LECTURE NOTES

CPSC 351 — Winter 2025

Theoretical Foundations of Computer Science II

Philipp Woelfel

Chapter 6: Decidability

Contents

6 Decidability

We have already seen that Turing machines are much more powerful than finite automata. In fact, the *Church-Turing thesis* states that any problem that can be solved with an algorithm can also be solved with a Turing machine (and vice versa). This is only a thesis, and not a theorem, because there is no clear-cut definition of "algorithm". Clearly it is much more tedious to devise a Turing machine for a problem, than a program in a high level programming language. But in order to understand what problems we can or cannot solve with modern programming languages, it suffices to answer this question for Turing machines.

We will show that certain problems cannot be solved by Turing machines or high level programming languages. We will again restrict our attention to *decision problems*, where each input has a "yes" or "no" output. As discussed before, these are equivalent to languages—the answer for an input x is "yes", if x is in the language, and "no" otherwise.

Definition 6.1. *A language is* Turing machine recognizable[1] *(short: TM-recognizable), if there exists a Turing machine that recognizes it.*

For the purpose of conciseness we will sometimes simply say *recognizable* instead of Turing machine recognizable.

Recall that to recognize a language L, a Turing machine must accept exactly the inputs $x \in L$. So for each input $x \in L$, the Turing machine reaches an accepting state. For an input $x \notin L$, the Turing machine either rejects or it never halts. But we normally require that a computer program always halts, for every input. Therefore, it makes sense to require the same from a Turing machines.

Definition 6.2. *We say a Turing machine* decides *language L, if it recognizes L and halts for every input. Language L is* Turing machine decidable *(short: TM-decidable), if there exists a Turing machine that decides L.*

[1] In some textbooks this is also known as *semi-decidable*.

6.1 Diagonalization

Before we prove that some language is not decidable, we will give a simple example that will illustrate core idea.

Once upon a time, there was a remote village, in which an election for a new village chief was scheduled to take place. The current village chief suggested a new procedure to the village council:

> *Each villager can nominate any number of villagers for the chief position, by submitting a list of nominees. Self-nominations are allowed. If any villager submits a list that contains exactly the names of all villagers who did not nominate themselves, then that villager shall be the new chief. In case of multiple such submissions, one of the successful candidates will be chosen at random.*

A heated debate about the proposal began in the council. But then, one councilperson, who, incidentally, had a computer science degree, spoke up: "No villager can possibly find the correct list!". Other council members were confused. "Why?", they asked. The computer scientist explained.

> *Imagine, there is a villager who finds the correct list. For the purpose of this thought experiment, let us call her Alice. What if Alice self-nominates? That means she put her own name on her list of nominees. But then her list is wrong, because Alice can only nominate villagers who don't self-nominate, and that excludes herself.*
>
> *So then, maybe Alice does not self-nominate? Then she does not put her own name on her list of nominees. In that case, her list is also wrong, because Alice must add everyone to her list, who does not nominate themselves, and that includes Alice.*
>
> *No matter what Alice does, self-nominate or not, her list will be wrong. In conclusion, no villager can provide a correct list.*

The council members shake their heads in confusion. It is late at night, everyone is tired, and there lives nobody called Alice in the village, anyway. Despite enraged protests of the computer scientist, the council accepts the village chief's proposal. And to this day, no other chief has been elected.

6.1.1 The Diagonal Language

We will now translate the above example into the world of Turing machines. The *Diagonal Language* is

$$L_D = \{\langle M\rangle \mid M \text{ is a Turing machine that does not accept } \langle M\rangle\}.$$

Thus, a Turing machine for L_D takes as input the encoding of another Turing machine M. If M accepts its own encoding, $\langle M \rangle$, then L_D must not accept $\langle M \rangle$ (it must either reject or not halt). On the other hand, if M does not accept $\langle M \rangle$, then L_D must accept $\langle M \rangle$. In other words, L_D does the opposite (w.r.t. accepting or not accepting) of what M does on the input $\langle M \rangle$.

Of course, we could design a Turing machine that takes a Turing machine encoding $\langle M \rangle$ as an input, then simulate M on $\langle M \rangle$, and return the opposite answer of what M returns. This can be achieved by using the Universal Turing machine as a subfunction. However, if M does not halt on $\langle M \rangle$, then the simulation does also not halt, and our algorithm not return anything. Thus, such a Turing machine would not be able to recognize L_D. In fact, there exists no Turing machine that recognizes L_D:

Theorem 6.3. *The Diagonal Language is not Turing machine recognizable.*

Proof. We will prove this by contradiction. Assume that L_D is recognizable. Then there is a Turing machine M_D that recognizes L_D. We investigate what M_D does, if it receives its own description, $\langle M_D \rangle$, as input.

