Models of Sequential Computation

CHAPTER 13

Programming is often considered as an art, but it raises mostly practical questions like “which language should I choose”, “which algorithm is adequate”, and “what is an ergonomic user interface”. The more thoughtful programmer will pose general questions on computing like:

- Is there an algorithm for every problem?
- What is an “ideal” computer?
- How does a “minimal” computer look like?
- Can I proof the correctness of my program?

These are questions about theoretical Computer Science. It deals with the foundation of informatics: computational models, complexity, proofing the correctness of programs in particular, and not about hardware issues. The foundations of hardware are Physics and Electronics, but the foundation of informatics is Mathematics.

Computer Science imparts mostly practical knowledge but those interested in deeper insight should read this chapter.

Lets start with the question of computational models that are minimal in terms of hardware, but sufficient general to execute all possible algorithms. Does it make sense to think about such models?

Yes, we want to find out the principles common to every technical computer.

There exist a couple of minimal models and we will look at the most prominent one, the Turing machine, called after the British computer scientist Alan Turing (* June 23, 1912, † June 7, 1954).

13.1 Turing Machines

Turing studied in the beginning of the 1930^{th} a simple hypothetical machine that has only a very limited functionality. Many attempt have been made to find machines that can compute any algorithm, but those machines were found equivalent to the Turing machine in the sense that they had all the same

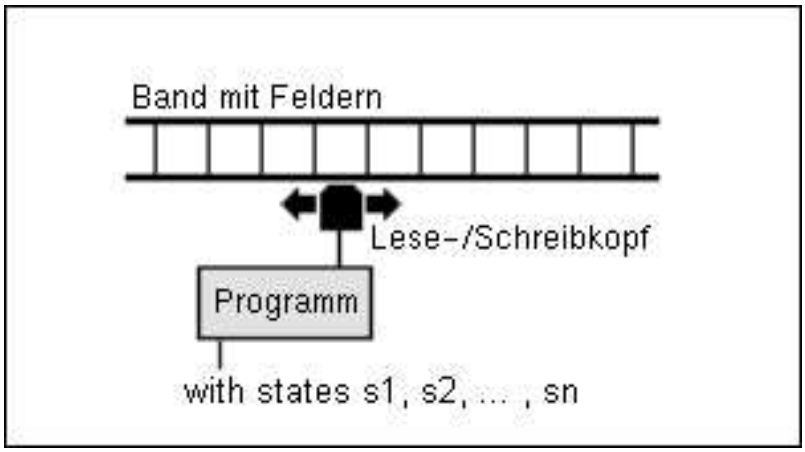

Figure 13.1 Turing machine with a *n*-state control unit, infinite storage band, and read/write-head

computing capabilities. This gave a strong indication for *Church's hypothesis* that "the Turing machine can execute every computable function" and underlined the importance of the Turing's research.

A *computable function* is a function for which an algorithm exists to compute the function value.

The Turing machine consist of only three elements:

- an unlimited storage "band" where each storage cell contain one value from a finite alphabet V or is empty (⊔)
- a read/write-head to read and write a data value from/to the "band"
- a control unit containing a program that operates the read/write head and changes its state.

Initially the band contains a finite number of cells with symbols (data) from the alphabet V. Outside the data the band is empty. The read/write-head is positioned on the first nonempty cell as in the Figure 13.1.

The value of the cell is read and depending on this value and the current state of the control unit a new (or the old) value is written into the cell, the state is updated, and then the head is moved one position left or right to the next cell. At the next position again, the cell's data is read, a new value is written, and the head moves to the next cell (left or right). This continues until the machine reaches it final (halt) state.

More formally, the program of the Turing machine is a function of two parameters, the current cell value and the current state. The output of the function is a new cell value, a head movement of one cell left or right, and a new state.

Lets make a little example: For simplicity we chose as alphabet the binary digits $\{0, 1\}$. Our program should produce an even parity for any data on the band. The program is given in form of a table where the rows are defined by the state of the control unit and the columns are the possible data values

state	0	1	⊔
even	(even,0,r)	(odd,1,r)	(halt,0,n)
odd	(odd,0,r)	(even,1,r)	(halt,1,n)

Figure 13.2 Transition function for a Turing machine for even parity generation

(symbols) on the band.

We will demonstrate the operation with the data 10101.

Our initial state is *even* and the reading starts at the first position with a value of 1. In the table of Figure 13.2 we find for the state *even* the action (odd,1,r) which means "leave the cell value unchanged, change the state to *odd* and move the head one cell to the right" At the new position we read 0 and which results together with the state *odd* the result is (odd,0,r). The next step yields (even,1,r), the following (even,0,r), and (odd,1,r). Now we are past the last data and read an empty cell. In our table we have (halt,1,n) in column ⊔ and row *odd*. We write a 1, set the state to *halt* and stop the program. The new data on the band has now an even number of 1s, four 1s to be exact.

