Lec 23 Computability

Decision problem — yes / no given input

Elements
- Domain
- Instance
- Property

Domain	Instance	Property
chess	some chess board	does white have guaranteed win
graphs	a graph	is there cycle?
SML	e	is e well-typed
		does e reduce to value

<u>Def</u> Given property P on type D, a decision procedure for P is a function $f : D \to bool$ s.t.

1. $f(x) \hookrightarrow true$ if P holds for x
2. $f(x) \hookrightarrow false$ if P doesn't hold for x

Equivalently:
1. $f(x) \hookrightarrow true \iff$ P holds for x
2. f is total

<u>Def</u> If a decision procedure exists for P, then P is decidable
else P undecidable

The halting problem — whether a programme halts

Domain : $(int \to int) * int$ for now
Instance : (f, x)
Property : $f\ x \hookrightarrow$ value v

call it HALT

Thm HALT is undecidable

Proof ← proof by diagonalisation

Suppose H : (int → int) → bool is a decision procedure for HALT.
So
- H(g, x) ↪ true if g(x) valuable
  ↪ false else
- H is total

Then let

```
fun loop () = loop ()                : unit → α
fun diag (x : int) : int =
  if H (diag , x) then loop ()
  else 0
```

Consider H(diag, 0)

Suppose " ↪ true

diag (0) ⇒ if H (diag , x) then loop () else 0
⇒ loop ()

So " ↪ false because diag 0 not valuable

Suppose " ↪ false

diag (0) ⇒ if H (diag , x) then loop () else 0
⇒ 0

So " ↪ true because diag 0 ⇒ 0

Proof by diagonalisation

Recall: Cantor : all ways to list nums in ℕ not countable

Reduction argument

Suppose P and Q are properties

<u>Def</u> If P reduceable to Q if we have a decision for Q, f_Q, we can implement a decision procedure f_P for P.

Consider:

HALTO – for some func f : (int → int), f 0 is valuable.

Idea: HALT can reduce to HALTO

Write HALT ≤ HALTO
at least as hard as HALT

<u>Thm</u> HALTO is undecidable

<u>Proof</u>

Suppose we have z is a decision procedure for HALTO

Implement a procedure for HALT:

```
fun H (g : int → int , x : int) : bool =
    z (fn _ ⇒ g x)
```

... then we just implemented a decision procedure for HALT,
so z can't have existed.

Turns out we can have problems that are more impossible than others.

Semi-decision procedure

Def A semi-decision procedure for P is a function
$f : D \to bool$ s.t.

1. $f(x) \hookrightarrow true \Leftrightarrow$ P holds for x.
(so if answer is no f can loop, but if yes f must return true)

Def P is recursively enumerable (r.e.) aka semi-decidable if P has a semi-decision procedure

Thm HALT is semi-decidable

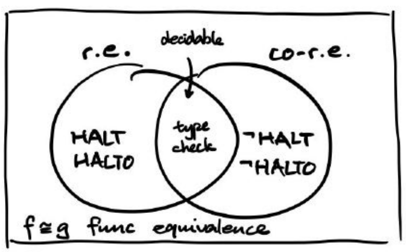
Proof

```
fun H'(g: int -> int , x: int) : bool  =  (g x ; true)
```

So HALT is r.e.

Def P is co-recursively enumerable (co-r.e.) if $\neg P$ is r.e.

Then...



Thm If P decidable, $\neg P$ is decidable

We can just take negation after deciding P

Thm If P is r.e. and co-r.e., then P is decidable

Run decision for P and ¬P simultaneously, then we know once one of them finishes.

Neither r.e. nor co-r.e.

Consider

EQUIV
- Domain : (int → int) * (int → int)
- Instance : (f, g)
- Property : $f \cong g$

Thm EQUIV is not r.e. and not co-r.e.

Proof

(not r.e) Suppose E_q is semi-decision procedure for EQUIV

WTW a semi-decision procedure for ¬HALTO

```
fun S(h: int → int): bool =
  Eq ((fn y => (h 0; y)), (fn y => loop ()))
```

(not co-r.e.) Suppose notE_q is semi-decision procedure for ¬EQUIV

WTW a semi-decision procedure for ¬HALTO

```
fun S'(h: int → int, x: int): bool =
  notEq((fn y => (h 0; y)), (fn y => y))
```

□

Bonus : Rice's Theorem ?

Domain : (int → int)
Instance : f
Property : true

trivil. Just return true

Domain : (int → int)
Instance : f
Property : false

trivil. Just return false

<u>Thm</u> Only these two decidable. Anything else on (int → int) undecidable
Viz. If P is non-trivil property on int → int then P undecidable

<u>Proof</u>

Let y, n s.t. $P(y)$, $\neg P(n)$. Suppose D is decision procedure for P

```
fun l (_: int) : int = loop ()
```

Suppose $P(l)$

```
fun H (f: int → int ): bool =
  not (D(( fn x ⇒ (f x ; n x ))))
```

if f(0) halts $H(f) \cong$ not $(D(n))$
$\cong$ true

Suppose $\neg P(l)$

```
fun H (f: int → int ): bool =
  D(( fn x ⇒ (f x ; y x )))
```