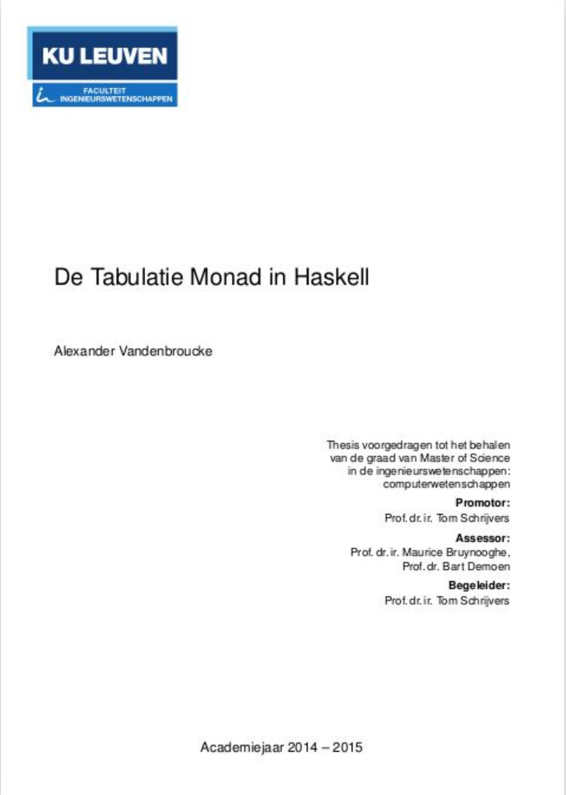
Fixing Non-determinism

Alexander Vandenbroucke



KU LEUVEN

FACULTEIT INGENIEURSWETENSCHAPPEN

De Tabulatie Monad in Haskell

Alexander Vandenbroucke

Thesis voorgedragen tot het behalen
van de graad van Master of Science
in de ingenieurswetenschappen:
computerwetenschappen

Promotor:
Prof. dr. ir. Tom Schrijvers

Assessor:
Prof. dr. ir. Maurice Bruynooghe,
Prof. dr. Bart Demoen

Begeleider:
Prof. dr. ir. Tom Schrijvers

Academiejaar 2014 – 2015

Fixing Non-determinism

Alexander Vandenbroucke
KU Leuven
alexander.vandenbroucke@kuleuven.be

Tom Schrijvers
KU Leuven
tom.schrijvers@kuleuven.be

Frank Piessens
KU Leuven
frank.piessens@kuleuven.be

Abstract

Non-deterministic computations are conventionally modelled by lists of their outcomes. This approach provides a concise declarative description of certain problems, as well as a way of generically solving such problems.

However, the traditional approach falls short when the non-deterministic problem is allowed to be recursive: the recursive problem may have infinitely many outcomes, giving rise to an infinite list.

Yet there are usually only finitely many distinct *relevant* results. This paper shows that this set of interesting results corresponds to a least fixed point. We provide an implementation based on algebraic effect handlers to compute such least fixed points in a finite amount of time, thereby allowing non-determinism and recursion to meaningfully co-occur in a single program.

Categories and Subject Descriptors D.3 [*Programming Languages*]

Keywords Haskell, Tabling, Effect Handlers, Logic Programming, Non-determinism, Least Fixed Point

1. Introduction

Non-determinism [19] models a variety of problems in a declarative fashion, especially those problems where the solution depends on the exploration of different choices. The conventional approach represents non-determinism as lists of possible outcomes. For instance, consider the semantics of the following non-deterministic expression:

$$1 \mathbin{?} 2$$

This expression represents a non-deterministic choice (with the operator ?) between 1 and 2. Traditionally we model this with the list [1, 2]. Now consider the next example:

$$\begin{aligned} swap\ (m, n) &= (n, m) \\ pair &= (1, 2) \mathbin{?} swap\ pair \end{aligned}$$

The corresponding Haskell code is:

```
swap :: [(a, b)] → [(b, a)]
swap c = [(m, n) | (n, m) ← c]
pair :: [(Int, Int)]
pair = [(1,2)] ++ swap pair
```

This is an executable model (we use >>> to denote the prompt of the GHCi Haskell REPL):

```
>>> pair
[(1, 2), (2, 1), (1, 2), (2, 1) ...
```

We get an infinite list, although only two distinct outcomes ((1, 2) and (2, 1)) exist. The conventional list-based approach is clearly inadequate in this example. In this paper we model non-determinism with sets of values instead, such that duplicates are implicitly removed. The expected model of *pair* is then the set {(1, 2), (2, 1)}. We can execute this model:[1]

```
swap :: (Ord a, Ord b) ⇒ Set (a, b) → Set (b, a)
swap = map (λ(m, n) → (n, m))
pair :: Set (Int, Int)
pair = singleton (1, 2) `union` swap pair
>>> pair
fromList ...
```

Haskell lazily prints the first part of the *Set* constructor, and then has to compute union infinitely many times. As an executable model of non-determinism it clearly remains inadequate: it fails to compute the solution {(1, 2), (2, 1)}.

This paper solves the problem caused by the co-occurence of non-determinism and recursion, by recasting it as the least fixed point problem of a *different* function. The least fixed point is computed explicitly by iteration, instead of implicitly by Haskell's recursive functions.

The contributions of this paper are:

- We define a monadic model that captures both non-determinism *and* recursion. This yields a finite representation of recursive non-deterministic expressions. We use this representation as a light-weight (for the programmer) embedded Domain Specific Language to build non-deterministic expressions in Haskell.
- We give a denotational semantics of the model in terms of the least fixed point of a semantic function $\mathcal{R}[\![\cdot]\!]$. The semantics is subsequently implemented as a Haskell function that interprets the model.
- We generalize the denotational semantics to arbitrary complete lattices. We illustrate the added power on a simple graph problem, which could not be solved with the more specific semantics.
- We provide a set of benchmarks to demonstrate the expressivity of our approach and evaluate the performance of our implementation.

---

[1] Here map comes from `Data.Set`

non-determinism

=

multiple outcomes

1 ? 2

1 ? 2

non-deterministic choice

1 ? 2

non-deterministic choice

=

{1,2}

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

non-deterministic choice

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

non-deterministic choice + recursion

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

non-deterministic choice + recursion

=

?

