Turing Machines: Limits of Decidability

COMP1600 / COMP6260

Dirk Pattinson Victor Rivera
Australian National University

Semester 2, 2021

Interlude: What can Haskell programs do?

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

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


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

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

```
> 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`?

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

```
> 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 flashback. A machine M accepts w, if running M on w *terminates* and leaves M in an accepting state.

From TM's to Haskell Programs (and maybe back?)

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 $L(M) = L(f)$?

From TMs to Haskell Programs.

Q. For every TM M, can we write a Haskell function `f :: String -> Bool` so that $L(M) = L(f)$?

A. Easy. Implement / install / download a TM simulator, e.g. `https://hackage.haskell.org/package/turing-machines-0.1.0.1/src/src/Automaton/TuringMachine.hs`

From Haskell programs to Turing machines.

Q. For every Haskell function `f :: String -> Bool`, is there is a TM M such so that $L(M) = L(f)$?

A. We would need to simulate the evaluation of Haskell programs in a Turing machine. This is hairy. But if we believe the Church-Turing thesis (which we do), it's possible in principle.

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(s) = True` if `s` is a syntactically correct Haskell program that satisfies the hello world spec
- `hello-world-check(s) = False` if `s` 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))
```

Observation: For every initial value ≥ 1, this sequence eventually reaches 1 (try it!).

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?

Contrived and the Hello World Spec

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

Hello World Spec: A program should:

- return `True` if the argument is equal to `"hello world"`
- return `False` otherwise.

In particular, it should always return something!

Hence `contrived` is correct iff there's a 1 in `collatz n` for all $n \geq 1$.

Sneaky

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

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


Apparently Simple.

- does a program satisfy the hello world spec?

Seemingly complicated.

- does `collatz n` contain a 1 for all $n \geq 1$?

Connection (by sneakily inserting `collatz`)

- *if* we can check whether a program satisfies hello word spec, *then* we can check whether `collatz n` contains a 1.

Bombshell Revelation

Collatz conjecture.

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

This is an unsolved problem in maths, 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.

What problems can we solve in principle?

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

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

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

(repeating our earlier definition.)

Example

```
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.

- 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. If this gives 'True', terminate and return True, otherwise go to the next pair.

(We could implement this by inserting a evaluation step counter into a Haskell interpreter)

Discussion

Algorithm (short form) given `w :: String`:

- check that p defines a program p of type `String -> Bool`
- simulate execution of p for increasingly more steps on increasingly longer strings
- return `True` if the simulation returns `True`

Observation.

- we never return `False`, so non-acceptance is non-termination
- at runtime, we can't distinguish between not yet accepted, or rejected.

Discussion.

- Recursive enumerability is *weak*, and comparatively easy: just need to terminate on positive instances, can ignore negative instances
- Stronger, and more difficult (and later): require that acceptor `String -> Bool` always terminates.

Second Example


```
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 $W = L(f)$?

W is not Recursively Enumerable

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


Let's assume that there is `f :: String -> Bool` with $W = L(f)$. Let `sc` :: String be the source code of `f`.

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

- Because $W = L(f) = \{\mathtt{w :: String} \mid \mathtt{f\ w = True}\}$ we have that `f sc = True`.
- Because `f sc = True`, `sc` $\notin W$ (contrary to our assumption).

So case 1 cannot apply.

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

- Then either `sc` is not syntactically correct or `f sc` does not eval to `True`.
- as `sc` is syntactically correct, it must be that `f sc = True`.
- By definition of W, this means that `sc` $\in W$ (contary to our assumption).

So case 2 can't apply, either, and therefore `f` cannot exist.

Back to Turing Machines

Preview. We will now do the same with Turing machines instead of Haskell programs. But why?

Mathematical Precision.

- what does 'evaluate' and 'terminate' mean, formally?

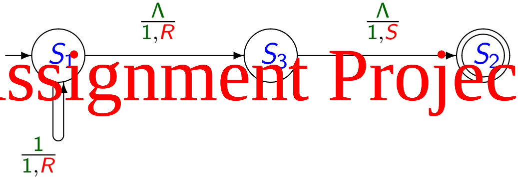
Simplicity of Model

- to make the above precise, we have to dive deep into Haskell
- many features of Haskell are not used

Applicability to Other Models

- Using the TM model, we drill down to the very basics
- Can apply similar reasoning to any language that is Turing complete.

Coding a TM onto tape – example



The transitions are

0	1	00	1	0	1	00	1	00
0	1	000	1	000	1	00	1	00
000	1	000	1	00	1	00	1	000

Recall: States, Symbols, Directions.

So the TM as a whole is encoded by the binary string

01001010010 11 0100010001001 11 000100010010 01000

The coding plays the role of 'source code' in the Haskell examples.

Strings and Machines

TM Encoding

- a TM may have several encodings (re-order states or symbols)
- some strings may not be encodings at all, e.g. 110.

Questions. Given binary string w

- Is w an encoding of a TM? *(easy)*
- If w *is* an encoding of a TM M, does M accept the string w, or reject it?

Q. Why does this make sense?

- it amounts to asking whether a TM accepts itself
- key step to exhibit non-recursively enumerable languages

A language that is not recursively enumerable

Definition. L_d is the set of binary strings w such that

$$w \text{ is a code of a Turing Machine } M \text{ that rejects } w$$

- d for 'diagonal'
- 'reject' means halting in a non-final state, or non-termination.

Theorem. L_d is not recursively enumerable, i.e there is no TM that accepts *precisely* L_d.

Proof (Sketch)

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

A language that is not recursively enumerable ctd.

Two Possibilities.

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.

Reflecting on this example

Reflections on the proof.

- the language L_d is artificial, designed for easy contradiction proof
- but it is *easy* to define, like any other language (using basic maths)
- if we believe the Church-Turing thesis, there is *no* program that would be able to answer the question $w \in L_d$.

Questions.

- are all non-recursively enumerable languages 'weird' (like L_d)?
- are the problems that are not computable but of interest in practice?

The Halting Problem

Halting Problem.

- Given a Turing machine M and input w does M **halt** (either accepting or rejecting) on w?

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!

Recursively Enumerable vs. Recursive Languages

Recursively Enumerable Languages.

A language L is *recursively enumerable* if there is a Turing machine M so that M accepts precisely all strings in L ($L = L(M)$)

- if $w \notin L$, the TM may never terminate and give an answer ...
- also called *semidecidable*
- can enumerate all elements of the language, but cannot be sure whether a string eventually occurs

Recursive Languages.

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

- *always* gives a yes/no answer
- also called *decidable*

Example.

- the language L_d is not recursively enumerable
- the halting problem is recursively enumerable
- ... but not recursive (as we will see next)

Recursive Problems in Haskell

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

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

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.

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

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

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

(We will define this more formally with TMs later.)

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.

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

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

Some Answers

DFAs accepting the empty string.

- decidable: just check whether the initial state is accepting.

DFAs accepting at least one string

- decidable: compute the set of reachable states

