CS420

Introduction to the Theory of Computation

Lecture 22: Undecidability

Tiago Cogumbreiro

Today we will learn...

- Turing Machine theory in Coq
- Undecidability
- Unrecognizability

> Section 4.2

Turing Machine theory in Coq

Turing Machine theory in Coq

- **What?** I am implementing the Sipser book in Coq.
- **Why?**
  - So that we can dive into any proof at any level of detail.
  - So that you can inspect any proof and step through it on your own.
  - So that you can ask why and immediately have the answer.

Do you want to help out?

UMass Boston

Why is proving important to CS?

- **Generality is important.**
  Whenever we implement a program, we are implicitly proving some notion of correctness in our minds (the program is the proof).
- **Rigour is important.**
  The importance of having precise definitions. Fight ambiguity!
- **Assume nothing and question everything.**
  In formal proofs, we are pushed to ask why? And we have a framework to understand why.
- **Models are important.**
  The basis of formal work is abstraction (or models), e.g., Turing machines as models of computers; REGEX vs DFAs vs NFAs.

What follows is a description of our Coq implementation

Turing Machine Theory in Coq

Unspecified input/machines

For the remainder of this module we leave the input (string) and a Turing Machine unspecified.

```
Variable input: Type.
Variable machine: Type.
```

Turing Machine Theory in Coq

Running a TM

We can run any Turing Machine given an input and know whether or not it accepts, rejects a given input. We leave running a Turing Machine unspecified.

```
Parameter Exec: machine → input → bool → Prop.

Parameter exec_exists:
  forall m i,
  (exists b, Exec m i b) \/ (forall b, ~ Exec m i b).
```

Properties

- A machine may execute a return either `true` or `false`
- A machine may be unable to execute a given input (eg, the machine loops forever)

What is a language?

A language is a predicate: a formula parameterized on the input.

```
Definition lang := input → Prop.
```

Defining a set/language

Set builder notation

$$L = \{x \mid P(x)\}$$

Functional encoding

$$L(x) \stackrel{\texttt{def}}{=} P(x)$$

Defining membership

Set membership

$$x \in L$$

Functional encoding

$$L(x)$$

Example

Set builder example

$$L = \{a^n b^n \mid n \geq 0\}$$

Functional encoding

$$L(x) \stackrel{\texttt{def}}{=} \exists n, x = a^n b^n$$

The language of a TM

Set builder notation

The language of a TM can be defined as:

$$L(M) = \{w \mid M \text{ accepts } w\}$$

Functional encoding

$$L_M(w) \stackrel{\texttt{def}}{=} M \text{ accepts } w$$

In Coq

```
Definition Lang (m:machine) : lang := fun i ⇒ Exec m i true.
```

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/umb-svl/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

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

- `Call` runs a Turing machine on a given input (only needed for main results)
- `Ret` rejects/accepts (pick one) the given input
- `Seq p q` runs program p, if p terminates, then run q

  Notation:

  ```
  mlet x ← p1 in p2 ≡ Seq p1 (fun x ⇒ p2)
  ```

UMass Boston

Run (part 1)

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

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

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

$$\frac{\text{Exec}\ m\ i\ b}{\texttt{Run}(\texttt{Call}\ m\ i)\ b}$$

Run (part 2)

3. If we run program `p` and get a result r_1 and `p` terminates with `b` and we run `(p b)` and get a result r_2, then sequencing `p` with `q` returns result r_2

$$\frac{\texttt{Run}\ p\ b_1 \qquad \texttt{Run}\ (q\ b_1)\ b_2}{\texttt{Run}\ (\texttt{Seq}\ p\ q)\ b_2}$$

