Languages and Machines

L10: Decidability (Part I)

Jorge A. Pérez

Bernoulli Institute for Mathematics, Computer Science, and AI
University of Groningen, Groningen, the Netherlands

May 23, 2024

Outline

Effective Computation

- How to distinguish between the computable and non-computable?

Effective Computation

- How to distinguish between the computable and non-computable?
- Mathematicians approached the notion via several formalisms:

Effective Computation

- How to distinguish between the computable and non-computable?
- Mathematicians approached the notion via several formalisms:
  - ▶ Turing machines (TMs)
  - ▶ Combinatory logic
  - ▶ The λ-calculus (functional programming!)

  Then 'computable' would mean computable by, e.g., a TM.

Effective Computation

- How to distinguish between the computable and non-computable?
- Mathematicians approached the notion via several formalisms:
  - ▶ Turing machines (TMs)
  - ▶ Combinatory logic
  - ▶ The λ-calculus (functional programming!)

  Then 'computable' would mean computable by, e.g., a TM.
- These formalisms are all equivalent — they embody the same notion of **effective computation**, from different angles (<u>Effective</u> as in: complete, mechanical, deterministic)

Effective Computation

- How to distinguish between the computable and non-computable?
- Mathematicians approached the notion via several formalisms:
  - Turing machines (TMs)
  - Combinatory logic
  - The λ-calculus (functional programming!)

  Then 'computable' would mean computable by, e.g., a TM.
- These formalisms are all equivalent — they embody the same notion of **effective computation**, from different angles (Effective as in: complete, mechanical, deterministic)
- Deterministic TMs are arguably closer to actual computers than the other formalisms

Church-Turing's Thesis

- While formalisms such as TMs, combinatory logic, λ-calculus, etc, are vastly dissimilar, they have a striking commonality
- **Effective computability**: they capture our intuition about what it means to be effectively computable — no more and no less

Church-Turing's Thesis

- While formalisms such as TMs, combinatory logic, λ-calculus, etc, are vastly dissimilar, they have a striking commonality
- **Effective computability**: they capture our intuition about what it means to be effectively computable — no more and no less
- **Church-Turing's thesis**: computability is not just TMs, nor Java, nor the λ-calculus, but the 'common spirit' they embody
- Not a theorem, but an observation (perhaps unsurprising today)

Church-Turing's Thesis

- While formalisms such as TMs, combinatory logic, λ-calculus, etc, are vastly dissimilar, they have a striking commonality
- **Effective computability**: they capture our intuition about what it means to be effectively computable — no more and no less
- **Church-Turing's thesis**: computability is not just TMs, nor Java, nor the λ-calculus, but the 'common spirit' they embody
- Not a theorem, but an observation (perhaps unsurprising today)
- TMs were a key step towards acceptance of the Church-Turing's thesis: they were the first readily programmable model
- Of course TMs do not address all possible aspects of computation (say,

Church-Turing's Thesis

- While formalisms such as TMs, combinatory logic, λ-calculus, etc, are vastly dissimilar, they have a striking commonality
- **Effective computability**: they capture our intuition about what it means to be effectively computable — no more and no less
- **Church-Turing's thesis**: computability is not just TMs, nor Java, nor the λ-calculus, but the 'common spirit' they embody
- Not a theorem, but an observation (perhaps unsurprising today)
- TMs were a key step towards acceptance of the Church-Turing's thesis: they were the first readily programmable model
- Of course TMs do not address all possible aspects of computation (say, interactivity or randomness)

Church-Turing's Thesis

- While formalisms such as TMs, combinatory logic, λ-calculus, etc, are vastly dissimilar, they have a striking commonality
- **Effective computability**: they capture our intuition about what it means to be effectively computable — no more and no less
- **Church-Turing's thesis**: computability is not just TMs, nor Java, nor the λ-calculus, but the 'common spirit' they embody
- Not a theorem, but an observation (perhaps unsurprising today)
- TMs were a key step towards acceptance of the Church-Turing's thesis: they were the first readily programmable model
- Of course TMs do not address all possible aspects of computation (say, interactivity or randomness) but they do capture a robust notion of effective computability