```
n = 0;
reach n = { q0 } -- initial state, reachable in zero steps
repeat
  n := n + 1
  reach (n) = { s | can reach s in one step from
                 reach (n-1) } -- reachble in n steps
until reach (n) = reach (n-1)    -- no new states found
reach := reach n
```


- check whether `reach n` contains a final state.

Other Problems. see Tutorial.

Termination Checking

Halting Problem (in Haskell).

```
H = {(w :: String, i :: String) | w is syntactically correc
                     and defines f: String -> Bool
                     so that f i terminates }
```

H is recursively enumerable

```
H = {(w :: String, i :: String ) | w is syntactically correc
                     and defines f: String -> Bool
                     so that 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`.

Meta Programming.

- need to write a Haskell interpreter in Haskell
- this can be done (ghc is written in ghc)
- later: universal Turing machines

Via Church Turing Thesis.

- we can write a Haskell interpreter
- by Church-Turing, this can 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 `d` exists with $L(\mathtt{d}) = \mathtt{H}$ ...

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


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

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

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

Case 1. `P sc` terminates.

- then `(sc, sc)` is in `H`.
- then `d (sc, sc)` evaluates to `True`
- 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 d exists with $L(\mathtt{d}) = \mathtt{H}$ ...

Detour. If we can define d, then we can define P.


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

(infinite recursion whenever `d (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.
- then d(sc, sc) returns `False`
- then `P sc` does terminate. This can't be either!

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

The Halting Problem

General Formulation (via Church Turing Thesis)

‘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 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 }
```


Is `T` recursively enumerable? Even recursive?

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.

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)

The Halting Problem as a Language Problem

Slight Modification of universal TM:

- U_1 is a universal TM with all states accepting
- hence if U_1 halts, then U_1 accepts.

Halting Problem formally

- Is $L(U_1)$ recursive?

Observation.

- all problems can be expressed as language problems
- we know that $L(U_1)$ is recursively enumerable – by definition

Q. Is $L(U_1)$ even recursive?

- can we design a "better" TM for $L(U_1)$ that *always* halts?

The Halting Problem is Undecidable

Theorem. The halting problem is undecidable.

Proof (Sketch).

- suppose we had a TM H *that always terminates* so that $L(H) = L(U_1)$ (H for halt)
- construct a new TM P (for paradox)

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

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

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

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

Contradiction in both cases, so H cannot exist.

Reflections on the proof

Positive Information.

- to show that a language is (semi-) decidable, one usually needs to exhibit an algorithm. This generates information (the algorithm)

Negative Information.

- to show that a language is *not* decidable, assume that there is a TM for it, and show a contradiction. This (just) shows impossibility.

Reduction.

- standard proof technique
- assume that a TM exists for a language L
- *reduce* L to a known undecidable language
- so that a solution for L would give a solution to a known undecidable problem

Example.

- if a TM for language L existed, we could solve the halting problem!
- many other undecidable problems ...

Total Turing Machines

Question. Is there a TM T (for total) that

- always terminates
- takes an encoding of a TM M as input
- accepts if M terminates *on all inputs* ?

Reduction Strategy.

- Suppose we had such a TM T
- for a TM M and string w define a new TM M_w that ignores its input and runs like M would on w
- running T on M_w tells us whether M halts on w
- so we would have solved the halting problem
- since the halting problem cannot be solved, T cannot exist.

The Chomsky Hierarchy

Recall. Classification of language according to complexity of grammars

- regular languages – FSA's
- context-free languages – PDA's
- context-sensitive languages
- recursively enumerable languages – TM's

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

- they sit between context sensitive and recursively enumerable
- and are recognised by *total* TMs that halt on every input.

Structure vs Property

- all other automata had a clear cut definition
- total TMs have a *condition* attached

Problem.

- cannot *test* whether this condition is fulfilled
- so the definition is mathematical, not computational

Back to the Entscheidungsproblem

Q. Can we design an algorithm that *always terminates* and checks whether a mathematical formula is a theorem?

- this is the *Entscheidungsproblem* posed by Hilbert
- and what Alan Turing's original paper was about

More detail.

- mathematical formula means statement of first-order logic
- proof means proof in natural deduction (or similar)

Ramifications.

- all mathematicians could be replaced by machines

Turing's Result.

- the set of first-order formulae that are provable is *not* recursive.
- the existence of a TM that computes the Entscheidungsproblem leads to a contradiction

Other Approaches.

- Church showed the same (using the λ-calculus) in 1932
- was not widely accepted as λ-calculus is less intuitive