It is left to the reader as an exercise to show that in the case of even number of 1s the parity bit will be 0.

It is not easy to believe that such a machine can execute any algorithm. We will not proof it here, but we will at least try to make it plausible.

First we limit our alphabet as before to the binary digits. Using dual code we can express any number or using ASCII-code we can code numbers and roman letters. This should be sufficient to formulate any algorithm.

Now we need to show that we can compute any formula. It is known from Mathematics that arithmetic operation can all all be expressed with the four basic calculating operations (addition, subtraction, multiplication, division). The derivation of a function f at point x for instance is computed as limit of the expression:

$$f'(x) := \lim_{\epsilon \to 0}(f(x+\epsilon) - f(x-\epsilon))/2 * \epsilon(\epsilon > 0)$$

if the limit exists.

This is not a proof but an example where a higher operation is expressed (here defined) by elementary operations.

Division can be reduced to multiplication of the reciprocal and multiplication is noting than repeated addition. Furthermore the addition of integer numbers like $3+2$ results from the incrementing the number 0 first 3 and then 2 times.

All we need to show is, that we can store the number 0 on the band and are able to increment a number.

To store 0 is very simple. When the Turing machine is started, the head is under the leftmost nonempty position. We write there the value 0 and change the initial state to *next* and move one cell right. We read the next value and if it is now empty (⊔) then we write empty on the band and move right again. This is repeated until the value read is empty where we set the state to *halt* and terminate the program.

The result on the band is one 0 and empty (⊔) to the left and right.

Let the numbers be written on band with the least significant bit to the leftmost position and let the significance be 2^n for position $n(n > 0)$ to right. To show that we can increment an even value is trivial because we just write the digit 1 to the start position and stop the program. For odd and even numbers we present the flow chart in Figure 13.3 and the state table as alternative notation in Figure 13.4.

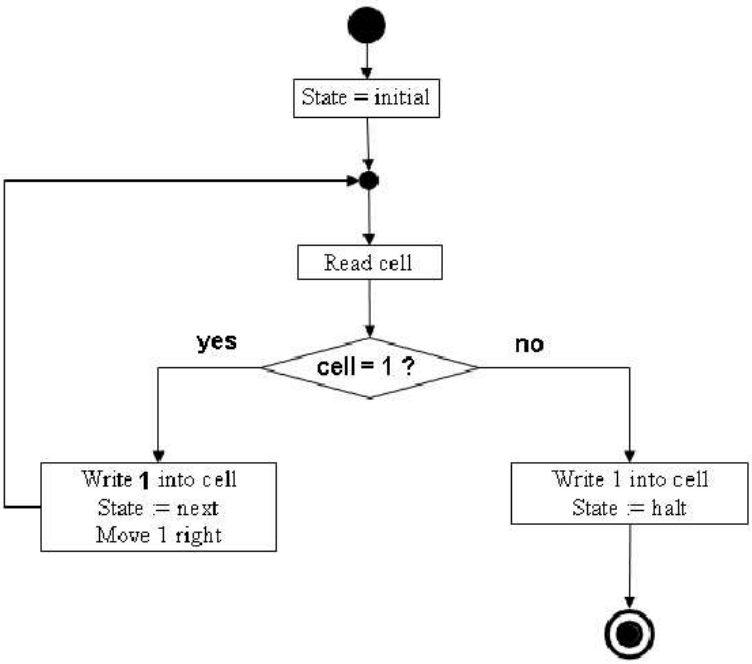

Figure 13.3 Flowchart to increment an odd number with the Turing machine

state	0	1	⊔
initial	(halt,1,n)	(next,1,r)	n.a.
next	(halt,1,n)	(next,1,r)	(halt,1,n)

Figure 13.4 Transition function for a Turing machine that increments an integer

We will learn an other notation for describing the behavior of machines with a finite number of states in Section 13.3.

Now that we have shown how to increment a number with the Turing machine we can do any algorithmic operation, and therefore produce any algorithm. In other words, it is a universal machine as potential as a real electronic computer in terms of the problems it can solve. But, real computers are much more complex. The Turing machine does not have any interactive device or any possibility for interrupts. For theoretical investigations however it serves as an object of study. Any results we get from there are applicable to a real computer.

