

# Computability and Decidability

Klaus Sutner  
Carnegie Mellon University

Fall 2011

# Outline

- 1 Computability
- 2 Decidability
- 3 Halting
- 4 A Hierarchy

## Recall: Computability

- Register machines can be used to define RM-computable functions. We will often simply refer to these functions as **computable functions**.
- We are interested in computable functions as solutions for certain types of **computational problems**.
- There are three basic types of such problems (this small taxonomy is important in particular in complexity theory).

## Decision Problems

### Definition

A **decision problem** consists of a set of **instances** and a subset of **Yes-instances**.

Problem:	<b>Primality</b>
Instance:	A natural number $x$ .
Question:	Is $x$ a prime number?

Here the set of all instances is  $\mathbb{N}$  and the set of Yes-instances is  $\{p \in \mathbb{N} \mid p \text{ prime}\}$ .

The answer (solution) to any decision problem is just one bit (true or false).

## Function Problems

### Definition

A **function problem** consists of a set of **instances** and, for each instance  $I$ , a **solution**  $\text{sol}(I)$ .

Problem:     **Next Prime**  
Instance:     A natural number  $x$ .  
Solution:     The least prime  $p > x$ .

Instances are again  $\mathbb{N}$  and the solution for  $x \in \mathbb{N}$  is  $\text{sol}(x) = p$  where  $p$  is the appropriate prime (uniquely determined).

We insist that there always is a solution (otherwise  $\text{sol}$  would be a partial function).

## Search Problems

### Definition

A **search problem** consists of a set of **instances** and, for each instance  $I$ , a set of **solutions**  $\text{sol}(I)$ .

So here “the” solution is not required to be unique. And, we allow for  $\text{sol}(I)$  to be empty. Very convenient in practice.

Problem:     **Factor**

Instance:    A natural number  $x$ .

Solution:    A natural number  $z$ ,  $1 < z < x$ , dividing  $x$ .

Note that the set of solutions is empty if  $x$  is prime.

## Connections

Note that **Primality** and **Next Prime** are closely connected: an algorithm for one problem can be turned into an algorithm for the other.

**Factor** is a bit different, though: even if we can test primality easily it is not at all clear how to produce a factor (if the given number fails to be prime).

In fact, we now know that Primality is easy in the sense that there is a polynomial time algorithm for it, but we fervently hope that Factor will turn out to be hard (certain cryptographic methods will fail otherwise).

## Instances (or: Don't Cheat)

The input must be given in explicit, unobfuscated form: unary, binary, decimal, Roman numerals, . . . .

Not allowed are tricks like the following:

*Encode natural number  $n$  as  $2n + 1$  whenever  $n$  is prime; and as  $2n$  when  $n$  is compound.*

With this “input convention” primality testing would be trivial, but the coding procedure requires a lot of computational effort.

A similar trick could be used to trivialize any decision problem.

## Solving a Problem

- To solve a **decision problem**, an algorithm has to accept each instance of the problem as input, and return “Yes” or “No” depending on whether the instance is a Yes-instance.
- To solve a **function problem**, an algorithm has to accept each instance  $I$  of the problem as input, and return the unique  $\text{sol}(I)$ .
- To solve a **search problem**, an algorithm has to accept each instance  $I$  of the problem as input, and return either an element of  $\text{sol}(I)$  or “No” if  $\text{sol}(I)$  is empty (the instance has no solutions).
- We interpret “algorithm” as “computable function.”

- Computability

- ② Decidability

- Halting

- A Hierarchy

## Decision as Function Problems

Our definition of computability focuses on functions. However, we also need to be able to talk about sets and relations.

### Definition

The **characteristic function**  $\text{char}_R : \mathbb{N}^k \rightarrow \{0, 1\}$  of a relation  $R \subseteq \mathbb{N}^k$  is defined by:

$$\text{char}_R(x) = \begin{cases} 1 & x \in R \\ 0 & \text{otherwise.} \end{cases}$$

So a characteristic function is just a glorified bit-vector.

