CS420

Introduction to the Theory of Computation

Lecture 23: Undecidable problems

Tiago Cogumbreiro

Today we will learn...

Decidability of

- The Halting Problem
- Emptiness for TM
- Regularity
- Equality

> Section 5.1

Recap

Decidable languages:

- $A_{DFA}, A_{REX}, A_{NFA}, A_{CFG}$

```
def A_DFA(D, w):
  return D accepts w
```

$$A_{DFA} = \{\langle D, w\rangle \mid D \text{ accepts } w\}$$

- E_{DFA}, E_{CFG}

```
def E_DFA(D):
  return L(D) = {}
```

$$E_{DFA} = \{\langle D\rangle \mid L(D) = \emptyset\}$$

- EQ_{DFA}

```
def EQ_DFA(D1, D2):
  return L(D1) = L(D2)
```

$$EQ_{DFA} = \{\langle N_1, N_2\rangle \mid L(N_1) = L(N_2)\}$$

Exercise 1

Prove or falsify the following statement: EQ_{REX} is undecidable.

Exercise 1

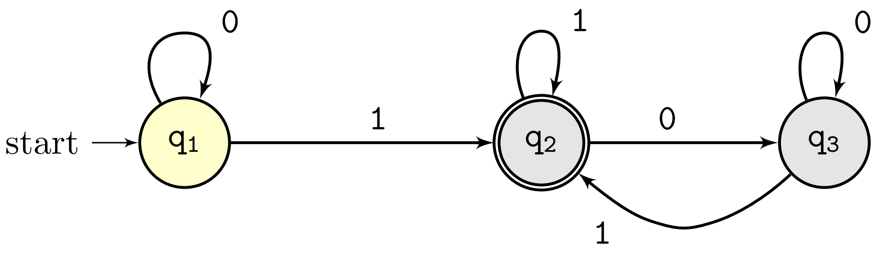
Prove or falsify the following statement: EQ_{REX} is undecidable.

Proof. False. EQ_{REX} is decidable, as given by the following pseudo code, where EQ_DFA is the decider of EQ_{DFA} and REX_TO_DFA is the conversion from a regular expression into a DFA.

```
def EQ_REX(R1, R2):
  return EQ_DFA(REX_TO_DFA(R1), REX_TO_DFA(R2))
```

Exercise 2

Let D be the DFA below




```
def A_DFA(D, w): return D accept w
def E_DFA(D): return L(D) == {}
def EQ_DFA(D1, D2): return L(D1) == L(D2)
```

- Exercise 2.1: Is $\langle D, 0100\rangle \in A_{DFA}$?
- Exercise 2.2: Is $\langle D, 101\rangle \in A_{DFA}$?
- Exercise 2.3: Is $\langle D\rangle \in A_{DFA}$?
- Exercise 2.4: Is $\langle D, 101\rangle \in A_{REX}$?
- Exercise 2.5: Is $\langle D\rangle \in E_{DFA}$?
- Exercise 2.6: Is $\langle D, D\rangle \in EQ_{DFA}$?
- Exercise 2.7: Is $101 \in A_{REX}$?

Exercise 3

Recall that DFAs are closed under $\cap$. Prove the following statement.

If A is regular, then X_A decidable.

$$X_A = \{\langle D \rangle \mid D \text{ is a DFA} \land L(D) \cap A \neq \emptyset\}$$

Exercise 3

> Recall that DFAs are closed under $\cap$. Prove the following statement.

If A is regular, then X_A decidable.

$$X_A = \{\langle D \rangle \mid D \text{ is a DFA} \wedge L(D) \cap A \neq \emptyset\}$$

Proof. If A is regular, then let C be the DFA that recognizes A. Let `intersect` be the implementation of $\cap$ and `E_DFA` the decider of E_{DFA}. The following is the decider of X_A.

```
def X_A(D):
  return not E_DFA(intersect(C, D))
```

Theorem 4.22

L decidable iff L recognizable and L co-recognizable

Theorem 4.22

L decidable iff L recognizable and L co-recognizable

Proof. We can divide the above theorem in the following three results.

1. If L decidable, then L is recognizable. **(Proved.)**
2. If L decidable, then L is co-recognizable. **(Proved.)**
3. If L recognizable and L co-recognizable, then L decidable.

Part 3. If L recognizable and $\overline{L}$ recognizable, then L decidable.

We need to extend our mini-language of TMs

`plet b ← P1 \\ P2 in P3`

Runs P1 and P2 in parallel.

- If P1 and P2 loop, the whole computation loops
- If P1 halts and P2 halts, pass the success of both to P3
- If P1 halts and P2 loops, pass the success of P1 to P3
- If P1 loops and P2 halts, pass the success of P2 to p3

```
Inductive par_result :=
| pleft: bool → par_result
| pright: bool → par_result
| pboth: bool → bool → par_result.
```

Part 3. If L recognizable and $\overline{L}$ recognizable, then L decidable.

Proof.

