

Theory of Computation

Prof. Michael Mascagni



Florida State University
Department of Computer Science

Coding Programs by Numbers

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

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

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

Arranging Variables and Labels

- ▶ The variables are arranged in the following order

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

- ▶ The labels are arranged in the following order

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

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

Coding Instructions by Numbers

Let I be an instruction of language \mathcal{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

$$\text{IF } V \neq 0 \text{ GOTO } L'$$

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

$$V \leftarrow V \quad \text{if } I(r(q)) = 0$$

$$V \leftarrow V + 1 \quad \text{if } I(r(q)) = 1$$

$$V \leftarrow V - 1 \quad \text{if } I(r(q)) = 2$$

$$\text{IF } V \neq 0 \text{ GOTO } L' \quad \text{if } j = I(r(q)) - 2 > 0$$

and L' is the j th label in the list.

Coding Programs by Numbers, Finally

Let a program \mathcal{P} consists of the instructions l_1, l_2, \dots, l_k . Then we set

$$\#(\mathcal{P}) = [\#(l_1), \#(l_2), \dots, \#(l_k)] - 1$$

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

Coding Programs by Numbers, Examples

Consider the following “nowhere defined” program \mathcal{P}

```
[A] X ← X + 1
    IF X ≠ 0 GOTO A
```

Let l_1 and l_2 , respectively, be the first and the second instruction in \mathcal{P} , then

$$\#(l_1) = \langle 1, \langle 1, 1 \rangle \rangle = \langle 1, 5 \rangle = 21$$

$$\#(l_2) = \langle 0, \langle 3, 1 \rangle \rangle = \langle 0, 23 \rangle = 46$$

Therefore

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

Coding Programs by Numbers, Examples

What is the program whose number is 199?

