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

Institute of Information Science, Academia Sinica

Department of Information Management, National Taiwan University

Week 4, Spring 2008

About This Course Note

- It is prepared for the course *Theory of Computation* taught at the National Taiwan University in Spring 2008.
- 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/`

  and released under a Creative Commons "Attribution-ShareAlike 2.5 Taiwan" license:

  `http://creativecommons.org/licenses/by-sa/2.5/tw/`

One-One Functions

- A function is *one-one* if, for all x, y in the domain of f, $f(x) = f(y)$ implies $x = y$.
- That is, if $x \neq y$, then $f(x) \neq f(y)$.
- Function $f(n) = n^2$ is one-one.
- Function $u_1^2(x_1, x_2) = x_1$ is not one-one as, for example, both $u_1^2(0, 0)$ and $u_1^2(0, 1)$ map to 0.

Onto Functions

- If the range of f is the set S, then we say f is an *onto* function with respect to S, or simply that f is *onto* S.
- Function $f(n) = n^2$ is onto the set of perfect squares $\{n^2 \mid n \in N\}$, but is not onto N.
- Let $S_1 \times S_2$ be domain of function $u_1^2(x_1, x_2) = x_1$, then function $u_1^2(x_1, x_2)$ is onto S_1.

Programs Accepting Any Number of Inputs

- We permit each program to be used with any number of inputs.
- If the program has n input variables, but only $m < n$ are specified, the remaining $n - m$ input variables are assigned the value 0 and the computation proceeds.
- On the other hand, if $m > n$ values are specified, then the extra input values are ignored.

Programs Accepting Any Number of Inputs, Examples

- Consider the following program $\mathscr{P}$ that computes $x_1 + x_2$,

$$
\begin{array}{ll}
 & Y \leftarrow X_1 \\
 & Z \leftarrow X_2 \\
[B] & \text{IF } Z \neq 0 \text{ GOTO } A \\
 & \text{GOTO } E \\
[A] & Z \leftarrow Z - 1 \\
 & Y \leftarrow Y + 1 \\
 & \text{GOTO } B
\end{array}
$$

- We have

$$
\begin{aligned}
\psi^{(1)}_{\mathscr{P}}(r_1) &= r_1 + 0 = r_1 \\
\psi^{(3)}_{\mathscr{P}}(r_1, r_2, r_3) &= r_1 + r_2
\end{aligned}
$$

Pairing Functions

- There is a one-one and onto function from $N \times N$ to N (with domain $N \times N$ and range N). This function is called a pairing function.
- That is, we can map a pair of numbers to a single number, and back, without losing information. Likewise, we can compute from any number a pair of numbers, and back, without missing anything.
- The primitive recursive function

$$\langle x, y \rangle = 2^x(2y + 1)\dot{-}1$$

  is a pairing function.
- $\langle 0, 0 \rangle = 0, \langle 1, 0 \rangle = 1, \langle 0, 1 \rangle = 2, \ldots$

The Pairing Function $\langle x, y \rangle = 2^x(2y+1) \dot{-} 1$

- Note that $2^x(2y+1) \neq 0$, so

$$\langle x, y \rangle + 1 = 2^x(2y+1)$$

- If z is any given number, then there is a *unique* solution x, y to the equation $\langle x, y \rangle = z$.
- Namely, x is the largest number such that $2^x | (z+1)$, and y is then the solution of the equation $2y + 1 = (z+1)/2^x$.
- The pairing function thus defines two functions l and r such that $x = l(z)$ and $y = r(z)$.

The Pairing Function $\langle x, y \rangle = 2^x(2y+1) \dot{-} 1$, Continued

If $\langle x, y \rangle = z$, then $x, y < z + 1$. Hence, $l(z) \leq z$, and $r(z) \leq z$.

We can write

$$\begin{aligned} l(z) &= \min_{x \leq z}[(\exists y)_{\leq z}(z = \langle x, y \rangle)], \\ r(z) &= \min_{y \leq z}[(\exists x)_{\leq z}(z = \langle x, y \rangle)], \end{aligned}$$

so that $l(z)$ and $r(z)$ are primitive recursive functions.