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

non-deterministic choice

+

recursion

=

{(1,2)

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

non-deterministic choice + recursion

=

{(1,2),(2,1)}

```
swap (x,y) = (y,x)
pair = (1,2) ? swap pair
```

non-deterministic choice + recursion

=

the least fixed point

{(1,2),(2,1)}

```haskell
swap :: (Ord a, Ord b)
     => Set (a,b) -> Set (b,a)
swap = Set.map (\(x,y) -> (y,x))

pair :: Set (Int, Int)
pair = singleton (1,2) `union` swap pair
```

```haskell
swap :: (Ord a, Ord b)
     => Set (a,b) -> Set (b,a)
swap = Set.map (\(x,y) -> (y,x))

pair :: Set (Int, Int)
pair = singleton (1,2) `union` swap pair
```

```
ghci> pair
fromList [
```

program hangs

Fixing the Model: Syntax

```
data ND a
  = Success a
  | Fail
  | Or (ND a) (ND a)
```

A succesful computation

```
data ND a
  = Success a
  | Fail
  | Or (ND a) (ND a)
```

A succesful computation

```
data ND a
  = Success a
  | Fail
  | Or (ND a) (ND a)
```

A failed computation

A succesful computation

```
data ND a
  = Success a
  | Fail
  | Or (ND a) (ND a)
```

A failed computation

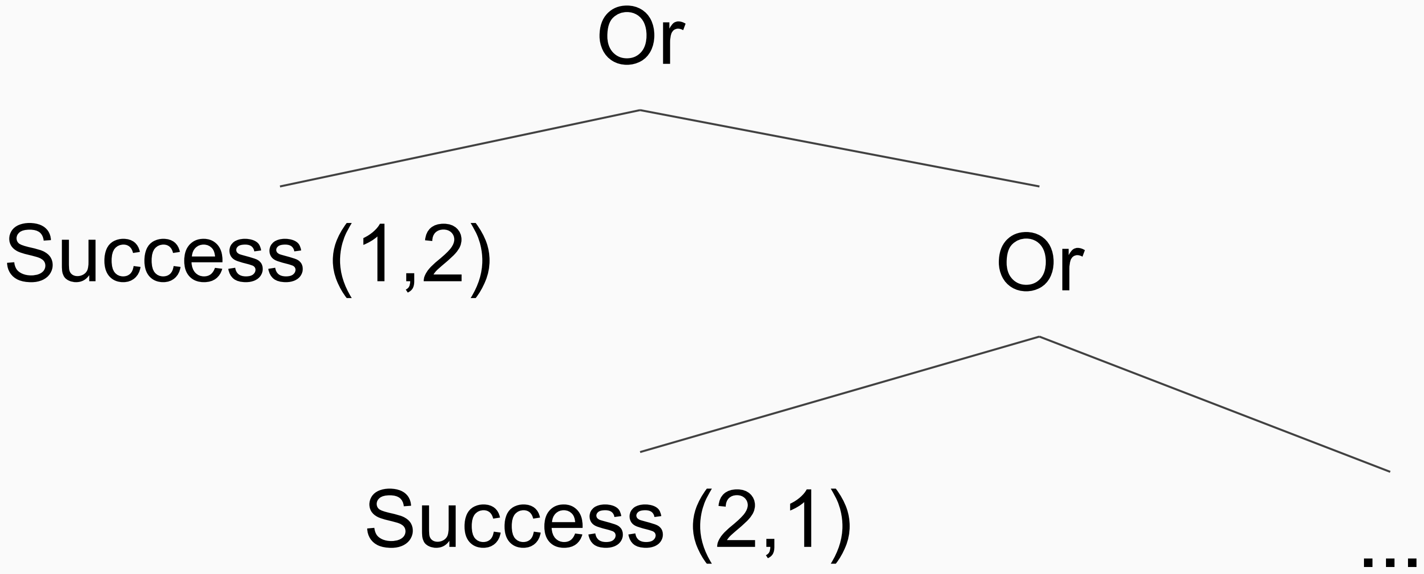
A non-deterministic choice

A free monad

```
instance Monad ND where
  return = Success
  Success a >>= f = f a
  Fail      >>= f = Fail
  Or l r    >>= f = Or (l >>= f) (r >>= f)
```

```
data ND a
  = Success a
  | Fail
  | Or (ND a) (ND a)
```

```
pairND :: ND (Int,Int)
pairND = return (1,2)`Or`do (x,y) <- pairND
                            return (y,x)
```



Or

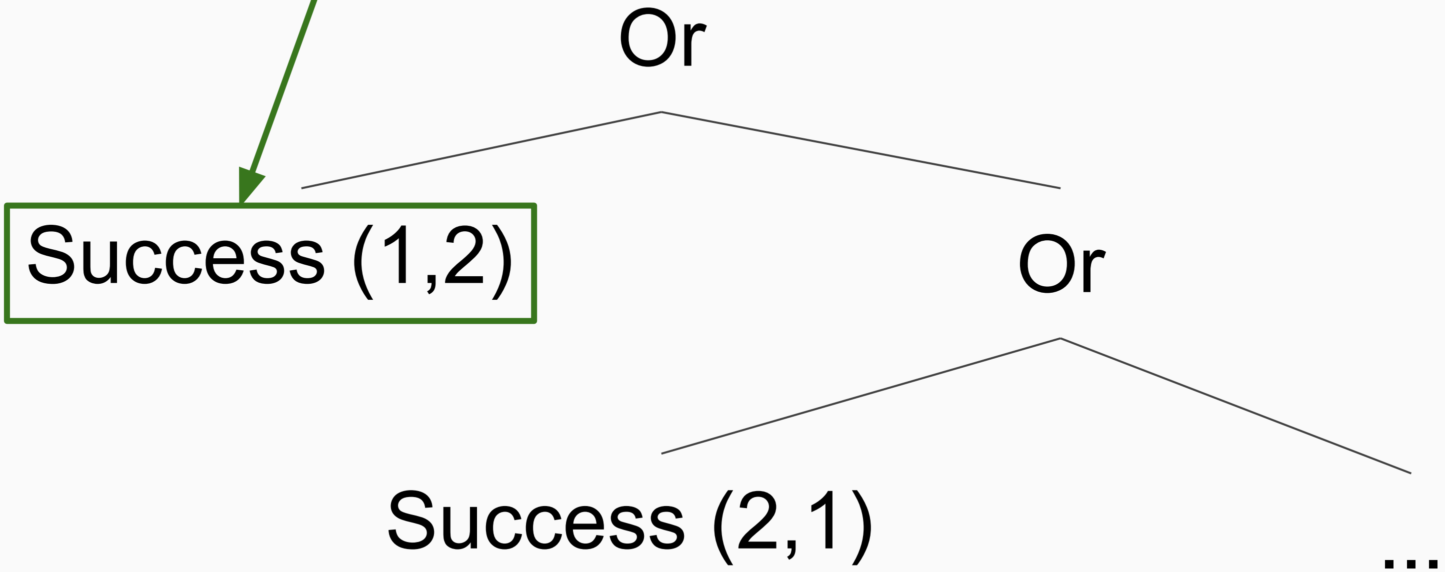
Success (1,2)