Universality

- **Programs as data**:
  TMs (but also all other models) are powerful enough that
  programs can be written to read/manipulate other programs
  (suitably encoded as data)
  - Think: Compilers and interpreters

Universality

- **Programs as data**:
  TMs (but also all other models) are powerful enough that
  programs can be written to read/manipulate other programs
  (suitably encoded as data)
  - Think: Compilers and interpreters
- TMs can interpret input strings as descriptions of other TMs
  (see next lecture!)
- A **universal machine** U is constructed to take an encoded
  description of another machine M and a string x as input.
  U can perform a step-by-step simulation of M on input x
- This is computers as we know them today!

Self-Reference

- A consequence of universality, and key to the discovery of uncomputable problems
- Observation: there are uncountably many **decision problems** but countably many TMs
- Extremely powerful: Gödel's incompleteness theorem, whose proof exploits self-reference
  (Idea: Construct the provable sentence "I am not provable")

Some Terminology

Recall: A TM is **always terminating** (or **total**) if it halts on (accepts or rejects) all inputs

A language (set of strings) L is

- **recursive**
  if $L = L(M)$ for some <u>always terminating</u> TM M
- **recursively enumerable (r.e.)**
  if $L = L(M)$ for some TM M

Some Terminology

Recall: A TM is **always terminating** (or **total**) if it halts on (accepts or rejects) all inputs

A language (set of strings) L is

- **recursive**
  if $L = L(M)$ for some <u>always terminating</u> TM M

- **recursively enumerable (r.e.)**
  if $L = L(M)$ for some TM M

Alternatively, let P be a **property** of strings.

- P is decidable
  if the set of all strings having P is recursive: there is a total TM that
  accepts strings that have P and rejects those that don't

- P is semi-decidable
  if the set of strings having P is r.e.: there is a TM that
  accepts x if x has P and *rejects or loops if not*

Some Terminology

Recursive and **recursively enumerable** are best applied to sets, while **decidable** and **semi-decidable** to properties

- Property P is decidable $\Leftrightarrow$ Set $\{x \mid P(x)\}$ is recursive
- Set A is recursive $\Leftrightarrow$ "$x \in A$" is decidable

Similarly:

- Property P is semi-decidable $\Leftrightarrow$ Set $\{x \mid P(x)\}$ is r.e.
- Set A is r.e. $\Leftrightarrow$ "$x \in A$" is semi-decidable

Outline

Decision problems

A question that expects an answer 'yes' or 'no', depending on some given **instance** (positive or negative).

We would like to have procedures (programs, TMs) to answer correctly this question in all cases.

Decision problems

A question that expects an answer 'yes' or 'no', depending on some given **instance** (positive or negative).

We would like to have procedures (programs, TMs) to answer correctly this question in all cases.

Examples:

1. Given a graph, is there a path between two of its nodes?
2. Is a natural n the difference between two prime numbers?
3. Given a CFG G and a string w, do we have $w \in L(G)$?
4. Given a CFG G, does $L(G)$ contain a palindrome?
5. Given a TM M and a string w, does it hold that $w \in L(M)$?
6. Given a program P, does the call of P with input I terminate?

Decidable and semi-decidable problems

A problem is

- **decidable** if there is a procedure (a program or TM) able to answer the question correctly in all cases
- **semi-decidable** if there is a procedure that
  - for every positive instance terminates with answer 'yes'
  - for every negative instance terminates with answer 'no' or loops

Examples

1. Given a graph, is there a path between two of its nodes?

2. Is a natural n the difference between two prime numbers?

3. Given a CFG G and a string w, do we have $w \in L(G)$?

4. Given a CFG G, does $L(G)$ contain a palindrome?

5. Given a TM M and a string w, does it hold that $w \in L(M)$?

6. **The halting problem:**
   Given a program P, does the call of P with input I terminate?