For example, the characteristic function  $\mathbb{N} \rightarrow \mathbf{2}$  of the set of primes looks like this

0011010100010100010100...

## Decidable Sets and Relations

### Definition

A set  $R \subseteq \mathbb{N}^k$  is **decidable** if the characteristic function  $\text{char}_R$  is computable.

Likewise, one says that a decision problem is **solvable** if the set of Yes-instances is decidable.

This simply means that there is some algorithm that, given an instance of the problem, computes for a finite amount of time, and then returns the correct Yes or No answer.

## Again: Instances

One might wonder what a decision algorithm is supposed to do with input that is not an instance of the problem in question.

There are two choices:

- We don't care: the behavior of the algorithm can be arbitrary.
- More realistic: the algorithm rejects bad input. This is perfectly reasonable since the collection of all inputs is always decidable. Something very fishy is going on if it is not.

## Canonical Representations

One might worry that dealing with specific data structures is messy, but in all practical cases there seems to be a canonical, natural choice. Essentially, all that is needed is:

- Natural numbers are given in binary.
- Nested lists of objects (hereditarily finite lists).

The sequence numbers from last week can handle the encoding of all finitary objects; in a natural, effective (computationally safe) manner.

Of course, efficiency is an entirely different story. If you want to get mileage out of existing physical realizations of computation you need something like  $\mathbb{C}$ .

## Closure Properties

### Lemma

*The decidable sets are closed under intersection, union and complement.*

*Proof.*

Consider two decidable sets  $A, B \subseteq \mathbb{N}^k$ . We have two RM-programs  $P_A$  and  $P_B$  that decide membership.

Idea: Run both  $P_A$  and  $P_B$  on input  $x$ , returning output  $b_A$  and  $b_B$  where  $0 \leq b_A, b_B \leq 1$ .

For intersection return  $\min(b_A, b_B)$ ,

for union return  $\max(b_A, b_B)$ ,

for the complement of  $A$  return  $1 - b_A$ .

□

## Digression: Boolean Algebra

Note that we have used arithmetic to express logical operations.

conjunction     $\min(b_A, b_B)$      $b_A \wedge b_B$

disjunction     $\max(b_A, b_B)$      $b_A \vee b_B$

negation         $1 - b_A$              $\neg b_A$

The arithmetic can easily be handled by a register machine.

The fact that arithmetic can express logic to some degree is the cause of endless headaches – and the source of beautiful hacks in languages such as C.

## How Bad Can It Be?

Here is an innocent question:

Is every decision problem  $A \subseteq \mathbb{N}$  decidable?

If this were true we could construct, for any  $A \subseteq \mathbb{N}$ , a RM-program  $P_A$  that, on input any number  $x \in \mathbb{N}$ , halts with output  $\text{char}_A(x)$ .

Unfortunately, the answer is **NO**.

Here is a cardinality argument due to Cantor that shows that there are lots of undecidable problems.

## Counting

### Theorem

*There are undecidable decision problems.*

*Proof.*

There are uncountably many subsets of  $\mathbb{N}$  (to be precise:  $2^{\aleph_0}$ ).

But there are only countably many RM-programs (recall our coding machinery).

Hence uncountably many problems  $A \subseteq \mathbb{N}$  are not decidable.

□

Note that this argument leaves a bitter after-taste: it does not produce any concrete undecidable problem. Here is a more constructive approach.

- Computability
- Decidability
- ③ Halting
  - A Hierarchy

# Halting

We would like a concrete decision problem that is undecidable.  
Preferably one that is “natural.”

We will see many examples in a while, but it is not entirely clear how to get started: what is the most obvious undecidable problem?

It seems like a fair guess that we should consider register machines themselves – after all, we want a problem that cannot be handled by any register machine whatsoever.

As it turns out, the right question is: “Does a given RMP halt on some particular input?”

