Lecture Notes on Great Ideas in Theoretical Computer Science

Yuxin Deng
East China Normal University
June 12, 2018

Contents

1 Introduction 2

2 Algorithm 2
  2.1 Euclid’s Algorithm 2
  2.2 Modern Concept of Algorithm 3

3 Logic 4
  3.1 Propositional Logic 4
  3.2 Predicate Logic 5
  3.3 Gödel’s Incompleteness Theorems 6

4 The Lambda Calculus 6
  4.1 Syntax 6
  4.2 Semantics 8

5 Turing Machines 10
  5.1 The Computational Model 10
  5.2 The Halting Problem 12

6 P and NP Problems 13

7 Randomized Algorithms 16

8 Formal Methods 18
  8.1 Formal Specification 18
  8.2 Process Algebras 19
  8.3 Temporal Logic 21

9 Cryptography 22
  9.1 Basic Concepts 22
  9.2 RSA 23

10 Quantum Computation 24
  10.1 Quantum Bits 24
1 Introduction

Quite a lot of undergraduate students have the misconception that computer science is a discipline mostly about computer programming. Indeed, in their curricula a great emphasis is put on improving their programming skills. However, this course has a very different flavor. As we will see in the course, there has been a great many profound theories closely related to mathematics lying behind computer programs that support all kinds of ordinary applications we find in our desktops, tablets, smart phones etc. More precisely, this course gives an introduction to some of the greatest ideas of theoretical computer science. Starting with examples of computational thinking such as Euclid’s algorithm, the course will progress through propositional and predicate logics, Gödel’s theorems, the lambda calculus, Turing machines, the P versus NP problem, randomized algorithms, formal methods, cryptography, and quantum computing. These are a few selected topics covered by the course, which is implemented as a series of seminars organized in various forms including discussion, oral presentations, and lectures. The students are encouraged to actively participate in each seminar so that they can understand, appreciate, and debate about the implications of many of these ideas.

The lecture notes are by no means a survey of the greatest ideas in theoreticl computer science. They merely provide a brief introduction to some ideas in order to stimulate more discussion. Moreover, the selection of the topics is very subjective. The contents of the notes are drawn from various sources (cf. the references); in particular, [1] is a great inspiration.

2 Algorithm

Using the Euclid’s algorithm as an example, we introduce the idea of algorithm design with recursion and the divide-and-conquer paradigm. Then we briefly give some historical notes about modern theory of algorithms.

2.1 Euclid’s Algorithm

One of the oldest algorithms is Euclid’s algorithm that provides an efficient method for computing the greatest common divisor (GCD) of two natural numbers. The key idea of the algorithm is that if a number $p$ divides both $a$ and $b$, then it also divides $a - b$. (Note that if $a = np$ and $b = mp$ for some numbers $n$ and $m$ then $a - b = (n - m)p$.) Therefore, finding the GCD of two numbers, say 513 and 399, is the same as finding the GCD of 399 and 513 - 399 = 114. Now 114 is smaller than 399 so the computation has made progress. By repeatedly using this observation, we see that

$$GCD(513, 399) = GCD(399, 114) = GCD(285, 114) = \cdots = GCD(57, 57) = 57$$

The algorithm can be described by the pseudocode in Algorithm 1. Essentially, what the above algorithm does is to compute the following recursive function

$$GCD(a, b) = \begin{cases} 
  a & \text{if } a = b \\
  GCD(a - b, b) & \text{if } a > b \\
  GCD(a, b - a) & \text{if } a < b.
\end{cases} \quad (1)$$

If a problem can be formulated by a recursive function like (1), then it will be easy to solve it by an algorithm. The process of formulating a problem while keeping recursion in mind is an
Algorithm 1: Euclid’s GCD algorithm

<table>
<thead>
<tr>
<th>Algorithm 1: Euclid’s GCD algorithm</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>input</strong>: Two positive integers (a) and (b)</td>
</tr>
<tr>
<td><strong>output</strong>: The GCD of (a) and (b)</td>
</tr>
<tr>
<td><strong>begin</strong></td>
</tr>
</tbody>
</table>
| \[\text{while } a \neq b \text{ do}\]
| \[\text{if } a < b \text{ then}\]
| \[\text{swap } a \text{ with } b;\]
| \[a := a - b;\]
| \[\text{end}\]
| \[\text{return } a;\]
| **end** |

Example of computational thinking, because recursion is used in many places in computer science. More generally, sometimes we need to split a complex problem into several sub-problems of the same type until they become simple enough to be solved directly. Then we combine the solutions to the sub-problems so as to yield a solution to the original problem. This is an important algorithm design paradigm called divide and conquer.

**Exercise 2.1** Design an algorithm to sort a list of natural numbers using the divide-and-conquer paradigm.

2.2 Modern Concept of Algorithm

Although Euclid’s algorithm goes back to 300 BC, it was not until 1930s that the modern theory of algorithms was introduced by Alonzo Church, Alan Turing, and other pioneers of computer science. The theory was developed in response to a challenge posed by the mathematician David Hilbert. Hilbert asked whether all the problems of mathematics can be solved by some algorithms. The original problem given in 1928 is called the entscheidungsproblem. To solve that problem requires a clear answer to the following question: what does it mean for a function \(f: \mathbb{N} \rightarrow \mathbb{N}\) to be computable? At that time there was no formal definition of algorithms. Therefore, the class of functions computable by an algorithm was also vague. Intuitively, we could say a function \(f\) is computable if there is a pencil-and-paper method allowing a trained person to calculate \(f(n)\), for any given \(n\). But how to make the concept of a pencil-and-paper method formal? Three researchers made efforts in answering this question.

- Gödel defined the class of general recursive functions and postulated that a function is intuitively computable if and only if it is general recursive.
- Church defined an idealized programming language called the lambda calculus and postulated that a function is intuitively computable if and only if it can be written as a lambda term.
- Turing defined an idealized computer for doing calculations, which is now called a Turing machine and postulated that a function is intuitively computable if and only if it can be computed by such a machine.

Church, Kleene, Rosser, and Turing proved that all three computational models were equivalent to each other. In other words, the models define the same class of functions. Church and Turing showed that a general solution to the entscheidungsproblem is impossible, assuming that the intuitive concept of computable functions by an algorithm corresponds to the mathematical concept of functions computable by Turing machines (or equivalently, the functions expressible in the lambda calculus).
calculus). This assumption is now known as the Church-Turing thesis. The importance of this thesis lies in the fact that it turns the vague study of algorithms into a rigorous mathematical study. Amazingly, eighty years have passed since the thesis was put forward, and we still see no evidence against the thesis!

**Exercise 2.2** Read the articles of Church [4] and Turing [12].

## 3 Logic

Logic plays a crucial role in computer science. Its key areas of applications include computability theory, artificial intelligence for automated reasoning, software engineering for validating the correctness of software specifications, just to name a few. In this section, we briefly review two basic forms of logic: proposition logic and predicate logic. They belong to symbolic logic and study symbolic abstractions that capture the formal features of logical inference.

### 3.1 Propositional Logic

Propositions are just statements and propositional logic studies how to combine statements to form other statements. The earliest logician is probably Aristotle, who formalized the concept of syllogism.

All men are mortal, Socrates is man, so Socrates is mortal.

In propositional logic, we can formalize the above reasoning by the transitivity of implications:

If both $A \rightarrow B$ and $B \rightarrow C$ are true, then so is $A \rightarrow C$.

**Definition 3.1** Propositions are inductively by the following two rules:

1. Propositional letters, written $A, B, \ldots$, are propositions.
2. If $\alpha$ and $\beta$ are propositions, then $\alpha \land \beta$, $\alpha \lor \beta$, $\neg \alpha$ and $\alpha \rightarrow \beta$ are propositions.

What we defined above is the syntax of propositional logic. We now turn to the semantic part.

**Definition 3.2** A truth assignment $A$ is a function that assigns to each propositional letter $A$ a unique truth value $A(A) \in \{T, F\}$. A truth valuation $V$ with respect to $A$ is a function that assigns to each proposition $\alpha$ a unique truth value such that

1. $V(\neg \alpha) = T$ iff $V(\alpha) = F$;
2. $V(\alpha \land \beta) = T$ iff $V(\alpha) = T$ and $V(\beta) = T$;
3. $V(\alpha \lor \beta) = T$ iff $V(\alpha) = T$ or $V(\beta) = T$;
4. $V(\alpha \rightarrow \beta) = T$ iff either $V(\alpha) = F$ or $V(\alpha) = V(\beta) = T$.

A proposition $\alpha$ is valid if for any valuation $V$, we have $V(\alpha) = T$. Two propositions $\alpha$ and $\beta$ are logically equivalent if for every valuation $V$, we have $V(\alpha) = V(\beta)$.

In order to reason about propositions, we use the following axioms and an inference rule.

1. $\alpha \rightarrow (\beta \rightarrow \alpha)$
2. $(\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))$
(3) \((\neg\beta \rightarrow \neg\alpha) \rightarrow ((\neg\beta \rightarrow \alpha) \rightarrow \beta)\)

where \(\alpha, \beta\) and \(\gamma\) can be any propositions. This is a list of \textit{axiom schemes}. The axioms are all instances of these schemes. The only rule of inference is called \textit{modus ponens}, written as follows:

\[
\frac{\alpha \quad \alpha \rightarrow \beta}{\beta}.
\]

The axioms and the rule give rise to a Hilbert-style proof system. The proposition \(\alpha\) is provable if it can be derived by using the axioms and the inference rule. The proof system is sound and complete.

**Theorem 3.1** A proposition \(\alpha\) is provable if and only if it is valid.

### 3.2 Predicate Logic

In ordinary life, we often talk about properties of objects. For example, when we are talking about natural numbers we may say “\(x\) is a prime number greater than 59”. In the formal language of predicate logic, we would use a predicate \(\psi(x)\) to describe this property. Here we take the natural numbers as our domain of discourse and \(x\) is a \textit{variable} that can be instantiated by an object in the domain of discourse. As with propositions, complex predicates can be built out of simple predicates by using connectives such as \(\neg\), \(\land\) and \(\lor\). In addition, predicate logic uses functions and two other constructors:

1. the \textit{universal quantifier}, \(\forall\), which means “for all”;
2. the \textit{existential quantifier}, \(\exists\), which means “there exists”.

