Infinity and beyond ...

- Consider the following Haskell definition:

```
ones = 1 : ones
twos = 2 : twos
```

- How do we prove the following property:

  `twos == map (+1) ones`

  Does it even make sense ?

The usual trick

- We could make the property finite

  `take n twos == take n (map (+1) ones)`

- This can be proven by induction on `n`.
- But is a proof like this, for arbitrary `n` satisfactory ?

Building our intuition

```
twos == map (+1) ones
(2:twos) == map (+1) (1:ones)
(2:2:twos) == map (+1) (1:1:ones)
...
(2:2:...2:twos) == map (+1) (1:1:...:1:ones)
(2:2:...2:twos) == (1+1):(1+1):...:(1+1):map (+1) ones
(2:2:...2:twos) == (1+1):(1+1):...:(1+1):twos  ?
```

All we see are sequences of twos — can we interpret this as a proof principle?

Laziness and Types

- Our challenge here (`ones` and `twos`) involves laziness
- Laziness is just about reduction order, right?
- No, laziness has big implications for how we interpret types.

The "meaning" of recursive types

- Consider the following type declaration:
  `data List = Nil | Cons Int List`
- Drop the `data` keyword, so now its "just" an equation:

  `List = Nil | Cons Int List`
- What about this equation?

  `Nil | Cons Int List = List`
- Both are equivalent, but, for someone moving from "mainstream" programming to Haskell, they sub-consciously favour the second form (??)

The second form

- Why might we (you?) subconsciously favour this?

  `Nil | Cons Int List = List`
- It suggests the following?

  `(Nil | Cons Int List) -> List`

  So?
- This suggest that our original data declaration was defining `List` as something that can be built from simpler, given components, in two ways:

```
nil  :: ()              -> List
cons :: (Int,List)      -> List
list :: () | (Int,List) -> List
     -- suggestive, not proper Haskell
```

The first form

- `List = Nil | Cons Int List`

  Ok, now what?
- Add an arrow

  `List -> (Nil | Cons Int List)`

  What might this suggest?
- From a `List` we can obtain either `Nil`, or `Cons i is` for some values of `i` and `is`.
- ```
  poke :: List -> () | (Int,List)
    -- also suggestive and not proper Haskell
  ```
- We can view a `List` as an entity[1] (with state) that we can "poke", and we will get back one of two possible responses:
  - Nothing (a.k.a., `Nil` or `()`), with nothing further left to poke; or
  - A pair of an `Int` and a `List`, with the possiblility that we can do another poke on the returned list.

[1] a.k.a. "process"

Poking a List

```
poke [2,4,6,8,...] = (2,[4,6,8,...])
poke [4,6,8,...] = (4,[6,8,...])
poke [6,8,...] = (6,[8,...])
```

We can plot this as a state diagram:

$$\underset{l2}{\circ} \xrightarrow[2]{} \underset{l4}{\circ} \xrightarrow[4]{} \underset{l6}{\circ} \xrightarrow[6]{} \underset{l8}{\circ} \xrightarrow[8]{} \underset{\ldots}{\circ}\cdots$$

where
`l2 = [2,4,6,8,...]`,
`l4 = [4,6,8,...]`,
etc ..., are *States*.
```

Lazy Lists (a.k.a "Streams") as Producers

- We interpret definition `ones = 1:ones` as saying:
  - `ones` can 'produce' a `1`
  - its behaviour afterwards is that of `ones`
- We interpret the following part of `map`'s definition `map f (x:xs) = (f x):(map f xs)` as:
  - whenever its argument 'produces' a value, apply `f` to it
  - then repeat
- A strict (non-lazy) view of datatypes and recursive functions is as a way of 'building' values from basic bits
- We are viewing lazy values as pre-existing 'producer's of component values.
  - `x:xs` is viewed as generating an `x`, with `xs` capturing the producer's future behaviour.
  - `[]` is a producer that has terminated.
    But note that no producer is obliged to eventually stop.

Producer Equivalence

- Two producers are equivalent if they produce the same values
- In effect once they produce a value, they end up in *corresponding* states where they still produce the same values
- A proof sets up a relation between corresponding states that have the same property

Viewing `ones` and `twos` as Processes (I)

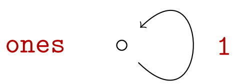
- A process `xx` of type `[t]` can display two behaviours:
  - Terminate now (`[]`)

    $\texttt{[]} \quad \bullet$

  - Produce `x::t` and then behave like `xs::[t]` (`x:xs`)

    $\texttt{x:xs} \quad \circ \xrightarrow{\texttt{x}} \circ \quad \texttt{xs}$

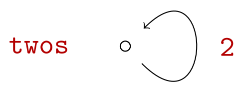
  - Each dot marks a *state* of the system.
    We adopt the convention that black dots are terminating states.
- So, `ones = 1:ones` acts like:



Viewing `ones` and `twos` as Processes (II)

- A process `map f xx` behaves like `xx`, except that a production of `x` by `xx` becomes a production of `f x` by `map f xx`.
- So, `map (+1) ones` acts like:

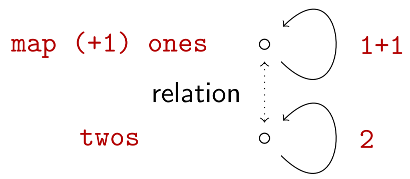
- Obviously, `twos` looks like:



Comparing Systems

- We say two systems are bisimilar if their behaviour, viewed as sequences of the arrow labels, cannot be used to distinguish one from the other.
- In order to show two systems are bisimilar we have to find a relation on states (dots like $\bullet$ or $\circ$) such that any two states so related have the following properties:
  - Both states have the same number of outgoing arrows with the same labels
  - Arrows with the same labels lead to states that are themselves in the relation.
- This is a proof that the two systems are bisimilar using a technique known as *co-induction*.

Co-inductive Proof that `map (+1) ones = twos`

- For our case there is only one state in each, both with arrows labelled with 2, back into the same states, so the relation we look for is the only one possible, relating `map (+1) ones` to `twos`.
- 

- Remember, `1+1=2` — just in case you have forgotten!! Referential Transparency still applies.

More complicated Example

A Program:

