COMP1600, week 11:

Turing Machines:

Limits of Decidability

convenors: Dirk Pattinson, Pascal Bercher

lecturer: Pascal Bercher

slides based on those by: Dirk Pattinson

(with contributions by Victor Rivera and previous colleagues)

Australian
National
University

Semester 2, 2024

Overview of Week 11

- Introduction
- Recap on Languages – Based on Haskell
- Proving Program Properties (Example)
- Recursively Enumerable Problems
- (Un)Decidable Problems

Introduction

Disclaimer

Many of the slides this week use Haskell

- You should be able to read them even if you can't code Haskell! Basically like pseudo code...
- What we say about Haskell transfers to TMs, it's just a bit more "practical" that way.
- You do not need to be able write Haskell code,
  - neither to understand what's going on (as long as you can read it),
  - nor in the tutorial exercises or the assessments,
  - or in the exam or quiz.

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:

  equivalent terms:

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:
  - decidable problems (can "answer the question" in finite time)

equivalent terms: recursive

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:
  - decidable problems (can "answer the question" in finite time)
  - undecidable problems (the opposite of decidable!)

equivalent terms: recursive/non-recursive,

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:
  - decidable problems (can "answer the question" in finite time)
  - undecidable problems (the opposite of decidable!)
  - semi-decidable problems (can give "all yes-answers" in finite time)
    (note: can be decidable and semi-decidable, in fact, one implies the other)

  equivalent terms: recursive/non-recursive, recursively enumerable

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:
  - decidable problems (can "answer the question" in finite time)
  - undecidable problems (the opposite of decidable!)
  - semi-decidable problems (can give "all yes-answers" in finite time)
    (note: can be decidable and semi-decidable, in fact, one implies the other)
  - not even semi-decidable (i.e., TMs can't express them!)

  equivalent terms: recursive/non-recursive, recursively enumerable

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:
  - decidable problems (can "answer the question" in finite time)
  - undecidable problems (the opposite of decidable!)
  - semi-decidable problems (can give "all yes-answers" in finite time)
    (note: can be decidable and semi-decidable, in fact, one implies the other)
  - not even semi-decidable (i.e., TMs can't express them!)

  equivalent terms: recursive/non-recursive, recursively enumerable
- how these properties of *languages* relate to properties of TMs.

What this Week is About

You should learn/understand:

- The limits of computation: some problems even TM's can't express
  I.e., no matter how hard you try, the respective TM will make an error!
- Even those problems (i.e., languages) that can be "expressed" by TMs sometimes can't be "decided" (i.e., they might not terminate).
- More specifically, you should master the relationship between:
  - decidable problems (can "answer the question" in finite time)
  - undecidable problems (the opposite of decidable!)
  - semi-decidable problems (can give "all yes-answers" in finite time)
    (note: can be decidable and semi-decidable, in fact, one implies the other)
  - not even semi-decidable (i.e., TMs can't express them!)

  equivalent terms: recursive/non-recursive, recursively enumerable
- how these properties of *languages* relate to properties of TMs.

Call for action: Please engage with all tutorial exercises!!
Check the sample solutions carefully.

Recap on Languages – Based on Haskell

Language of Haskell Programs

Observation. Turing machines 'recognise' strings. Haskell functions of type `String -> Bool` also recognise strings. For a Haskell program

```
p :: String -> Bool
```

we can define $L(p) = \{\texttt{w} :: \texttt{String} \mid \texttt{p w} = \texttt{True}\}$.

Question. Given the Haskell programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

which of the following are true?

- $L(p)$ and $L(q)$ are the same, as non termination is non acceptance.
- $L(p)$ and $L(q)$ are not the same, as q does not always terminate.

Language of Haskell Programs, cont'd

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Recall. $L(p) = \{\texttt{w} :: \texttt{String} \mid \texttt{p w} = \texttt{True}\}$.

Q. If `p w` doesn't terminate, does it make sense to say that `p w = True`? Probbaly not. What about `p w = False`? Does it even matter?

Language of Haskell Programs, cont'd

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Recall. $L(p) = \{\texttt{w} :: \texttt{String} \mid \texttt{p w} = \texttt{True}\}$.

Q. If `p w` doesn't terminate, does it make sense to say that `p w = True`? Probbaly not. What about `p w = False`? Does it even matter?

Put differently:

- if `p w` does not terminate, then `w` is not in $L(p)$.
- if `p w` does terminate, and evaluates to `False`, then `w` is not in $L(p)$.
- The only way in which `w` can be in $L(p)$ is if `p w` terminates *and* evaluates to `True`.