For instance, to continue our example above, we may state the property “there exists some prime number \(x\) greater than 59” as \((\exists x)\psi(x)\). Note that we can also view \(\exists\) as a defined symbol and replace \((\exists x)\psi(x)\) by \(\neg(\forall x)\neg\psi(x)\).

The semantics of predicate logic is similar to that of proposition logic, with the main difference that we need to fix a nonempty domain for assigning values to variables.

A proof system for predicate logic contains the following axioms:

1. \(\alpha \rightarrow (\beta \rightarrow \alpha)\)
2. \((\alpha \rightarrow (\beta \rightarrow \gamma)) \rightarrow ((\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow \gamma))\)
3. \((\neg\alpha) \rightarrow (\alpha \rightarrow \beta)\)
4. \((\forall x)\alpha(x) \rightarrow \alpha(t)\) for any term \(t\) substitutable for \(x\) in \(\alpha\)
5. \((\forall x)(\alpha \rightarrow \beta) \rightarrow (\alpha \rightarrow (\forall x)\beta)\) if \(\alpha\) contains no free occurrences of \(x\).

Concerning inference rules, we now have two: besides the \textit{modus ponens} rule which already occurred in proposition logic, we have a \textit{generalization} rule:

\[
\frac{(\forall x)\alpha}{\alpha}.
\]

It turns out that the five axioms and the two rules of inference constitute a sound and complete proof system. The first proof of the completeness theorem for predicate logic was given by Gödel.

**Theorem 3.2** A predicate formula \(\alpha\) is provable if and only if it is valid.

**Exercise 3.1** Read the textbook [8], where the proofs of soundness and completeness for propositional logic and predicate logic can be found.
3.3 Gödel's Incompleteness Theorems

In 1931 Kurt Gödel announced two theorems of mathematical logic that demonstrate the inherent limitations of every formal axiomatic system containing basic arithmetic. The first incompleteness theorem states that no consistent system of axioms whose theorems can be listed by an effective procedure is capable of proving all facts about the natural numbers. Here consistency means that there is no statement such that both its negation and itself are provable from the axioms, and being effective means there is a computational method (e.g. an algorithm or a computer program) that mechanically applying the axioms and rules to confirm precisely when something is a proof. For any such system, there will always be statements about the natural numbers that are true, but that are unprovable within the system. To prove those statements, we need a system with more expressive power, but then there will be true statements unprovable within that system, and so on.

To prove the first incompleteness theorem, Gödel considered the sentence: “This sentence is not provable.” If the sentence is true, then it means that this is an unprovable sentence. If the sentence is false, then it means that the sentence is provable, which can’t happen because in a sound system of logic it is impossible to prove a falsehood. In Gödel’s proof, each sentence is encoded as a natural number (usually called Gödel number) and the notion of provability itself can be encoded by Gödel numbers. The relation between a sentence and its proof becomes an arithmetic relation between their respective Gödel numbers.

The second incompleteness theorem is an extension of the first. It shows that if a consistent system whose theorems can be effectively listed is capable of proving certain basic facts about the natural numbers, then one particular arithmetic truth the system cannot prove is the consistency of the system itself.

The key idea in proving the second incompleteness theorem is to show that various facts about provability used in the proof of the first incompleteness theorem can be formalized within the system using a predicate for provability. In particular, the proof that the consistency of the system implies the provability of the sentence “This sentence is not provable” can be formalized within the system, and therefore that sentence can be proved in the system, which gives rise to a contradiction.

4 The Lambda Calculus

The lambda calculus is a very simple yet powerful programming language. Many modern programming languages can be viewed as extensions of it. For example, practical functional programming languages including Lisp, Scheme, Haskell and ML extend the lambda calculus with various features such as data types, input/output, side effects, objects etc.

4.1 Syntax

The citizens in the realm of the lambda calculus are called lambda terms defined as follows.

**Definition 4.1** Assume an infinite set \( \mathcal{V} \) of variables, denoted by \( x, y, z \ldots \). The lambda terms are generated by the following Backus-Naur Form:

\[
M, N ::= x \mid (MN) \mid (\lambda x.M)
\]

Alternatively, the set of lambda terms is the smallest set \( \Lambda \) satisfying the three rules below, assuming the infinite set \( \mathcal{V} \) of variables.

- If \( x \in \mathcal{V} \) then \( x \in \Lambda \) (variables)
- If \( M, N \in \Lambda \) then \( (MN) \in \Lambda \) (applications)
Table 1: Rules for $\alpha$-equivalence

<table>
<thead>
<tr>
<th>$M = M$</th>
<th>$M = M'$</th>
<th>$N = N'$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$MN = MN'$</td>
<td>$M = M'$</td>
<td>$\lambda x. M = \lambda x. M'$</td>
</tr>
<tr>
<td>$\lambda x. M = \lambda y. M{y/x}$</td>
<td>$y \notin M$</td>
<td>$\lambda x. M = \lambda y. M{y/x}$</td>
</tr>
</tbody>
</table>

- If $x \in V$ and $M \in \Lambda$ then $(\lambda x. M) \in \Lambda$ (lambda abstractions)

For example, both $xx$ and $(\lambda x. xy)$ are lambda terms, so is $(\lambda f. (f(x)))$.

We often omit outermost parentheses. For example, we write $MN$ instead of $(MN)$. Applications always associate to the left, i.e. $MNP$ means $(MN)P$. The body of a lambda abstraction (the part after the dot) extends as far to the right as possible. For example, $\lambda x. MN$ means $\lambda x. (MN)$ rather than $(\lambda x. M)N$. Multiple lambda abstractions like $\lambda x. \lambda y. \lambda z. M$ can be abbreviated as $\lambda xyz. M$.

An occurrence of a variable $x$ inside $\lambda x. N$ is said to be bound. In that case, the corresponding $\lambda x$ is called a binder, and the subterm $N$ is the scope of the binder. A variable occurrence that is not bound is said to be free. For example, in the term $(\lambda x. xy) x$, the variable $y$ is free, but $x$ has both a free and a bound occurrence. More formally, we define the set of free variables of a term $M$, written $FV(M)$, in a recursive way called induction on the structure of $M$ or simply structural induction.

$$FV(x) = \{x\}$$
$$FV(MN) = FV(M) \cup FV(N)$$
$$FV(\lambda x. M) = FV(M) \setminus \{x\}$$

Given two variables $x, y$ and a term $M$, we write $M\{y/x\}$ for the term obtained by renaming $x$ as $y$ in $M$.

$$x\{y/x\} \equiv y$$
$$z\{y/x\} \equiv z, \text{ if } x \neq z$$
$$(MN)\{y/x\} \equiv (M\{y/x\})(N\{y/x\})$$
$$(\lambda x. M)\{y/x\} \equiv \lambda y. (M\{y/x\})$$
$$(\lambda z. M)\{y/x\} \equiv \lambda z. (M\{y/x\}), \text{ if } x \neq z$$

The renaming operation defined above replaces all occurrences of $x$ by $y$. If one term can be obtained from the other by only renaming bound variables, then they are $\alpha$-equivalent.

**Definition 4.2** The binary relation $\equiv$ on lambda terms is the smallest relation defined by the six rules in Table 1.

For example, the two terms $\lambda x. xy$ and $\lambda z. zy$ are identified up to $\alpha$-equivalence. In the sequel, we will not distinguish $\alpha$-equivalent lambda terms, which is fixed as a convention.

We have defined a renaming operation to change one variable into another. Going a bit further, we now define a substitution operation that allow us to replace a free variable by a lambda term. In doing so we should be careful in order to prevent existing free variables from being captured.

**Definition 4.3** The (capture-avoiding) substitution of $N$ for free occurrences of $x$ in $M$, denoted
M\{N/x\}, is defined below:

\[
\begin{align*}
  x\{N/x\} & \equiv N \\
y\{N/x\} & \equiv y, \quad \text{if } x \neq y \\
(MP)\{N/x\} & \equiv (M\{N/x\})(P\{N/x\}) \\
(\lambda x.M)\{N/x\} & \equiv \lambda x.M \\
(\lambda y.M)\{N/x\} & \equiv \lambda y.(M\{y'/y\}\{N/x\}), \quad \text{if } x \neq y \text{ and } y \notin FV(N) \\
(\lambda y.M)\{N/x\} & \equiv \lambda y'.(M\{y'/y\}\{N/x\}), \quad \text{if } x \neq y, y \in FV(N), \text{ and } y' \text{ fresh.}
\end{align*}
\]

Here a variable is called fresh if it has not been used before. For example, in the substitution \((\lambda x.xy)\{xz/y\}\) we would like to avoid the free occurrence of \(x\) in the term \(xz\) from being captured by \(\lambda x\), thus we rename the bound variable \(x\) in \(\lambda x.xy\) into a fresh variable \(x'\) before the substitution and obtain \(\lambda x'.x'(xz)\) after the substitution.

### 4.2 Semantics

We now turn to the operational semantics of the lambda calculus. According to the semantics, lambda terms are evaluated by performing \(\beta\)-reductions. A term of the form \((\lambda x.M)N\) is called a \(\beta\)-redex. It reduces to \(M\{N/x\}\) (the reduct) in one step. If the latter has a redex, it can also reduce. A lambda term without any \(\beta\)-redex is in \(\beta\)-normal form. For example, the term \((\lambda x.xy)(\lambda x.x)\) can reduce as follows, where each redux is underlined.

\[
\begin{align*}
(\lambda x.xy)(\lambda z.z) & \rightarrow_\beta (\lambda z.z)y \\
& \rightarrow_\beta y
\end{align*}
\]

Note that some terms may never reach a normal form. For example,

\[
(\lambda x.xx)(\lambda x.xx) \rightarrow_\beta (\lambda x.xx)(\lambda x.xx) \rightarrow_\beta \cdots
\]

Therefore, the term \((\lambda x.xx)(\lambda x.xx)\) has no normal form.