```
data Parity = Even | Odd

pflop = Even:Odd:pflop

from n = n:(from (n+1))

parity n = if even n then Even else Odd
```

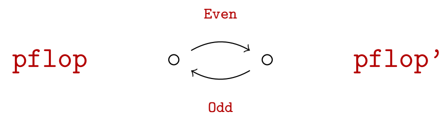
A Property: `pflop = map parity (from 0)`

State Diagram for `pflop`

We rewrite the definition of `pflop`, which exposes two states:

```
pflop = Even:pflop'
pflop' = Odd:pflop
```

The state diagram is:

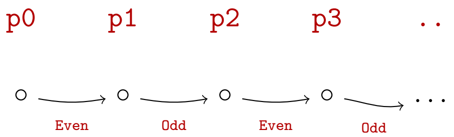


State Diagram for map parity (from 0)

We partially evaluate (lazily) the expression:

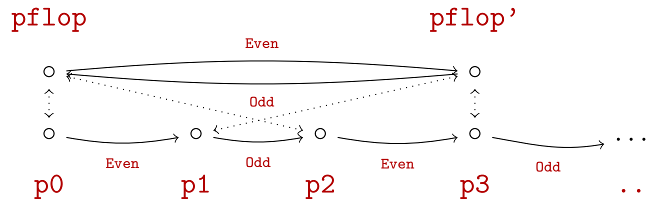
```
map parity (from 0)
= map parity (0:from 1)
= (parity 0) : map parity (from 1)
= (parity 0) : map parity (1: from 2)
= (parity 0) : ( parity 1) : map parity (from 2)
= (parity 0) : ( parity 1) : map parity (2: from 3)
= (parity 0) : ( parity 1) : (parity 2): map parity (from 3)
= Even : Odd : Even : map parity (from 3)
= ...
```

The state diagram is:



Relating pflop and parity n

- We relate pflop to p0,p2,p4, etc.
- We relate pflop' to p1,p3,p5, etc.



- From pflop there is one arrow labelled Even to pflop'
- From pn, there is one arrow labelled Even or Odd to pn+1, depending on if n is even or odd.
  - Arrows out of p0, p2, p4, are labelled with Even, `even n ==> parity n = Even`
  - Arrows out of p1, p3, p5, are labelled with Odd, `odd n ==> parity n = Odd`
- We have our required bisimulation relation

Co-Induction: Summary

- Induction is used to reason about recursively defined functions/structures
  - Must have base cases
  - Proofs only apply to finite structures/terminating functions
  - Relatively straightforward
- Co-induction also reasons about co-recursively defined functions/structures
  - No base case is needed
  - Proofs apply to both finite and infinite structures, and both terminating and non-terminating functions.
  - Counter-intuitive proof technique, requiring the invention/discovery of an appropriate bisimulation relation ("eureka step").

Co-Induction vs Induction

- Consider recursive definition:

```
data List = Nil | Cons Int List
```

- Inductive interpretation: a recipe to construct a List.

```
Nil :: () -> List
Cons :: (Int,List) -> List
```

For symmetry with below, we view Nil as a constant function, and consider Cons as un-curried.

- Co-Inductive interpretation: ways in which a List can be taken apart, or interpeted as a process producing stuff.

```
Nil :: List -> ()
Cons :: List -> (Int, List)
```

Note how the arrow gets flipped!