First assume that M_D accepts $\langle M_D \rangle$. Then, according to the definition of the Diagonal Language, $\langle M_D \rangle$ is not in L_D. Since M_D recognizes L_D, it must not accept any input that is not in L_D, and so specifically it must not accept the input $\langle M_D \rangle$. This is a contradiction.

Now assume that M_D does not accept $\langle M_D \rangle$. Then, again by the definition of L_D, $\langle M_D \rangle \in L_D$. Hence, since M_D recognizes L_D, it must accept the input $\langle M_D \rangle$. We have again arrived at a contradiction. □

6.1.2 The Python Diagonal Problem

Similar as for Turing machines, we can define decision problems that we cannot solve using high level languages, such as Python. For simplicity, we will assume, unless otherwise noted, that each Python function returns True or False, if it terminates.

Definition 6.4. *We say a Python function* `func` recognizes *a language L, if* `func(arg)` *returns* True *exactly for the strings* `arg` $\in L$. *It* decides *language L, if it recognizes L and always terminates (for every argument* `arg`*). Language L is Python-decidable (Python-recognizable), if exists is a Python function that decides (recognizes) it.*

Sometimes we pass a Python function `func2` as an argument to another Python function `func`, i.e., we call `func(func2)`. For simplicity, we do not distinguish between the functions `func2` itself (as a Python object), and its source code. We can do so without loss of generality, because on one hand the source code uniquely defines the function, and on the other hand, Python allows us to obtain the source code of any function using the `inspect` primitive (see Figure 6.1).

```python
import inspect
def add(x,y):
    return x+y
sourceCode = inspect.getsource(add)
```

Figure 6.1: This Python code assigns the source code defines function `add` and assigns its source code to the variable `sourceCode`.

The *Python Diagonal Language*, L_{Diag}, contains every Python method `func`, such that `func(func)` does not return `True`. With the same arguments as for the Diagonal Language, we can easily see that there can be no Python method that recognizes L_{Diag}.

Suppose there is such a method, and assume it is called `diag`. Then we can run

```
diag(diag.py)
```

If this call returns `True`, then `diag` is in L_{Diag}, because by the assumption that `diag` recognizes L_{Diag}, `diag(func)` must return `True` if and only if `func` $\in L_{Diag}$.

If it does not return `True` (i.e., it either returns `False` or the call does not terminate), then, since `diag` recognizes L_{Diag}, the argument of the function call, `diag`, is not in L_{Diag}. But if `diag` is not in L_{Diag}, then by definition of the language `diag(diag)` must return `True`—a contradiction.

Theorem 6.5. *There is no Python function that recognizes L_{Diag}.*

6.2 Undecidable Problems

The Python Diagonal Language is artificially defined in such a way that no Python function can recognize it. But it is not very useful for anything else. We will now consider more natural decision problems, and prove that they are not decidable. The Python Diagonal Language will serve as a "base case" for our impossibility proofs: We will show for these more natural decision problems, that if they can be decided by a Python function, then so can the Python Diagonal Language.

Turing machines are the standard computational model for proving that certain problems cannot be solved. According to the Church-Turing Thesis, Turing machines are as powerful as high level languages, such as Python. Hence, it does not matter if we prove that a language is not TM-decidable or not Python-decidable. Therefore, instead of Python-decidable and -recognizable will sometimes just say decidable and recognizable, respectively.

In the following, we will consider two variants of the same problem. One is expressed using Python functions, the other one using Turing machines. We will then show for the first variant that it cannot be solved by any Python program, and for the second, that it is undecidable, i.e., it cannot be solved by any Turing machine. But the arguments are essentially the same.

6.2.1 The Python Accepting Problem

The Python Accepting Problem is to decide, if the source code of a Python method returns True for a specific input. The corresponding language is

$$L_{Accepting} = \{(\texttt{func}, \texttt{arg}) \mid \texttt{func} \text{ is a Python function and } \texttt{func(arg)} \text{ returns True}\}$$

Thus, a Python method `acceptRec` thatrecognizes this language takes as arguments a Python function `func` and an argument `arg` If we call `acceptRec(func,arg)` then the function must return True if and only if `func(arg)` returns True.