Language of Haskell Programs, cont'd

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Recall. $L(p) = \{\texttt{w} :: \texttt{String} \mid \texttt{p w} = \texttt{True}\}$.

Q. If `p w` doesn't terminate, does it make sense to say that `p w = True`? Probbaly not. What about `p w = False`? Does it even matter?

Put differently:

- if `p w` does not terminate, then `w` is not in $L(p)$.
- if `p w` does terminate, and evaluates to `False`, then `w` is not in $L(p)$.
- The only way in which `w` can be in $L(p)$ is if `p w` terminates *and* evaluates to `True`.

Q. Now there's a slight difference to TMs. Which?

Language of Haskell Programs, cont'd

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Recall. $L(p) = \{\texttt{w} :: \texttt{String} \mid \texttt{p w} = \texttt{True}\}$.

Q. If `p w` doesn't terminate, does it make sense to say that `p w = True`? Probbaly not. What about `p w = False`? Does it even matter?

Put differently:

- if `p w` does not terminate, then `w` is not in $L(p)$.
- if `p w` does terminate, and evaluates to `False`, then `w` is not in $L(p)$.
- The only way in which `w` can be in $L(p)$ is if `p w` terminates *and* evaluates to `True`.

Q. Now there's a slight difference to TMs. Which?
A. TMs don't need to terminate (=halt) to accept.

Language of Haskell Programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Slogan.

Non-Termination or Termination with value False = Non-Acceptance.

For the programs above, that means $L(p) = L(q)$.

Language of Haskell Programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Slogan.

Non-Termination or Termination with value False = Non-Acceptance.

For the programs above, that means $L(p) = L(q)$.

TM Recap.

- In our definition, non-acceptance is the same as rejection.
  Others might define rejection as non-accept plus halting. (We don't!)

Language of Haskell Programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Slogan.

Non-Termination or Termination with value False = Non-Acceptance.

For the programs above, that means $L(p) = L(q)$.

TM Recap.

- In our definition, non-acceptance is the same as rejection.
  Others might define rejection as non-accept plus halting. (We don't!)
- Acceptance:
  - A TM accepts a word if we can reach an accepting state with it as input.

Language of Haskell Programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Slogan.

Non-Termination or Termination with value False = Non-Acceptance.

For the programs above, that means $L(p) = L(q)$.

TM Recap.

- In our definition, non-acceptance is the same as rejection.
  Others might define rejection as non-accept plus halting. (We don't!)
- Acceptance:
  - A TM accepts a word if we can reach an accepting state with it as input.
  - The language of a TM is the words accepts.

Language of Haskell Programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Slogan.

Non-Termination or Termination with value False = Non-Acceptance.

For the programs above, that means $L(p) = L(q)$.

TM Recap.

- In our definition, non-acceptance is the same as rejection.
  Others might define rejection as non-accept plus halting. (We don't!)
- Acceptance:
  - A TM accepts a word if we can reach an accepting state with it as input.
  - The language of a TM is the words accepts.
- Rejection:
  - can be *explicit* by halting without acceptance.

Language of Haskell Programs

```
p :: String -> Bool
p s  = even (length s)
```

```
q :: String -> Bool
q s | even (length s) = True
    | otherwise = q(s)
```

Slogan.

Non-Termination or Termination with value False = Non-Acceptance.

For the programs above, that means $L(p) = L(q)$.

TM Recap.

- In our definition, non-acceptance is the same as rejection.
  Others might define rejection as non-accept plus halting. (We don't!)
- Acceptance:
  - A TM accepts a word if we can reach an accepting state with it as input.
  - The language of a TM is the words accepts.
- Rejection:
  - can be *explicit* by halting without acceptance.
  - can be *implicit* by looping forever (without accepting).

TM's vs. Haskell Programs

Q. Can TMs do more or less than Haskell acceptors?

- for every TM M, we can write a Haskell function `f :: String -> Bool` so that $L(M) = L(f)$?
- for every Haskell function `f :: String -> Bool` there is a TM M such so that $L(M) = L(f)$?

TM's vs. Haskell Programs

Q. Can TMs do more or less than Haskell acceptors?

- for every TM M, we can write a Haskell function `f :: String -> Bool` so that $L(M) = L(f)$?
- for every Haskell function `f :: String -> Bool` there is a TM M such so that $L(M) = L(f)$?

Q. How would one *prove* them?

TM's vs. Haskell Programs

Q. Can TMs do more or less than Haskell acceptors?

- for every TM M, we can write a Haskell function `f :: String -> Bool` so that $L(M) = L(f)$?
- for every Haskell function `f :: String -> Bool` there is a TM M such so that $L(M) = L(f)$?