## Enumerating Machines

Using the sequence number machinery from last time, we can effectively enumerate all possible register machines as a list  $(P_e)_{e \geq 0}$ .

Effective means that the function  $e \mapsto P_e$  is “computable.”

Why the quotation marks? Because we have only defined computability for  $\mathbb{N}$ , and register machine programs are not numbers—they could be considered ASCII texts or some such. So, strictly speaking, to say that the map  $e \mapsto P_e$  is computable is simply meaningless.

## Effectiveness

What we mean to say is that if we went to the trouble of defining computability not just on  $\mathbb{N}$  but on larger discrete universes, then the map would turn out to be computable. We won't bother.

The important part is that the translation from  $\mathbb{N}$  to RMPs is entirely natural. For example, we could:

- Sort all RMPs in length-lex order, let  $P_e$  be the  $e$ th program on this list.
- Let  $P_e$  be the program  $Q$  if the sequence number of  $Q$  is  $e$ . Otherwise,  $P_e$  is `inc 0 0`.

# Indices

## Definition

The number  $e$  is called an **index** for program  $P_e$  (with respect to the given enumeration).

So an index is simply a translation of a program into the world of arithmetic.

We can compute on these indices. For example, there is a computable function  $f$  such that  $f(e)$  is the number of registers used in  $P_e$ .

Or there is a computable function  $g$  such that  $P_{g(e)}$  is a program that computes the same function as  $P_e$  but conforms to certain syntactic constraints (input/output registers nicely numbered, no bad labels, never gets stuck except in halting instructions, ...)

## Convergence, A Pain

Working with computable functions or programs one has to deal with non-terminating computations.

Write  $P(x) \downarrow$  if  $P$  on input  $x$  terminates and produces some output, and  $P(x) \uparrow$  when the computation fails to terminate.

Note that  $P(x) = Q(x)$  should be interpreted as:

- either  $P(x) \downarrow$  and  $Q(x) \downarrow$  and the output is the same; or
- $P(x) \uparrow$  and  $Q(x) \uparrow$ .

## Total Recall: URM

We have seen how to construct a universal register machine  $\mathcal{U}$ : on input  $e = \langle P \rangle$  and  $x$ , the URM simulates  $P$  on input  $x$ .

More precisely,

- the URM takes as input an index  $e$  and an argument  $x$ , simulates  $P_e$  on  $x$  and
- if  $P_e(x) \downarrow$  returns the proper output, and
- fails to terminate if  $P_e(x) \uparrow$ .

## The Halting Problem

Problem:     **Halting**  
Instance:    Index  $e$  and input  $x$ .  
Question:    Does program  $P_e$  halt on input  $x$ ?

Note that this is a perfectly well-defined decision problem; the instance here consists of two natural numbers.

The answer for a particular instance depends on the underlying enumeration ( $P_e$ ) but that will not affect the decidability status of Halting.

# Halting is Undecidable

## Theorem

*The Halting Problem is undecidable.*

*Proof.*

Otherwise we could write a RM-program  $Q$  that, on input  $e$ ,

- halts on output 0 if  $P_e$  on input  $e$  fails to halt, and
- halts on output  $y + 1$  if  $P_e$  on input  $e$  halts on output  $y$ .

$Q$  is a RM-program, so it has some index  $q$ .

But running  $Q = P_e$  on input  $q$  produces a contradiction.

## The Contradiction

First note that  $Q$  is total: it halts on all inputs by definition.

But then

$$Q(q) = P_q(q) + 1 = Q(q) + 1$$

Contradiction.



## In Pseudo-Code

```
// machine Q

if( halt(e,e) )
    return eval(e,e) + 1;
else
    return 0;
```

Here  $\text{halt}(e,x)$  is the halting tester that exists by assumption, and  $\text{eval}(e,x)$  is just the URM: run  $P_e$  on input  $x$ .