Or

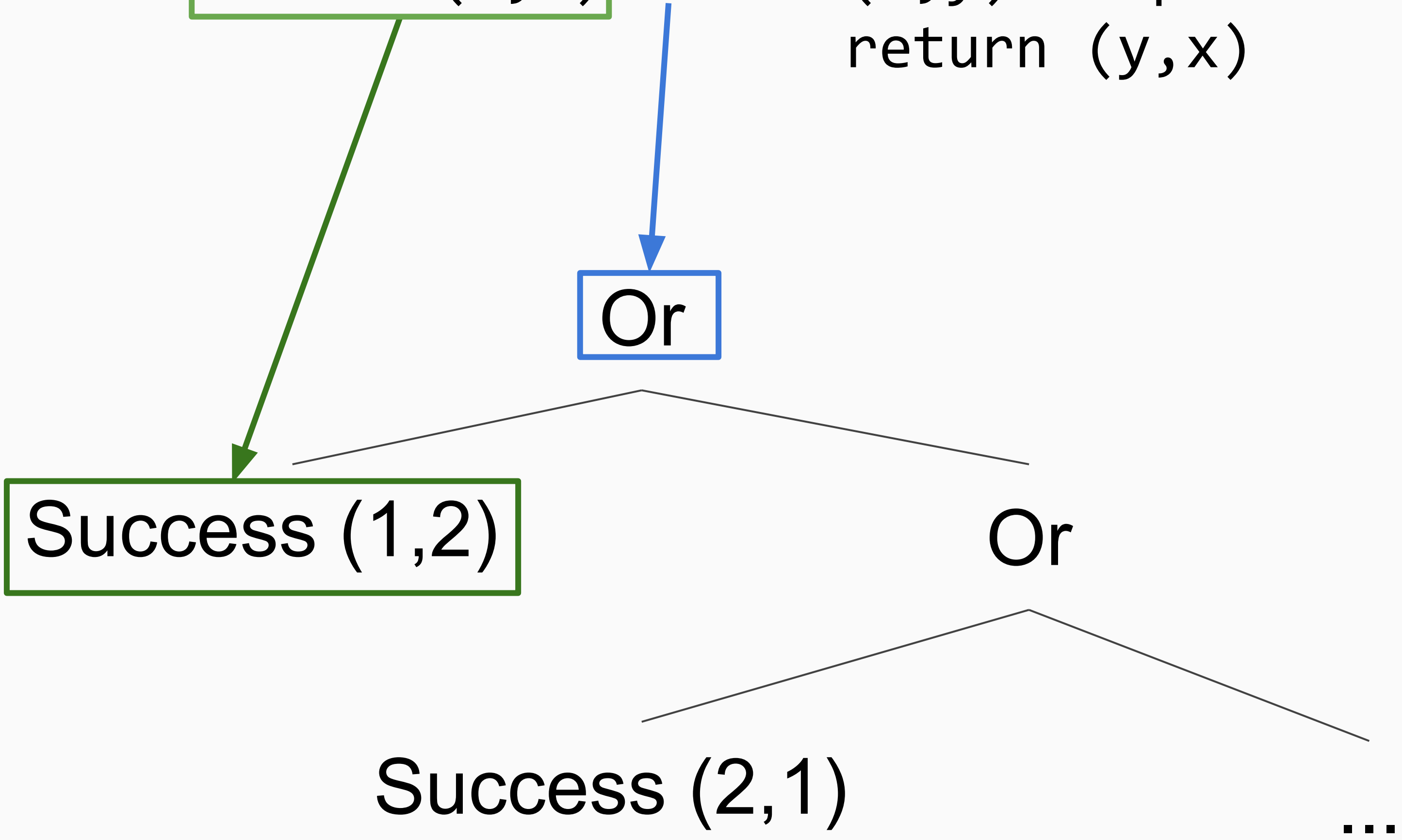
Success (2,1)

...

```
pairND :: ND (Int,Int)
pairND = return (1,2)`Or`do (x,y) <- pairND
                            return (y,x)
```



```
pairND :: ND (Int,Int)
pairND = return (1,2)`Or`do (x,y) <- pairND
                            return (y,x)
```