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

$$3 = \langle 2, 0 \rangle = \langle 2, \langle 0, 0 \rangle \rangle$$

$$2 = \langle 0, 1 \rangle = \langle 0, \langle 1, 0 \rangle \rangle$$

We conclude that \mathcal{P} is the following program

```
[B] Y ← Y
      Y ← Y
      Y ← 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)!

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

We define predicate HALT(x, y) such that

HALT(x, y) \Leftrightarrow program number y eventually halts on input x .

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

We define predicate HALT(x, y) such that

HALT(x, y) \Leftrightarrow program number y eventually halts on input x .

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

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

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

We define predicate HALT(x, y) such that

HALT(x, y) \Leftrightarrow program number y eventually halts on input x .

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

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

Note that HALT(x, y) is a total function.

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

We define predicate HALT(x, y) such that

HALT(x, y) \Leftrightarrow program number y eventually halts on input x .

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

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

Note that HALT(x, y) is a total function.

But, is HALT(x, y) computable?

HALT(x, y) Is Not Computable

HALT(x, y) Is Not Computable

Theorem 2.1. HALT(x, y) is not a computable predicate.

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 \mathcal{P} :

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

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 \mathcal{P} :

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

It is clear that

$$\Psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} \text{undefined} & \text{if HALT}(x, x) \\ 0 & \text{if } \sim \text{HALT}(x, x). \end{cases}$$

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 \mathcal{P} :

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

It is clear that

$$\Psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} \text{undefined} & \text{if HALT}(x, x) \\ 0 & \text{if } \sim \text{HALT}(x, x). \end{cases}$$

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

$$\text{HALT}(x, y_0) \Leftrightarrow \Psi_{\mathcal{P}}^{(1)}(x) \text{ is defined} \Leftrightarrow \mathcal{P} \text{ halts on } x \Leftrightarrow \sim \text{HALT}(x, x)$$

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 \mathcal{P} :

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

It is clear that

$$\Psi_{\mathcal{P}}^{(1)}(x) = \begin{cases} \text{undefined} & \text{if HALT}(x, x) \\ 0 & \text{if } \sim \text{HALT}(x, x). \end{cases}$$

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

$$\text{HALT}(x, y_0) \Leftrightarrow \Psi_{\mathcal{P}}^{(1)}(x) \text{ is defined} \Leftrightarrow \mathcal{P} \text{ halts on } x \Leftrightarrow \sim \text{HALT}(x, x)$$

Let $x = y_0$, we arrive at

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

which is a contradiction. □

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

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

Let's be precise on what have be proved.

“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 \mathcal{S} . It is a predicate on the computational behavior of the programs, i.e., whether a program y of language \mathcal{S} will halt on input x .

“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 \mathcal{S} . It is a predicate on the computational behavior of the programs, i.e., whether a program y of language \mathcal{S} will halt on input x .
- ▶ It is shown *there exists no program in language \mathcal{S} that computes HALT(x, y)*.

“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 \mathcal{L} . It is a predicate on the computational behavior of the programs, i.e., whether a program y of language \mathcal{L} will halt on input x .
- ▶ It is shown *there exists no program in language \mathcal{L} 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 \mathcal{L} .

“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 \mathcal{L} . It is a predicate on the computational behavior of the programs, i.e., whether a program y of language \mathcal{L} will halt on input x .
- ▶ It is shown *there exists no program in language \mathcal{L} 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 \mathcal{L} .
- ▶ But can HALT(x, y) be expressed in languages other than \mathcal{L} ? 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 \mathcal{P} and an input to the program, can determine whether or not the given program will eventually halt on the given input.

The Unsolvability of Halting Problem

There is no algorithm that, given a program of \mathcal{P} 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 Unsolvability of Halting Problem

There is no algorithm that, given a program of \mathcal{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 \mathcal{S} that computes $HALT(x, y)$,*” as an algorithm can refer to a method in any formalism of computation.

The Unsolvability of Halting Problem

There is no algorithm that, given a program of \mathcal{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 \mathcal{S} that computes $HALT(x, y)$,*” as an algorithm can refer to a method in any formalism of computation.
- ▶ However, language \mathcal{S} can be shown to be as powerful as any known computational formalism. Therefore, we reason that if no program in \mathcal{S} can solve it, no algorithm can.

Church's Thesis

Any algorithm for computing on numbers can be carried out by a program of \mathcal{L} .

Church's Thesis

Any algorithm for computing on numbers can be carried out by a program of \mathcal{L} .

- ▶ This assertion is called *Church's Thesis*.

Church's Thesis

Any algorithm for computing on numbers can be carried out by a program of \mathcal{L} .

- ▶ 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.

Church's Thesis

Any algorithm for computing on numbers can be carried out by a program of \mathcal{L} .