Q. How would one *prove* them?

A.

First:

- write a universal TM in Haskell (just like in Power Point!)
- E.g., `https://hackage.haskell.org/package/turing-machines-0.1.0.1/src/src/Automaton/TuringMachine.hs`

Second:

- write a Haskell interpreter in a TM (sigh!)
- or trust the Church-Turing thesis...

Proving Program Properties (Example)

Language Recogniser Hello World

Let's implement our first string recogniser in Haskell:

```
simple :: String -> Bool
simple s = (s == "hello world")
```

Hello World Spec. `p :: String -> Bool` satisfies hello world spec, if:

- `p ("hello world") = True`
- `p(s) = False`, if `s != "hello world"`.

Q. Can we (in principle) write a Haskell program

```
hello-world-check :: String -> Bool
```

such that:

- `hello-world-check(code) = True` if code is a syntactically correct Haskell program that satisfies the hello world spec.
- `hello-world-check(code) = False` if code is either not syntactically correct, or does not satisfy the hello world spec.

Interlude: Weird Integer Sequences

This unrelated function just computes an infinite sequence of Integers

```
collatz :: Int  -> [Int]
collatz n | even n = n:(collatz (n `div` 2))
          | otherwise = n:(collatz (3 * n + 1))
```

This problem is also mentioned in a video by by Veritasium (again):
`https://www.youtube.com/watch?v=094y1Z2wpJg`

Conjecture For every initial value ≥ 1, the sequence ends in $(4, 2, 1)^{\infty}$.
(Has been confirmed/tested for all numbers until $2^{68} \approx 2.95 \times 10^{20}$)

```
*Main> take 50 $ collatz 17
[17,52,26,13,40,20,10,5,16,8,4,2,1,4,2,1,4,2,1,4,2,1,4,2,
1,4,2,1,4,2,1,4,2,1,4,2,1,4,2,1,4,2,1,4,2,1,4,2,1,4]
*Main> take 60 $ collatz 439
[439,1318,659,1978,989,2968,1484,742,371,1114,557,1672,836,418,
209,628,314,157,472,236,118,59,178,89,268,134,67,202,101,304,15
2,76,38,19,58,29,88,44,22,11,34,17,52,26,13,40,20,10,5,16,8,4,2
,1,4,2,1,4,2,1]
```

Contrived and the Hello World Spec

Contrived Hello World Recogniser.

```
contrived :: String -> Bool
contrived s = 1 `elem` (collatz (1 + length s)) &&
              (s == "hello world")
```

Q. Does `contrived` satisfy the hello world spec?

Hello Word Spec. A program should:

- return `True` if the argument is equal to `"hello world"`
- return `False` otherwise. (In particular, it *should* terminate.

In particular, it should always return something!

But does it?

Sneaky

Again, does `contrived` specify the demanded specification?

```
contrived :: String -> Bool
contrived s = 1 `elem` (collatz (1 + length s)) &&
              (s == "hello world")
```

Let's see:

- If input is: "hello world", we get:
  - "s == "hello world" is true, so no problem there.
  - We call collatz with $n = 12$ (length of hello world $+ 1$), which gives true.

Sneaky

Again, does `contrived` specify the demanded specification?

```
contrived :: String -> Bool
contrived s = 1 `elem` (collatz (1 + length s)) &&
              (s == "hello world")
```

Let's see:

- If input is: "hello world", we get:
  - "s == "hello world" is true, so no problem there.
  - We call collatz with $n = 12$ (length of hello world + 1), which gives true.
- If the input is *not* "hello world" we get:
  - "s == "hello world" will be false (but it's (probably) evaluated second),
  - we call collatz with arbitrary length, so truth value is not known. But it's truth value doesn't matter anyway, does it? It might diverge! And hence never terminate.

Bombshell Revelation

Collatz conjecture.

Does `collatz n` contain a 1 for every $n \geq 1$?

This is an unsolved problem in mathematics, see, e.g.,

`https://en.wikipedia.org/wiki/Collatz_conjecture/`

Interpretation.

- this doesn't make it *impossible* that we can write hello-world-check,
- but we would have to be more clever than generations of mathematicians.
- This did not show the limitations of *computation*, but shows that it might be hard to prove termination.

Recursively Enumerable Problems

What problems can we solve in principle?

Definition. A *problem* over an alphabet Σ is a set of strings over Σ. For Haskell, we consider $\Sigma = \texttt{Char}$.