## Diagonalization

This argument is a modification of Cantor's set-theory proof that the real numbers are uncountable.



It is rather surprising that the entirely non-constructive cardinality argument due to Cantor also turns out to be the main tool in establishing non-computability results.

## A Variant

It follows from the proof that the apparently easier version

Problem:     **Halting II**  
Instance:    Index  $e$ .  
Question:    Does program  $P_e$  halt on input  $e$ ?

is also undecidable.

It does not help at all that the argument in the full version here is fixed to be  $x = e$ .

## No Way Around It

Note that the Halting problems are always undecidable, no matter how the effective enumeration is chosen.

For example, there is no convenient enumeration  $Q_e$  such that  $Q_e$  halts on  $e$  iff  $e$  is even.

Any complicated property of programs translates into a complicated property of indices.

A similar problem arises in complexity theory. For example, there seems to be no reasonable way to encode finite graphs as numbers so that a graph is Hamiltonian iff its index is even. The same is true for more realistic data structures.

## What's Really Going On?

There is an easily decidable relation  $T(e, x, t)$  and an easily computable function  $U$  such that

$$P_e(x) \downarrow \iff \exists t T(e, x, t)$$

$$P_e(x) = U(\min(t \mid T(e, x, t)))$$

$T(e, x, t)$  essentially means: program  $P_e$  on input  $x$  halts after at most  $t$  steps. More precisely ...

## Kleene's $T$ Predicate

$T(e, x, t)$

- $e$  the index of a register machine  $P$
- $x$  an argument for  $P$
- $t$  a sequence number that codes the halting computation of  $P$  on input  $x$ .

$T$  is decidable since, given a coded sequence of configurations  $C_0, C_1, \dots, C_n$  it is not hard to check whether this really is a proper halting computation of  $P$  on input  $x$ .

Given the right  $t$  it is easy to read off the output of the computation.

## Unbounded Search

So where does undecidability come in?

The reason Kleene's  $T$  fails to yield the decidability of halting is that we cannot bound the search for  $t$  in any computable fashion: given  $e$  and  $x$  there is no way to predict the number of steps in the computation of  $P_e$  on  $x$ .

Note that this is not a problem in any real algorithm: given some particular input we can always compute ahead of time how long the algorithm will run.

## Still Unhappy?

Halting may seem like a somewhat unsatisfactory example of an undecidable problem: it already involves computable functions, it does not deal with a question at least superficially unrelated to computability.

Though anyone who has ever written code would have to admit that Halting is really quite natural.

How about other undecidable problems that are of independent interest? Perhaps something that was studied even before the concept of an algorithm was invented?

## Hilbert's 10th Problem

Perhaps the most famous example of an undecidability result in mathematics is Hilbert's 10th problem, the insolubility of Diophantine equations.

A Diophantine equation is a polynomial equation with integer coefficients:

$$P(x_1, x_2, \dots, x_n) = 0$$

The problem is to determine whether such an equation has an **integer solution**.

**Theorem (Y. Matiyasevic, 1970)**

*It is undecidable whether a Diophantine equation has an integer solution.*

## The Proof

The proof is too complicated to be presented here, but the main idea is simple enough:

*Show that decidability of Hilbert's 10th problem implies decidability of the Halting problem.*

This technique of reducing one (hard) problem to another is hugely important in complexity theory.

It works (to a degree) even if problems are not provably hard: at least lots of other problems are just as hard.

## Bounding Roots

One might think that integer roots of  $P(x_1, \dots, x_n)$  should be bounded by some reasonably simple function of  $n$ , the degree  $d$  of  $P$ , and the largest coefficient.

Perhaps something wacko like

$$(n! \max(|c_i|))^{n^{d+17}}$$

More precisely, if we could compute a bound on the size of the potential roots every Diophantine set would be decidable:

- First compute the bound on the size, then
- do a brute-force search over the finitely possible values.