1. Let M_1 recognize L from assumption L recognizable
2. Let M_2 recognize $\overline{L}$ from assumption $\overline{L}$ recognizable
3. Build the following machine

```
Definition par_run M1 M2 w :=            (* M1 and M2 are parameters of the machine *)
  plet b ← Call M1 w \\ Call M2 w in     (* Call M1 with w and M2 with w in parallel *)
  match b with
  | pleft true    ⇒ ACCEPT
  | pboth true _  ⇒ ACCEPT               (* If M1 accepts w, accept *)
  | pright false ⇒ ACCEPT                (* If M2 rejects w, accept *)
  | _ ⇒ REJECT                           (* Otherwise, reject *)
  end.
```

4. Show that par_run M1 M2 recognizes L and is a decider.

Part 3. If L recognizable and $\overline{L}$ recognizable, then L decidable.

Point 4: Show that `par_run M1 M2` recognizes L and is a decider.

- 1. Show that `par_run M1 M2` recognizes L: `par_run M1 M2` accepts w iff $L(w)$
- 1.1. `par_run M1 M2` accepts w, then $w \in L$
- 1.2. $w \in L$, then `par_run M1 M2` accepts w case analysis on `run M2` with w

```
Definition par_run M1 M2 w :=
  plet b ← Call M1 w \\ Call M2 w in
  match b with
  | pleft true
  | pright false
  | pboth true _ ⇒ ACCEPT
  | _ ⇒ REJECT
  end.
```

- `M1` recognizes L
- `M2` recognizes $\overline{L}$
- Lemma `par_mach_lang`

Part 3. If L recognizable and $\overline{L}$ recognizable, then L decidable.

Point 4: Show that `par_run M1 M2` recognizes L and is a decider.

1. Show that `par_run M1 M2` recognizes L: `par_run M1 M2` accepts w iff $L(w)$
   1. If `par_run M1 M2` accepts w, then $w \in L$ by case analysis on `Call M1 w \\ Call M2 w`:
      - `M1` halts and `M2` loops. `M1` must accept, thus $w \in L$
      - `M2` halts and `M1` loops. `M2` must reject, but both cannot reject (contradiction).
      - `M1` and `M2` halt. `M1` must accept, thus $w \n L$.
   2. $w \in L$, then `par_run M1 M2` accepts w. `M1` accepts w. Case analysis call `M2` with w.
      - `M2` accept w: both cannot accept, contradiction.
      - `M2` reject w: par-call yields `pboth true false`, returns `Accept`.
      - `M2` loops w: par-call yields `bleft true`, returns `Accept`

(1) understand execution of a program by observing its output; (2) understand execution by observing its input

Part 3. If L recognizable and $\overline{L}$ recognizable, then L decidable.

Point 4: Show that `par_run M1 M2` recognizes L and is a decider.

2. Show that `par_run M1 M2` decides L

   (Walk through the proof of** `recognizable_co_recognizable_to_decidable`**...)

Homework 7 tutorial

Basic definitions

Run, recognizes

Running a Turing Machine

Use `run` to let a Turing `m` execute input `i`. Returns a `result`.

```
Inductive result := Accept | Reject | Loop.
```

Run, recognizes

Running a Turing Machine

Use `run` to let a Turing `m` execute input `i`. Returns a `result`.

```
Inductive result := Accept | Reject | Loop.
```

Recognizes

> A Turing machine `m` recognizes a language `L` if `m` accepts the same inputs as those in language `L`.

```
Definition Recognizes m L := forall i, run m i = Accept <-> L i.
```

- Use constructor `recognizes_def` to build `Recognizes m L`

Recognizable

Definition 3.5: Recognizable

> Call a language Turing-recognizable if some Turing machine recognizes it.

```
Definition Recognizable L := exists m, Recognizes m L.
```

- Use constructor `recognizable_def` to build `Recognizable L`

Decides

A Turing machine `m` decides a language L if:

1. `m` recognizes L
2. `m` is a decider

```
Definition Decides m L := Recognizes m L /\ Decider m.
```

- Use `decides_def` to build `Decides m L`

Decider

A Turing machine that never loops for all possible inputs.

```
Definition Decider m := forall i, run d i <> Loop.
```

- Use `decider_def` to build `Decider m`

Decidable

Definition 3.6

> Call a language Turing-decidable or simply decidable if some Turing machine decides it.

```
Definition Decidable L := exists m, Decides m L.
```

- Use `decidable_def` to build `Decidable L`

Summary

Term	*Usage*	*Coq*	*Constructor*
Run	run a TM with a given input i	`run m i`	N/A
Recognizes	a TM recognizes a language	`Recognizes m L`	`recognizes_def`
Recognizable	a language is recognizable	`Recognizable L`	`recognizable_def`
Decides	a TM decides a language	`Decides m L`	`decides_def`
Decider	a TM is a decider	`Decider m`	`decider_def`
Decidable	a language is decidable	`Decidable L`	`decidable_def`

Prog

A DSL for composing Turing Machines

Specifying TMs with `Prog`

- `Prog` is a **domain-specific** language (DSL) that allow us to compose Turing machines
- `Prog` gives an unique opportunity for CS420 students to study complex Theoretical Computer Science problems in a (hopefully) intuitive framework
- All theorems studied in this course are fully proved; students can see all details at their own time, interactively
- The proofs follow the structure of the book as close as possible

Did you know?