13.2 Decidability

The next principal issue we will investigate is the question if we can decide beforehand if a Turing machine will eventually stop (i.e. reach its *halt* state) or not. If we could solve this question theoretically we could proof for any programm if it will terminate or if it will run in an infinite loop.

This *halting problem* is closely related to the question what can be computed. If somebody wants to know if an method of calculation is algorithmic he can try to run it on the Turing machine. If the machine stops, the problem is *decidable*. In fact, the German mathematician David Hilbert believed in 1900 that any mathematical defined problem is decidable. But

Kurt Gödel (Austrian mathematician, * 1906, † 1978) proofed in his famous incompleteness theorem that "any sufficiently complex formal system is either contradictory or incomplete". Applied to the question whether all problems (which are sufficiently complex) can be solved by an algorithm, Gödel's theorem states that there are problems for which no algorithmic solution exist, i.e. are undecidable.

The most prominent undecidable problem is the *halting problem* of the Turing machine: "Can we create a program that checks any program if it will terminate or not?" The answer is NO. Turing proved that a general algorithm to solve the halting problem for all possible inputs is undecidable. Because of its fundamental importance we will sketch the proof-by-contradiction using pseudo-code. We assume there is a function *isHalting* that can test any other program if it terminates.

```
function isHalting (program, input)
{
        if program(input) terminates
            then return YES
            else return NO
}
```

Now we construct the program *test* that takes a program as input:

```
function test (program)
{
        while isHalting(program, program) = YES {do nothing}
}
```

Finally we run test with itself as parameter:

```
test (test)
```

The program *test* will only terminate when it is not terminating. This is a contradiction that proofs the impossibility to decide if a program will terminate for all possible inputs. The meaning of "decide" is that no program can be written that finds out if a arbitrary given program will terminate for all possible inputs. Applied to the Turing machine, this is the *halting problem* which turns out to be undecidable.

13.3 Finite State Machines

A Turing machine is a kind of "automata" that takes an input from the band and runs autonomously from its initial state via a finite number of other states to a final state. We have describe the behavior as a state transition table as in Figure 13.2.

We find automata everywhere not only in an abstract Turing machine. Coffee makers, ticket and soft drink distributers, access control, and gambling automata are some examples. Characteristic for those machines is that their actions of depend on its current state. Take a parking ticket distributer, it will only produce a ticket when you have put some coins into the machine and pressed the "print" button. The behavior of such a machine is best visualized by a state chart as in Figure 13.5.

The corresponding state matrix is shown in Figure 13.6. The events are put in the first line and the states are written in the first column. The transitions are marked in the crossing cells of state and event because the next state is defined by the current state and an event. Some event/state combinations are "illegal", i.e. they cannot occur or nothing will happen. Thats why they are marked with a hyphen (-).

The table notation is handy to program a state machine but the state transition chart is more human readable.

A *finite state machine* is an automata that

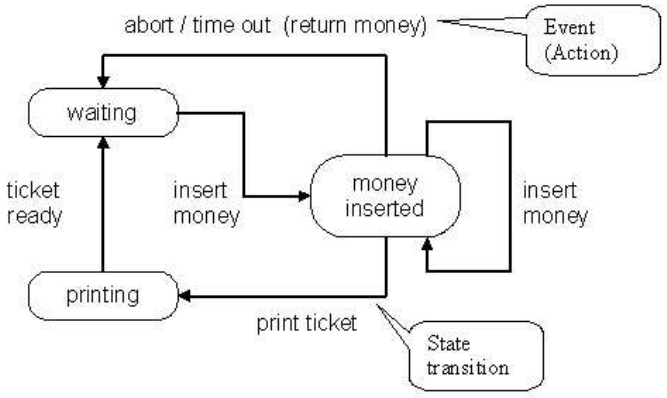

Figure 13.5 State transition diagram of a parking ticket distributer in UML notation

state	insert coin	abort	print ticket	time o
waiting	money inserted	-	-	-
money inserted	money inserted	waiting	printing	waiti
printing	-	-	-	-

Figure 13.6 Transition function matrix for a parking ticket distributer

- accepts a finite set of input data, called *events*,
- has a finite number of states (incl. one initial state and some final states)
- changes its state in dependence of the input (event) and the current state.

The set of state transitions as described in a state chart or matrix is called a *state transition function* $\delta : E \times S \rightarrow S$, where E is the set of events and S is the set of states.

If a finite state machine terminates it has received a legal sequence of events. These events could be symbols of an alphabet for instance. If we interpret the sequence of symbols as legal words of an formal language the state machine works as syntax checker. It accepts only legal language elements. Finite state machines are therefore often called *cognitive automata* and implemented in compilers. As example consult Figure 13.7 that accepts any alphanumeric string that is beginning with a letter.