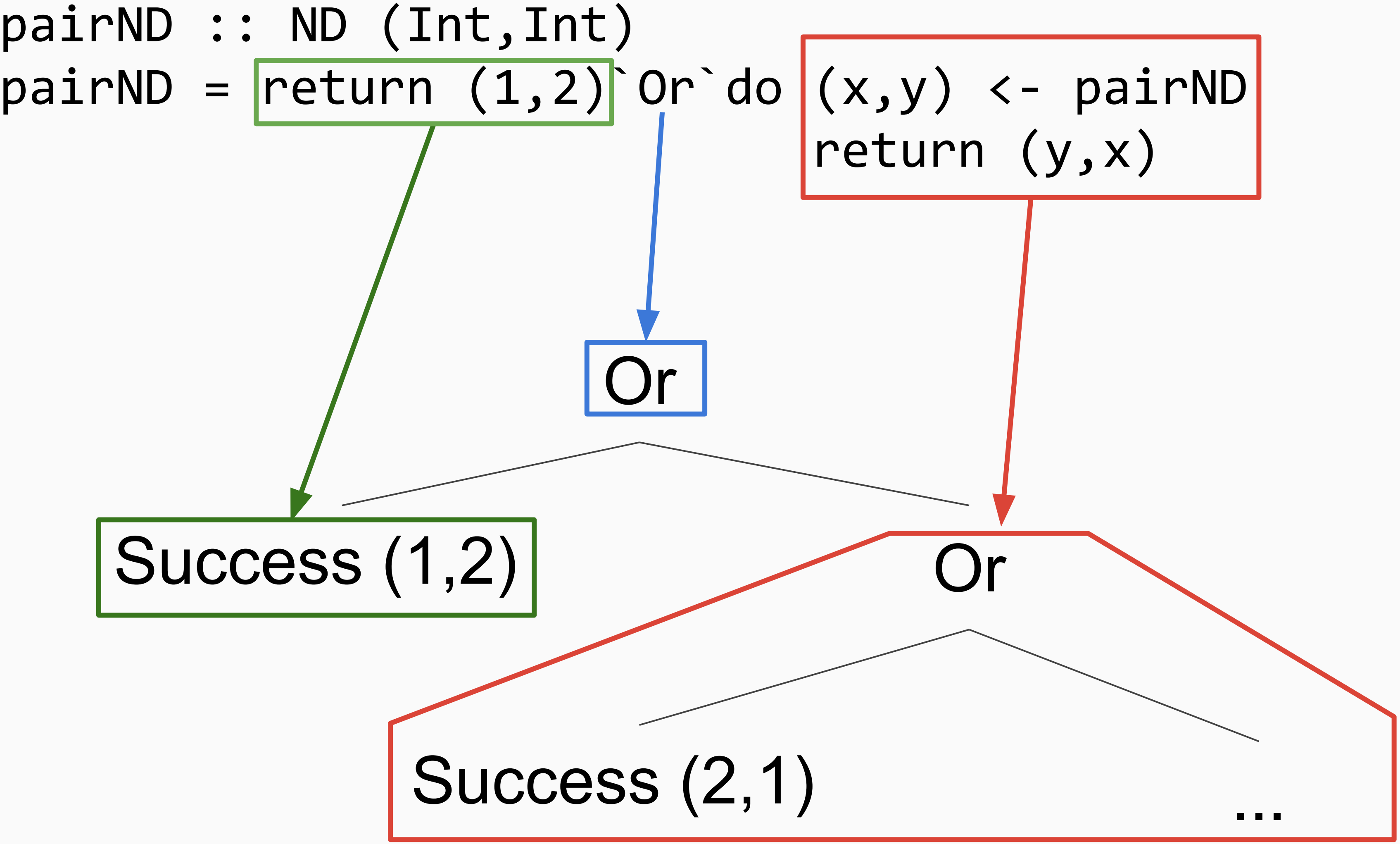
Or

Success (1,2)

Or

Success (2,1)

...



```
pairND :: ND (Int,Int)
pairND = return (1,2)`Or`do (x,y) <- pairND
                            return (y,x)
```

Or

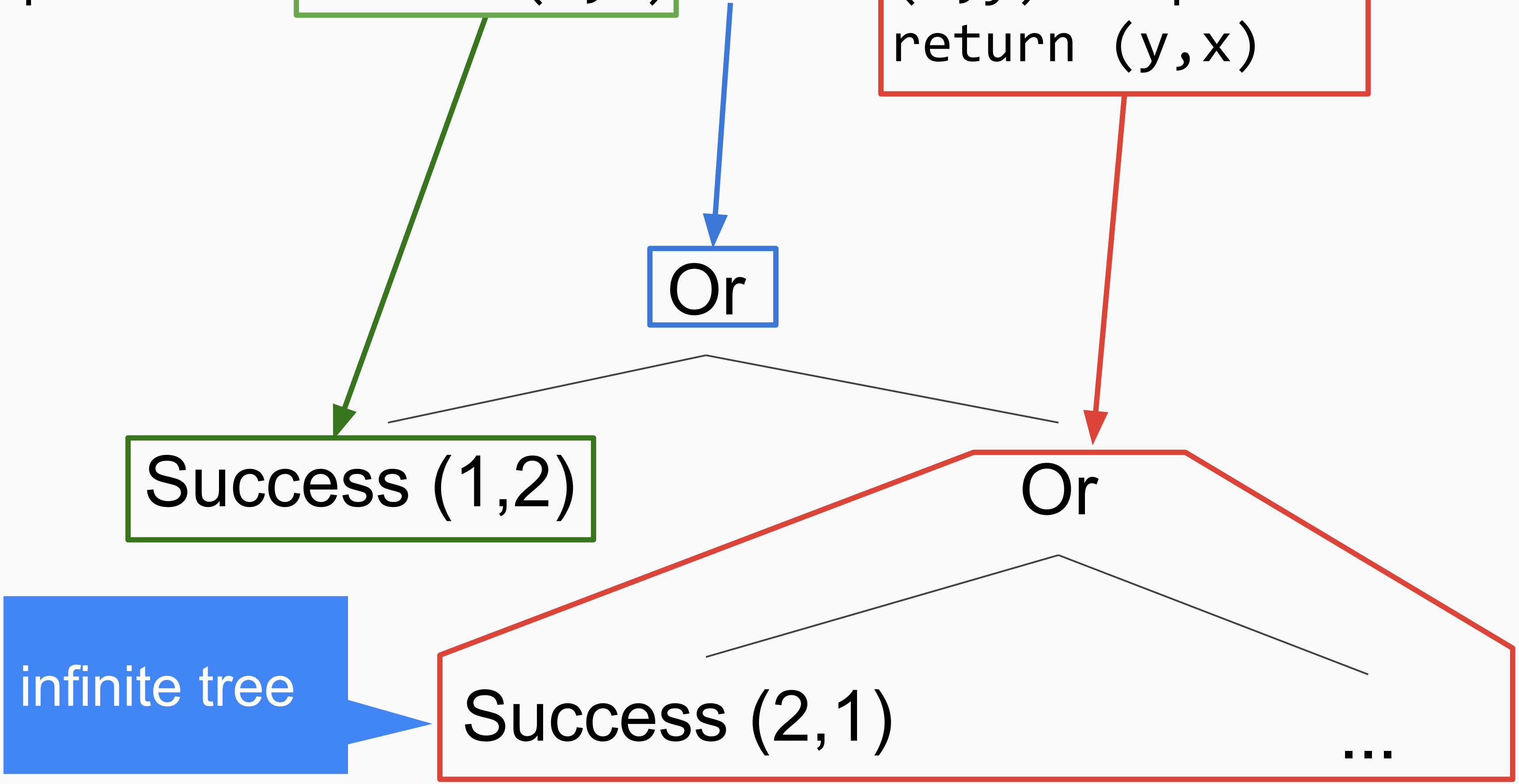
Success (1,2)

