \ldots$$

  and their state is coded by the Gödel number $[y, x_1, z_1, x_2, z_2, \ldots]$.
- Let variable S in program $\mathscr{U}_n$ store the current state of program $\mathscr{P}$ coded in the above manner.
- Let variable K in program $\mathscr{U}_n$ store the number such that the Kth instruction of program $\mathscr{P}$ is about to be executed.
- Let variable Z in program $\mathscr{U}_n$ store the instruction sequence of program $\mathscr{P}$ coded as a Gödel number.

Setting Up

As program $\mathscr{U}_n$ computes $\Phi^{(n)}(X_1, \ldots, X_n, X_{n+1})$, we begin $\mathscr{U}_n$ by setting up the initial environment for program (number) X_{n+1} to execute:

$Z \leftarrow X_{n+1} + 1$
$S \leftarrow \prod_{i=1}^{n} (p_{2i})^{X_i}$
$K \leftarrow 1$

- If $X_{n+1} = \#(\mathscr{P})$, where $\mathscr{P}$ consists of instructions $I_1, \ldots, I_m$, then Z gets the value $[\#(I_1), \ldots, \#(I_m)]$.
- S is initialized as $[0, X_1, 0, X_2, \ldots, 0, X_n]$ which gives the first n input variables their appropriate values and gives all other variables the value 0.
- K, the instruction counter, is given the initial value 1.

Decoding Instruction

We first see if the execution of program $\mathscr{P}$ shall halt. If not, we fetch the Kth instruction and decode the instruction.

$[C]$ IF $K = Lt(Z) + 1 \ \vee \ K = 0$ GOTO F
$U \leftarrow r((Z)_k)$
$P \leftarrow p_{r(U)+1}$

- If the computation has ended, GOTO F, where the proper value will be output. (The case for $K = 0$ will be explained later.)

- $(Z)_k = \langle a, \langle b, c\rangle\rangle$ is the number of the Kth instruction. Thus $U = \langle b, c\rangle$ is the code of the statement to be executed.
- The variable mentioned in the statement is the $(r(U) + 1)$th in our list S, and its current value is stored as the exponent to which P divides S.

Instruction Execution

IF $l(U) = 0$ GOTO N
IF $l(U) = 1$ GOTO A
IF $\sim (P|S)$ GOTO N
IF $l(U) = 2$ GOTO M

- If $l(U) = 0$, the instruction is a dummy $\boxed{V \leftarrow V}$ and the computation does nothing. Hence, it goes to N (for *Nothing*).
- If $l(U) = 1$, the instruction is $\boxed{V \leftarrow V + 1}$. The computation goes to A (for *Add*) to add 1 to the exponent on P in the prime power factorization of S.
- If $l(U) \neq 0, 1$, the instruction is either $\boxed{V \leftarrow V - 1}$, or $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$. In both cases, if $V = 0$, the computation does nothing so goes to N. This happens when P is not a divisor of S.
- If $P|S$ and $l(U) = 2$, the computation goes to M (for *Minus*).

Branching

$K \leftarrow \min_{i \leq Lt(Z)}[l((Z)_i) + 2 = l(U)]$
GOTO C

- If $l(U) > 2$ and $P|S$, the current instruction is of the form $\boxed{\text{IF } V \neq 0 \text{ GOTO } L}$ where V has a nonzero value and L is the label whose position in our label list is $l(U) - 2$.
- The next instruction should be the first with this label.
- That is, K should get as its value the least i for which $l((Z)_i) = l(U) - 2$. If there is no instruction with the appropriate label, K gets the 0, which will lead to termination the next time through the main loop.
- Once the instruction counter K is adjusted, the execution enters the main loop by $\boxed{\text{GOTO } C}$.

Subtraction and Addition

$$
\begin{array}{ll}
[M] & S \leftarrow \lfloor S/P \rfloor \\
 & \text{GOTO } N \\
[A] & S \leftarrow S \cdot P \\
[N] & K \leftarrow K+1 \\
 & \text{GOTO } C
\end{array}
$$

- 1 is subtracted from the variable by dividing S by P.
- 1 is added to the variable by multiplying S by P.
- The instruction counter is increased by 1 and the computation returns to the main loop to fetch the next instruction.

Finalizing

$$
\begin{array}{ll}
[F] & Y \leftarrow (S)_1
\end{array}
$$

- One termination, the value of Y for the program being simulated is stored at the exponent on p_1 in S.

$\mathscr{U}_n$, Finally

$$
\begin{array}{ll}
 & Z \leftarrow X_{n+1}+1 \\
 & S \leftarrow \prod_{i=1}^{n}(p_{2i})^{X_i} \\
 & K \leftarrow 1 \\
[C] & \text{IF } K = Lt(Z)+1 \ \vee \ K = 0 \text{ GOTO } F \\
 & U \leftarrow r((Z)_k) \\
 & P \leftarrow p_{r(U)+1} \\
 & \text{IF } l(U) = 0 \text{ GOTO } N \\
 & \text{IF } l(U) = 1 \text{ GOTO } A \\
 & \text{IF } \sim (P|S) \text{ GOTO } N \\
 & \text{IF } l(U) = 2 \text{ GOTO } M \\
 & K \leftarrow \min_{i \le Lt(Z)}[l((Z)_i)+2 = l(U)] \\
 & \text{GOTO } C \\
[M] & S \leftarrow \lfloor S/P \rfloor \\
 & \text{GOTO } N \\
[A] & S \leftarrow S \cdot P \\
[N] & K \leftarrow K+1 \\
 & \text{GOTO } C \\
[F] & Y \leftarrow (S)_1
\end{array}
$$

Notations

For each $n > 0$, the sequence

$$\Phi^{(n)}(x_1, \ldots, x_n, 0), \Phi^{(n)}(x_1, \ldots, x_n, 1), \ldots$$

enumerates all partially computable functions of n variables. When we want to emphasize this aspect we write

$$\Phi_y^{(n)}(x_1, \ldots, x_n) = \Phi^{(n)}(x_1, \ldots, x_n, y)$$

It is often convenient to omit the superscript when $n = 1$, writing

$$\Phi_y(x) = \Phi(x, y) = \Phi^{(1)}(x, y).$$