A problem P is *recursively enumerable* if there is a Haskell function `f :: String -> Bool` such that

$$P = L(f) = \{\texttt{w} :: \texttt{String} \mid \texttt{f w} = \texttt{True}\}$$

(recall that this doesn't imply termination on the `False` values.)

Definition. If a language is recognised by a Turing machine, then it is called *recursively enumerable*. We also call this *semi-decidable*.

Vocabulary:

- Recognition is the same as acceptance.
- a *problem* is the same as a *language*.

Example

```
L = { w :: String | w is syntactically correct Haskell
                    and defines a function String -> Bool that
                    accepts at least one string }
```

Is L recursively enumerable?

Example for Recursive Enumerability

```
L = { w :: String | w is syntactically correct Haskell
                    and defines a function String -> Bool that
                    accepts at least one string }
```

To see that L is recursively enumerable: given `w :: String`

- Check whether `w` is syntactically correct by running it through a Haskell compiler. Then use a type checker for input/output types.
- Now, consider the infinite list of pairs

```
(0, 0)                    -- all pairs that add to 0
(0, 1), (1, 0)            -- all pairs that add to 1
(0, 2), (1, 1), (2, 0)    -- all pairs that add to 2
...
```

  and walk through the list of all pairs. Whenever we see (i, j), run `w` for i computation steps on all strings `s` of length j, i.e., run `w` for
  - 0 steps on ϵ
  - 0 steps on $a, \ldots, Z$, 1 step on ϵ
  - 0 steps on $aa, \ldots, ZZ$, 1 step on $a, \ldots, Z$, 2 steps on ϵ
  - ...

  If this gives 'True', terminate and return True, otherwise continue.

Discussion on Recursive Enumerability

Observation.

- We never returned `False` (in the example), so rejection was by non-termination.
- At runtime, we cannot distinguish between not yet accepted, or rejected.

Discussion on Recursive Enumerability

Observation.

- We never returned `False` (in the example), so rejection was by non-termination.
- At runtime, we cannot distinguish between not yet accepted, or rejected.

Conclusion.

- Recursive enumerability is *weak*, and comparatively easy: just need to terminate on positive instances, can ignore negative instances.
- This doesn't imply that we *need* to loop forever, but we can if we must. (For some problems, we indeed *cannot* terminate.

Discussion on Recursive Enumerability

Observation.

- We never returned `False` (in the example), so rejection was by non-termination.
- At runtime, we cannot distinguish between not yet accepted, or rejected.

Conclusion.

- Recursive enumerability is *weak*, and comparatively easy: just need to terminate on positive instances, can ignore negative instances.
- This doesn't imply that we *need* to loop forever, but we can if we must. (For some problems, we indeed *cannot* terminate.
- Stronger, and more difficult: require that acceptor `String -> Bool` always terminates.
  - Such an acceptor is called *decider* or *decision procedure*
  - Languages that have an acceptor are called *decidable* (others *undecidable*)

Example 2

```
W = { w :: String |  w is syntactically correct haskell
                     and defines f :: String -> Bool
                     and f w doesn't evaluate to True }
```

Q. Is W recursively enumerable, i.e. can we write a Haskell function `f :: String -> Bool` such that $W = L(f)$?

Example 2

```
W = { w :: String |  w is syntactically correct haskell
                     and defines f :: String -> Bool
                     and f w doesn't evaluate to True }
```

Q. Is W recursively enumerable, i.e. can we write a Haskell function `f :: String -> Bool` such that $W = L(f)$?

A. Assume the answer is yes and let `sc :: String` be the source code of `f`.

Case 1: $\texttt{sc} \in W$.

- Since $W = L(f) = \{\texttt{w} :: \texttt{String} \mid \texttt{f w} = \texttt{True}\}$ we get `f sc = True`
- Because `f sc = True`, $\texttt{sc} \notin W$ (contradiction!)

So case 1 cannot apply.

Example 2

```
W = { w :: String |  w is syntactically correct haskell
                     and defines f :: String -> Bool
                     and f w doesn't evaluate to True }
```

Q. Is W recursively enumerable, i.e. can we write a Haskell function `f :: String -> Bool` such that $W = L(f)$?

A. Assume the answer is yes and let `sc :: String` be the source code of `f`.

Case 1: `sc` $\in W$.

- Since $W = L(f) = \{\texttt{w} :: \texttt{String} \mid \texttt{f w} = \texttt{True}\}$ we get `f sc = True`
- Because `f sc = True`, `sc` $\notin W$ (contradiction!)