Or

Success (2,1)

...

```
pairND :: ND (Int,Int)
pairND = return (1,2)`Or`do (x,y) <- pairND
                            return (y,x)
```



Or

Success (1,2)

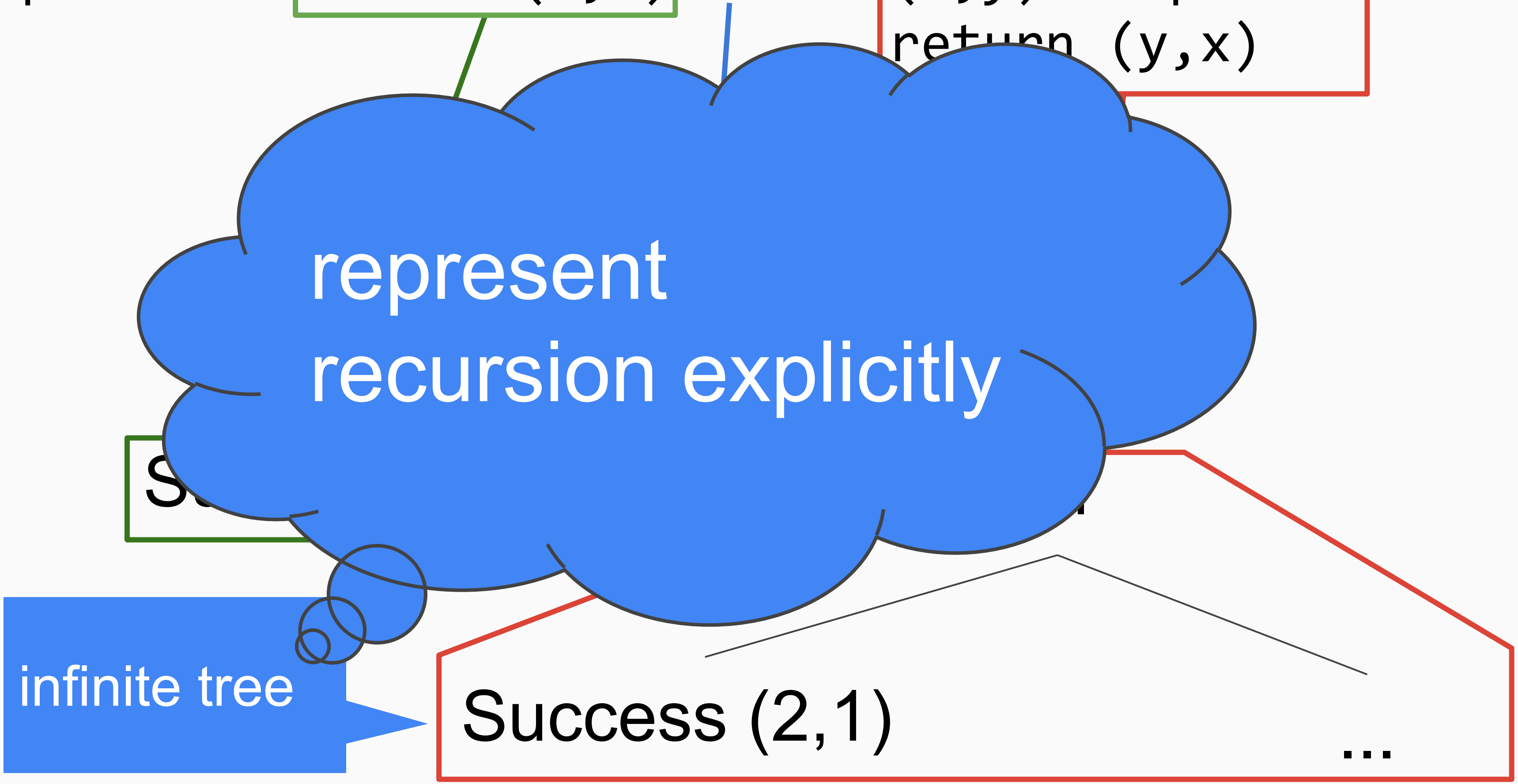
Or

Success (2,1)

...

infinite tree

```
pairND :: ND (Int,Int)
pairND = return (1,2)`Or`do (x,y) <- pairND
                            return (y,x)
```