We now give a formal definition of \(\beta\)-reduction.

**Definition 4.4** The single-step \(\beta\)-reduction is the smallest relation \(\rightarrow_\beta\) satisfying:

\[
\begin{align*}
(\lambda x.M)N & \rightarrow_\beta M\{N/x\} \\
M & \rightarrow_\beta M' \\
MN & \rightarrow_\beta MN
\end{align*}
\]

We write \(M \rightarrow_\beta M'\) if \(M\) can be transformed into \(M'\) by zero or more reduction steps; formally, the relation \(\rightarrow_\beta\) is the reflexive transitive closure of \(\rightarrow_\beta\).

The lambda calculus has a very simple syntax and semantics. Amazingly, it is expressive enough to encode many data structures.

We first introduce two lambda terms to represent the boolean values true and false. Let \(T = \lambda xy.x\) and \(F = \lambda xy.y\) In order to encode the boolean function AND, let and be the term \(\lambda ab.abF\). Then we have the following reductions:

\[
\begin{align*}
\text{and TT} & \rightarrow_\beta T \\
\text{and TF} & \rightarrow_\beta F \\
\text{and FT} & \rightarrow_\beta F \\
\text{and FF} & \rightarrow_\beta F
\end{align*}
\]

Of course, this kind of encoding is not unique. For example, The AND function can also be encoded as \(\lambda ab.bab\).
Exercise 4.1 Below is an encoding of some other boolean functions.

\[
\begin{align*}
\text{not} &= \lambda a. a \text{FT} \\
\text{or} &= \lambda ab. a \text{T}b \\
\text{xor} &= \lambda ab. a (b \text{FT})b \\
\text{if-then-else} &= \lambda x.x
\end{align*}
\]

Verify that they work as expected. For example, we have

\[
\begin{align*}
\text{if-then-else } TMN &\rightarrow_\beta M \\
\text{if-then-else } FN &\rightarrow_\beta N
\end{align*}
\]

for any terms \(M\) and \(N\).

Let \(f\) and \(x\) be two variables, and \(n \geq 0\) is a natural number. We write \(f^n x\) for the term \(f(f(\ldots (fx)\ldots))\), where \(f\) occurs \(n\) times. The \textit{n}th Church numeral is the lambda term \(\pi = \lambda fx. f^n x\).

\[
\begin{align*}
0 &= \lambda fx.x \\
1 &= \lambda fx.fx \\
2 &= \lambda fx.f(fx) \\
\ldots
\end{align*}
\]

The successor function can be defined by the term \(\text{succ} = \lambda nfx. (nfx)(fx)\). When we apply it to the numeral \(\pi\), the following reduction steps can be observed.

\[
\begin{align*}
\text{succ } \pi &= (\lambda nfx. (nfx)(fx))((\lambda fx. f^n x)) \\
&\rightarrow_\beta \lambda fx. f((\lambda fx. f^n x)fx) \\
&\rightarrow_\beta \lambda fx. f(f^n x) \\
&= \lambda fx. f^{n+1} x \\
&= n + 1
\end{align*}
\]

Exercise 4.2 Addition and multiplication may be defined as follows. Let \(\text{add} = \lambda nmfx. nf(f^m x)\) and \(\text{mult} = \lambda nfx. n(fx)\). Show that

\[
\begin{align*}
\text{add } \pi \pi &\rightarrow_\beta n + m \\
\text{mult } \pi \pi &\rightarrow_\beta n \cdot m
\end{align*}
\]

A useful function is \(\text{iszero}\) that tests if a number is zero or not. Let \(\text{iszero} = \lambda nxy. n(\lambda z. y)x\) and verify that \(\text{iszero}(0) = \text{true}\) and \(\text{iszero}(n+1) = \text{false}\).

Exercise 4.3 The predecessor function is much more difficult. A possible definition is the following one.

\[
\text{pred} = \lambda nfx. n(\lambda gh. h(gf))(\lambda u. x)(\lambda u. u)
\]

Verify that

\[
\begin{align*}
\text{pred } \pi &\rightarrow_\beta \pi \\
\text{pred } 0 &\rightarrow_\beta 0
\end{align*}
\]

So far we have seen some simple boolean and arithmetic functions. Imagine now we would like to define something more complicated, e.g. the factorial function.

\[
\text{fact}(n) = \begin{cases} 
1 & \text{if } n = 0 \\
 n \cdot \text{fact}(n - 1) & \text{if } n > 0 
\end{cases}
\]

This function is recursively defined. To encode it as a lambda term, we should first express recursion in the language of the lambda calculus. This can be done by introducing \textit{fixed points}.

For any function \(f\), if \(f(x) = x\) for some \(x\), then we say \(x\) is a fixed point of \(f\). Analogously, if \(F\) and \(M\) are lambda terms with \(FN \rightarrow_\beta N\), then we say \(M\) is a fixed point of \(F\).
Theorem 4.1  In the lambda calculus, every term $F$ has a fixed point.

Proof: Let $\Theta = AA$ where $A = \lambda xy. y(xx)$. For any lambda term $F$, we claim that $\Theta F$ is a fixed point of $F$. This can be seen as follows.

\[
\begin{align*}
\Theta F &= AAF \\
&= (\lambda xy.y(xx))AF \\
&\rightarrow_\beta F(AAF) \\
&= F(\Theta F)
\end{align*}
\]

Thus $\Theta F$ is a fixed point of $F$. $\square$

Now we are ready to define recursive functions. Consider the factorial function again. If we have a term $\text{fact}$ to represent the function $\text{fact}$, the following equation should be satisfied.

\[
\text{fact } n = \text{if-then-else } (\text{iszero } n)(\top)(\text{mult } n(\text{fact } (\text{pred } n)))
\]

Let us rewrite the equation so as to leave $\text{fact}$ alone on the left hand side.

\[
\begin{align*}
\text{fact } &= \lambda n. \text{if-then-else } (\text{iszero } n)(\top)(\text{mult } n(\text{fact } (\text{pred } n))) \\
\text{fact } &= (\lambda f. \lambda n. \text{if-then-else } (\text{iszero } n)(\top)(\text{mult } n(f(\text{pred } n)))) \text{fact}
\end{align*}
\]

To solve the equation above, we use the fixed point combinator $\Theta$ in the proof of Theorem 4.1 and define

\[
\text{fact } = \Theta(\lambda f. \lambda n. \text{if-then-else } (\text{iszero } n)(\top)(\text{mult } n(f(\text{pred } n))))
\]

Exercise 4.4 Show that $\text{fact } n \rightarrow_\beta \text{m}$, where $m = \text{fact } (n)$, for any natural number $n$.

Exercise 4.5 Consider the four commonly used data structures: pairs, tuples, lists and trees. Show that all of them can be encoded in the lambda calculus.

Exercise 4.6 The book of Barendregt [3] gives a comprehensive description and explanation of the lambda calculus. Check it out from the university library. For a quick introduction to the lambda calculus, the nice lecture notes of Selinger [10] are recommended.

5 Turing Machines

5.1 The Computational Model

A Turing machine is a mathematical model of computation that describes an abstract machine that manipulates symbols on a strip of tape according to a table of rules. The main ingredients of a Turing machine are the following:

- A **tape** divided into many cells. Each cell contains a symbol from some finite alphabet with a special **blank** symbol. The tape is arbitrarily extensible to the right, which serves as an unlimited memory of the machine.

- A **head** that can read and write symbols on the tape. It can also move both to the left and to the right.

- A **state register** that stores the current state of the machine. Initially the machine is in a special start state.

- A finite **table** of instructions described as transition rules. According to the current state and the symbol being scanned by the head, the machine applies a rule to write a symbol in the current cell, move the head one cell to the left or right, and go to a new state.
In Figure 1, we show the schematic of a Turing machine. The state register and the instructions are stored in the control unit.

A formal definition of Turing machines is given below.

**Definition 5.1** A Turing machine is a 7-tuple \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})\), where

- \(Q\) is a finite set of states,
- \(\Sigma\) is a finite set of input symbols not containing the blank symbol \(b\),
- \(\Gamma\) is a finite set of tape alphabet symbols with \(b \in \Gamma\) and \(\Sigma \subseteq \Gamma\),
- \(\delta : Q \times \Gamma \to Q \times \Gamma \times \{L,R\}\) is the transition function,
- \(q_0 \in Q\) is the start state,
- \(q_{\text{acc}} \in Q\) is the accept state,
- \(q_{\text{rej}} \in Q\) is the reject state, where \(q_{\text{rej}} \neq q_{\text{acc}}\).