- ▶ 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 \mathcal{L} .*

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

- ▶ This shall not be too surprising, as it is easy to construction short programs of \mathcal{S} such that it is very difficult to tell whether they will ever halt.

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

- ▶ This shall not be too surprising, as it is easy to construction short programs of \mathcal{S} such that it is very difficult to tell whether they will ever halt.
- ▶ Example: Fermat's last theorem.

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

- ▶ This shall not be too surprising, as it is easy to construction short programs of \mathcal{S} such that it is very difficult to tell whether they will ever halt.
- ▶ Example: Fermat's last theorem.
- ▶ Example: Goldbach's conjecture.

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

- ▶ This shall not be too surprising, as it is easy to construction short programs of \mathcal{P} 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 \mathcal{P} 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 \mathcal{P} of language \mathcal{S} that will search all positive integers x, y, z and numbers $n > 2$ for a solution to the equation $x^n + y^n = z^n$.

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 \mathcal{P} of language \mathcal{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 \mathcal{P} never halts if and only if Fermat's last theorem is true.

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 \mathcal{P} of language \mathcal{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 \mathcal{P} never halts if and 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

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 \mathcal{P} of language \mathcal{L} 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 \mathcal{P} never halts if and 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: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5, \dots$

Goldbach's Conjecture

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

- ▶ Check: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5, \dots$
- ▶ Is there a counterexample?

Goldbach's Conjecture

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

- ▶ Check: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5, \dots$
- ▶ Is there a counterexample?
- ▶ Let's write a program \mathcal{P} in \mathcal{I} to search for a counterexample!

Goldbach's Conjecture

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

- ▶ Check: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5, \dots$
- ▶ Is there a counterexample?
- ▶ Let's write a program \mathcal{P} in \mathcal{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:

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

Goldbach's Conjecture

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

- ▶ Check: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5, \dots$
- ▶ Is there a counterexample?
- ▶ Let's write a program \mathcal{P} in \mathcal{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 \mathcal{P} never halts is equivalent to Goldbach's conjecture.

Goldbach's Conjecture

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

- ▶ Check: $4 = 2 + 2$, $6 = 3 + 3$, $8 = 3 + 5, \dots$
- ▶ Is there a counterexample?
- ▶ Let's write a program \mathcal{P} in \mathcal{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 \mathcal{P} never halts is equivalent to Goldbach's conjecture.
- ▶ The conjecture is still open; nobody knows yet whether \mathcal{P} will eventually halt.

Compute with Numbers of Programs

- ▶ Programs taking programs as input: Compilers, interpreters, evaluators, Web browsers,

Compute with Numbers of Programs

- ▶ Programs taking programs as input: Compilers, interpreters, evaluators, Web browsers,
- ▶ Can we write a program in language \mathcal{S} to accept the number of another program \mathcal{P} , as well as the input x to \mathcal{P} , then compute $\Psi_{\mathcal{P}}^{(1)}(x)$ as output?

Compute with Numbers of Programs

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

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \dots, x_n, y) = \Psi_{\mathcal{P}}^{(n)}(x_1, \dots, x_n), \quad \text{where } \#(\mathcal{P}) = y.$$

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \dots, x_n, y) = \Psi_{\mathcal{P}}^{(n)}(x_1, \dots, x_n), \quad \text{where } \#(\mathcal{P}) = y.$$

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

Universality

For each $n > 0$, we define

$$\Phi^{(n)}(x_1, \dots, x_n, y) = \Psi_{\mathcal{P}}^{(n)}(x_1, \dots, x_n), \quad \text{where } \#(\mathcal{P}) = y.$$

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

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

$$\Psi_{\mathcal{U}_n}^{(n+1)}(x_1, \dots, x_n, x_{n+1}) = \Phi^{(n)}(x_1, \dots, x_n, x_{n+1}).$$

The programs \mathcal{U}_n are called universal.

“Computer Organization” of \mathcal{U}_n

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

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

and their state is coded by the Gödel number

$$[y, x_1, z_1, x_2, z_2, \dots].$$

- ▶ Let variable S in program \mathcal{U}_n store the current state of program \mathcal{P} coded in the above manner.
- ▶ Let variable K in program \mathcal{U}_n store the number such that the K th instruction of program \mathcal{P} is about to be executed.
- ▶ Let variable Z in program \mathcal{U}_n store the instruction sequence of program \mathcal{P} coded as a Gödel number.

Setting Up

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

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

Setting Up

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

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

- ▶ If $X_{n+1} = \#(\mathcal{P})$, where \mathcal{P} consists of instructions l_1, \dots, l_m , then Z gets the value $[\#(l_1), \dots, \#(l_m)]$.

Setting Up

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

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

- ▶ If $X_{n+1} = \#(\mathcal{P})$, where \mathcal{P} consists of instructions l_1, \dots, l_m , then Z gets the value $[\#(l_1), \dots, \#(l_m)]$.