## No Way

Alas, finding bounds even in concrete cases turns out to be quite difficult. Try to find a solution for

$$x^2 - 991y^2 - 1 = 0.$$

Of course,  $x = 1, y = 0$  is a trivial solution. The smallest positive solution here is

$$x = 379516400906811930638014896080$$

$$y = 12055735790331359447442538767$$

## Exercise (in futility)

*Try to find a positive solution to  $313(x^3 + y^3) = z^3$ .*

## Different Rings

Note that the choice of  $\mathbb{Z}$  as ground ring is important here. We can ask the same question for polynomial equations over other rings  $R$  (always assuming that the coefficients have simple descriptions).

- $\mathbb{Z}$ : undecidable
- $\mathbb{Q}$ : **major open problem**
- $\mathbb{R}$ : decidable
- $\mathbb{C}$ : decidable

Decidability of Diophantine equations over the reals is a famous result by A. Tarski from 1951, later improved by P. Cohen.

### Exercise

*Why does undecidability over  $\mathbb{Z}$  not simply imply undecidability over  $\mathbb{Q}$ ?  
What is the obstruction?*

- Computability

- Decidability

- Halting

- ④ A Hierarchy

## The Halting Set

Fix some effective enumeration  $(P_e)_{e \geq 0}$  of all RM-programs. Let

$$K = \{ e \in \mathbb{N} \mid P_e \text{ halts on } e \}$$

From our previous results,  $K$  is already undecidable.

- So how complicated is the Halting Problem?
- Are there more complicated problems, or is this as bad as it gets?

## Semi-Decidable Sets

### Definition

A set is **semi-decidable** if there is a RM-program for it that, on input  $x$ , halts if  $x$  belongs to the set, and fails to halt otherwise.

We will call such a method a **semi-decision procedure**.

### Lemma

*The Halting set  $K$  is semi-decidable.*

*Proof.* Use a URM to simulate the computation of  $P_e$  on input  $e$ . If  $P_e$  halts in the simulation, halt too. Otherwise just keep running forever.  $\square$

## Example: Diophantine Sets

Call a set  $S \subseteq \mathbb{N}$  **Diophantine** if there is a polynomial with integer coefficients  $P(z, \vec{x})$  such that

$$z \in S \iff P(z, \vec{x}) = 0 \text{ has integer solution}$$

Note that every Diophantine set is trivially semi-decidable.

Matiyasevic showed the opposite implication: every semi-decidable set is already Diophantine.

It was known prior to Matiyasevic's result that all semi-decidable sets are "Diophantine" with respect to exponential polynomials (which can involve terms  $x^y$ ), but getting rid of these is very difficult.

## Decidable vs. Semi-Decidable

### Lemma

*A set is decidable if, and only if, the set and its complement are both semi-decidable.*

*Proof.*

Let  $A$  be decidable. The program  $P$  that decides membership in  $A$  can easily be turned into two programs that semi-decide membership in  $A$  and  $\mathbb{N} - A$ .

For the opposite direction, suppose we have two programs  $P_1$  and  $P_2$  that semi-decide  $A$  and  $\mathbb{N} - A$ . Given  $x$ , run both programs in parallel on  $x$ . One of them must halt, output 0 or 1 correspondingly and halt.  $\square$

## Closure for Semi-Decidable

### Lemma

*The semi-decidable sets are closed under union and intersection.*

*Proof.*

Suppose we have two programs  $P_A$  and  $P_B$  that semi-decide  $A$  and  $B$ , and some input  $x$ .

For union, run both programs in parallel on  $x$ . If one of them halts, also halt.

For intersection, keep going until the second one also halts, then halt.  $\square$

## Aside: Complements

We make no claims about complement here, otherwise all semi-decidable sets would be decidable. Since  $K$  is semi-decidable but not decidable it follows from the lemma that  $\mathbb{N} - K$  is not even semi-decidable.

This is not an issue with decidable (as opposed to semi-decidable) sets: the complement of a decidable set is obviously again decidable. But semi-algorithms are very different from algorithms.