We could describe all possible descriptors with a simple regular expression as introduced in Section 8.4. If l is a letter and d is a digit then the regular expression is:

$descriptor := l(l \cup d)^*$.

This is not a coincidence but it can be proved that any finite state machine is equivalent to an regular expression. There exists a language hierarchy and correspondence between grammar and automata (see Figure 13.8) that was categorized by Noam Chomsky (* 1928, professor emeritus of the MIT)

The classification forms an inclusion hierachy: Type-3 $\subset$ Type-2 $\subset$ Type-1 $\subset$ Type-0.

Type-3 grammars define the most simple languages that generate words made of sequences of symbols (terminals). This is used by text processing. With Type-2 grammars it is possible to build a language with non-terminals using a kind of parentheses. A Type-1

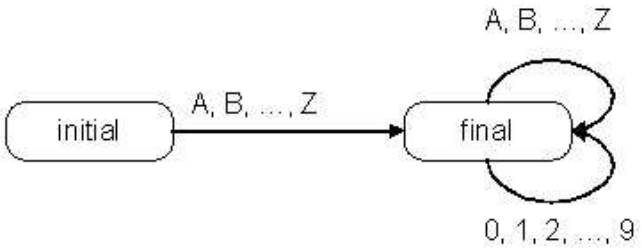

Figure 13.7 Finite state machine to identify descriptors that begin with a letter

language class	grammar	automata
Type-0	unlimited	general Turing machine
Type-1	context sensitive	Linear-bounded non-deterministic
Type-2	context free	stack machine
Type-3	regular	finite automata

Figure 13.8 Chomsky hierarchy of languages

grammar produces a language with elements that can be used only in a certain context. This is the case in most programming languages because the use of a variable depends on its prior declared type. Type-0 grammars do not restrict the structure of the language.

13.4 Verification

When we build an automata or any software system it is always a big question if the system will exactly do what is "expected". The first problem then is to formally specify what is expected. If we have no formal and unambiguous specification there is now way to make sure we know exactly what the program should do.

With a formal specification the output of a program is defined as function of the input. Applying a formal process of verification avoids the risk of inaccurate conclusions that may occur with intuitive reasoning.

The cardinal problem of this approach is that in general we do not know if our verification process terminates. In fact, we come back to the problem of decidability (Section 13.2). We have the situation of a mathematical defined problem with input and output but we do not know if we have an algorithm for it. So there is no guaranty that our verification will terminate. This is the reason why formal verification (proof) is only possible for small and simple programs. For complex programs we only *test* some representative situations and hope that all other cases behave as well or do not matter.

A formal verification of a program's correctness is based on the specification. To proof that a program works correctly we have to assume some input. The *preconditions* describes the set of possible input data to start the program. The next step is to follow the propagation of the input during the processing of the program. Many research has been done to analyze the program structure in order to see how the output state is affected by the input.

Here a little example: Let a, b be input values and the program shall only consist of the following if-then-else structure:

```
if (a > b)
    then return y := b/a
    else return y := a/b
```

The program should output the value of y. If $a, b > 0$ then we expect results between 0 and 1. How can we proof this? This is done by two preconditions.

- $\{a, b \mid a > b > 0\}$
- $\{a, b \mid 0 < a \leq b\}$

As the second precondition is the negation of the first one for any positive value of a and b there is no other possibility to consider. But there are more cases when we allow negative values for a or b. Now we have to follow the program flow. In the first case the "if-condition" is true and the "then-branch" will be executed. This results in a positive $y := b/a < 1$. In the second case the "if-condition" is false and the "else-branch" is executed resulting in a positive $y := a/b \leq 1$. We get indeed an output between 0 (excluding 0) and 1.

We formulate the desired output as *postcondition* and describe it as the set of values $\{y \mid 0 < y \leq 1\}$

The formulation of the pre- and postcondition for strictly negative input is left to the student as an exercise.

13.5 Testing

If we try verify a real program against a formal specification this would turn out to be a great challenge. So in commercial program development only some typical and important situations are checked.

This can be done not by a program analysis which is called a *structured walkthrough*. If the code is explained by its programmer to an other expert this is called *code review* or *code inspection*.

The other possibility of testing is to run the program or part of it with test input. In both cases it is characteristic that there is **no proof** of correctness. *Testing* only checks a program partially, but hopefully all relevant situations. There is an empirical argument discovered by the sociologist Vilfredo Pareto (* 1848, † 1923) that in that time "20% of population possessed 80% of the wealth". This statement has been generalized later by Joseph M. Juran to the *Pareto principle*. Applied to the testing of programs it is assumed "that we can find 80% of the errors with 20% of effort". This relationship between effort and result is visualized in Figure 13.9.