Such a method `acceptRec()` exists, if termination is not required in case that the correct answer is False: `acceptTest(func,arg)` can simply simulate each step of the execution of `func(arg)`. In fact, a Python interpreter does exactly that, and the following implementation recognizes the language $L_{Accepting}$:

```
def acceptRec(func,arg):
    return func(arg)
```

Clearly, `acceptRec(func,arg)` returns the same value that `func(arg)` returns. But if `func(arg)` does not terminate, then neither does `acceptTest(func,arg)`.

Theorem 6.6. *$L_{Accepting}$ is Python-recognizable.*

If we want to have an algorithm that determines if `func(str)` returns True, then naturally we would like such an algorithm to always terminate. But, as we will show now, such an algorithm does not exist.

Theorem 6.7. *$L_{Accepting}$ is not Python-decidable.*

Proof. We will prove this by contradiction: Assume there is a function `acceptDec()` that decides $L_{Accepting}$. I.e., a call of `acceptDec(func,arg)` will eventually return True if `func(str)` returns True, and it will return False otherwise (even if `func(str)` does not terminate).

We will now use `acceptDec()` as a subroutine in a new program, which solves the Python Diagonal Problem:

```python
def diag(func):
    return not acceptDec(func,func)
```

This Python function simply returns the negation of `acceptDec(func,func)`. Thus, if `func(func)` returns True, then `diag(func)` returns False, and vice versa. Hence, `diag(func)` returns True if and only if `func` $\in L_{Diag}$. By the assumption that `acceptDec` always decides $L_{Accepting}$, it always terminates, and so `diag` also always terminates. Hence, `diag` decides L_{Diag}, and thus it obviously also recognizes L_{Diag}. This contradicts Theorem 6.5. □

6.2.2 The Turing Machine Accepting Problem

We will now translate the Python Accepting Problem into the Turing Machine world. Here, the Accepting Problem is the problem to decide if a given Turing machine accepts a given input. The corresponding language is

$$L_A = \{\langle M, w\rangle \mid M \text{ is a Turing machine that accepts } w\}.$$

This language is exactly what the Universal Turing machine U recognizes: U takes as input $\langle M, w\rangle$, where M is a Turing machine, and then it simulates M on w. Hence, U accepts the input if and only if M accepts w. Thus, we obtain:

Theorem 6.8. *L_A is Turing machine recognizable.*

Observe that U does not halt, if M does not halt, so U recognizes, but does not decide L_A. In fact, no Turing machine can decide L_A.

Theorem 6.9. *L_A is not decidable.*

Proof. For the purpose of contradiction, assume there is a Turing machine M_A that decides L_A. I.e., if $x \in L_A$, then M_A accepts the input x, and if $x \notin L_A$, then M_A rejects x. We will now present an algorithm that uses M_A as a subfunction, and decides the Diagonal Language—see Algorithm 1.

```
Input: The encoding $\langle M\rangle$ of a Turing machine
1 Simulate $M_A$ on $\langle M, M\rangle$            // $M_A$ is a TM that decides $L_A$
2 if $M_A$ accepts then reject
3 if $M_A$ rejects then accept
```

Algorithm 1: Deciding the Diagonal Language with a TM for the Accepting Problem.

The algorithm takes as input the description $\langle M\rangle$ of a Turing machine. It then simulates M_A on $\langle M, M\rangle$. Once M_A halts and accepts or rejects, the algorithm negates the answer of M_A: If M_A accepts, our algorithm rejects, and if M_A rejects, then our algorithm accepts.

We will now show that Algorithm 1 decides the Diagonal Language L_D. Consider an input $\langle M\rangle$ for the algorithm. If $\langle M\rangle \in L_D$, then this means that Turing machine M does not accept if it is given its own description as input. Thus, $\langle M, M\rangle \notin L_A$, and so M_A rejects $\langle M, M\rangle$. This means that Algorithm 1 accepts, which is the correct answer for the question if $\langle M\rangle$ is in L_D.

Now assume that $\langle M\rangle \notin L_D$. Then M accepts $\langle M\rangle$, so $\langle M, M\rangle \in L_A$. Hence, M_A accepts, and thus Algorithm 1 rejects.