Examples

1. Given a graph, is there a path between two of its nodes?
   (decidable)
2. Is a natural n the difference between two prime numbers?

3. Given a CFG G and a string w, do we have $w \in L(G)$?
   (decidable)
4. Given a CFG G, does $L(G)$ contain a palindrome?

5. Given a TM M and a string w, does it hold that $w \in L(M)$?

6. **The halting problem:**
   Given a program P, does the call of P with input I terminate?

Examples

1. Given a graph, is there a path between two of its nodes?
   (decidable)
2. Is a natural n the difference between two prime numbers?
   (semi-decidable)
3. Given a CFG G and a string w, do we have $w \in L(G)$?
   (decidable)
4. Given a CFG G, does $L(G)$ contain a palindrome?

5. Given a TM M and a string w, does it hold that $w \in L(M)$?

6. **The halting problem:**
   Given a program P, does the call of P with input I terminate?

Examples

1. Given a graph, is there a path between two of its nodes?
   (decidable)
2. Is a natural n the difference between two prime numbers?
   (semi-decidable)
3. Given a CFG G and a string w, do we have $w \in L(G)$?
   (decidable)
4. Given a CFG G, does $L(G)$ contain a palindrome?
   (not decidable)
5. Given a TM M and a string w, does it hold that $w \in L(M)$?
   (not decidable)
6. **The halting problem:**
   Given a program P, does the call of P with input I terminate?
   (not decidable)

Outline

The halting problem for TMs (1/3)

Theorem
The halting problem for TMs is undecidable.
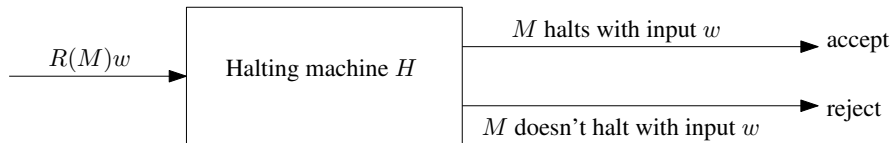
Idea for a proof by contradiction.

1. Assume there is a TM H that solves the halting problem.
A string is accepted by H if

- the input consists of two strings, $R(M)$ and w.
  $R(M)$ is the **representation** of a TM M, and w is the input to M

- the computation of M with input w halts.

Otherwise, H rejects the input.

The halting problem for TMs (1/3)

Theorem
The halting problem for TMs is undecidable.

Idea for a proof by contradiction.

1. Assume there is a TM H that solves the halting problem.
A string is accepted by H if

- the input consists of two strings, $R(M)$ and w.
  $R(M)$ is the **representation** of a TM M, and w is the input to M

- the computation of M with input w halts.

Otherwise, H rejects the input.

Graphically:

2. Modify H to build another TM, called K: the computations of K are the same as H, but K loops indefinitely whenever H terminates in an accepting state, i.e., whenever M halts on w.

The halting problem for TMs (2/3)

2. Modify H to build another TM, called K: the computations of K are the same as H, but K loops indefinitely whenever H terminates in an accepting state, i.e., whenever M halts on w.

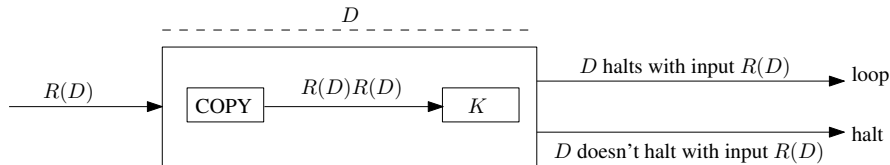3. Combine K with a "copy machine" to build another TM, called D, with $D(M) = K(M, M)$, as follows:
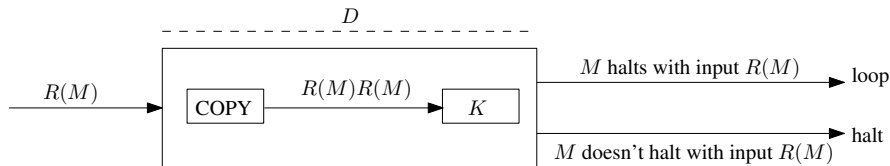
If the call $D(M)$ terminates, then the call $M(M)$ won't terminate