So case 1 cannot apply.

Case 2: `sc` $\notin W$.

- Either `sc` is not syntactically correct or `f sc` doesn't eval to `True`
- Since `sc` is syntactically correct, it must be that `f sc = True`

Example for not being Rec. Enumerable

```
W = { w :: String |  w is syntactically correct haskell
                     and defines f :: String -> Bool
                     and f w doesn't evaluate to True }
```

Q. Is W recursively enumerable, i.e. can we write a Haskell function `f :: String -> Bool` such that $W = L(f)$?

A. Assume the answer is yes and let `sc :: String` be the source code of `f`.

Case 1: `sc` $\in W$.

- Since $W = L(f) = \{\texttt{w} :: \texttt{String} \mid \texttt{f w} = \texttt{True}\}$ we get `f sc = True`
- Because `f sc = True`, `sc` $\notin W$ (contradiction!)

So case 1 cannot apply.

Case 2: `sc` $\notin W$.

- Either `sc` is not syntactically correct or `f sc` doesn't eval to `True`
- Since `sc` is syntactically correct, it must be that `f sc = True`
- By definition of W, this means that `sc` $\in W$ (contradiction!)

So case 2 can't apply either! Thus, W is not recursively enumerable.

Back to Turing Machines

We now switch back to TMs. Why?

- Haskell code was just for motivation; to show it's "not just theory"
- TMs are a nice, simple framework (confirm early motivation).

Now what?

- Recall that we are able to encode each TM as a string.
- That way, we can put TM codes into TMs! (Just like Haskell code can analyze Haskell code)
- So we can (t)ask a TM to analyze a TM – hence even itself!

Non-Recursively Enumerable Language, Revisited

Definition. $L_d = \{w \mid w \textit{ is a code of a Turing Machine } M \textit{ that rejects } w\ \}$ is the *d*iagonal language. (In analogy to Cantor's diagonal argument.)

Theorem. L_d is not recursively enumerable, i.e., no TM can accepts it.

Proof

- Suppose for contradiction that TM M exists with $L(M) = L_d$.
- M has a binary encoding C (pick one of them)
- Question: is $C \in L_d$?

(Language and proof are identical to before for Haskell; next slide for TMs.)

Non-Recursively Enumerable Language, Revisited cont'd

Two Possibilities. (recall: C is the code of M)

Option 1. $C \in L_d$

- then M accepts C because M accepts all strings in L_d
- but L_d contains only those TM (codes) w that *reject* w
- hence $c \notin L_d$ – contradiction!

Option 2. $C \notin L_d$.

- then M rejects C because M rejects all strings not in L_d.
- but L_d contains all the encodings w for TMs that reject w
- so $C \in L_d$ – contradiction!

As we get a contradiction either way, our assumption that L_d can be recognised by a TM must be false.

In Short. There cannot exist a TM whose language is L_d.

Summary

Some reflections on recursive enumerability:

- We saw (just now!) problems which are not recursively enumerable! But it was a bit artificial...
  - There are other – less weird – problems that also can't be solved by a TM.
  - Some are on the current exercise sheet!
- We saw problems that are recursively enumerable, but they required unbounded runtime.
- So ... are there problems that are recursively enumerable (i.e., can be expressed by a TM) but that can't be solved by a terminating TM? (Maybe even one we have seen before?)

(Un)Decidable Problems

The Halting Problem

Halting Problem.

- Given a Turing machine M and input w, does M **halt** on w? (We don't mind whether M accepts or rejects.)

Blue Screen of Death

- answering this question could lead to auto-testing
- programs that don't get stuck in infinite loops ...

Partial Answers.

- can give an answer for *some* pairs M, w
- e.g., if M accepts straight away, or has no loops
- difficulty: answer for *all* M, w.

The Halting Problem – a First Stab

First Attempt at solving the halting problem

- Feed M the input w, sit back, and see if it halts!

Critique.

- this is a *partial* decision procedure
- if M halts on w, will get an answer
- will get *no* answer if M doesn't halt!

Comparison with L_d

- this is *better* than L_d
- for L_d, we cannot guarantee *any* answer at all!
- in contrast, the above procedure proves that H is at least semi-decidable.

Recursive Languages

Definition.
A language L is *recursive* if there is a Turing machine *that halts on all inputs* and accepts the language, $L = L(M)$.

- *always* gives a yes/no answer
- also called *decidable*. Now, *undecidable* just means not decidable!

Alternative Definition.
Note that we could have also defined recursive as being recursively enumarable plus the extra requirement that the recognising TM halts on all inputs. (Which also highlights that semi-decidability does not imply undecidability.)