To summarize: Algorithm 1 decides the Diagonal Language, provided that M_A decides the accepting problem. (Obviously, there exists a Turing machine for Algorithm 1: Such a Turing machine only needs to replace its input $\langle M\rangle$ with $\langle M, M\rangle$, and then simulate M_A.) Hence, the Diagonal Language is decidable, and therefore also Turing machine recognizable. This contradicts Theorem 6.3. In particular, our assumption that a Turing machine M_A exists, which decides L_A, is wrong. Hence, L_A is not decidable. □

6.2.3 The Python Halting Problem

The Python Halting Problem is similar to the Python Accepting Problem, except that now we only want to know if a given Python function terminates for some input. The corresponding language is

$$L_{Halting} = \{(\texttt{func}, \texttt{arg}) \mid \texttt{func} \text{ is a Python function such that } \texttt{func(arg)} \text{ terminates}\}.$$

It is easy to see that the following Python program recognizes $L_{Halting}$:

```python
def haltingRec(func,arg):
    func(arg)
return True
```

If $(\texttt{func}, \texttt{arg}) \in L_{Halting}$, then the call of `func(arg)` terminates, and `haltingRec(func,arg)` returns True. Otherwise, `func(arg)` does not terminate, and so `haltingRec(func,arg)` does not return True (because it also does not terminate).

Theorem 6.10. $L_{Halting}$ *is Python-recognizable.*

But similar to the Python Accepting Problem, $L_{Halting}$ is not decidable. The idea is this: If there is a Python function `haltingDec` that decides $L_{Halting}$, then we can also solve the Python Accepting Problem. Given an input $(\texttt{func}, \texttt{arg})$ for the Python Accepting Problem, we first use *haltingDec* to check if `func(arg)` terminates. If not, then clearly `func(arg)` does not return

True On the other hand, if `func(arg)` does terminate, then it is safe call `func(arg)` and return the same result.

Theorem 6.11. *$L_{Halting}$ is not Python-decidable.*

Proof. For the purpose of contradiction, assume there is a Python function *haltingDec* that decides $L_{Halting}$. Consider the following Python function:

```
def acceptDec(func,arg):
    if haltingDec(func,arg):
        return func(arg)
    else:
        return False
```

We will now show that the above Python function `acceptDec` decides $L_{Accepting}$.

Consider an input (`func`, `arg`). If (`func`, `arg`) $\in L_{Accepting}$, then `func(arg)` returns True, and in particular the function call terminates. Thus, (`func`, `arg`) $\in L_{Halting}$, so *haltingDec* calls `func(arg)` and also returns True.

Now assume (`func`, `arg`) $\notin L_{Accepting}$. If `func(arg)` terminates, it returns False. Then as before *haltingDec* calls `func(arg)` and returns the same value as `func(arg)`, which is now False. On the other hand, if `func(arg)` does not terminate, then (`func`, `arg`) $\notin L_{Halting}$, so the call of $haltingDec(func, arg)$ returns False. Thus, our algorithm also returns False.

Thus, our algorithm always gives the correct answer, and so it decides $L_{Accepting}$. This contradicts Theorem 6.7. Hence, our assumption that *haltingDec* decides $L_{Halting}$ must be wrong. □

6.2.4 The Python Accept-All Problem

The Python Accept-All Problem is to determine if a Python function always returns True, not matter what argument is passed to it:

L_{Σ^*} = {`func` | `func` is a Python function and `func(arg)` returns True for any argument `arg`}.

We will show that there is no Python function that decides this problem.

Theorem 6.12. *The Python Accept-All Problem is Python-undecidable.*

Proof. Again, we will prove this by contradiction. So assume there is a Python function `acceptAllDec`, such that

```
acceptAllDec(func)
```

returns True if `func(arg)` returns True for every argument `arg`, and otherwise returns False.

We will use `acceptAllDec` as a subfunction to solve the Python Accepting Problem. To that end, we will create a Python method `acceptDec`, which takes as input a Python function `func` and an argument `arg`. It then defines a new Python function `func2`, which ignores the argument passed to it, and simply calls `func(arg)` and returns the result. After that, `acceptDec(func,arg)` calls `acceptAllDec(func2)` and returns whatever the subfunction call returns.

```
def acceptDec(func,arg):
    def func2(arg2):
        return func(arg)
    return acceptAllDec(func2)
```

