Learning Objectives:

- See a fundamental result in computer science and mathematics: the existence of uncomputable functions.
- See the canonical example for an uncomputable function: *the halting problem*.
- Introduction to the technique of *reductions* which will be used time and again in this course to show difficulty of computational tasks.
- Rice's Theorem, which is a starting point for much of research on compilers and programming languages, and marks the difference between *semantic* and *syntactic* properties of programs.

10

Is every function computable?

> *"A function of a variable quantity is an analytic expression composed in any way whatsoever of the variable quantity and numbers or constant quantities."*, Leonhard Euler, 1748.

We saw that NAND programs can compute every finite function. A natural guess is that NAND++ programs could compute every infinite function. However, this turns out to be *false*, even for functions with 0/1 output. That is, there exists a function $F : \{0,1\}^* \to \{0,1\}$ that is *uncomputable*! This is actually quite surprising, if you think about it. Our intuitive notion of a "function" (and the notion most scholars had until the 20th century) is that a function f defines some implicit or explicit way of computing the output $f(x)$ from the input x.[1] The notion of an "uncomputable function" thus seems to be a contradiction in terms, but yet the following theorem shows that such creatures do exist:

Theorem 10.1 — Uncomputable functions. There exists a function $F^* : \{0,1\}^* \to \{0,1\}$ that is not computable by any NAND++ program.

Proof. The proof is illustrated in Fig. 10.1. We start by defining the following function $G : \{0,1\}^* \to \{0,1\}$:

For every string $x \in \{0,1\}^*$, if x satisfies **(1)** x is a valid representation of a NAND++ program P_x and **(2)** when the program P_x is executed on the input x it halts and produces an output, then we define $G(x)$ as the first bit of this output. Otherwise (i.e., if x is not a valid representation of a program, or the program P_x never halts on x) we define $G(x) = 0$. We define $F^*(x) := 1 - G(x)$.

[1] In the 1800's, with the invention of the Fourier series and with the systematic study of continuity and differentiability, people have starting looking at more general kinds of functions, but the modern definition of a function as an arbitrary mapping was not yet universally accepted. For example, in 1899 Poincare wrote "we have seen a mass of bizarre functions which appear to be forced to resemble as little as possible honest functions which serve some purpose. ... they are invented on purpose to show that our ancestor's reasoning was at fault, and we shall never get anything more than that out of them".

We claim that there is no NAND++ program that computes F^*. Indeed, suppose, towards the sake of contradiction, that there was some program P that computed F^*, and let x be the binary string that represents the program P. Then on input x, the program P outputs $F^*(x)$. But by definition, the program should also output $1 - F^*(x)$, hence yielding a contradiction. ■

Figure 10.1: We construct an uncomputable function by defining for every two strings x, y the value $1 - P_y(x)$ which equals to 0 if the program described by y outputs 1 on x, and equals to 1 otherwise. We then define $F^*(x)$ to be the "diagonal" of this table, namely $F^*(x) = 1 - P_x(x)$ for every x. The function F^* is uncomputable, because if it was computable by some program whose string description is x^* then we would get that $P_{x^*}(x^*) = F(x^*) = 1 - P_{x^*}(x^*)$.

> The proof of Theorem 10.1 is short but subtle. I suggest that you pause here and go back to read it again and think about it - this is a proof that is worth reading at least twice if not three or four times. It is not often the case that a few lines of mathematical reasoning establish a deeply profound fact - that there are problems we simply *cannot* solve and the "firm conviction" that Hilbert alluded to above is simply false.

The type of argument used to prove Theorem 10.1 is known as *diagonalization* since it can be described as defining a function based on the diagonal entries of a table as in Fig. 10.1. The proof can be thought of as an infinite version of the *counting* argument we used for showing lower bound for NAND progams in Theorem 5.4. Namely, we show that it's not possible to compute all functions from

$\{0,1\}^* \rightarrow \{0,1\}$ by NAND++ programs simply because there are more functions like that then there are NAND++ programs.

10.1 The Halting problem

Theorem 10.1 shows that there is *some* function that cannot be computed. But is this function the equivalent of the "tree that falls in the forest with no one hearing it"? That is, perhaps it is a function that no one actually *wants* to compute.