Example.

- the language L_d is not recursively enumerable (or not semi-decidable)
- the halting problem is recursively enumerable, but not recursive (i.e., semi-decidable but not decidable)

Recursive Problems in Haskell

Recall. A problem $P \subseteq \Sigma^*$ is recursively enumerable if

$$P = \{\texttt{w :: String | f w = True}\}$$

for some function `f :: String -> Bool`.

(The formal definition is the one via Turing machines.)

Criticism. We may never know that a string is rejected (since we can also reject by non-acceptance, i.e., by looping forever).

Definition. A problem $P \subseteq \Sigma^*$ is recursive if

$$P = \{\texttt{w :: String | f w = True}\}$$

for some function `f :: String -> Bool` *that always terminates*.

Examples

Encoding.

- We have seen how Turing machines can be encoded as strings.
- Similarly, DFAs can be encoded as strings (we don't make this explicit).
- This means that we can used DFAs and TMs as inputs to. problems.

Q. Which of the following problems is recursive? Recursively enumerable?

1. $\{s \mid s \text{ is a code of a DFA that accepts } \epsilon\}$
2. $\{s \mid s \text{ is a code of a DFA that accepts at least one string}\}$
3. $\{s \mid s \text{ is a code of a TM that accepts } \epsilon\}$
4. $\{s \mid s \text{ is a code of a TM that accepts at least one string}\}$

Examples

Encoding.

- We have seen how Turing machines can be encoded as strings.
- Similarly, DFAs can be encoded as strings (we don't make this explicit).
- This means that we can used DFAs and TMs as inputs to. problems.

Q. Which of the following problems is recursive? Recursively enumerable?

1. $\{s \mid s \text{ is a code of a DFA that accepts } \epsilon\}$
2. $\{s \mid s \text{ is a code of a DFA that accepts at least one string}\}$
3. $\{s \mid s \text{ is a code of a TM that accepts } \epsilon\}$
4. $\{s \mid s \text{ is a code of a TM that accepts at least one string}\}$

A. First two: recursive! Last two: see tutorials.

H is recursively enumerable

```
H = {(w :: String, i :: String ) | w valid Haskell
                  and defines f :: String -> Bool
                  and f i terminates }
```

Algorithm to check whether `(w, i)` is in `H`:

- check whether `w` is correct Haskell
- check whether `w` defines `f :: String -> Bool`
- run the function `f` on input `i`.
- (We had the same proof already for TMs)

Meta Programming.

- need to write a Haskell interpreter in Haskell
- can be done! (Glasgow Haskell Compiler (ghc) is written in Haskell)

Or via Church Turing Thesis.

- we know that we can write a Haskell interpreter (there are some!)
- by Church-Turing, this can also be done in a TM
- as Haskell is Turing-complete, this can be done in Haskell

H is not recursive

```
H = {(w :: String, i :: String ) | w valid Haskell
                    and defines f :: String -> Bool
                    and f i terminates }
```

Impossibility Argument assume total `t` exists with $L(\texttt{t}) = \texttt{H}$ ...

Detour. If we can define `t`, then we can define `P`.

```
P :: String -> Bool
P w = if t (w, w) then P w else True
```

(infinite recursion whenever `t (w, w) = True`)

Let `sc` be the source code of `P`.

Case 1. `P sc` terminates.

- then `(sc, sc)` is in `H`. (Due to H's definition!)
- then `t (sc, sc)` returns `True`. (Since $L(\texttt{t}) = \texttt{H}$)
- then `P sc` doesn't terminate. But this can't be!

H is not recursive

```
H = {(w :: String, i :: String ) | w valid Haskell
                    and defines f :: String -> Bool
                    and f i terminates }
```

Impossibility Argument assume total `t` exists with $L(\texttt{t}) = \texttt{H}$ ...

Detour. If we can define `t`, then we can define `P`.

```
P :: String -> Bool
P w = if t (w, w) then P w else True
```

(infinite recursion whenever `t (w, w) = True`)

Let `sc` be the source code of `P`.

Case 2. `P sc` does not terminate.

- then `(sc, sc)` is not in H. (Again due to H's definition!)
- then `t (sc, sc)` returns `False` (Again since $L(\texttt{t}) = \texttt{H}$)
- then `P sc` does terminate. This can't be either!

As a conclusion, the function t (that decides H) cannot exist.

The Halting Problem

General Formulation

There is no program that always terminates, and determines whether (another) program terminates on a given input.‘

Interpretation.