represent
recursion explicitly

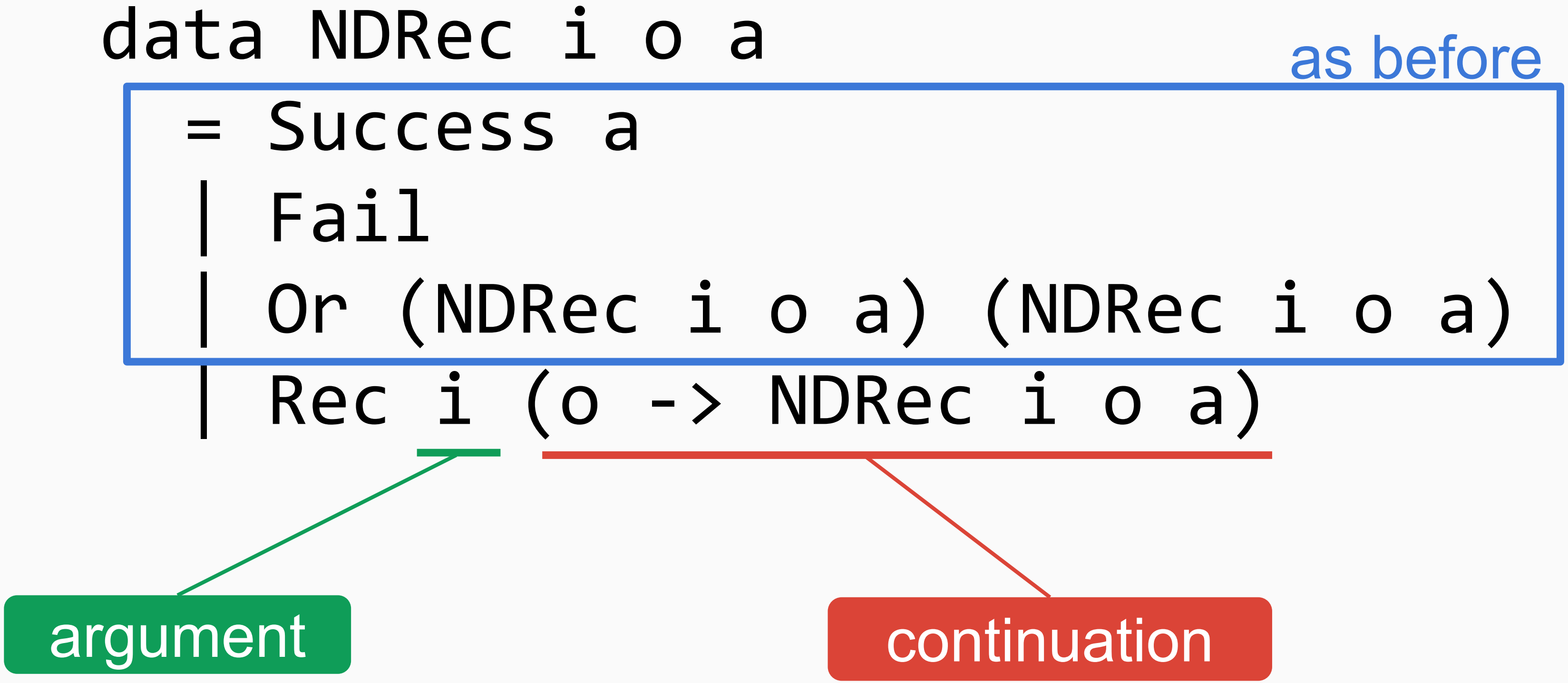
S

infinite tree

Success (2,1)

...

Explicit Recursion



```
data NDRec i o a
  = Success a
  | Fail
  | Or (NDRec i o a) (NDRec i o a)
  | Rec i (o -> NDRec i o a)
```

as before

argument

continuation

Explicit Recursion

```
data NDRec i o a
  = Success a
  | Fail
  | Or (NDRec i o a) (NDRec i o a)
  | Rec i (o -> NDRec i o a)
```

```
rec :: i -> NDRec i o o
rec i = Rec i Success
```

Explicit Recursion

```
data NDRec i o a
  = Success a
  | Fail
  | Or (NDRec i o a) (NDRec i o a)
  | Rec i (o -> NDRec i o a)
```

```
choice :: [NDRec i o a] -> NDRec i o a
choice = foldr Or Fail
```

Explicit Recursion

```
data NDRec i o a
  = Success a
  | Fail
  | Or (NDRec i o a) (NDRec i o a)
  | Rec i (o -> NDRec i o a)
```

```
choose :: [a] -> NDRec i o a
choose = choice . map return
```

Explicit Recursion

```
data NDRec i o a
  = Success a
  | Fail
  | Or (NDRec i o a) (NDRec i o a)
  | Rec i (o -> NDRec i o a)
```

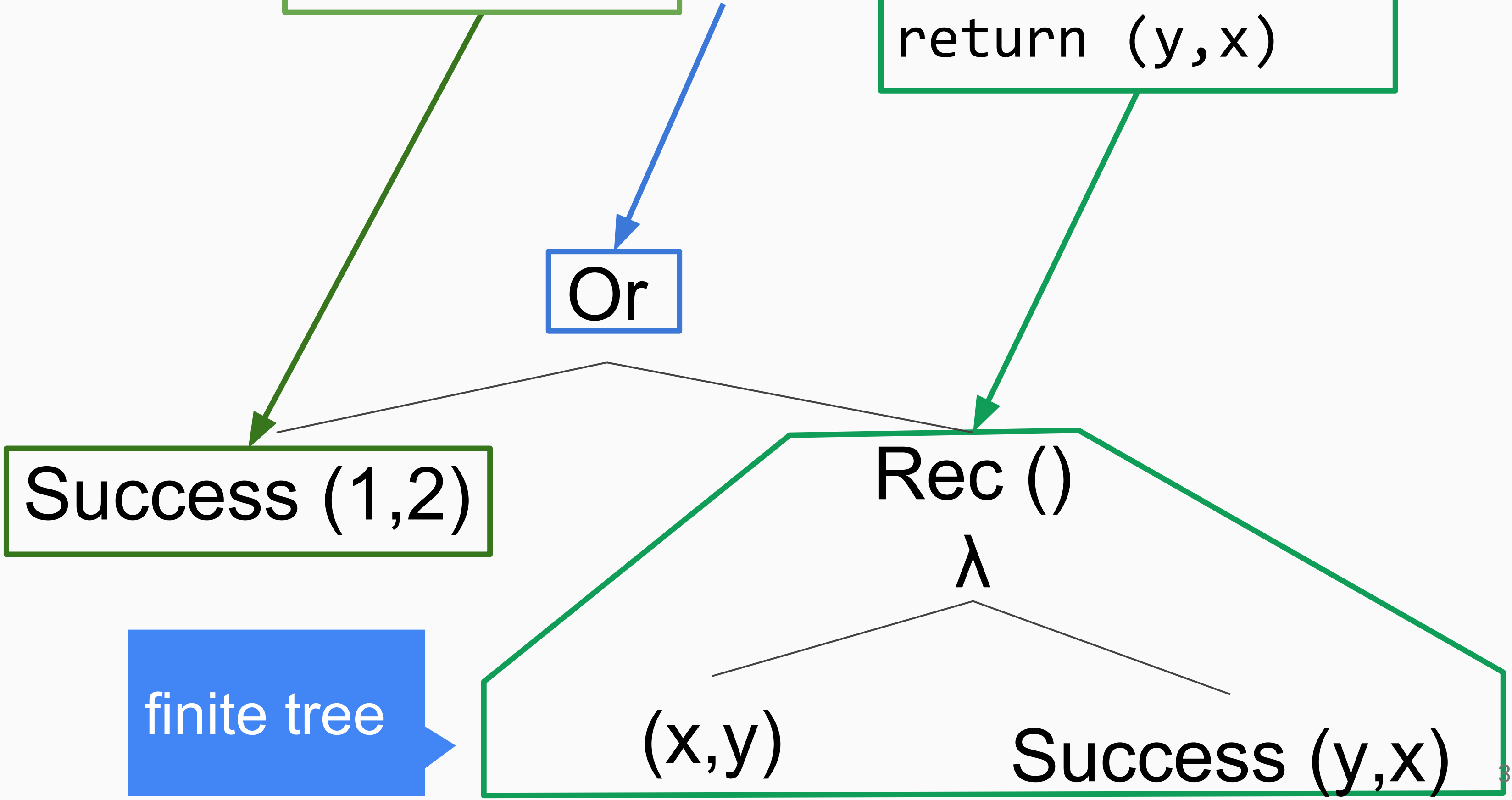
```
guard :: Bool -> NDRec i o ()
guard b = if b then return () else Fail
```