The halting problem for TMs (3/3)

4. The input to D may be the representation of any TM, even D itself. Adapting the diagram in the previous slide:

Thus, $D(D)$ terminates iff $D(D)$ doesn't terminate.
A contradiction, derived from the assumption that there is a machine H that solves the halting problem.

The halting problem, without input

A seemingly simpler problem, which is also undecidable.

Given a program P <u>without input</u>, is there a program Q that can decide whether or not P terminates?

1. Assume Q does indeed exist, and is an always terminating program with boolean output.
2. Hence, $Q(P)$ terminates iff the call P terminates.
3. Define a "linker" L: a program that calls program P_i with input I. That is, $L(P_i, I) = P_i(I)$.
4. $L(P_i, I)$ is a program without input, for any P_i and I.
5. Thus, $Q(L(P_i, I))$ terminates iff the call $P_i(I)$ terminates
6. Define a program Q' such that $Q'(P_i, I) = Q(L(P_i, I))$
7. Q' would decide the halting problem—a contradiction

Outline

Problems and languages

As mentioned earlier:

- Languages can be recursive or recursively enumerable
- Decision problems can be decidable or semi-decidable

We can relate problems and languages:

- Given a decision problem P, we can define a language L_P that consists of its positive instances.
- We need a function *encode* that transforms problem instances into a suitable alphabet.
- This way, the decision problem P is reduced to the problem of constructing a TM that accepts the language L_P.

Taking Stock

- Effective computation and Church-Turing's thesis
- Universality and self-reference
- A language (set of strings) is recursive or recursively enumerable
- A property can be decidable or semi-decidable
- Decision problems
- Accepting a language is a decision problem; every decision problem corresponds to a language (via an encoding function)
- The halting problem is not decidable, even without input

Taking Stock

- Effective computation and Church-Turing's thesis
- Universality and self-reference
- A language (set of strings) is recursive or recursively enumerable
- A property can be decidable or semi-decidable
- Decision problems
- Accepting a language is a decision problem; every decision problem corresponds to a language (via an encoding function)
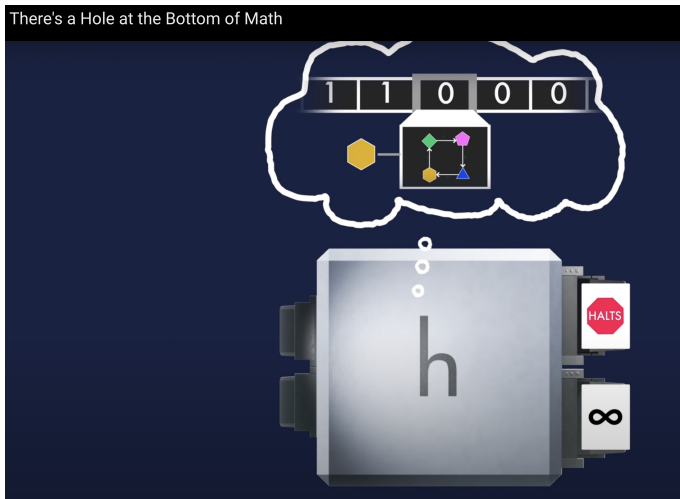- The halting problem is not decidable, even without input

Next lecture:

- A Universal Turing machine
- Acceptance of the empty string (the blank tape problem)
- Undecidability results

A Suggestion

You Can't Prove Everything That's True

https://www.youtube.com/watch?v=HeQX2HjkcNo