There are problems that cannot be solved (decided) algorithmically.

- ‘solve’ means by a program that doesn’t get stuck
- Halting problem is one example.

(We have argued in terms of Haskell programs. Will do this via TMs next)

Hello World Spec

Flashback. `p :: String -> Bool` satisfies hello world spec, if:

- `p ("hello world") = True`
- `p(s) = False`, if `s != "hello world"`.

Earlier.

- checking whether `p` satisfies hello world spec is hard.

Now.

- checking whether `p` satisfies hello world spec is *impossible*.

Hello World Spec

Recall. `p :: String -> Bool` satisfies hello world spec, if:

- `p ("hello world") = True`
- `p(s) = False`, if `s != "hello world"`.

Impossibility argument. If there was

```
hello-world-check ::  String -> Bool
```

Define

```
halt :: String * String -> Bool
halt w i = hello-world-check aux
where aux s = (s == "hello world") &&
              (w i = True || w i = False).
```

Observation

- if `hello-world-check` were to exist, we could solve the Halting problem
- general technique: *reduction*, i.e., use a hypothetical solution to a problem to solve one that is unsolvable.

Total Functions

Question. Consider the set

```
T = { w :: String |
          w is valid Haskell
          and defines f :: String -> Bool,
          and f x terminates for all x :: String }
```

Q. Is `T` recursively enumerable? Even recursive?

A. We will see:

- It's not recursive (just like the Halting problem.)
- It's *not even recursively enumerable!*
  - *Intuitively:* We can't give a semi-decision procedure for the yes-instances because there's no criterion to terminate with "yes" as there are infinitely many inputs which we have to test. This is different from the standard version of the halting problem where we can return "yes" as soon as the respected program terminates.
  - *More formally:* Our tutorials!

Back to TMs: The Universal TM

TMs that simulate other TMs

- given a TM M, it's easy to work out what M does, given some input
- it is an *algorithm*. If we believe the Church-Turing thesis, this can be accomplished by (another) TM. (People actually did this.)

Universal TM

- is a TM that accepts two inputs: the *coding* of a TM M_s and a string
- it *simulates* the execution of M_s on w
- and accepts if and only if M_s accepts w.

Construction of a universal TM

- keep track of current state and head position of M_s
- scan the TM instructions of M_s and follow them
- (this requires lots of coding but is possible/was done)

The Halting Problem is Undecidable

Theorem. $H = \{(m, w) \mid m \text{ is the code of a TM } M, \text{ and } M \text{ halts on } w\}$

(Note that we might sometimes just write M instead of m as it's clear that we put the encoding (m) of M into that set, not the actual TM.)

The Halting Problem is Undecidable

Theorem. $H = \{(m, w) \mid m \text{ is the code of a TM } M, \text{ and } M \text{ halts on } w\}$

(Note that we might sometimes just write M instead of m as it's clear that we put the encoding (m) of M into that set, not the actual TM.)

Proof (Sketch).

- suppose we had a TM T_H *that always terminates* so that $L(T_H) = H$
- construct a new TM P (for paradox)

Construction of P: P takes *one* input, an encoding of a TM

- If T_H accepts (m, m) (i.e. if M halts on its own encoding m), loop forever.
- If T_H rejects (m, m), halt.

Q. does P halt on input (an encoding of) P?

The Halting Problem is Undecidable

Theorem. $H = \{(m, w) \mid m \text{ is the code of a TM } M, \text{ and } M \text{ halts on } w\}$

(Note that we might sometimes just write M instead of m as it's clear that we put the encoding (m) of M into that set, not the actual TM.)

Proof (Sketch).

- suppose we had a TM T_H *that always terminates* so that $L(T_H) = H$
- construct a new TM P (for paradox)

Construction of P: P takes *one* input, an encoding of a TM

- If T_H accepts (m, m) (i.e. if M halts on its own encoding m), loop forever.
- If T_H rejects (m, m), halt.

Q. does P halt on input (an encoding of) P?

- **No** – then T_H *accepted* (P, P), so P should have halted on input P.
- **Yes** – then T_H *rejected* (P, P), so P should *not* have halted on P.

Contradiction in both cases, so T_H cannot exist.

Total Turing Machines

We know that it's undecidable to check whether a TM halts on a given input. But maybe it's easier *for all inputs*?

Q. Is there a TM M_T (for total) that

- always terminates (=total),
- takes an encoding m of a TM M as input, and
- accepts if M terminates *on all inputs*?

Or phrased differently,

Total Turing Machines

We know that it's undecidable to check whether a TM halts on a given input. But maybe it's easier *for all inputs*?