Observe that the function `func2(arg2)` defined in a call of `acceptDec(func,arg)` returns exactly the same as `func(arg)`, no matter what argument `arg2` is passed to it. Moreover, if `func(arg)` does not terminate, then neither does `func2(arg2)`. Thus, if `func(arg)` returns True, then and only then `func2(arg2)` always returns True, for every argument `arg2`. Hence, if `func(arg)` returns True, then and only then `acceptAllDec(func2)` returns True, and then and only then `acceptDec(func,arg)` returns True. Moreover, `acceptDec` always terminates (by the assumption that `acceptAllDec` decides the Python Accept-All Problem, and thus also always terminates). Hence, `acceptDec` decides the Python Accepting Problem. This contradicts Theorem 6.7. □

6.3 Reductions

In the previous sections we have shown that several problems are undecidable. All of those proofs, except for the one for the Diagonal Language, were *reductions*. A reduction from problem L_1 to problem L_2 establishes that if we can solve L_2, then we can also solve L_1. Thus, it shows that L_1 is "not harder" than L_2. Most importantly, if we already know that L_1 is unsolvable, then it follows that L_2 is also unsolvable.

For example, in the proof of Theorem 6.7, we showed that if there is a Python function that decides $L_{Accepting}$, then there is also a Python function that decides L_{Diag}. Thus, we have *reduced* L_{Diag} to $L_{Accepting}$. Similarly, in the proofs of Theorems 6.11 and 6.12 we reduced $L_{Accepting}$ to $L_{Halting}$ and $L_{\Sigma/ast}$, respectively.

Definition 6.13. *Let L_1 and L_2 be two languages. A* Python reduction *from L_1 to L_2 is a Python function that decides L_1 using regular Python code as well as zero or more calls to an "oracle" subfunction that is assumed to decide L_2. We say L_1 is* Python reducible *to L_2, or short $L_1 \leq_{Py} L_2$, if there exists a Python reduction from L_1 to L_2.*

The Python functions in the proofs of Theorems 6.7, 6.11 and 6.12 show that

$$L_{Diag} \leq_{Py} L_{Accepting} \leq_{Py} L_{Halting} \text{ and } L_{Accepting} \leq_{Py} L_{\Sigma^*}.$$

Theorem 6.14. *Let L_1 and L_2 be languages, where $L_1 \leq_{Py} L_2$. If L_1 is not Python-decidable, then L_2 is not Python-decidable.*

Proof. We prove the contrapositive: If L_2 is Python-decidable, then L_1 is Python-decidable. Suppose L_2 is Python-decidable. Then there exists a Python function `func2` that decides L_2. Since $L_1 \leq_{Py} L_2$, there exists a Python function `func1` that decides L_1, using `func2` as an "oracle" subfunction. Since `func2` can be implemented in Python, L_1 is Python-decidable. □

6.3.1 Mapping Reductions

The Python reduction from L_A to L_{Σ^*} in the proof of Theorem 6.7 is of a special form: The Python function solves the Python Accepting Problem, by taking the input (`func`, `arg`), and *transforming* it into a new argument `func2`, which is then passed to the "oracle" Python function `acceptAllDec`. It then returns exactly the return value of that subfunction call. This is called a *mapping reduction.*

Specifically, a mapping reduction from a language L_1 to a language L_2 is a Python function that decides L_1 in the following way: Given the argument x, it computes a new value $f(x)$, and then calls an "oracle" Python subfunction that decides L_2 with the argument $f(x)$. Finally, it returns the result of that subfunction call.

Clearly, that Python function is correct for an input x, if

$$x \in L_1 \Leftrightarrow f(x) \in L_2.$$

Thus, a more mathematical way of defining this concept is as follows.

Definition 6.15. *Let L_1 and L_2 be languages over the alphabet Σ. A* mapping reduction *from L_1 to L_2 is a function $f : \Sigma^* \to \Sigma^*$, such that*

1. *f can be computed by a Python function; and*
2. *$x \in L_1$ if and only if $f(x) \in L_2$.*

If there is a mapping reduction from L_1 to L_2, then we write $L_1 \leq_m L_2$.

A mapping reduction from L_1 to L_2 immediately translates into a Python function that solves L_1 using an "oracle" Python function `func2` that decides L_2: Given an input x, the algorithm simply computes $f(x)$ and then returns `func2(`$f(x)$`)`.

Observation 6.16. *If $L_1 \leq_m L_2$, then $L_1 \leq_{Py} L_2$.*

6.3.2 The Python Equivalence Problem

Two Python functions are equivalent, if they recognize the same language. The Python Equivalence Problem is to decide if two given Python functions are equivalent:

$$L_{\text{PEQ}} = \{(\texttt{func1}, \texttt{func2}) \mid \texttt{func1} \text{ and } \texttt{func2} \text{ are equivalent Python functions}\}.$$

That is, $(\texttt{func1}, \texttt{func2}) \in L_{\text{PEQ}}$, if and only if for any input `arg`, either `func1` and `func2` both accept `arg`, or both do not accept `arg`.