**Example 5.1** Consider the Turing machine \((Q, \Sigma, \Gamma, \delta, q_0, q_{\text{acc}}, q_{\text{rej}})\), where

- \(Q = \{Q_n | 0 \leq n \leq 7\} \cup \{q_{\text{acc}}, q_{\text{rej}}\}\),
- \(\Sigma = \{0, 1, \#\}\) and \(\Gamma = \{0, 1, \#, b, c\}\).
- The start, accept, and reject states are \(q_0\), \(q_{\text{acc}}\) and \(q_{\text{rej}}\), respectively.
- The transition function \(\delta\) is given by the following rules

\[
\begin{align*}
(q_0, 0, q_1, c, R) &\quad (q_0, 1, q_2, c, R) &\quad (q_0, \#, q_7, \#, R) \\
(q_1, 0, q_1, 0, R) &\quad (q_1, 1, q_1, 1, R) &\quad (q_1, \#, q_3, \#, R) \\
(q_2, 0, q_2, 0, R) &\quad (q_2, 1, q_2, 1, R) &\quad (q_2, \#, q_4, \#, R) \\
(q_3, 0, q_5, c, L) &\quad (q_3, c, q_3, c, R) &\quad \\
(q_4, 1, q_5, c, L) &\quad (q_4, c, q_4, c, R) &\quad (q_5, \#, q_6, \#, L) \\
(q_5, 0, q_5, 0, L) &\quad (q_5, 1, q_5, 1, L) &\quad (q_5, c, q_5, c, L) \\
(q_6, 0, q_6, 0, L) &\quad (q_6, 1, q_6, 1, L) &\quad (q_6, c, q_0, c, R) \\
(q_7, c, q_7, c, R) &\quad (q_7, b, q_{\text{acc}}, b, R) &\quad \\
\end{align*}
\]

plus some transitions going to the reject state, which occur implicitly whenever a state lacks a transition for a particular symbol under the read-write head. For example, we did not write a rule for state \(q_3\) with input symbol 1 because we implicitly assume the rule \((q_3, 1, q_{\text{rej}}, 1, R)\). That is, when the machine is in state \(q_3\) and reads the input symbol 1, it goes to state \(q_{\text{rej}}\) and the head moves right.
What this Turing machine does is to accept strings of the form \( w \# w \) where \( w \in \{0, 1\}^* \). The basic idea is to compare the symbols in the corresponding cells on the two sides of the \( \# \) and determine whether they match. To keep track of which symbols have been checked, the machine replaces each symbol already examined by the mark \( c \). If all the symbols except for the \( \# \) have been changed into \( c \), which means that the two sides of the \( \# \) are exactly the same, then the machine goes into the accept state. If it finds a mismatch, it will enter a reject state.

**Exercise 5.1** Verify that the Turing machine in Example 5.1 indeed accepts strings of the form \( w \# w \).

There are many variants of the Turing machine model. For example, in Definition 5.1 the transition function requires the head to move to the left or right after each step. It makes sense to allow the head the ability to stay in the same position. We could also give a Turing machine multiple tapes instead of one, with each tape having its own head for reading and writing. Another variant is to add nondeterminism. In a certain state, a nondeterministic Turing machine reads an input symbol and then may proceed according to several possibilities because several transition rules are applicable. Interestingly, it turns out that the Turing machine model is very robust because all those reasonable variants have the same expressive power as the original model — they recognize the same set of strings.

**Exercise 5.2** Consider multitape Turing machines and nondeterministic Turing machines. Why can they be simulated by single-tape deterministic Turing machines?

Probably the most striking result about Turing machines is the existence of universal Turing machines (UTM). When started on a tape with a description encoding another Turing machine, say \( M \), followed by the input to \( M \), a UTM is capable of simulating the computation of \( M \) and producing the same result as \( M \) would when started on that input. A UTM can be viewed as a programmable computer. When given a program (a description of another machine), it behaves as if it were that machine while processing the input.

**Exercise 5.3** How to encode the specification of a Turing machine so that it can be understood by a UTM?

A computational system is called Turing-complete if it can simulate a UTM. For example, the lambda calculus and all general-purpose languages in wide use such as C and Java are Turing complete.

### 5.2 The Halting Problem

For a simple program like

\[
\textbf{while (true) do skip} \quad (2)
\]

it is easy to see that an execution of this program will never halt. However, if a complex program is given, we are not able to quickly answer whether it will halt. Suppose we are looking for three positive integers \( a, b \) and \( c \) satisfying the equation

\[
a^3 + b^3 = c^3. \quad (3)
\]

We might try every possible triple \( (a, b, c) \) with \( a + b + c = 3, a + b + c = 4, \ldots \), one by one in order, and stop when a solution is found. Writing a computer program to implement this procedure is easy. Then we would like to run the program and go for a cup of tea before returning back to check the outcome of the program. However, if the program has not stopped, it is unknown whether the
program will eventually halt but we have not waited long enough or it will run forever. In fact, by
Fermat’s Last Theorem, which took more than three centuries for mathematicians to work out a
proof, we now know that the equation (3) has no solution. Therefore, our program attempting to
solve it will never halt.

The Halting problem is the problem of determining, from a description of an arbitrary computer
program and an input, whether the program will eventually stop running or continue to run forever.
In 1936, Turing proved that there exists no general algorithm to solve the halting problem.

**Theorem 5.1** There is no algorithm to solve the halting problem.

**Proof:** We give a proof using the Turing machine model. First note that it is easy to design a Turing
machine $L$ that, like the program in (2), simply loops forever. Let us write $M \downarrow w$ if running $M$ on
$w$ halts, otherwise we write $M \not\downarrow w$. Suppose for a contradiction that some Turing machine
$H$ exists that can decide either $M \downarrow w$ or $M \not\downarrow w$, for any Turing machine $M$ and input $w$. The running of $H$
will result in either the accept state or the reject state, corresponding to the two cases above. The
input string of $H$ is an encoding of the description of $M$ and $w$, written $\langle M, \langle M \rangle \rangle$.

Now we construct a new Turing machine $D$ with $H$ and $L$ as subroutines. The following is a
description of $D$.

\[
D = \text{"On input } \langle M \rangle, \text{ where } M \text{ is a Turing machine:} \\
1. \text{Run } H \text{ on input } \langle M, \langle M \rangle \rangle. \\
2. \text{If } H \text{ is in its accept state then run } L. \\
   \text{If } H \text{ is in its reject state then does nothing."}
\]

We see that $D$ halts iff $H$ does not accept $\langle M, \langle M \rangle \rangle$. More specifically,

\[
D \downarrow \langle M \rangle \iff H \text{ does not accept } \langle M, \langle M \rangle \rangle \iff M \not\downarrow \langle M \rangle.
\]

Now let us run $D$ with its own description $\langle D \rangle$ as input. In that case we have

\[
D \downarrow \langle D \rangle \iff D \not\downarrow \langle D \rangle
\]

which is a contradiction. Thus our assumption of the existence of $H$ must be wrong. \qed

The proof is analogous to Cantor’s diagonal argument for showing the uncountability of the set
of all infinite sequences of binary digits. Suppose we make a table listing all Turing machines down
the rows, $M_1, M_2, \ldots$ and all their descriptions across the columns $\langle M_1 \rangle, \langle M_2 \rangle, \ldots$, as in Figure 2.
The Turing machine $H$ can tell if $M_i$ halts on input $M_j$. Therefore, for the entry at row $i$ and
column $j$ we put $\downarrow$ if $M_i \downarrow \langle M_j \rangle$ and $\not\downarrow$ if $M_i \not\downarrow \langle M_j \rangle$. By our assumption, $H$ is a Turing machine
and so is $D$. Therefore, it must occur on the list $M_1, M_2, \ldots$. Let $D = M_k$ for some $k \geq 1$. What
$D$ does is to compute the opposite of the diagonal entries. The contradiction occurs when we try to
fill in the entry at row $k$ and column $k$, where neither $\downarrow$ nor $\not\downarrow$ can be filled in because $M_k \downarrow \langle M_k \rangle$ iff
$M_k \not\downarrow \langle M_k \rangle$.

### 6 P and NP Problems

To estimate the running time of an algorithm, we often use asymptotic analysis, which considers the
running time of an algorithm when it is run on large inputs. Let us first introduce the asymptotic
notation or big-O notation.

**Definition 6.1** Let $f$ and $g$ be functions $f, g : \mathbb{N} \to \mathbb{R}^+$. We write $f(n) = O(g(n))$ if there are $c$
and $N$ such that for every integer $n \geq N$,

\[
f(n) \leq cg(n).
\]
Intuitively, \( f(n) = O(g(n)) \) means that \( f \) is less than or equal to \( g \) if we ignore differences up to constant factor. For example, if \( f(n) = 7n^8 + 5n^6 + 344n^2 + 2089 \), then we write \( f(n) = O(n^8) \).

**Definition 6.2** The class \( P \) consists of languages that are decidable in polynomial time on a deterministic single-tape Turing machine. The class \( NP \) consists of languages that are decidable by some nondeterministic polynomial time Turing machine.

**Theorem 6.1** Let \( t(n) \) be a function, where \( t(n) \geq n \). Then every \( t(n) \) time nondeterministic single-tape Turing machine has an equivalent \( 2^{O(t(n))} \) time deterministic single-tape Turing machine.

The question of whether \( P = NP \) is one of the greatest unsolved problems in mathematics and theoretical computer science.

The following problems are in P:

1. Given two natural numbers \( x \) and \( y \), calculate their greatest common divisor.
2. The primality testing problem is to determine whether a given number is prime.
3. Given a directed graph and two specific nodes \( s, t \), determine whether there exists a directed path from \( s \) to \( t \).
4. Given a weighted graph, find the shortest paths between nodes.
5. The Chinese postman problem is to find a shortest closed path that visits every edge of an undirected graph. The problem was originally studied by the Chinese mathematician Meigu Guan in 1960.
6. Maximize a linear function subject to linear inequality constraints (linear programming).

The following problems are in NP:

1. Determine if a natural number is composite, i.e., it is the product of two numbers greater than 1.
2. A Hamiltonian path in a directed graph \( G \) is a directed path that goes through each node exactly once. Determine if \( G \) contains a Hamiltonian path from node \( s \) to node \( t \) (HAMPATH).
3. A \( k \)-clique in an undirected graph is a subgraph with \( k \) nodes, wherein every two nodes are connected by an edge. Given a graph \( G \) and a specified size \( k \), determine whether \( G \) contains a \( k \)-clique (CLIQUE).
4. An independent set of a graph is a set of nodes such that every edge in the graph has at most one endpoint in the set. Given an undirected graph and a number \( k \), the independent set problem asks whether the graph contains an independent set of size \( k \) (INDEPENDENT-SET).
A *vertex cover* of an undirected graph $G$ is a subset of nodes where every edge of $G$ touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover with $k$ nodes, where $k$ is a specified size (VERTEX-COVER).

A subset of the nodes of a graph $G$ is a *dominating set* if every other node of $G$ is adjacent to some node in the subset. The dominating set problem asks whether a graph has a dominating set with $k$ nodes, where $k$ is a specified size (DOMINATING-SET).

Given two graphs $G$ and $H$, determine whether $G$ contains a subgraph that is isomorphic to $H$ (SUBGRAPH-ISO).