Testing is classified mainly in two categories:

- black-box testing
- white-box testing

A black-box test is a pure input/output test. We supply some input to the program, module, or function and see if it reacts or gives output as desired. The test input is not based on the program structure. To avoid a lot of unnecessary input as so call *boundary value analysis* can help to reduce the amount of input cases. For example if the program is expected to accept values between 0 and 1 then the boundary values and values slightly outside the boundaries are taken for testing.

Black-box testing is usually done by the people who are not programmers but users. They need no understanding of the internals of a program.

Black-box testing has become popular with the software development methodology called *eXtreme Programming (XP)*. Following XP this black-box testing is called *unit tests*. A

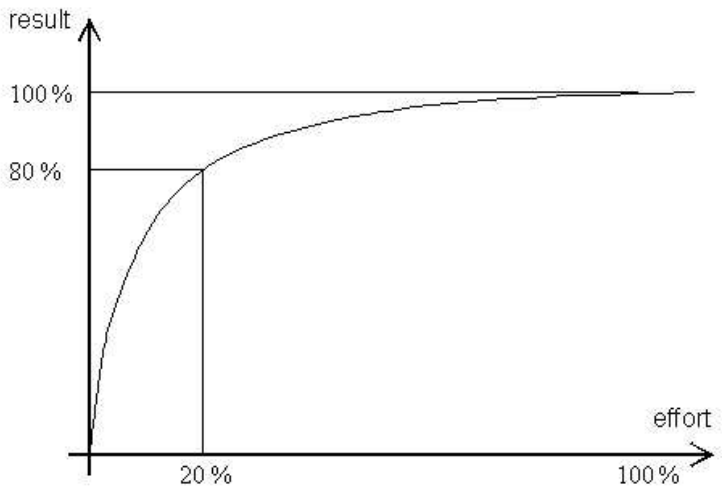

Figure 13.9 Pareto principle (80:20 rule)

unit test is created by the programmer prior to the development of the function or class itself. This is called *test first* in XP.

The programmer supplies all input values and states the postconditions (here called *assertions*) and the unit test framework generates the test environment. The test can be run by just pressing a start button. The programmer stops developing the function or class when the unit test is passed. This automated unit testing is also useful when a software is developed under a "collective code ownership". A programmer may change or extend the code as long as all unit tests are passed.

The white-box testing requires insight into the program, module, or function under test. This is why only programmers do this testing. The input for the tests is chosen fully aware of the program structure. If we consider our if-then-else example of the previous section we understand that for positive a and b we have to check both cases, where $a > b > 0$ and its negation. So white-box testing is closer to verification and can cover situations that otherwise would not be considered.

White-box testing is usually done on a function or class level, compared to black-box testing that is done mainly on a program or module level.

13.6 Summary

Review Terms

- Turing machine
  - band
  - empty cell
  - read/write head
  - control unit
  - state (initial, halt)
- Church's hypothesis
- computable function
- Decidability
  - halting problem
  - proof by contradiction
  - control unit

- state (initial, halt)
- finite state machine
  - event
  - state transition
  - state transition function
  - cognitive automata
- Chomsky hierarchy
- verification
  - precondition
  - postcondition
  - assertion
- Pareto priciple
- testing
  - black-box
  - white-box
  - structured walkthrough
  - code inspection
  - code review
  - boundary value analysis
- eXtreme Programming (XP)
  - unit test
  - test first

Exercises

13.1 Show that the summation algorithm of Chapter 2, Figure **??** has a space complexity of $O(1)$. State more precisely the constant $const$ in storage cell units for the given implementation in Figure **??**.

13.2 Specify pre- and postcondition for the following program structure:

switch (a)
case ($a > 0$): return $y := 1/a$
case ($a = 0$): return $y := a$
case ($a < 0$): return $y := -1/a$

Bibliographical Notes

Turing reformulated in his seminal work "On Computable Numbers, with an Application to the Entscheidungsproblem" Turing published in 1936 the results from Kurt G"odel. Turing proofed that any mathematical problem could be solved by his machine if an algorithm exists. This result was even more astonishing because at that time there was no computer available.

There are many good books available on theoretical computer science, namely van Leeuwen [ed] (1994), Hromkovic [2004], and Sch , but a much more entertaining introduction to formal languages and recursion is the book of Douglas R. Hofstadter Hofstadter [1980].

Much can be found about XP on the Web Wells , Jeffries an about their creators Kent Beck, Ward Cunningham and Ron Jeffries N. .