Run in Coq

```
Inductive Run: prog → bool → Prop :=
| run_call:
  (** Run a turing machine m. *)
  forall m i b,
  Exec m i b →
  Run (Call m i) b
| run_ret:
  (** We can directly return a result *)
  forall b,
  Run (Ret b) b
| run_seq:
  (** If p terminates and returns b, then we can
     proceed with the execution of q b. *)
  forall p q b1 b2,
  Run p b1 →
  Run (q b1) b2 →
  Run (Seq p q) b2.
```

```
Goal exists b, Run (Ret true) b. Proof. Admitted.

Goal exists b, Run (Ret false) b. Proof. Admitted.

Goal forall b, Run (Ret true) b → b = true. Proof. Admitted.

Goal exists b, Run (mlet x ← Ret true in Ret true) b. Proof. Admitted.

Goal exists b, Run (mlet x ← Ret true in Ret false) b. Proof. Admitted.

Goal forall p q b1, Run (mlet x ← p in q) b1 → exists b2, Run (mlet x ← q in p) b2.
Proof. Admitted.
```

```
Inductive Loop: prog → Prop :=
| loop_tur:
  (** When the turing machine loops, calling it loops *)
  forall m i,
  (forall b, ~ Exec m i b) →
  Loop (Call m i)
| loop_seq_l:
  (** If p terminates and returns b, then we can
      proceed with the execution of q b. *)
  forall p q,
  Loop p →
  Loop (Seq p q)
| loop_seq_r:
  (** If p terminates and returns b, then we can
      proceed with the execution of q b. *)
  forall p q b,
  Run p b →
  Loop (q b) →
  Loop (Seq p q).
```

```
Inductive Halt : prog → Prop :=
| halt_ret:
  (** We can directly return a result *)
  forall b,
  Halt (Ret b)
| halt_call:
  (** Run a turing machine m. *)
  forall m i b,
  Exec m i b →
  Halt (Call m i)
| halt_seq:
  (** If p terminates and returns b, then we can
     proceed with the execution of q b. *)
  forall p q b,
  Run p b →
  Halt (q b) →
  Halt (Seq p q).
```

Recognizes

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

```
Definition Recognizes (p: input → prog) (L:lang) :=
  forall i, Run (p i) true ↔ L i.
```

- Use `recognizes_def`, or `unfold` to build `Recognizes p L`

Recognizable

- Call a language (Turing-)recognizable if some `prog` recognizes it.

```
Definition Recognizable (L:lang) : Prop :=
  exists p, Recognizes p L.
```

Decides

A program `p` decides a language `L` if:

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

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

Decider

A program that never loops for all possible inputs.

```
Definition Decider (p:input → prog) := forall i, Halt (p i).
```

Decidable

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

Summary

Term	*Usage*	*Coq*	*Constructor*
Run	Run a program `p` that outputs `b`	`Run p b`	`Print Run.`
Recognizes	a program recognizes a language	`Recognizes p L`	`recognizes_def`
Recognizable	a language is recognizable	`Recognizable L`	`recognizable_def`
Decides	a program decides a language	`Decides p L`	`decides_def`
Decider	a program is a decider	`Decider p`	`decider_def`
Decidable	a language is decidable	`Decidable L`	`decidable_def`

Recognizes

We give a formal definition of recognizing a language. We say that M recognizes L if, and only if, M accepts w whenever $w \in L$.

```
Definition Recognizes (m:machine) (L:lang) := forall w, run m w = Accept <-> L w.
```

Examples

- Saying M recognizes $L = \{a^n b^n \mid n \geq 0\}$ is showing that there exist a proof that shows that all inputs in language L are accepted by M and vice-versa.
- Trivially, M recognizes $L(M)$.

We will prove 4 theorems

- Theorem 4.11 A_{TM} is undecidable
- Theorem 4.22 L is decidable if, and only if, L is recognizable ***and*** co-recognizable
- Corollary 4.23 $\overline{A}_{TM}$ is unrecognizable
- Corollary 4.18 Some languages are unrecognizable

Why?