Given a collection of natural numbers $x_1, ..., x_k$ and a target number $t$, determine whether the collection contains a subcollection that adds up to $t$; that is, for some $\{y_1, ..., y_l\} \subseteq \{x_1, ..., x_k\}$ we have $\sum y_i = t$ (SUBSET-SUM).

Given a set of items, each with a weight and a value, the *knapsack problem* whether there exists a collection of items so that a total value of at least $v$ is achieved without exceeding the total weight $w$ (KNAPSACK).

A *coloring* of a graph is an assignment of colors to its nodes so that no two adjacent nodes are assigned the same color. Determine if a graph has a coloring with three colors (3COLOR).

In the early 1970s Stephen Cook and Leonid Levin discovered some problems in NP whose individual complexity is related to that of the entire class. If a polynomial time algorithm exists for any of these problems, all problems in NP would be polynomial time solvable. These problems are called *NP-complete*.

Suppose we have some boolean variables that can take values 0 or 1. The boolean operations $\land$, $\lor$ and $\neg$ are defined as follows:

\[
\begin{align*}
0 \land 0 &= 0 & 0 \lor 0 &= 0 & \neg 0 &= 1 \\
0 \land 1 &= 0 & 0 \lor 1 &= 1 & \neg 1 &= 0 \\
1 \land 0 &= 0 & 1 \lor 0 &= 1 \\
1 \land 1 &= 1 & 1 \lor 1 &= 1
\end{align*}
\]

A *Boolean formula* is an expression involving Boolean variables and operations, e.g.

\[
(\neg x_1 \land x_2 \land x_3) \lor (x_3 \land \neg x_4 \land x_5).
\]

It is satisfiable if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The *satisfiability problem* (SAT) is to test whether a Boolean formula is satisfiable.

**Theorem 6.2 (Cook-Levin)** SAT is NP-complete.

The problems HAMPATH, CLIQUE, INDEPENDENT-SET, VERTEX-COVER, DOMINATING-SET, SUBGRAPH-ISO, SUBSET-SUM, KNAPSACK, and 3COLOR are all known to be NP-complete.

Some problems are believed to sit between P and NP-complete. A famous example is factoring. Many cryptographic protocols are based on the difficulty of factoring large composite integers. Neither the existence nor non-existence of polynomial time algorithms for factoring has been proved. The problem is clearly in class NP but has not been proved to be in, or not in, NP-complete.

A related concept is *NP-hardness*, which refers to a class of problems that are, informally, at least as hard as the hardest problem in NP. More precisely, a problem is *NP-hard* if all problems in NP are polynomial time reducible to it, even though it may not be in NP itself. Figure 3 shows the relationship between the four classes P, NP, NP-Hard, and NP-Complete.
The traveling salesperson problem asks the following question: “given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the original city?” It is an NP-hard problem.

**Exercise 6.1** Give two examples of problems that are in P and two in NP.


7 Randomized Algorithms

A randomized algorithm receives, in addition to its input data, a stream of random bits that it can use for the purpose of making random choices. Even for a fixed input, different runs of a randomized algorithm may generate different outcomes. Randomization has been recognized as an extremely useful tool for constructing algorithms. It finds applications in a wide range of areas, including number theory, algebra, graph theory, pattern matching, selection, sorting, searching, computational geometry, combinatorial enumeration, and parallel and distributed computation [7]. Very often the execution time or space requirement of a randomized algorithm is smaller than that of the best deterministic algorithm for the same problem.

Let us take a look at the famous sorting algorithm Quicksort. It uses random partitioning, which is an important tool for the construction of randomized divide-and-conquer algorithms. To sort a set of elements $X = \{x_1, x_2, ..., x_n\}$, the algorithm proceeds as follows:

**Algorithm 2: Quicksort**

<table>
<thead>
<tr>
<th>input :</th>
<th>A set of elements $S$</th>
</tr>
</thead>
<tbody>
<tr>
<td>output:</td>
<td>Set $S$ sorted</td>
</tr>
<tr>
<td>begin</td>
<td>Draw an element $r$ at random from the set $S$;</td>
</tr>
<tr>
<td></td>
<td>Compare each element with $r$ so as to form two sets: $S_1 = {s \in S \mid s &lt; r}$ and $S_2 = {s \in S \mid s &gt; r}$;</td>
</tr>
<tr>
<td></td>
<td>Recursively, sort $S_1$ and $S_2$.</td>
</tr>
<tr>
<td>end</td>
<td></td>
</tr>
</tbody>
</table>
Ideally, we would like the element \( r \), often called the splitter, to divide \( X \) into two sets of equal size. The random choice of a splitter does not guarantee such a division. But it can be shown that the expected execution time of the above quicksort with random partition is \( 2n \ln n + O(n) \). This performance is close to the lower bound \( n \log_2 n \) for the number of comparisons needed to sort \( n \) elements.

This is an example of a Las Vegas algorithm, which is a randomized algorithm that always gives correct results. In other words, a Las Vegas algorithm does not gamble with the correctness of the result, but the resources used for the computation. In contrast, a Monte Carlo algorithm is a randomized algorithm whose output may be incorrect with a small probability. It typically applies in situations where the algorithm makes a decision or a classification, and its output is either yes or no. If the answer is yes, then it confirms with probability larger than \( \frac{1}{2} \), but if the answer is no, the algorithm will never give a definitive result; however, its failure to give a yes in a long series of trials gives strong evidence that the correct answer is no. Let us consider an example of testing polynomial equality.

Suppose we have two complicated polynomials of degree \( d \) and want to test whether they are equal or not.

\[
(3 - x)^7(x^3 - 5)^{19} - (x - 9)^{78} \equiv (x - 2)^6(x^2 + 3x)^{460}
\]

A naive way is to expand both polynomials and check whether the coefficients match, which could take exponential time. Alternatively, we might do the following: pick random values for the argument \( x \), plug in the values for the variables, and evaluate the two polynomials. If we obtain two different values, we would be able to conclude that the polynomials are not equal. If we repeat the procedure and keep getting the same values, we would not be able to conclude that the polynomials are equal, but our confidence in it would increase with each round of repetition. Here it is important to know how many different values of \( x \) can make the two polynomials evaluate to the same result. The answer is \( d \) according to the fundamental theorem of algebra — a non-constant polynomial of degree \( d \) has at most \( d \) roots. Therefore, if two polynomials are different, then with high probability they will differ at any random point where we choose to evaluate them, as long as \( x \) is chosen from a field much larger than the degree \( d \).

Let us consider another example, which is about testing equality of long strings. Suppose Alice and Bob live far away from each other, but they can communicate over a reliable but costly communication channel. Suppose Alice has a very long string of bits \( x \), Bob has a long string \( y \), and they want to know whether \( x = y \). A straightforward way for testing this equality would be for Alice to send \( x \) across the channel to Bob. Then Bob compares \( x \) and \( y \) locally and let Alice know whether they are equal. But this solution is very expensive in terms of the use of the communication channel. An alternative solution would be for Alice to derive from \( x \) a much shorter string that could serve as a “fingerprint” of \( x \); this short string could be obtained by applying some standard hashing or checksum technique. Alice then sends the fingerprint to Bob, who determines whether the fingerprints of \( x \) and \( y \) are equal. In case of inequality, Bob notifies Alice that \( x \neq y \). Otherwise, Bob would assume that \( x = y \) and notify Alice. In this solution, much less data is transmitted across the channel, but there exists the possibility of obtaining a false match, in which \( x \) and \( y \) are not the same but their fingerprints are the same.

For the above idea to work, Alice and Bob need to agree on a fingerprinting function. For example, they could choose a prime \( p \) and use the modulo operation

\[
H_p(x) = H(x) \mod p
\]

where \( H(x) \) is the integer represented by the bit string \( x \). If \( p \) is not too large, the fingerprint \( H_p(x) \) will be a short string of bits. Note that it is a good idea for \( p \) to be determined by randomization every time the equality of two strings is to be checked. Otherwise, if \( p \) is held fixed, there are certain pairs of \( x \) and \( y \) that always give the same fingerprint.
Algorithm 3: Testing equality of long strings

\begin{algorithm}
begin
    Alice draws \( p \) at random from the set of primes less than a certain value \( M \);
    Alice sends \( p \) and \( H_p(x) \) to Bob;
    Bob checks whether \( H_p(x) = H_p(y) \) and confirms the equality or inequality of the strings \( x \) and \( y \).
end
\end{algorithm}

If the prime \( p \) is chosen from a suitably large range, then for any strings \( x \) and \( y \) the probability that the algorithm will fail is extremely small; moreover, the probability can be further reduced by repeating the algorithm with several independent random choices of \( p \). To be precise, let \( \pi(N) \) be the number of primes less than \( N \). In number theory, it is a known fact that \( \pi(N) \) is asymptotic to \( N/\ln N \). It is also known that, if \( m < 2^n \), then the number of primes that divide \( m \) is less than \( \pi(n) \), except when \( n \) is very small. Now let us analyze the probability that the above algorithm will fail. A failure occurs when there is a false match, i.e. \( x \neq y \) but \( H_p(x) = H_p(y) \). This means that \( p \) divides \( |H(x) - H(y)| \), an integer less than \( 2^n \), where \( n \) is the length of the long strings. Hence

\[ Pr[\text{failure}] = \frac{|\{ p : p < M, p \text{ is prime, } p \text{ divides } |H(x) - H(y)|\}|}{\pi(M)} \leq \frac{\pi(n)}{\pi(M)}. \]

If the strings \( x \) and \( y \) are 100,000 bits long and the fingerprint has 32 bits, then substituting \( n = 100,000 \), \( M = 2^{32} \) into the above formula yields the probability of failure less than \( 10^{-4} \). Therefore, the algorithm is practically effective.

8 Formal Methods

In this section, we first give a brief introduction to the concept of formal specification, and then we give two examples of formal methods that could be used for formally specifying concurrent systems.