- ▶ S is initialized as $[0, X_1, 0, X_2, \dots, 0, X_n]$ which gives the first n input variables their appropriate values and gives all other variables the value 0.

Setting Up

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

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

- ▶ If $X_{n+1} = \#(\mathcal{P})$, where \mathcal{P} consists of instructions l_1, \dots, l_m , then Z gets the value $[\#(l_1), \dots, \#(l_m)]$.
- ▶ S is initialized as $[0, X_1, 0, X_2, \dots, 0, X_n]$ 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 \mathcal{P} shall halt. If not, we fetch the K th instruction and decode the instruction.

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

Decoding Instruction

We first see if the execution of program \mathcal{P} shall halt. If not, we fetch the K th 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.)

Decoding Instruction

We first see if the execution of program \mathcal{P} shall halt. If not, we fetch the K th 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 K th instruction. Thus $U = \langle b, c \rangle$ is the code of the statement to be executed.

Decoding Instruction

We first see if the execution of program \mathcal{P} shall halt. If not, we fetch the K th 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 K th 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 $I(U) = 0$ GOTO N

IF $I(U) = 1$ GOTO A

IF $\sim (P|S)$ GOTO N

IF $I(U) = 2$ GOTO M

Instruction Execution

IF $I(U) = 0$ GOTO N

IF $I(U) = 1$ GOTO A

IF $\sim (P|S)$ GOTO N

IF $I(U) = 2$ GOTO M

- ▶ If $I(U) = 0$, the instruction is a dummy $V \leftarrow V$ and the computation does nothing. Hence, it goes to N (for *Nothing*).

Instruction Execution

IF $I(U) = 0$ GOTO N

IF $I(U) = 1$ GOTO A

IF $\sim (P|S)$ GOTO N

IF $I(U) = 2$ GOTO M

- ▶ If $I(U) = 0$, the instruction is a dummy $V \leftarrow V$ and the computation does nothing. Hence, it goes to N (for *Nothing*).
- ▶ If $I(U) = 1$, the instruction is $V \leftarrow V + 1$. The computation goes to A (for *Add*) to add 1 to the exponent on P in the prime power factorization of S .

Instruction Execution

IF $I(U) = 0$ GOTO N

IF $I(U) = 1$ GOTO A

IF $\sim (P|S)$ GOTO N

IF $I(U) = 2$ GOTO M

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

Instruction Execution

IF $I(U) = 0$ GOTO N

IF $I(U) = 1$ GOTO A

IF $\sim (P|S)$ GOTO N

IF $I(U) = 2$ GOTO M

- ▶ If $I(U) = 0$, the instruction is a dummy $V \leftarrow V$ and the computation does nothing. Hence, it goes to N (for *Nothing*).
- ▶ If $I(U) = 1$, the instruction is $V \leftarrow V + 1$. The computation goes to A (for *Add*) to add 1 to the exponent on P in the prime power factorization of S .
- ▶ If $I(U) \neq 0, 1$, the instruction is either $V \leftarrow V - 1$, or $\text{IF } V \neq 0 \text{ GOTO } L$. 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 $I(U) = 2$, the computation goes to M (for *Minus*).

Branching

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

Branching

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

- ▶ If $I(U) > 2$ and $P|S$, the current instruction is of the form $\text{IF } V \neq 0 \text{ GOTO } L$ where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.

Branching

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

GOTO C

- ▶ If $I(U) > 2$ and $P|S$, the current instruction is of the form IF $V \neq 0$ GOTO L where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.
- ▶ The next instruction should be the first with this label.

Branching

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

GOTO C

- ▶ If $I(U) > 2$ and $P|S$, the current instruction is of the form IF $V \neq 0$ GOTO L where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.
- ▶ The next instruction should be the first with this label.
- ▶ That is, K should get as its value the least i for which $I((Z)_i) = I(U) - 2$. 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.

Branching

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

GOTO C

- ▶ If $I(U) > 2$ and $P|S$, the current instruction is of the form IF $V \neq 0$ GOTO L where V has a nonzero value and L is the label whose position in our label list is $I(U) - 2$.
- ▶ The next instruction should be the first with this label.
- ▶ That is, K should get as its value the least i for which $I((Z)_i) = I(U) - 2$. 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 GOTO C .

Subtraction and Addition

[M] $S \leftarrow \lfloor S/P \rfloor$
GOTO N

[A] $S \leftarrow S \cdot P$

[N] $K \leftarrow K + 1$
GOTO C

Subtraction and Addition

[M] $S \leftarrow \lfloor S/P \rfloor$
GOTO N

[A] $S \leftarrow S \cdot P$

[N] $K \leftarrow K + 1$
GOTO C

- ▶ 1 is subtracted from the variable by dividing S by P .

Subtraction and Addition

[M] $S \leftarrow \lfloor S/P \rfloor$
GOTO N

[A] $S \leftarrow S \cdot P$

[N] $K \leftarrow K + 1$
GOTO C

- ▶ 1 is subtracted from the variable by dividing S by P .
- ▶ 1 is added to the variable by multiplying S by P .

Subtraction and Addition

[M] $S \leftarrow \lfloor S/P \rfloor$
GOTO N

[A] $S \leftarrow S \cdot P$

[N] $K \leftarrow K + 1$
GOTO C

- ▶ 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

[F] $Y \leftarrow (S)_1$

Finalizing

[F] $Y \leftarrow (S)_1$

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

\mathcal{U}_n , Finally

$$Z \leftarrow X_{n+1} + 1$$

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

$$K \leftarrow 1$$

[C] IF $K = Lt(Z) + 1 \vee K = 0$ GOTO F

$$U \leftarrow r((Z)_k)$$

$$P \leftarrow p_{r(U)+1}$$

IF $I(U) = 0$ GOTO N

IF $I(U) = 1$ GOTO A

IF $\sim (P|S)$ GOTO N

IF $I(U) = 2$ GOTO M

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

GOTO C

[M] $S \leftarrow \lfloor S/P \rfloor$

GOTO N

[A] $S \leftarrow S \cdot P$

[N] $K \leftarrow K + 1$

GOTO C

[F] $Y \leftarrow (S)_1$

Notations

For each $n > 0$, the sequence

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

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

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

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

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