Assume that `PEQDec` is a Python function that decides L_{PEQ}. Then the following Python function decides L_{Σ^*}:

```python
def acceptDec(func):
    def alwaysTrue(arg):
        return True
    return PEQDec(func,alwaysTrue)
```

This is a mapping-reduction, because `acceptDec` makes exactly one funcion call to the "oracle" Python function `PEQDec` and then returns the same result. Moreover, `acceptDec` decides L_{Σ^*}, assuming that `PEQDec` decides L_{PEQ}: `alwaysTrue` is defined as a Python function that returns always `True`, and thus `PEQDec(func,alwaysTrue)` returns `True` if and only if `func` also always returns `True`. Hence, `acceptDec(func)` returns `True` if and only if `func` $\in L_{\Sigma^*}$.

Thus, we have shown that $L_{\Sigma^*} \leq_m L_{\text{PEQ}}$. Since mapping reductions are also Python reductions, it is also true that $L_{\Sigma^*} \leq_{Py} L_{\text{PEQ}}$. Hence, from Theorems 6.12 and 6.14 we obtain

Theorem 6.17. *The Python Equivalence Problem is undecidable.*

6.3.3 Proving Unrecognizability with Mapping Reduction

An advantage of a mapping reduction $L_1 \leq_m L_2$ is that if L_1 is not recognizable, then neither is L_2. This is in general not the case for Turing reductions.

Theorem 6.18. *Let L_1 and L_2 be languages over the alphabet Σ. If $L_1 \leq_m L_2$ and L_1 is not recognizable, then L_2 is not recognizable.*

Proof. We prove the contrapositive statement: If L_2 is recognizable, then so is L_1. Suppose L_2 is recognizable. Then there exists a Python function `func2` that recognizes L_2. Since $L_1 \leq_m L_2$, there exists a function $f : \Sigma^* \to \Sigma^*$ that can be computed by a Python function `transform`, such that

$$x \in L_1 \Leftrightarrow f(x) \in L_2. \tag{6.1}$$

Consider the following Python function:

```python
def func1(arg):
    return func2(transform(arg))
```

It is easy to see that `func1` recognizes L_1: If `arg` $\in L_1$, then `transform(arg)` $\in L_2$. By the assumption that `func2` recognizes L_2, `func2(transform(arg))` returns True. Thus, `func1(arg)` also returns True.

On the other hand, if `arg` $\notin L_1$, then `transform(arg)` $\notin L_2$. Then `func2(transform(arg))` either returns False or it does not terminate. Thus, `func1(arg)` either also returns False or does not terminate. □

Thus, we can use mapping reductions to prove that languages are not recognizable.

Theorem 6.19. *The complement of the Accepting problem, $\overline{L_{Accepting}}$, is not recognizable.*

Proof. We will show that $L_{Diag} \leq_m \overline{L_{Accepting}}$. Then the claim follows from Theorem 6.18 and the fact that L_{Diag} is not recognizable (Theorem 6.3).

Recall that

$$L_{Diag} = \{\texttt{func} \mid \texttt{func} \text{ is a Python function such that } \texttt{func(func)} \text{ does not return True}\}.$$

A slightly annoying technicality is that `func` might not be a properly encoded Python function. In that case `func` is not in L_{Diag}. We can use the Python primitive `callable(func)`, which return True if and only if `func` is a Python function.

The following is a reduction from L_{Diag} to $\overline{L_{Accepting}}$, assuming the "oracle" Python function `notAccept` decides $\overline{L_{Accepting}}$:

```
def diagRec(func):
    def alwaysTrue(arg):
        return True
    if not callable(func):
        return notAccept(alwaysTrue,alwaysTrue)
    else:
        return notAccept(func,func)
```

To see that this is correct, first assume that `func` $\in L_{Diag}$. Then `func` is a Python function, so `callable(func)` returns True. Moreover, `func(func)` does not return True, and so `notAccept(func,func)` returns True. Hence, `diagRec(func)` returns True, which is correct.