8.1 Formal Specification

As computer systems are being used increasingly in safety-critical applications, i.e., systems where a failure could result in the loss of human life, mass destruction of property, or significant financial loss, computer scientists have considered formal methods and their role in the specification and design phases of system development. In developing complex software systems, it is dangerous for a development team to undertake a project without stating clearly and precisely what is required of the system. This is called requirements specification in the software life cycle, with the aim to describe what the system is to do. To avoid ambiguity resulted from requirements specification in natural languages, formal methods have been introduced. They employ discrete mathematics to describe the function and architecture of a hardware or software system, and various forms of logic to reason about requirements, their interactions, and validity.

While programming languages are formal languages, they are generally not used in formal specification, as most languages do not have a full formal semantics, and they force us to address implementation issues before we have a clear description of what the system can do. Instead, we use the language of mathematics, which is universally understood and well established in notation. The key to the success of formal specification is to abstract away from the details of the implementation and consider only the essential relationships of the data, and we can model even the most complex systems using simple mathematical objects such as sets, relations, functions, etc. Below are some reasons why a mathematical approach may be beneficial in producing a specification:
• **Precision**: Natural language and diagrams can be very ambiguous. A mathematical notation allows the specifier to be very exact about what is specified. It also allows the reader of a specification to identify properties and problems.

• **Conciseness**: A formal specification is also very concise compared with an equivalent high-level language program. Such a specification can be an order of magnitude smaller than the program that implements it, and hence is that much easier to comprehend.

• **Abstraction**: A formal notation allows the writer to concentrate on the essential features of a system, ignoring those that are implementation details.

• **Reasoning**: Once a formal specification is available, mathematical reasoning is possible to aid in its validation.

Once a formal specification is done, it must be translated to a design, i.e., a clear plan for implementation of the system specification, and eventually into its equivalent in a programming language. This approach is known as refinement.

Roughly speaking, formal specification languages can be divided into three classes:

1. **Model-oriented approaches** as exemplified by ASM (Abstract State Machines), B-Method, RAISE (Rigorous Approach to Industrial Software Engineering), VDM (Vienna Development Method) and the Z notation. These approaches involve the derivation of an explicit model of the systems desired behavior in terms of abstract mathematical objects.

2. **Property-oriented approaches** using axiomatic semantics, which use first-order predicate logic to express preconditions and postconditions of operations over abstract data types, and algebraic semantics (such as the OBJ family of languages including CafeOBJ), which are based on multisorted algebras and relate properties of the system in question to equations over the entities of the system.

3. **Process algebras** such as CSP (Communicating Sequential Processes) and CCS (Calculus of Communicating Systems), which have evolved to meet the needs of concurrent, distributed, and real-time systems, and which describe the behavior of such systems by describing their algebras of communicating processes.

A branch of formal methods known as model checking allows systems to be tested exhaustively. Most computer-based systems are far too complicated to test completely because the number of ways the system could be used is far too large. However, a number of techniques, Binary Decision Diagrams (BDDs), for example, allow relatively efficient checking of significant systems, especially for hardware. Mechanical tools exist to handle BDDs and other model-checking approaches efficiently. SPIN is one of the leading general model-checking tools that is widely used.

**Exercise 8.1** Read the excellent survey of formal methods by Hinchey et al. [6], from which the above summary about formal specification is extracted.

### 8.2 Process Algebras

Process algebras generally are made up of the following: a language for describing systems, a behavioral equivalence or inequivalence that allows comparison of system behaviors, and axioms that allow for proofs of equivalence between systems. Some algebras include a refinement ordering to determine whether one system is a refinement of another. Process algebras usually use a handshake mechanism (rendezvous) between processes via a port or channel. One process may wait for data at a channel until another process sends data over the channel (or vice versa). Once the data is
exchanged, both processes continue. The receiving process then may execute different processes based on the data received. Process algebras are often used to determine if deadlock and livelock exist in communicating systems. Internal, noncommunications aspects of processes are not reflected by process algebras.

An an example, let us briefly introduce the Calculus of Communicating Systems (CCS) developed by Robin Milner for reasoning about concurrent systems. Assume a countably infinite collection $A$ of channel names. The set $\bar{A} = \{\bar{a} \mid a \in A\}$ is the set of complementary names (co-names for short). Let $L = A \cup \bar{A}$ be the set of labels, and $Act = A \cup \{\tau\}$ be the set of actions. By convention, we assume that $\bar{\bar{a}} = a$ for each label $a$.

**Definition 8.1** The collection of CCS expressions is given by the following grammar:

$$P, Q ::= K \mid \alpha.P \mid \sum_{i \in I} P_i \mid P|Q \mid P[f] \mid P\setminus L,$$

where $K$ is a constant in $\mathcal{K}$, $\alpha$ is an action in $Act$, $I$ is an index set, $L$ is a set of labels, and $f : Act \rightarrow Act$ is a relabeling function such that $f(\tau) = \tau$ and $f(\bar{a}) = f(a)$ for each label $a$.

We write $0$ for an empty sum of processes, i.e., $0 = \sum_{i \in \emptyset} P_i$ and $P_1 + P_2$ for a sum of two processes. Moreover, the behaviour of each process constant is given by a defining equation

$$K \overset{\text{def}}{=} P.$$

An example of a CCS expression is given by the counter defined below:

$$\text{Counter}_0 \overset{\text{def}}{=} \text{up.}\text{Counter}_1 \quad \text{Counter}_n \overset{\text{def}}{=} \text{up.}\text{Counter}_{n+1} + \text{down.}\text{Counter}_{n-1} \quad (n > 0).$$

For each $n \geq 0$, the process $\text{Counter}_n$ behaves like a counter whose value is $n$: the ‘up’ action increases the value of the counter by one, and the ‘down’ action decreases it by one. The operational behaviour of a process $P$ is given by transitions in the form $P \xrightarrow{\alpha} Q$ derived by the rules in Table 2.

For the counter defined above, we have the following sequence of transitions

$$\text{Counter}_0 \xrightarrow{\text{up}} \text{Counter}_1 \xrightarrow{\text{up}} \text{Counter}_2 \xrightarrow{\text{up}} \cdots$$

**Exercise 8.2** Draw a complete transition graph of the process $\text{Counter}_0$.

CCS can be used to represent systems at different levels of abstraction to give a high-level overview of a system as well as a detailed view. With its refinement ability, CCS can also maintain equivalences between higher level and lower level specifications. In addition, there is the ability to test that two specifications are bisimulations of each other. CCS also allows for proof of correctness as well as deadlock and livelock.
Definition 8.2 A binary relation \( R \) between processes is a bisimulation iff whenever \( s_1 R s_2 \) and \( \alpha \) is an action:

- if \( s_1 \xrightarrow{\alpha} s'_1 \), then there is a transition \( s_2 \xrightarrow{\alpha} s'_2 \) such that \( s'_1 R s'_2 \);
- if \( s_2 \xrightarrow{\alpha} s'_2 \), then there is a transition \( s_1 \xrightarrow{\alpha} s'_1 \) such that \( s'_1 R s'_2 \);

Two processes are bisimilar if there is a bisimulation that relates them.

Exercise 8.3 Consider the processes

\[ P \overset{\text{def}}{=} a.(b.0 + c.0) \quad \text{and} \quad Q \overset{\text{def}}{=} a.b.0 + a.c.0. \]

Show that \( P \) and \( Q \) are not bisimilar.

Exercise 8.4 Read the textbook [2], which gives a gentle introduction to concurrency theory.

8.3 Temporal Logic

Temporal logic is used to express time-related aspects of systems. There are both modal and predicate approaches to temporal logic. In temporal logic, an expression may be true now or will be true at some time in the future. There are two types of semantic models used: timed specifications based on linear time and branching time. With linear time a specification is a set of linear states with each state being part of a possible execution sequence (used in CSP traces). With branching time, states are a specification described as a tree structure of states, with each path in the tree a possible execution sequence (used in CCS).

Temporal logic has gone through several modifications by different people for application to different fields. It is widely used in model checking. The idea of temporal logic has also been added to other formal methods to give them the basis of time in those methods. A wide variety of temporal logics have been developed. The differences in the different temporal logics range from expressiveness, availability of support tools for executability, and verifiability.

Below we briefly introduce linear temporal logic (abbreviated LTL). It is often used to specify properties of interleaving sequences, modeling the execution of a program. It can be defined on top of propositional logic. The syntax of LTL is as follows:

1. Propositions are LTL formulas;
2. If \( \phi \) and \( \psi \) are formulas, then so are \( \neg \phi \), \( \phi \land \psi \), \( \phi \lor \psi \), \( \Box \phi \), \( \Diamond \phi \), \( \square \phi \) and \( \phi \cup \psi \).

An LTL formula is interpreted over an infinite sequence of states \( s_0 s_1 s_2 \cdots \). We often write \( \xi^k \) for the suffix of \( \xi = s_0 s_1 s_2 \cdots \) starting at \( s_k \), i.e., the sequence \( s_k s_{k+1} s_{k+2} \cdots \). It is convenient to define the semantics of LTL for any arbitrary suffix \( \xi^k \) of a sequence \( \xi \) in the following way:

- \( \xi^k \models \alpha \), where \( \alpha \) is a proposition, exactly when \( s_k \models \alpha \) in propositional logic;
- \( \xi^k \models \neg \phi \) when not \( \xi^k \models \phi \);
- \( \xi^k \models \phi \land \psi \) when \( \xi^k \models \phi \) and \( \xi^k \models \psi \);
- \( \xi^k \models \phi \lor \psi \) when \( \xi^k \models \phi \) or \( \xi^k \models \psi \);
- \( \xi^k \models \Box \phi \) when \( \xi^{k+1} \models \phi \);
- \( \xi^k \models \Diamond \phi \) when \( \xi^{k+1} \models \phi \);
\begin{itemize}
  \item $\xi^k \models \Diamond \phi$ when there is an $i \geq k$ with $\xi^i \models \phi$;
  \item $\xi^k \models \Box \phi$ when for every $i \geq k$, $\xi^i \models \phi$;
  \item $\xi^k \models \phi \cup \psi$ when there is an $i \geq k$ such that $\xi^i \models \psi$ and for all $j$, where $k \leq j < i$, $\xi^j \models \phi$.
\end{itemize}

The first line states that a proposition, i.e., an LTL formula without modal operators, is interpreted in the first state of the sequence $\xi^k$. The next three lines give the usual interpretation of Boolean operators. The rest of the definition define the meaning of modal operators.