- We will learn that we cannot write a program that decides if a TM accepts a string
- We can define decidability in terms of recognizability+complement
- There are languages that cannot be recognized by some program

Theorem 4.11

A_{TM} is undecidable

Proof idea

1. Assume solving A_{TM} is decidable and reach a contradiction.
2. Find a program for which it is impossible to decide

```python
def tricky(f):
  return not f(f)

print(tricky(lambda x: True)) # Output?
```

Proof idea

1. Assume solving A_{TM} is decidable and reach a contradiction.
2. Find a program for which it is impossible to decide

```
def tricky(f):
  return not f(f)

print(tricky(lambda x: True)) # Output?

# False
try:
   print(tricky(tricky)) # Output?
except RecursionError:
   print("could not run: tricky(tricky)")
```

Proof idea

1. Assume solving A_{TM} is decidable and reach a contradiction.
2. Find a program for which it is impossible to decide

```python
def tricky(f):
  return not f(f)

print(tricky(lambda x: True)) # Output?

# False
try:
   print(tricky(tricky)) # Output?
except RecursionError:
   print("could not run: tricky(tricky)")
```

Calling `tricky(tricky)` loops **forever**.

UMass Boston

Proof idea

Let the solver of A_{TM} be `returns_true` which takes a boolean function `f`, an argument `a`, and returns whether `f(a)` would return true. Function `returns_true` **halts** for every input.

```
def tricky_v2(f):
  return not returns_true(f, f)
```

1. What would the result of `tricky_v2(tricky_v2)` be?

Proof idea

Let the solver of A_{TM} be `returns_true` which takes a boolean function `f`, an argument `a`, and returns whether `f(a)` would return true. Function `returns_true` **halts** for every input.

```
def tricky_v2(f):
  return not returns_true(f, f)
```

1. What would the result of `tricky_v2(tricky_v2)` be?
2. Assume that `tricky_v2(tricky_v2)` **loops**

Proof idea

Let the solver of A_{TM} be `returns_true` which takes a boolean function `f`, an argument `a`, and returns whether `f(a)` would return true. Function `returns_true` **halts** for every input.

```
def tricky_v2(f):
  return not returns_true(f, f)
```

1. What would the result of `tricky_v2(tricky_v2)` be?
2. Assume that `tricky_v2(tricky_v2)` **loops**
3. `not return_true(tricky_v2, tricky_v2)` **loops**
   (replace function call by definition)

Proof idea

Let the solver of A_{TM} be `returns_true` which takes a boolean function `f`, an argument `a`, and returns whether `f(a)` would return true. Function `returns_true` **halts** for every input.

```
def tricky_v2(f):
  return not returns_true(f, f)
```

1. What would the result of `tricky_v2(tricky_v2)` be?
2. Assume that `tricky_v2(tricky_v2)` **loops**
3. `not return_true(tricky_v2, tricky_v2)` **loops**
   (replace function call by definition)
4. `not false` **loops**
   (`return_true(tricky_v2, tricky_v2) = false` from assumption 2)

Proof idea

Let the solver of A_{TM} be `returns_true` which takes a boolean function `f`, an argument `a`, and returns whether `f(a)` would return true. Function `returns_true` **halts** for every input.

```
def tricky_v2(f):
  return not returns_true(f, f)
```

1. What would the result of `tricky_v2(tricky_v2)` be?
2. Assume that `tricky_v2(tricky_v2)` **loops**
3. `not return_true(tricky_v2, tricky_v2)` **loops**
   (replace function call by definition)
4. `not false` **loops**
   (`return_true(tricky_v2, tricky_v2) = false` from assumption 2)
5. contradiction

Proof idea

1. Assume `tricky_v2(tricky_v2) = true`

Proof idea

1. Assume `tricky_v2(tricky_v2) = true`
2. `not return_true(tricky_v2, tricky_v2) = true`
   (replace function call by function body)

Proof idea

1. Assume `tricky_v2(tricky_v2) = true`
2. `not return_true(tricky_v2, tricky_v2) = true`
   (replace function call by function body)
3. `not true = true`
   (since from assumption 2, `return_true(tricky_v2, tricky_v2) = true`)