## Aside: Parallel Computation

The fact that we can interleave two computations is critical for this argument. Intuitively, this is fairly obvious but requires a bit of effort in a more formal context: a universal RM can take turns simulating two RMs.

This is the tip of an iceberg: we can simulate unboundedly many computations: at stage  $s$  we can simulate  $s$  steps of the computation of  $Q_e$ ,  $e < s$ , for all inputs  $x < s$ , given some effective list  $Q_e$ .

## Another Look

Fix some set  $A \subseteq \mathbb{N}$ . Suppose we add an instruction to our register machines:

- **query r k l**  
if  $[R_r] \in A$  goto  $k$ , otherwise goto  $l$ .

### Definition

A RM-machine with this added instruction is a **oracle register machine (ORM)** with oracle  $A$ .

We write  $P_e^A$  for the  $e$ th ORM with oracle  $A$ .

## Quoi?

- We are not interested in physical realizability here (though some flatliners in the hyper-computation community fail to comprehend that).
- If  $A$  is decidable we could actually build such an ORM: replace all the query instructions by calls to an RM that decides membership in  $A$ .
- At any rate, we can define what it means to be  $A$ -computable,  $A$ -decidable,  $A$ -semi-decidable by copying our old definitions, replacing RM by ORM everywhere.
- Note that  $K$  is trivially  $K$ -decidable. As is  $\mathbb{N} - K$ .

## Two Halting Sets

There are (at least) two natural ways to define a halting set:

$$K = \{ e \in \mathbb{N} \mid P_e \text{ halts on } e \}$$

$$K_0 = \{ \langle e, x \rangle \in \mathbb{N} \mid P_e \text{ halts on } x \}$$

- $K$  is clearly  $K_0$ -decidable: just check if  $\langle e, e \rangle \in K_0$ .
- But  $K_0$  is also  $K$ -decidable: Why?

## Halting Oracle Machines

So what what happens to the halting problem?

### Definition (The Jump)

The **jump** of  $A \subseteq \mathbb{N}$  is the halting set for ORM with oracle  $A$ :

$$A' = \{e \mid P_e^A(e) \downarrow\}$$

So  $\emptyset'$  is the ordinary halting set.

But  $\emptyset'' = K'$  is a new beast.

# Undecidability

## Lemma

$A'$  is  $A$ -semi-decidable but not  $A$ -decidable.

*Proof.*

The proof is verbatim the same as for the ordinary halting problem.

□

So we get an infinite hierarchy of more and more complicated problems:

$$\emptyset, \emptyset', \emptyset'', \emptyset''', \dots, \emptyset^{(n)}, \dots$$

## No Pipedream

Some of these iterated halting sets actually occur naturally in decision problems in mathematics. For example, consider the question

*Does  $P_e$  converge only on finitely many inputs?*

### Claim

*This problem is  $K$ -semi-decidable but not  $K$ -decidable.*

In reality, though, nothing much beyond  $\emptyset^{(4)}$  seems to play a role (except for problems entirely outside of this hierarchy).

## No End in Sight

- Even worse, we can collect all these sets into a single one, essentially by taking a disjoint union:

$$\emptyset^\omega = \{ \langle e, n \rangle \mid e \in \emptyset^{(n)} \}$$

- And nothing stops us from forming  $(\emptyset^\omega)'$ ,  $(\emptyset^\omega)''$  and so on. We can iterate the jump transfinitely often.
- Headache, anyone?

- We are only interested in combinatorial problems.
- Computability may seem to be an obvious notion, but its historical development required quite a bit of effort.
- Decidable and semi-decidable problems appear in many places in mathematics.
- The Halting Problem is the classical example of a semi-decidable problem that fails to be decidable.
- The notion of abstract computability does not translate directly into practical computability.
- But problems that are undecidable in general usually turn out to be very hard to deal with in practical special cases, too.