Explicit Recursion

```
data NDRec i o a
  = Success a
  | Fail
  | Or (NDRec i o a) (NDRec i o a)
  | Rec i (o -> NDRec i o a)
```

```
instance Monad (NDRec i o) where
  return = Success
  Rec i k >>= f = Rec i (\x -> k x >>= f)
```

```
pairND :: () -> ND (Int,Int)
pairND () = return (1,2)`Or`do (x,y) <- rec ()
                               return (y,x)
```



Or

Success (1,2)

Rec ()

λ

(x,y)

Success (y,x)

finite tree

Fixing the Model: Semantics

[| . |] : (I -> NDRec I O O) -> (I -> P(O))

[| . |]

$$[|\ .\ |] : \overbrace{(I \text{ -> } NDRec\ I\ O\ O)}^{\text{function constructing syntax tree}} \text{ -> } \underbrace{(I \text{ -> } P(O))}_{\text{function computing set of outcomes}}$$

R[| . |]

$$\text{R[| . |] : }\overbrace{\text{NDRec I O O}}^{\text{syntax tree}}\text{ -> }\overbrace{\text{(I -> P(O))}}^{\text{environment}}\text{ -> }\underbrace{\text{P(O)}}_{\text{set of outcomes}}$$

R[| . |]

$$R[|\ \text{Succes}\ x\ |](s) = \{x\}$$

$$R[|\ \text{Fail}\ |](s) = \varnothing$$

$$R[|\ \text{Or}\ l\ r\ |](s) = R[|\ l\ |](s) \cup R[|\ r\ |](s)$$

$$R[|\ \text{Rec}\ i\ k\ |](s) = \bigcup_{x \in s(i)} R[|\ k(x)\ |](s)$$

[| . |] and R[| . |]

$$\forall a: [\![f]\!](a) = R[\![f(a)]\!]([\![f]\!])$$

[| . |] and R[| . |]

$$\forall a: [\![f]\!](a) = R[\![f(a)]\!]([\![f]\!])$$

$$\Leftrightarrow$$

$$[\![f]\!] \text{ is a fixpoint of } \lambda s.\lambda a.R[\![f(a)]\!](s)$$

[| . |] and R[| . |]

$$\forall a\!: [|\ f\ |](a) = R[|\ f(a)\ |]([|\ f\ |])$$

$$\Leftrightarrow$$

$$[|\ f\ |] \text{ is a } \underset{\textcolor{red}{\text{least}}}{\textcolor{red}{\wedge}} \text{ fixpoint of } \lambda s.\lambda a.R[|\ f(a)\ |](s)$$

[| . |] and R[| . |]

$$\forall a: [\![f]\!](a) = R[\![f(a)]\!]([\![f]\!])$$

$$\Leftrightarrow$$

$$[\![f]\!] \text{ is a fixpoint of } \lambda s.\lambda a.R[\![f(a)]\!](s)$$

least: $s_1 \sqsubseteq s_2 \Leftrightarrow \forall x: s_1(x) \subseteq s_2(x)$

Example

```
pairND :: () -> ND (Int,Int)
pairND () = return (1,2)`Or`do (x,y) <- rec ()
                               return (y,x)
```

$$[|\ \texttt{pairND}\ |](())$$

$$=$$

$$\mathrm{lfp}(\lambda s.\lambda().R[|\ \texttt{pairND ()}\ |](s))(())$$

$$=$$

$$(\lambda().\{(1,2),(2,1)\}\ ())$$

$$=$$

$$\{(1,2),(2,1)\}$$

Implementation

[| . |]

```
runNDRec :: (Ord i, Ord o)
         => (i -> NDRec i o o) -> (i -> Set o)
runNDRec expr i0 = lfp step s0 ! i0 where
  s0 = M.singleton i0 empty
  step m = foldr (\k -> go k (expr k)) m (M.keys m)
```

R[| . |]

```
go :: (Ord i, Ord o)
   => i
   -> NDRec i o o
   -> (M.Map i (Set o) -> M.Map i (Set o))

go i (Success a) = M.insertWith union i (singleton x)
go i Fail        = id
go i (Or l r)    = go i r . go i l
go i (Rec j k)   = \m -> case M.lookup j m of
  Nothing -> M.insert j empty m
  Just s  -> foldr (go i . k) m (toList s)
```

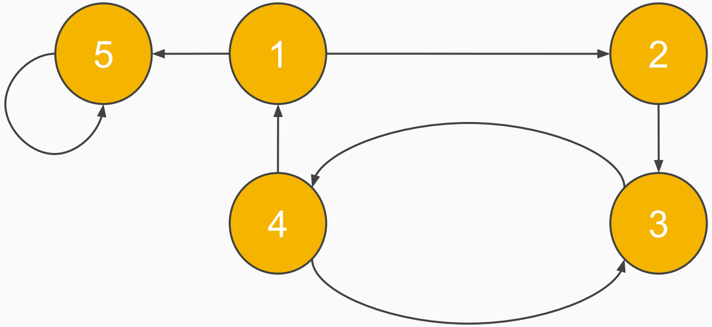
Implementation

```
ghci> runNDRec pairND ()
fromList [(1,2),(2,1)]
```

Success!