It turns out that there are natural uncomputable functions:

Theorem 10.2 — Uncomputability of Halting function. Let $HALT : \{0,1\}^* \rightarrow \{0,1\}$ be the function such that $HALT(P,x) = 1$ if the NAND++ program P halts on input x and equals to 0 if it does not. Then $HALT$ is not computable.

Before turning to prove Theorem 10.2, we note that $HALT$ is a very natural function to want to compute. For example, one can think of $HALT$ as a special case of the task of managing an "App store". That is, given the code of some application, the gatekeeper for the store needs to decide if this code is safe enough to allow in the store or not. At a minimum, it seems that we should verify that the code would not go into an infinite loop.

Proof. The proof will use the previously established Theorem 10.1 , as illustrated in Fig. 10.2. That is, we will assume, towards a contradiction, that there is NAND++ program P^* that can compute the $HALT$ function, and use that to derive that there is some NAND++ program Q^* that computes the function F^* defined above, contradicting Theorem 10.1. (This is known as a proof by *reduction*, since we reduce the task of computing F^* to the task of computing $HALT$. By the contrapositive, this means the uncomputability of F^* implies the uncomputability of $HALT$.)

Indeed, suppose that P^* was a NAND++ program that computes $HALT$. Then we can write a NAND++ program Q^* that does the following on input $x \in \{0,1\}^*$:[2]

1. Compute $z = P^*(x,x)$
2. If $z = 0$ then output 1.
3. Otherwise, if $z = 1$ then let y be the first bit of $EVAL(x,x)$ (i.e., evaluate the program described by x on the input x). If $y = 1$ then

[2] Note that we are using here a "high level" description of NAND++ programs. We know that we can implement the steps below, for example by first writing them in NAND« and then transforming the NAND« program to NAND++. Step 1 involves simply running the program P^* on some input.

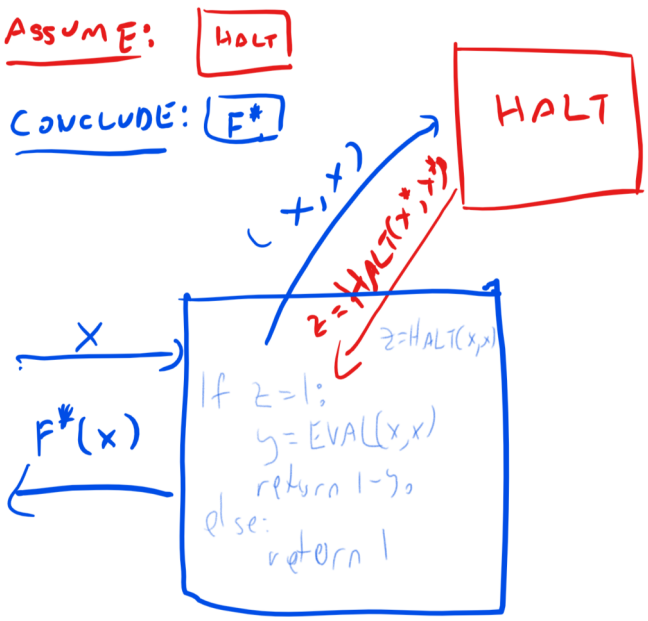

Figure 10.2: We prove that $HALT$ is uncomputable using a *reduction* from computing the previously shown uncomputable function F^* to computing $HALT$. We assume that we had an algorithm that computes $HALT$ and use that to obtain an algorithm that computes F^*.

output 0. Otherwise output 1.

Claim: For every $x \in \{0,1\}^*$, if $P^*(x,x) = HALT(x,x)$ then the program $Q^*(x) = F^*(x)$ where F^* is the function from the proof of Theorem 10.1.

Note that the claim immediately implies that our assumption that P^* computes $HALT$ contradicts Theorem 10.1, where we proved that the function F^* is uncomputable. Hence the claim is sufficient to prove the theorem.

Proof of claim:: Let x be any string. If the program described by x halts on input x and its first output bit is 1 then $F^*(x) = 0$ and the output $Q^*(x)$ will also equal 0 since $z = HALT(x,x) = 1$, and hence in step 3 the program Q^* will run in a finite number of steps (since the program described by x halts on x), obtain the value $y = 1$ and output 0.