- `gitlab.com/cogumbreiro/turing` is a research project that stemmed from trying to teach CS420 in a more compelling way (project-based, + interactive, + student-autonomous)
- This semester we are pushing the state-of-the-art of teaching Theoretical Computer Science
- Your input matters!

Turing programs Prog

```
Inductive Prog :=
    Seq : Prog → (bool → Prog) → Prog
  | Call : machine → input → Prog
  | Ret : result → Prog.
```

- `Seq` combines two programs
- `Call` runs a Turing machine on a given input
- `Ret` loops/rejects/accepts (pick one) for all inputs

Turing programs Prog

Notations

We use 3 notations to write shorter programs:

```
mlet x ← p1 in p2 := Seq p1 (fun x ⇒ p2)
           ACCEPT := Ret Accept
           REJECT := Ret Reject
             LOOP := Ret Loop
```

P-run (part 1)

1. Rule `run_ret`: the result of returning r (with `Ret r`) is r

$$\frac{}{\texttt{Run}\ (\texttt{Ret}\ r)\ r}$$

2. The result of calling a TM `m` is given by calling `run m i`.

$$\frac{\mathbf{run}(m, i) = r}{\texttt{Run}(\texttt{Call}\ m\ i)\ r}$$

P-run (part 2)

3. If we run program p and get a result r_1 and p terminates with b and we run $(\texttt{p b})$ and get a result r_2, then sequencing p with q returns result r_2

$$\frac{\texttt{Run}\ p\ r_1 \qquad \texttt{Dec}\ r_1\ b \qquad \texttt{Run}\ (q\ b)\ r_2}{\texttt{Run}\ (\texttt{Seq}\ p\ q)\ r_2}$$

4. If program p loops, then running p followed by q also loops:

$$\frac{\texttt{Run}\ p\ \texttt{Loop}}{\texttt{Run}\ (\texttt{Seq}\ p\ q)\ \texttt{Loop}}$$

P-run in Coq

```
Inductive Run: Prog → result → Prop :=
| run_ret:
  forall r,
  Run (Ret r) r
| run_call:
  Run (Call m i) (run m i)
| run_seq_cont:
  forall p q b r1 r2,
  Run p r1 →
  Dec r1 b →
  Run (q b) r2 →
  Run (Seq p q) r2
| run_seq_loop:
  forall p q,
  Run p Loop →
  Run (Seq p q) Loop
```

Why do we need P-run?

- Because `Prog` is inductively defined, we can reason about all possible ways in which we can **declare** a program (induction proofs)
- Because `Run` is inductively defined, we can also reason about all possible ways in which we can **run** a program
- `Prog` is already being informally used in the book, we are just making the meta-theory more **formal**!
- Proofs are easier (homework assignments have less technicalities/distractions)

P-Recognizes

Program `p` P-recognizes a language `L` if `p` accepts the same inputs as those in language `L`.

```
Definition PRecognizes p L := forall i, Run (p i) Accept <-> L i
```

- Use `p_recognizes_def` to build `PRecognizes p L`

P-Recognizable

> Call a language P-recognizable if some `Prog` recognizes it.

- There is no definition `PRecognizable`! We use `Recognizable` still.
- Use `p_recognizable_def` to build `Recognizable L` with a program!

P-Decides

A program p P-decides a language L if:

1. p P-recognizes L
2. p is a P-decider

```
Definition PDecides p L := PRecognizes p L /\ PDecider p.
```

- Use `p_decides_def` to build `PDecides p L`

P-Decider

A program that never loops for all possible inputs.

```
Definition PDecider p := forall i, PHalts (p i).
```

- Use `p_decider_def` to build `PDecider p`

P-Halts

```
Definition PHalts p := exists r : result, Run p r /\ r <> Loop
```

- Use `p_halts_def` to build `PHalts p`.

P-Decidable

> Call a program P-decidable or simply decidable if some program decides it.

- There is no definition `PDecidable`! We use `Decidable` still.
- Use `p_decidable_def` to build `Decidable L`

Summary

Term	*Usage*	*Coq*	*Constructor*
P-Run	run a program with a given input `i` and result `r`	`Run p i r`	`Print Run.`
P-Recognizes	a program recognizes a language	`PRecognizes p L`	`p_recognizes_def`
P-Recognizable	a language is recognizable	`Recognizable L`	`p_recognizable_def`
P-Decides	a program decides a language	`PDecides p L`	`p_decides_def`
P-Decider	a program is a decider	`PDecider p`	`p_decider_def`
P-Decidable	a language is decidable	`Decidable L`	`p_decidable_def`