The operator $\Diamond$ is called nexttime. The formula $\Diamond \phi$ holds in a sequence $s_k s_{k+1} \cdots$ when $\phi$ holds starting with the next state $s_{k+1}$. The modal operator $\Box$ is called eventually. The formula $\Box \phi$ holds in a sequence $\xi$ provided that there is a suffix of $\xi$ where $\phi$ holds. The modal operator $\Box$ is called always. The formula $\Box \phi$ holds in a sequence $\xi$ provided that $\phi$ holds in every suffix of $\xi$.

The operator $\cup$ is called until. Intuitively, $\phi \cup \psi$ says that $\phi$ holds until some point where $\psi$ holds.

We write $\xi \models \phi$ when $\xi^0 \models \phi$, that is, the complete sequence $\xi$ satisfies $\phi$.

Exercise 8.5 Read the textbook [5] to learn the basic ideas of model checking.

9 Cryptography

Cryptography is a black art that dates back thousands of years. A more recent example is the breaking of German code in World War II by a group of British mathematicians led by Alan Turing. In the last few decades the development of theoretical computer science has led to a revolution in the design of secret codes.

9.1 Basic Concepts

In the traditional way of communication, if Alice sends an encrypted message to Bob, then Bob has to know the secret key to decrypt it. The secret key is the piece of information used by the encrypting and decrypting algorithms. A short key may be discovered by a brute-force search of the entire space of possible keys. The only way to get perfect cryptographic security is using keys as long as the combined length of all messages sent, called one-time pad. Basically, every bit of a one-time pad is used just once to encrypt a bit of the message, and then discarded, which would not be practical if the message is very long. Even if cryptographic keys are short, their management is not easy. For example, every pair of parties that desires private communication needs to establish a joint secret key. The traditional communication uses a private-key cryptosystem, where the same key is used for both encryption and decryption.

Nowadays, public-key cryptosystems are widely used. There the decryption key is different from, and not easily computed from, the encryption key. Each individual only needs to establish a single pair of keys: an encryption key $E$ and a decryption key $D$. She keeps $D$ secret but publicizes $E$. Anyone who wants to send her a message can look up $E$ in the public directory, encrypt the message with it, and sends it to her. The first individual is the only person who knows $D$ and can decrypt that message.

One underpinning of the modern theory of cryptography is one-way function.

Definition 9.1 A one-way function is a function $f : \{0,1\}^n \to \{0,1\}^{p(n)}$ such that

\begin{enumerate}
  \item $f$ is computable in polynomial time;
  \item For all polynomial-time algorithms $A$,
    \[Pr_{x \in \{0,1\}^n}[f(A(f(x))) = f(x)]\]
\end{enumerate}
In other words, a polynomial-time algorithm should only be able to invert \( f \) with negligible probability. Here we do not use the requirement \( \mathbb{A}(f(x)) = x \) in order to rule out trivial functions like \( f(x) = 0 \).

The multiplication function \( \text{mult} \) is a candidate for a one-way function. For any \( w \in \{0,1\}^n \), let \( \text{mult}(w) \) be the string representing the product of the first and second halves of \( w \). That is, \( \text{mult}(w) = w_1 \cdot w_2 \), where \( w = w_1 w_2 \) such that \( |w_1| = |w_2| \), or \( |w_1| = |w_2| + 1 \) if \( |w| \) is odd. Despite a great amount of research into the integer factorization problem, no polynomial algorithm is known that can invert \( \text{mult} \).

9.2 RSA

RSA is one of the practical public-key cryptosystems widely used for secure data transmission. It is made of the initial letters of the surnames of Ron Rivest, Adi Shamir, and Leonard Adleman, who publicly described the algorithm in 1977.

Let us consider the simplest variant of RSA. Suppose you need to send your credit card number to Amazon.com. The following events could happen.

(1) Amazon first picks two large prime numbers, \( p \) and \( q \), with the condition that neither \( p - 1 \) nor \( q - 1 \) is divisible by 3. It then multiplies them together to obtain \( N = pq \) and sends it to you.

(2) On retrieving \( N \), you calculate \( y = x^3 \mod N \), where \( x \) is your credit card number, and send \( y \) back to Amazon.

(3) Amazon finds an integer \( k \) such that \( 3^k \equiv 1 \pmod{(p-1)(q-1)} \) and recover \( x \) from \( y \) as follows:

\[
y^k \mod N = x^{3k} \mod N = x \mod N. \tag{4}
\]

In the last step, Euler’s theorem discovered in the 1700s is used. It states that if \( a \) and \( n \) are relatively prime positive integers, then

\[
a^{\varphi(n)} \equiv 1 \pmod{n}
\]

where \( \varphi(n) \) is Euler’s totient function that gives the number of all integers from 1 to \( n \) that are relatively prime to \( n \). If two numbers \( m \) and \( n \) are relatively prime, then \( \varphi(mn) = \varphi(m)\varphi(n) \). In our example above, we have \( \varphi(N) = \varphi(p)\varphi(q) = (p-1)(q-1) \). It follows that

\[
x^{(p-1)(q-1)} \equiv 1 \pmod{N} \tag{5}
\]

The equality in (5) implies that, if Amazon can find an integer \( k \) with \( 3k = 1 \mod (p-1)(q-1) \) then (4) holds because

\[
x^{3k} = x^{c(p-1)(q-1)+1} = x \mod N,
\]

where \( c \) is some integer. The fact that neither \( p - 1 \) nor \( q - 1 \) is divisible by 3 implies that such an integer \( k \) must exist, and furthermore it can be found in polynomial time given \( p \) and \( q \), for example by using Euclid’s algorithm. Once Amazon has \( k \), it can compute \( y^k \mod N = x \) in polynomial time using repeated squaring, and thereby recover your credit card number \( x \). In this example, Amazon uses the public key \((N,3)\) and the private key \((N,k)\).

How secure is the system? Anyone who could factor \( N \) into \( pq \) could easily find the private key. Therefore, the security of the whole system is based on the intractability of factoring large integers.

\[1\text{If } x \text{ is not relatively prime to } N, \text{ we cannot use Euler’s theorem. This happens with a very low probability, but even in this case the equality in (4) still holds. See } \text{https://en.wikipedia.org/wiki/RSA_(cryptosystem)}. \]
10 Quantum Computation

In this section, we give a brief introduction to quantum computation. You are strongly suggested to read the famous book [9].

10.1 Quantum Bits

As we all know, a classical bit has a state, either 0 or 1. For quantum computation, the fundamental concept is quantum bit, or qubit for short. As expected, a qubit could be in the states $|0\rangle$ or $|1\rangle$. The notation $\langle | \rangle$ is called the Dirac notation widely used in the literature of quantum mechanics.

The main difference of qubits from bits is that a qubit can be in a superposition of states: $|\phi\rangle = \alpha|0\rangle + \beta|1\rangle$, which is a linear combination of states. Here $\alpha$ and $\beta$ are complex numbers. Therefore, the state of a qubit can be represented as a vector in a two-dimensional complex vector space. The special states $|0\rangle$ and $|1\rangle$ are the computational basis states. When we measure a qubit there are two possible results we can observe: 0 with probability $|\alpha|^2$, or $q$ with probability $|\beta|^2$, where the symbol $|\alpha|$ stands for the norm of the complex number $\alpha$. And the state of the qubit after the measurement becomes $|0\rangle$ or $|1\rangle$. For the probabilities to sum to one, we always have $|\alpha|^2 + |\beta|^2 = 1$. This is the normalization condition.

Now suppose we have two qubits. As one would expect, a two-qubit system has four computational basis states: $|00\rangle$, $|01\rangle$, $|10\rangle$, and $|11\rangle$. A pair of qubits can also exist in superpositions of these four states, and take the form $|\phi\rangle = \alpha_{00}|00\rangle + \alpha_{01}|01\rangle + \alpha_{10}|10\rangle + \alpha_{11}|11\rangle$.

The normalization condition requires $\sum_{x\in\{0,1\}^2} |\alpha_x|^2 = 1$, where the notation $\{0,1\}^2$ means the set of binary strings of length two. We could measure just a subset of the qubits, say the first qubit, and observe the result 0 with probability $|\alpha_{00}|^2 + |\alpha_{01}|^2$, leaving the post-measurement state $\frac{\alpha_{00}|00\rangle + \alpha_{01}|01\rangle}{\sqrt{|\alpha_{00}|^2 + |\alpha_{01}|^2}}$. 

A peculiar two-qubit state is the Bell state or EPR pair, after Einstein, Podolsky, and Rosen.

$$\sqrt{\frac{1}{2}} |00\rangle + |11\rangle$$

If we measure the first qubit, we obtain 0 with probability 1/2, leaving the post-measurement state $|00\rangle$, and 1 with probability 1/2, leaving the state $|11\rangle$. A measurement of the second qubit always gives the same result as the measurement of the first qubit. Therefore, the measurement outcomes are correlated. The two qubits in the Bell state are entangled.

More generally, we may have a system of $n$ qubits. The computational basis states of this system are of the form $|x_1x_2\cdots x_n\rangle$, and a quantum state of the system is given by $2^n$ complex numbers.