Otherwise, there are two cases. Either the program described by x does not halt on x, in which case $z = 0$ and $Q^*(x) = 1 = F^*(x)$. Or the program halts but its first output bit is not 1. In this case $z = 1$ but the value y computed by $Q^*(x)$ is not 1 and so $Q^*(x) = 1 = F^*(x)$. ■

P Once again, this is a proof that's worth reading more than once. The uncomputability of the halting problem is one of the fundamental theorems of computer science, and is the starting point for much of the investigations we will see later. An excellent way to get a better understanding of Theorem 10.2 is to do Exercise 10.1 which asks you to prove an alternative proof of the same result.

10.1.1 Is the Halting problem really hard?

Many people's first instinct when they see the proof of Theorem 10.2 is to not believe it. That is, most people do believe the mathematical statement, but intuitively it doesn't seem that the Halting problem is really that hard. After all, being uncomputable only means that $HALT$ cannot be computed by a NAND++ program. But programmers seem to solve $HALT$ all the time by informally or formally arguing that their programs halt. While it does occasionally happen that a program unexpectedly enters an infinite loop, is there really no way to solve the halting problem? Some people argue that *they* can, if they think hard enough, determine whether any concrete program that they are given will halt or not. Some have even argued that humans in general have the ability to do that, and hence humans have inherently superior intelligence to computers or anything else modeled by NAND++ programs (aka Turing machines).[3]

[3] This argument has also been connected to the issues of consciousness and free will. I am not completely sure of its relevance but perhaps the reasoning is that humans have the ability to solve the halting problem but they exercise their free will and consciousness by choosing not to do so.

The best answer we have so far is that there truly is no way to solve $HALT$, whether using Macs, PCs, quantum computers, humans, or any other combination of mechanical and biological devices. Indeed this assertion is the content of the Church-Turing Thesis. This of course does not mean that for *every* possible program P, it is hard to decide if P enter an infinite loop. Some programs don't even have loops at all (and hence trivially halt), and there are many other far less trivial examples of programs that we can certify to never enter an infinite loop (or programs that we know for sure that *will* enter such a loop). However, there is no *general procedure* that would determine for an *arbitrary* program P whether it halts or not. Moreover, there are some very simple programs for which it not known whether they halt or not. For example, the following Python program will halt if and only if Goldbach's conjecture is false:

```
def isprime(p):
   return all(p % i for i in range(2,p-1))
```

```
def Goldbach(n):
   return any( (isprime(p) and isprime(n-p))
        for p in range(2,n-1))

n = 4
while True:
   if not Goldbach(n): break
   n+= 2
```

Given that Goldbach's Conjecture has been open since 1742, it is unclear that humans have any magical ability to say whether this (or other similar programs) will halt or not.

Figure 10.3: XKCD's take on solving the Halting problem, using the principle that "in the long run, we'll all be dead".

10.1.2 Reductions

The Halting problem turns out to be a linchpin of uncomputability, in the sense that Theorem 10.2 has been used to show the uncomputability of a great many interesting functions. We will see several examples in such results in the lecture and the exercises, but there are many more such results in the literature (see Fig. 10.4).

The idea behind such uncomputability results is conceptually simple but can at first be quite confusing. If we know that $HALT$ is uncomputable, and we want to show that some other function $BLAH$ is uncomputable, then we can do so via a *contrapositive* argument (i.e., proof by contradiction). That is, we show that *if* we had a NAND++ program that computes $BLAH$ *then* we could have a NAND++ program that computes $HALT$. (Indeed, this is exactly how we showed

that $HALT$ itself is uncomputable, by showing this follows from the uncomputability of the function F^* from Theorem 10.1.)

For example, to prove that $BLAH$ is uncomputable, we could show that there is a computable function $R : \{0,1\}^* \rightarrow \{0,1\}^*$ such that for every $x \in \{0,1\}^*$, $HALT(x) = BLAH(R(x))$. Such a function is known as a *reduction*, because we are *reducing* the task of computing $HALT$ to the task of computing $BLAH$. The confusing part about reductions is that we are assuming something we *believe* is false (that $BLAH$ has an algorithm) to derive something that we *know* is false (that $HALT$ has an algorithm). For this reason Michael Sipser described such results as having the form "If pigs could whistle then horses could fly".