Pairing Function Theorem

Theorem 8.1. The functions $\langle x, y \rangle$, $l(z)$, and $r(z)$ have the following properties:

1. they are primitive recursive;
2. $l(\langle x, y \rangle) = x, \quad r(\langle x, y \rangle) = y$;
3. $\langle l(z), r(z) \rangle = z$;
4. $l(z), r(z) \leq z$.

Gödel Number

We define the *Gödel Number* of the sequence $(a_1, \ldots, a_n)$ to be the number

$$[a_1, \ldots, a_n] = \prod_{i=1}^{n} p_i^{a_i}$$

Thus, the the Gödel number of the sequence (3, 1, 5, 4, 6) is

$$[3, 1, 5, 4, 6] = 2^3 \cdot 3^1 \cdot 5^5 \cdot 7^4 \cdot 11^6$$

For each fixed n, the function $[a_1, \ldots, a_n]$ is clearly primitive recursive. Note that the Gödel numbering method encodes and decodes arbitrary finite sequences of numbers.

Uniqueness Property of Gödel Numbering

Theorem 8.2. If $[a_1, \ldots, a_n] = [b_1, \ldots, b_n]$, then

$$a_i = b_i$$

for all $i = 1, \ldots, n$. □

This result is an immediate consequence of the uniqueness of the factorization of integers into primes, sometimes referred to as the *unique factorization theorem*. Note that,

$$1 = 2^0 = 2^0 3^0 = 2^0 3^0 5^0 = \ldots,$$

hence it is natural to regard 1 as the Gödel number of the "empty" sequence (i.e., the sequence of length 0).

Function $(x)_i$

We now define a primitive recursive function $(x)_i$ so that if

$$x = [a_1, \ldots, a_n]$$

then $(x)_i = a_i$. We set

$$(x)_i = \min_{t \leq x}(\sim p_i^{t+1} | x)$$

Note that $(x)_0 = 0$, and $(0)_i = 0$ for all i.

Function $Lt(x)$

We also define the "length" function Lt,

$$Lt(x) = \min_{i \leq x}[(x)_i \neq 0 \ \& \ (\forall j)_{\leq x}(j \leq i \ \vee \ (x)_j = 0)]$$

For example, if $x = 20 = 2^2 \cdot 5^1 = [2, 0, 1]$ then $(x)_1 = 2, (x)_2 = 0, (x)_3 = 1$, but $(x)_4 = 0, (x)_5 = 0, \ldots, (x)_i = 0$, for all $i \geq 4$. So $Lt(20) = 3$. Note that $Lt(0) = Lt(1) = 0$.

If $x > 1$, and $Lt(x) = n$, then p_n divides x but no prime greater than p_n divides x.

Sequence Number Theorem

Theorem 8.3.

1. $$([a_1, \ldots, a_n])_i = \begin{cases} a_i & \text{if } 1 \leq i \leq n \\ 0 & \text{otherwise.} \end{cases}$$

2. $$[(x)_1, \ldots, (x)_n] = x \quad \text{if } n \geq Lt(x).$$

□

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

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

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

$V \leftarrow V$	if $l(r(q)) = 0$
$V \leftarrow V + 1$	if $l(r(q)) = 1$
$V \leftarrow V - 1$	if $l(r(q)) = 2$
IF $V \neq 0$ GOTO L'	if $j = l(r(q)) - 2 > 0$

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?

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

$$\begin{aligned} [B]\ & Y \leftarrow Y \\ & Y \leftarrow Y \\ & Y \leftarrow Y + 1 \end{aligned}$$

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

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

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 $\mathscr{P}$ 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 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 $\mathscr{P}$ 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 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 $\mathscr{P}$:

[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 $\mathscr{P}$:

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

It is clear that

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

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 $\mathscr{P}$:

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

It is clear that

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

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

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

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

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

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

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

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 $\mathscr{S}$. 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.

"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).

"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 $\mathscr{S}$.

"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.

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 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.

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 $\mathscr{S}$ has 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 $\mathscr{S}$.

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*.

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.

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 some 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.

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.

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.

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 $x^n + y^n = z^n$.

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.

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

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: $4 = 2 + 2,\ 6 = 3 + 3,\ 8 = 3 + 5, \ldots$

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?

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!

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:

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

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.

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:

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

- 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.