10.2 Quantum Circuits

The basic building blocks of classical circuits are logic gates such as NOT, AND, XOR etc. In the quantum setting, we have quantum gates. Consider the classical NOT gate, which changes the input data 0 to 1 and 1 to 0. The quantum NOT gate is analogous; it takes the state $\alpha|0\rangle + \beta|1\rangle$ to
\[ |A\rangle \quad \bullet \quad |A\rangle \]

\[ |B\rangle \quad \oplus \quad |B \oplus A\rangle \]

Figure 4: The CNOT gate

\[ \alpha |0\rangle + \beta |1\rangle \]. That is, the roles of \(|0\rangle\) and \(|1\rangle\) have been interchanged. A convenient way of representing this gate is to use the following matrix

\[
X \equiv \begin{bmatrix} 0 & 1 \\ 1 & 0 \end{bmatrix}
\]

If we write the quantum state \(\alpha |0\rangle + \beta |1\rangle\) as the vector

\[
\begin{bmatrix} \alpha \\ \beta \end{bmatrix},
\]

then the output from the quantum NOT gate is

\[
X \begin{bmatrix} \alpha \\ \beta \end{bmatrix} = \begin{bmatrix} \beta \\ \alpha \end{bmatrix}.
\]

Quantum gates on a single qubit can be represented by two by two matrix. To ensure that the output quantum state \(\alpha' |0\rangle + \beta' |1\rangle\) still satisfies the normalization condition, the matrix representation \(U\) of each quantum gate needs to be unitary, that is \(U^\dagger U = I\), where \(I\) is the identity matrix and \(U^\dagger\) is the adjoint of \(U\) obtained by transposing and then complex conjugating \(U\).

**Exercise 10.1** Besides the matrix \(X\), verify that the following matrices are also unitary.

\[
Y = \begin{bmatrix} 0 & -i \\ i & 0 \end{bmatrix}, \quad Z = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad H = \frac{1}{\sqrt{2}} \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}, \quad S = \begin{bmatrix} 1 & 0 \\ 0 & i \end{bmatrix}, \quad T = \begin{bmatrix} 1 & 0 \\ 0 & e^{i\pi/4} \end{bmatrix}
\]

The matrix \(H\) above represents the Hadamard gate, one of the most useful quantum gates. It turns \(|0\rangle\) into \((|0\rangle + |1\rangle)/2\) and turns \(|1\rangle\) into \((|0\rangle - |1\rangle)/2\).

We now consider a two-qubit quantum gate given in Figure 4, called the controled-NOT or CNOT gate. The input \(|A\rangle\) is the control qubit and \(|B\rangle\) is the target qubit. What the gate does is as follows. If the control qubit is 0 then the target qubit is left unchanged. If the control qubit is 1, then the target qubit is flipped. In other words,

\[
|00\rangle \rightarrow |00\rangle; \quad |01\rangle \rightarrow |01\rangle; \quad |10\rangle \rightarrow |11\rangle; \quad |11\rangle \rightarrow |10\rangle.
\]

More concisely, the action of the gate is \(|A, B\rangle \rightarrow |A, B \oplus A\rangle\), where \(\oplus\) is addition modulo two. We can also use the following matrix to describe it.

\[
U_{CN} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{bmatrix}
\]

**Exercise 10.2** Check that \(U_{CN}\) is a unitary matrix.
Figure 5: A quantum circuit to create Bell states

Figure 6: A quantum circuit for teleporting a qubit.

We might like to look for a quantum gate that can copy a quantum state. The action of the gate is as follows:

\[ \alpha |0\rangle + \beta |1\rangle \rightarrow (\alpha |0\rangle + \beta |1\rangle)(\alpha |0\rangle + \beta |1\rangle) = \alpha^2 |00\rangle + \alpha \beta |01\rangle + \alpha \beta |10\rangle + \beta^2 |11\rangle. \]

This operation is not possible because it is not linear. The final amplitudes \( \alpha^2, \beta^2 \) and \( \alpha \beta \) do not depend linearly on \( \alpha \) and \( \beta \). For an unknown quantum state, it is not possible to duplicate it. This is the no-cloning theorem in quantum mechanics.

Consider the circuit in Figure 5. It has a Hadamard gate followed by a CNOT. Its action is to put the top qubit in a superposition and let it be the control input of the CNOT.

\[
\begin{align*}
|00\rangle & \rightarrow \frac{|00\rangle + |11\rangle}{\sqrt{2}} \\
|01\rangle & \rightarrow \frac{|01\rangle + |10\rangle}{\sqrt{2}} \\
|10\rangle & \rightarrow \frac{|00\rangle - |11\rangle}{\sqrt{2}} \\
|11\rangle & \rightarrow \frac{|01\rangle - |10\rangle}{\sqrt{2}}
\end{align*}
\]

The four output states on the right are the Bell states.

10.3 Quantum Teleportation

Let us consider an example of quantum teleportation. Many years ago, Alice and Bob met and generated an EPR pair, each taking one qubit of the pair and then separated. Now they live far away from each other. Alice would like to send a qubit to Bob. The state of the qubit is unknown to Alice. Moreover, there is only a classical communication channel between Bob and her. How will she accomplish this mission?

Alice will use the solution described in Figure 6. She first interacts the unknown qubit, say \( |\psi\rangle \), with her half of the EPR pair, and measures the two qubits, obtaining one of four possible classical results, \( |00\rangle, |01\rangle, |10\rangle, \) and \( |11\rangle \). She sends this information to Bob over the classical channel. Upon
receiving the classical message, Bob performs one of four operations on his half of the EPR pair, and, magically, he can recover the original state $|\psi\rangle$.

We now examine the quantum circuit more closely. Suppose $\psi = \alpha|0\rangle + \beta|1\rangle$, where $\alpha$ and $\beta$ are unknown amplitudes. The input into the circuit is

$$
|\psi\rangle|\beta\rangle_0 = \frac{1}{\sqrt{2}}(\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|00\rangle + |11\rangle)),
$$

where the first two qubits are in the hands of Alice and the third qubit belongs to Bob. Alice sends her qubits to a CNOT gate, and then the quantum system of three qubits evolves into

$$
\frac{1}{\sqrt{2}}(\alpha|0\rangle(|00\rangle + |11\rangle) + \beta|1\rangle(|10\rangle + |01\rangle)),
$$

Shen then feeds the first qubit to Hadamard gate, obtaining

$$
\frac{1}{2}((0\rangle + |1\rangle)(|00\rangle + |11\rangle)) + \beta(|0\rangle - |1\rangle)(|10\rangle + |01\rangle)),
$$

which can be rewritten as

$$
\frac{1}{2}((0\rangle(\alpha|0\rangle + \beta|1\rangle) + |01\rangle(\alpha|1\rangle + \beta|0\rangle) + |10\rangle(\alpha|0\rangle - \beta|1\rangle) + |11\rangle(\alpha|1\rangle - \beta|0\rangle)).
$$

Depending on Alice’s measurement outcome, Bob’s qubit will be in one of four possible states. Using the measurement outcome received from Alice, Bob is able to manipulate his qubit so as to recover the original qubit $|\psi\rangle$. For example, if the message from Alice is 11 then Bob can apply first an $X$ and then a $Z$ gate, which change his qubit from $\alpha|1\rangle - \beta|0\rangle$ to $\alpha|0\rangle - \beta|1\rangle$ and then to $\alpha|0\rangle + \beta|1\rangle$.

Note that the teleportation did not allow faster than light communication because Alice’s measurement result was transmitted over a classical communication channel. The teleportation also did not violate the no-cloning theorem because after the teleportation the original qubit $|\psi\rangle$ was in one of the computational basis states $|0\rangle$ or $|1\rangle$.

### 10.4 Deutsch’s Algorithm

Suppose $f : \{0, 1\} \to \{0, 1\}$ is a function with a one-bit domain and range. It is constant if $f(0) = f(1)$. In order to determine whether $f$ is constant, how many evaluations of $f(x)$ is sufficient? At first sight, the answer seems to be two because we need to evaluate $f(0)$ and $f(1)$ and then compare them. Surprisingly, using a quantum circuit, we can solve the problem with only one evaluation!

Figure 7 implements Deutsch’s algorithm. Two input qubits are used. They are initialized as $|0\rangle$ and $|1\rangle$, respectively. When they pass through two Hadamard gates, we obtain

$$
|\psi_1\rangle = \frac{(|0\rangle + |1\rangle)\sqrt{2}(|0\rangle - |1\rangle)\sqrt{2}}{2}.
$$
Note that applying the gate $U_f$ to the state $|x⟩(|0⟩ - |1⟩)/2$ yields $(-1)^{f(x)}|x⟩(|0⟩ - |1⟩)/2$. If we apply $U_f$ to $|ψ_1⟩$, one of the following two possibilities will occur:

$$|ψ_2⟩ = \begin{cases} ±(\frac{|0⟩ + |1⟩}{\sqrt{2}})(\frac{|0⟩ - |1⟩}{\sqrt{2}}) & \text{if } f(0) = f(1) \\ ±(\frac{|0⟩ - |1⟩}{\sqrt{2}})(\frac{|0⟩ - |1⟩}{\sqrt{2}}) & \text{if } f(0) ≠ f(1). \end{cases}$$

When the first qubit goes through a Hadamard gate, we obtain

$$|ψ_3⟩ = \begin{cases} ±|0⟩(\frac{|0⟩ + |1⟩}{\sqrt{2}}) & \text{if } f(0) = f(1) \\ ±|1⟩(\frac{|0⟩ + |1⟩}{\sqrt{2}}) & \text{if } f(0) ≠ f(1). \end{cases}$$

Since $f(0) + f(1)$ is 0 if $f(0) = f(1)$, and 1 otherwise, we rewrite $|ψ_3⟩$ as

$$|ψ_3⟩ = ±|f(0) + f(1)⟩(\frac{|0⟩ - |1⟩}{\sqrt{2}}).$$

By measuring the first qubit we can determine $f(0) + f(1)$. Therefore, to determine the global property of $f(x)$, namely $f(0) + f(1)$, we have used only one evaluation of $f(x)$, which is more efficient than classical computation.

**Exercise 10.3** Consider a more general form of Deutsch’s problem. Let $f : \{0, 1\}^n → \{0, 1\}$ be an $n$-ary function. How many evaluations are needed to determine if $f(x)$ is constant for all values of $x$, or else $f(x)$ is balanced, i.e., equal to 1 for exactly half of all the possible $x$, and 0 the other half?

**References**