At the end of the day reduction-based proofs are just like other proofs by contradiction, but the fact that they involve hypothetical algorithms that don't really exist tends to make such proofs quite confusing. The one silver lining is that at the end of the day the notion of reductions is mathematically quite simple, and so it's not that bad even if you have to go back to first principles every time you need to remember what is the direction that a reduction should go in. (If this discussion itself is confusing, feel free to ignore it; it might become clearer after you see an example of a reduction such as the proof of Theorem 10.5.)

4

[4] TODO: clean up this figure

10.2 *Impossibility of general software verification*

The uncomputability of the Halting problem turns out to be a special case of a much more general phenomenon. Namely, that *we cannot certify semantic properties of general purpose programs*. "Semantic properties" mean properties of the function that the program computes, as opposed to properties that depend on the particular syntax. For example, we can easily check whether or not a given C program contains no comments, or whether all function names begin with an upper case letter. As we've seen, we cannot check whether a given program enters into an infinite loop or not.

But we could still hope to check some other properties of the program. For example, we could hope to certify that a given program M correctly computes the multiplication operation, or that no matter what input the program is provided with, it will never reveal some confidential information. Alas it turns out that the task of checking

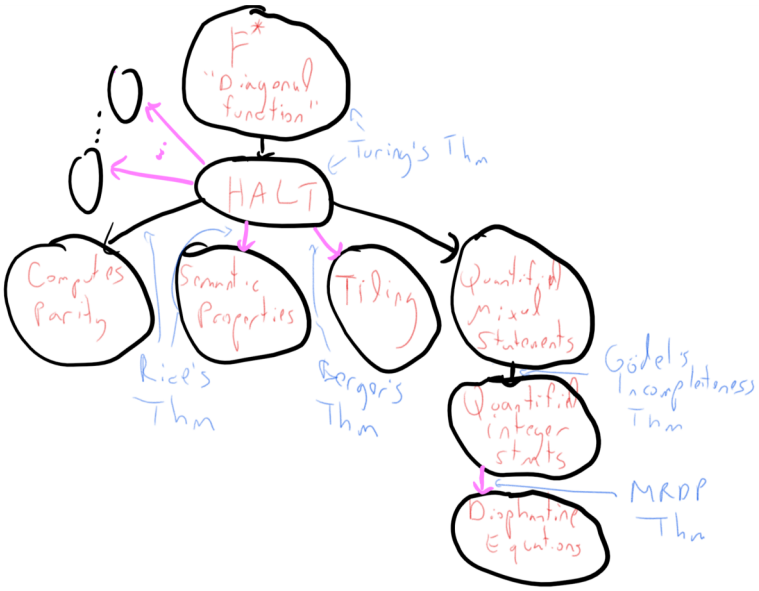

Figure 10.4: Some of the functions that have been proven uncomputable. An arrow from problem X to problem Y means that the proof that Y is uncomputable follows by reducing computing X to computing Y. Black arrows correspond to proofs that are shown in this and the next lecture while pink arrows correspond to proofs that are known but not shown here. There are many other functions that have been shown uncomputable via a reduction from the Halting function $HALT$.

that a given program conforms with such a specification is uncomputable. We start by proving a simple generalization of the Halting problem:

Theorem 10.3 — Halting without input. Let $HALTONZERO : \{0,1\}^* \to \{0,1\}$ be the function that on input $P \in \{0,1\}^*$, maps P to 1 if and only if the NAND++ program represented by P halts when supplied the single bit 0 as input. Then $HALTONZERO$ is uncomputable.

P The proof of Theorem 10.3 is below, but before reading it you might want to pause for a couple of minutes and think how you would prove it yourself. In particular, try to think of what a reduction from $HALT$ to $HALTONZERO$ would look like. Doing so is an excellent way to get some initial comfort with the notion of proofs by *reduction*, which is a notion that will recur time and again in this course.