Now assume `func` $\notin L_{Diag}$. If `func` is not a Python function, then `callable(func)` returns False, and the algorithm returns the result of `notAccept(alwaysTrue,alwaysTrue)`. Since `alwaysTrue` is defined as a Python function that always returns True, `alwaysTrue(alwaysTrue)` returns True, and so `notAccept(alwaysTrue,alwaysTrue)` returns False. Thus, `diagRec(func)` also returns False. If `func` is a Python function, then `func(func)` returns True by the definition of L_{Diag}. In this case `diagRec(func)` returns False, because `notAccept(func,func)` returns False. □

6.4 Union, Intersection, and Complement

It is easy to see that if two languages L_1 and L_2 are decidable, then so are $L_1 \cup L_2$, $L_1 \cap L_2$, and $\overline{L_1}$: Consider an input `arg`, for which we want to test if it is in one of these three languages. Since there are Python functions `func1` and `func2` that decide L_1 and L_2, respectively, we can run both of them on `arg`, one after the other. As both Python functions terminate eventually, we get an answer from each of them, and this answer lets us determine whether `arg` is in $L_1 \cup L_2$, $L_1 \cap L_2$, and $\overline{L_1}$, respectively. Thus, we have the following result.

Theorem 6.20. *The set of decidable languages is closed under union, intersection, and complement.*

For recognizable languages, the above approach works only for intersection, even though the argument is slightly more subtle. Suppose L_1 and L_2 are recognizable. Then there are Python functions `func1` and `func2` that recognize L_1 and L_2, respectively. The following Python function recognizes $L_1 \cap L_2$:

```
def intersection(arg):
    return func1(arg) and func2(arg)
```

If both, `func1(arg)` and `func2(arg)`, terminate, then clearly the return value `func1(arg) and func2(arg)` is correct. Otherwise, if one of `func1(arg)` and `func2(arg)` does not terminate, then `intersection(arg)` also does not terminate. This is fine, because in this case `arg` $\notin L_1 \cap L_2$.

Theorem 6.21. *The set of recognizable languages is closed under intersection.*

For union and complement, the above approach does not work. In fact, we have already seen that L_A is recognizable (Theorem 6.8), but $\overline{L_A}$ is not (Theorem 6.19).

Theorem 6.22. *The set of recognizable languages is not closed under complement.*

But what about union? If the set of recognizable languages were closed under complement, we could use De Morgan's rules to concluded from Theorem 6.21 that it is also closed under union. Unfortunately, due to Theorem 6.22, this is not an option. So we have to prove it the hard way.

Theorem 6.23. *The set of recognizable languages is closed under union.*

Proof. Suppose L_1 and L_2 are recognizable, so they are recognized by Python functions `func1` and `func2`, respectively. Consider the following code, which is not fully implemented (see the comments):

```python
def union(arg):
    #Execute the following if-statements in parallel:
    if func1(arg):
        return True
    if func2(arg):
        return True
    # Once both functions have terminated, and if none returned True:
    return False
```

It is possible to execute multiple functions in parallel in Python. But the syntactical overhead distracts from the core ideas, so the above code is not fully implemented. However, it is clear that this can be implemented such that if either `func1(arg)` or `func2(arg)` terminates and returns `True`, then `union(arg)` also returns `True`. Otherwise, if either both functions return `False` or do not terminate, then `union(arg)` either also does not terminate or returns `False`. Hence, `union` recognizes $L_1 \cup L_2$. □

We have seen that L_A is recognizable (Theorem 6.8), but not decidable (Theorem 6.9). Then we proved by reduction that $\overline{L_A}$ is not recognizable (Theorem 6.19). But this result is no accident: If a language L is not decidable, then either L or $\overline{L}$ cannot be recognizable.

Theorem 6.24. *Let L be a language. If L and $\overline{L}$ are recognizable, then L is decidable.*

Proof. Suppose we have Python functions `func1` and `func2` that recognize L and $\overline{L}$, respectively. Then we can define a Python function that takes an argument `arg` and executes `func1(arg)` and `func2(arg)` in parallel. As soon as one of the two function calls returns some value, ret, we know the correct answer: If `func1(arg)` returns ret, then that is the correct answer, and if `func2(arg)` returns ret, the negation of ret is the correct answer. Thus, as soon as one of `func1(arg)` and `func2(arg)` terminates, our program can return the correct result.