Q. Is there a TM M_T (for total) that

- always terminates (=total),
- takes an encoding m of a TM M as input, and
- accepts if M terminates *on all inputs*?

Or phrased differently, is the following set recursive?
$T = \{m \mid m \text{ is the encoding of a TM } M \text{ that halts on all inputs}\}$

Total Turing Machines

We know that it's undecidable to check whether a TM halts on a given input.
But maybe it's easier *for all inputs*?

Q. Is there a TM M_T (for total) that

- always terminates (=total),
- takes an encoding m of a TM M as input, and
- accepts if M terminates *on all inputs*?

Or phrased differently, is the following set recursive?
$T = \{m \mid m \text{ is the encoding of a TM } M \text{ that halts on all inputs}\}$

A. No! It's not recursive (i.e., it's undecidable).
(Proof on next slide.)

Proof that T is undecidable

Reduction Strategy.

- Suppose we had such a TM M_T that decides T.

Proof that T is undecidable

Reduction Strategy.

- Suppose we had such a TM M_T that decides T.
- We know: For an arbitrary TM M and an arbitrary string w, we can't decide whether M halts on w. So let's "solve" it anyway! (By using M_T, so we obtain a contradiction.)

Proof that T is undecidable

Reduction Strategy.

- Suppose we had such a TM M_T that decides T.
- We know: For an arbitrary TM M and an arbitrary string w, we can't decide whether M halts on w. So let's "solve" it anyway! (By using M_T, so we obtain a contradiction.)
- Given such (M, w), create a new TM M_w, that executes M on w. So, TM M_w ignores its input and runs like M would on w. Since M_w ignores its input, it either accepts all words or none!

Proof that T is undecidable

Reduction Strategy.

- Suppose we had such a TM M_T that decides T.
- We know: For an arbitrary TM M and an arbitrary string w, we can't decide whether M halts on w. So let's "solve" it anyway! (By using M_T, so we obtain a contradiction.)
- Given such (M, w), create a new TM M_w, that executes M on w. So, TM M_w ignores its input and runs like M would on w. Since M_w ignores its input, it either accepts all words or none!
- So, running M_T on M_w tells us whether M halts on w!

Proof that T is undecidable

Reduction Strategy.

- Suppose we had such a TM M_T that decides T.
- We know: For an arbitrary TM M and an arbitrary string w, we can't decide whether M halts on w. So let's "solve" it anyway! (By using M_T, so we obtain a contradiction.)
- Given such (M, w), create a new TM M_w, that executes M on w. So, TM M_w ignores its input and runs like M would on w. Since M_w ignores its input, it either accepts all words or none!
- So, running M_T on M_w tells us whether M halts on w!
- Contradiction, since we solved the halting problem. So M_T can't exist.

It's undecidable (i.e., not recursive), but maybe at least recursively enumerable? (I.e., says yes eventually when the answer is yes, but doesn't always say no when the answer is no.) We see that in the tutorials!

Conclusions

The Chomsky Hierarchy

Recall. Classification of language according to complexity of grammars

- regular languages – FSAs
- context-free languages – PDAs
- context-sensitive languages – LBAs (linearly bounded TMs)
- recursively enumerable languages – TMs

Q. Where do *recursive* languages sit in this hierarchy?
Are there automata for them?

The Chomsky Hierarchy

Recall. Classification of language according to complexity of grammars

- regular languages – FSAs
- context-free languages – PDAs
- context-sensitive languages – LBAs (linearly bounded TMs)
- recursively enumerable languages – TMs

Q. Where do *recursive* languages sit in this hierarchy?
Are there automata for them?

A. They sit between context sensitive and recursively enumerable and are recognised by *total* TMs, which halt on every input.

Structure vs **Property**

- all other automata had a clear cut definition
- total TMs have a *condition* attached (i.e., they always terminate)

Problem.

- cannot *test* whether this condition is fulfilled
- so the definition is based on a property, not a clear structure

Summary

This week, we learned:

- There are problems that are so hard that no TM can express them (those that are not recursively enumerable)
- Others can be expressed but might loop forever on no-instances (those that are recursively enumerable but undecidable)
- Others are even decidable, i.e., an algorithm always says (correctly) yes or no after a finite time.
- We saw (important) examples for such problems. Others are given in the tutorials.

The tutorial also discusses important properties such as the relationship between a language and its complement (regarding their complexity).

Next week, we look into decidable problems and analyse "more fine-grained" how hard it is to decide them. (Runtime!)