Proof of Theorem 10.3. The proof is by reduction from $HALT$. Suppose, towards the sake of contradiction, that $HALTONZERO$ is

computable. In other words, suppose towards the sake of contradiction that there exists an algorithm A such that $A(P') = HALTONZERO(P')$ for every $P' \in \{0,1\}^*$. Then, we will construct an algorithm B that solves $HALT$.

On input a program P and some input x, the algorithm B will construct a program P' such that $P'(0) = P(x)$ and then feed this to A, returning $A(P')$. We will describe this algorithm in terms of how one can use the input x to modify the source code of P to obtain the source code of the program P'. However, it is clearly possible to do these modification also on the level of the string representations of the programs P and P'.

Constructing the program P' is in fact rather simple. The algorithm B will obtain P' by modifying P to ignore its input and use x instead. In particular, for $n = |x|$, the program P' will have have variables `myx_0`,...,`my_`$\langle n-1 \rangle$ that are set to the constants `zero` or `one` based on the value of x. That is, it will contain lines of the form `myx_`$\langle i \rangle$`:=` $\langle x_i \rangle$ for every $i < n$. Similarly, P' will have variables `myvalidx_0`,...,`myvalidx_`$\langle n-1 \rangle$ that are all set to `one`. Algorithm B will include in the program P' a copy of P modified to change any reference to `x_`$\langle i \rangle$ to `myx_`$\langle i \rangle$ and any reference to `validx_`$\langle i \rangle$ to `myvalidx_`$\langle i \rangle$. Clearly, regardless of its input, P' always emulates the behavior of P on input x. In particular P' will halt on the input 0 if and only if P halts on the input x. Thus if the hypothetical algorithm A satisfies $A(P') = HALTONZERO(P')$ for every P' then the algorithm B we construct satisfies $B(P,x) = HALT(P,x)$ for every P, x, contradicting the uncomputability of $HALT$. ■

> **R** **The hardwiring technique** In the proof of Theorem 10.3 we used the technique of "hardwiring" an input x to a program P. That is, modifying a program P that it uses "hardwired constants" for some of all of its input. This technique is quite common in reductions and elsewhere, and we will often use it again in this course.

Once we show the uncomputability of $HALTONZERO$ we can extend to various other natural functions:

Theorem 10.4 — Computing all zero function. Let $ZEROFUNC : \{0,1\}^* \to \{0,1\}$ be the function that on input $P \in \{0,1\}^*$, maps P to 1 if and only if the NAND++ program represented by P outputs 0 on every input $x \in \{0,1\}^*$. Then $ZEROFUNC$ is uncomputable.

Proof. The proof is by reduction to $HALTONZERO$. Suppose, towards the sake of contradiction, that there was an algorithm A such that $A(P') = ZEROFUNC(P')$ for every $P' \in \{0,1\}^*$. Then we will construct an algorithm B that solves $HALTONZERO$. Given a program P, Algorithm B will construct the following program P': on input $x \in \{0,1\}^*$, P' will first run $P(0)$, and then output 0.

Now if P halts on 0 then $P'(x) = 0$ for every x, but if P does not halt on 0 then P' will never halt on every input and in particular will not compute $ZEROFUNC$. Hence, $ZEROFUNC(P') = 1$ if and only if $HALTONZERO(P) = 1$. Thus if we define algorithm B as $B(P) = A(P')$ (where a program P is mapped to P' as above) then we see that if A computes $ZEROFUNC$ then B computes $HALTONZERO$, contradicting Theorem 10.3 . ■

We can simply prove the following:

Theorem 10.5 — Uncomputability of verifying parity. The following function is uncomputable

$$COMPUTES\text{-}PARITY(P) = \begin{cases} 1 & P \text{ computes the parity function} \\ 0 & \text{otherwise} \end{cases} \tag{10.1}$$

We leave the proof of Theorem 10.5 as an exercise.

10.2.1 *Rice's Theorem*

Theorem 10.5 can be generalized far beyond the parity function and in fact it rules out verifying any type of semantic specification on programs. We define a *semantic specification* on programs to be some property that does not depend on the code of the program but just on the function that the program computes.