Now observe that one of the two functions will eventually terminate: if `arg` is in L, then `func1(arg)` eventually terminates (and returns `True`), and otherwise the same is true for `func2(arg)`. Thus, our algorithm always terminates and returns the correct answer. I.e., it decides L. □

6.5 Exercises

6.1 In a huge library, every book has a unique title on the cover and some (not necessarily unique) text inside. Except for one special book, titled *The Catalogue*, which is blank inside. A librarian offers you a job. He asks that you fill The Catalogue with a list of some of the titles of the books in the library. Precisely, the title of a book in the library should be listed in The Catalogue, if and only if the text of the book does not contain its title. The librarian promises you a huge reward, should you finish the job without errors. If, however, you don't fill The Catalogue completely, or if you add an incorrect entry, then you will receive no payment.

diagonalization

Should you accept the job offer?

6.2 Let $[0, 1]$ denote the real numbers between (and including) 0 and 1. The goal of this exercise is to prove that this set is not countable. Assume it is countable. That means, we can write down all real numbers in that interval, one on top of the other, in decimal notation with infinitely many decimal digits. For example, the list could look like so:

diagonalization

$$\begin{array}{l}
0.\ 5\ 6\ 5\ 6\ 5\ 6\ 5\ 6\ldots \\
0.\ 5\ 3\ 8\ 1\ 9\ 0\ 0\ 0\ldots \\
0.\ 7\ 1\ 1\ 1\ 1\ 1\ 1\ 1\ldots \\
0.\ 3\ 1\ 2\ 7\ 7\ 0\ 0\ 9\ldots \\
0.\ 1\ 2\ 4\ 3\ 3\ 0\ 0\ 0\ldots \\
0.\ 2\ 0\ 0\ 0\ 0\ 0\ 0\ 0\ldots \\
0.\ 4\ 3\ 1\ 9\ 0\ 7\ 2\ 1\ldots \\
\ \vdots
\end{array}$$

Prove that this is not possible. *Hint:* It has something to do with the highlighted digits on the diagonal.

6.3 Define the Turing-Machine Halting Problem, similarly to the Python Halting Problem. Show that the Turing-Machine Halting Problem is not TM-decidable.

decidability

6.4 Show that the following languages are not Python-decidable.

decidability,reductions

(a) $L_{NeverHalt}$ contains all Python functions `func`, such that `func(arg)` does not terminate for any argument `arg`.

(b) $L_\emptyset$ contains all Python functions `func`, such that `func(arg)` does not return True for any argument `arg`

(c) L_∞ is the language that contains all Python functions `func`, such that there are infinitely many arguments `arg` for which `funcarg` returns True.

(d) $L_{DoubleOrNothing}$ is the language of all triples (`func1`,`func2`,`arg`), such that `func1` and `func2` are Python functions, and either the two function calls `func1(arg)` and `func2(arg)` both return True, or none of them returns True.

6.5 Note that in Python it is possible to obtain the source code of a function, using the `inspect` module. For example,

reductions

```
import inspect
def alwaysTrue(arg):
  return True
code = inspect.getsource(alwaysTrue)
print(code)
```

prints the function definition of `alwaysTrue`, i.e., it produces the following printout:

```
def alwaysTrue(arg):
  return True
```

We say the i-th line in the code of a Python function `func` is *useless*, if there exists no argument `arg`, such that `func(arg)` executes line i. Let L_1 be the language that contains all pairs (`func`, i), such that the i-th line in the source code of `func` (i.e., the code returned from `inspect.getsource(alwaysTrue)`) is useless. Further, let L_2 be the language that contains all Python functions `func` whose source code has some useless line.

(a) Prove $L_\emptyset \leq_{Py} L_1$. (For the definition of $L_\emptyset$, see Exercise 6.5.)

(b) Discuss if the reduction in part (a) is a mapping reduction. If not, can you find a mapping reduction?

(c) Prove $L_2 \leq_{Py} L_1$.

(d) Challenge Question: Prove $L_1 \leq_{Py} L_2$.

6.6 Let L be the language that accepts all pairs (`func1`,`func2`), such that `func1` and `func2` are Python functions, and for every argument `arg` either `func1(arg)` or `func2(arg)` does not return True.

mapping reductions

Prove the following statements:

(a) $L_\emptyset \leq_m L$.

(b) $L \leq_m L_\emptyset$.

(For the definition of $L_\emptyset$, see Exercise 6.5.)

6.7 Suppose L_2 is recognizable and $L_1 \leq_{Py} L_2$. Show that this does not imply that L_1 is recognizable (by giving a counter example). Where does the equivalent of the proof of Theorem 6.14 fail for recognizable languages?

mapping reductions, recognizability