For example, consider the following two C programs

```
int First(int k) {
   return 2*k;
}

int Second(int n) {
   int i = 0;
   int j = 0
   while (j<n) {
      i = i + 2;
```

```
        j= j + 1;
    }
    return i;
}
```

`First` and `Second` are two distinct C programs, but they compute the same function. A *semantic* property, such as "computing a function $f : \mathbb{N} \to \mathbb{N}$ where $f(m) \geq m$ for every m", would be either *true* for both programs or *false* for both programs, since it depends on the *function* the programs compute and not on their code. A *syntactic* property, such as "containing the variable `k`" or "using a `while` operation" might be true for one of the programs and false for the other, since it can depend on properties of the programs' *code*.

Often the properties of programs that we are most interested in are the *semantic* ones, since we want to understand the programs' functionality. Unfortunately, the following theorem shows that such properties are uncomputable in general:

Theorem 10.6 — Rice's Theorem (slightly restricted version). We say that two strings P and Q representing NAND++ programs *have the same functionality* if for every input $x \in \{0,1\}^*$, either the programs corresponding to both P and Q don't halt on x, or they both halt with the same output.

We say that a function $F : \{0,1\}^* \to \{0,1\}$ is *semantic* if for every P and Q that have the same functionality, $F(P) = F(Q)$. Then the only semantic computable total functions $F : \{0,1\}^* \to \{0,1\}$ are the constant zero function and the constant one function.

Proof. We will illustrate the proof idea by considering a particular semantic function F. Define $MONOTONE : \{0,1\}^* \to \{0,1\}$ as follows: $MONOTONE(P) = 1$ if there does not exist $n \in \mathbb{N}$ and two inputs $x, x' \in \{0,1\}^n$ such that for every $i \in [n]$ $x_i \leq x'_i$ but $P(x)$ outputs 1 and $P(x') = 0$. That is, $MONOTONE(P) = 1$ if it's not possible to find an input x such that flipping some bits of x from 0 to 1 will change P's output in the other direction from 1 to 0. We will prove that $MONOTONE$ is uncomputable, but the proof will easily generalize to any semantic function. For starters we note that $MONOTONE$ is not actually the all zeroes or all one function:

- The program INF that simply goes into an infinite loop satisfies $MONOTONE(INF) = 1$, since there are no inputs x, x' on which $INF(x) = 1$ and $INF(x') = 1$.
- The program PAR that we've seen, which computes the XOR or

parity of its input, is not monotone (e.g., $PAR(1,1,0,0,\ldots,0) = 0$ but $PAR(1,0,0,\ldots,0) = 0$) and hence $MONOTONE(PAR) = 0$.

(It is important to note that in the above we talk about *programs* INF and PAR and not the corresesponding functions that they compute.)

We will now give a reduction from $HALTONZERO$ to $MONOTONE$. That is, we assume towards a contradiction that there exists an algorithm A that computes $MONOTONE$ and we will build an algorithm B that computes $HALTONZERO$. Our algorithm B will work as follows:

1. On input a program $P \in \{0,1\}^*$, B will construct the following program Q: "on input $z \in \{0,1\}^*$ do: a. Run $P(0)$, b. Return $PAR(z)$".
2. B will then return the value $1 - A(Q)$.

To complete the proof we need to show that B outputs the correct answer, under our assumption that A computes $MONOTONE$. In other words, we need to show that $HALTONZERO(P) = 1 - MONOTONE(Q)$. However, note that if P does *not* halt on zero, then the program Q enters into an infinite loop in step a. and will never reach step b. Hence in this case the program Q has the same functionality as INF.[5] Thus, $MONOTONE(Q) = MONOTONE(INF) = 1$. If P *does* halt on zero, then step a. in Q will eventually conclude and Q's output will be determined by step b., where it simply outputs the parity of its input. Hence in this case, Q computes the non-monotone parity function, and we get that $MONOTONE(Q) = MONOTONE(PAR) = 0$. In both cases we see that $MONOTONE(Q) = 1 - HALTONZERO(P)$, which is what we wanted to prove. An examination of this proof shows that we did not use anything about $MONOTONE$ beyond the fact that it is semantic and non-trivial (in the sense that it is not the all zero, nor the all-ones function). ■

[5] Note that the program Q has different code than INF. It is not the same program, but it does have the same behavior (in this case) of never halting on any input.

10.2.2 Is software verification doomed?

Programs are increasingly being used for mission critical purposes, whether it's running our banking system, flying planes, or monitoring nuclear reactors. If we can't even give a certification algorithm that a program correctly computes the parity function, how can we ever be assured that a program does what it is supposed to do? The key insight is that while it is impossible to certify that a *general* program conforms with a specification, it is possible to write a program

in the first place in a way that will make it easier to certify. As a trivial example, if you write a program without loops, then you can certify that it halts. Also, while it might not be possible to certify that an *artbirary* program computes the parity function, it is quite possible to write a particular program P for which we can mathematically *prove* that P computes the parity. In fact, writing programs or algorithms and providing proofs for their correctness is what we do all the time in algorithms research.

The field of *software verification* is concerned with verifying that given programs satisfy certain conditions. These conditions can be that the program computes a certain function, that it never writes into a dangeours memory location, that is respects certain invariants, and others. While the general tasks of verifying this may be uncomputable, researchers have managed to do so for many interesting cases, especially if the program is written in the first place in a formalism or programming language that makes verification easier. That said, verification, especially of large and complex programs, remains a highly challenging task in practice as well, and the number of programs that have been formally proven correct is still quite small. Moreover, even phrasing the right theorem to prove (i.e., the specification) if often a highly non-trivial endeavor.

10.3 Lecture summary

- Unlike the finite case, there are actually functions that are *inherenently uncomputable* in the sense that they cannot be computed by *any* NAND++ program.
- These include not only some "degenerate" or "esoteric" functions but also functions that people have deeply cared about and conjectured that could be computed.
- If the Church-Turing thesis holds then a function F that is uncomputable according to our definition cannot be computed by any finite means.

10.4 Exercises

Exercise 10.1 — Halting problem. Give an alternative, more direct, proof for the uncomputability of the Halting problem. Let us define $H : \{0,1\}^* \rightarrow \{0,1\}$ to be the function such that $H(P) = 1$ if, when we interpret P as a program, then $H(P)$ equals 1 if $P(P)$ halts

(i.e., invoke P on its own string representation) and $H(P)$ equals 0 otherwise. Prove that there no program P^* that computes H, by building from such a supposed P^* a program Q such that, under the assumption that P^* computes H, $Q(Q)$ halts if and only if it does not halt.[6] ■

[6] **Hint:** See Christopher Strachey's letter in the biographical notes.

Exercise 10.2 For each of the following two functions, say whether it is decidable (computable) or not:

1. Given a NAND++ program P, an input x, and a number k, when we run P on x, does the index variable `i` ever reach k?
2. Given a NAND++ program P, an input x, and a number k, when we run P on x, does P ever write to an array at index k?

■

10.5 Bibliographical notes

[7]

[7] TODO: Add letter of Christopher Strachey to the editor of The Computer Journal. Explain right order of historical achievements. Talk about intuitionistic, logicist, and formalist approaches for the foundations of mathematics. Perhaps analogy to veganism. State the full Rice's Theorem and say that it follows from the same proof as in the exercise.

The diagonlization argument used to prove uncomputability of F^* is of course derived from Cantor's argument for the uncountability of the reals. In a twist of fate, using techniques originating from the works Gödel and Turing, Paul Cohen showed in 1963 that Cantor's Continuum Hypothesis is independent of the axioms of set theory, which means that neither it nor its negation is provable from these axioms and hence in some sense can be considered as "neither true nor false".[8] See here for recent progress on a related question.

[8] The Continuum Hypothesis is the conjecture that for every subset S of $\mathbb{R}$, either there is a one-to-one and onto map between S and $\mathbb{N}$ or there is a one-to-one and onto map between S and $\mathbb{R}$. It was conjectured by Cantor and listed by Hilbert in 1900 as one of the most important problems in mathematics.

10.6 Further explorations

Some topics related to this lecture that might be accessible to advanced students include: (to be completed)

10.7 Acknowledgements