Lattices

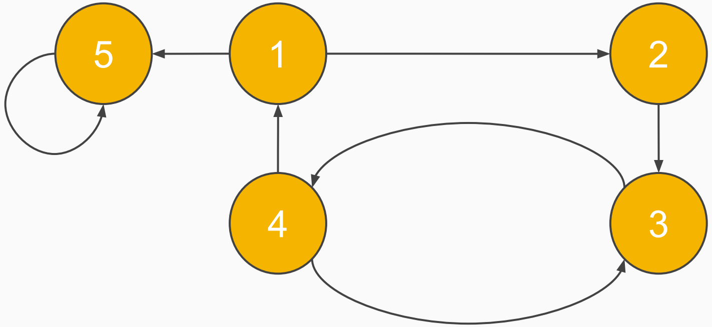
Distance in a cyclic graph



```
data Node = Node {
  label :: Int
  adj   :: [Node]
}
```

```
[n1,n2,n3,n4,n5] = [
  Node 1 [n2,n5], Node 2 [n3],
  Node 3 [n4], Node 4 [n1,n3],
  Node 5 [n5]]
```

Distance in a cyclic graph



```
dist :: Num a => Node -> Node -> NDRec Node a a
dist dst src | src == dst = return 0
             | otherwise  = choose (adj src)
                            >>= rec
                            >>= return . (+ 1)
```

Distance in a cyclic graph

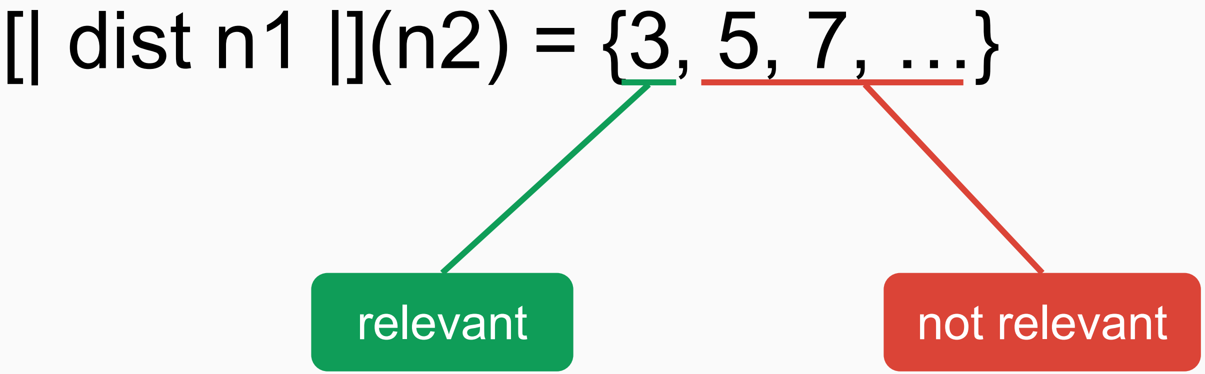
```
ghci> runNDRec (dist n1) n2
fromList [
```

The program hangs.

Distance in a cyclic graph

$$[|\ \text{dist n1}\ |](\text{n2}) = \{\underline{3}, \underline{5, 7, \ldots}\}$$

Distance in a cyclic graph

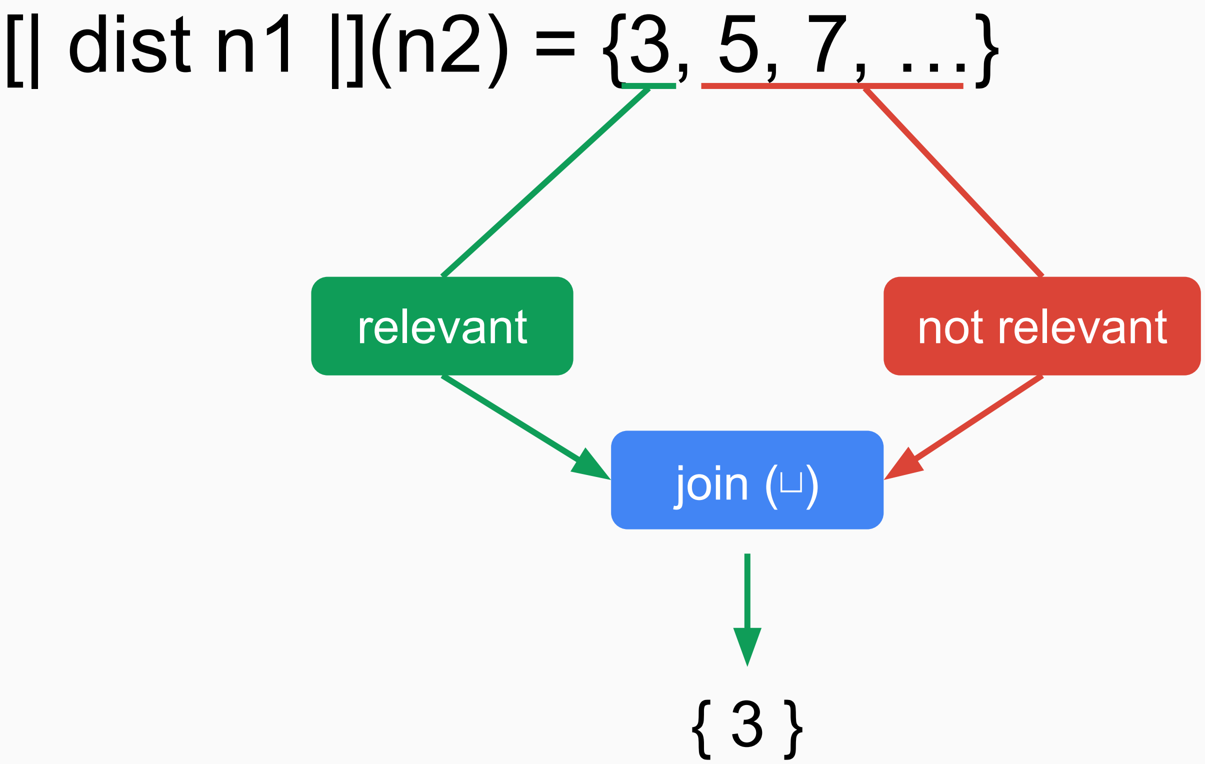


[| dist n1 |](n2) = {3, 5, 7, …}

relevant

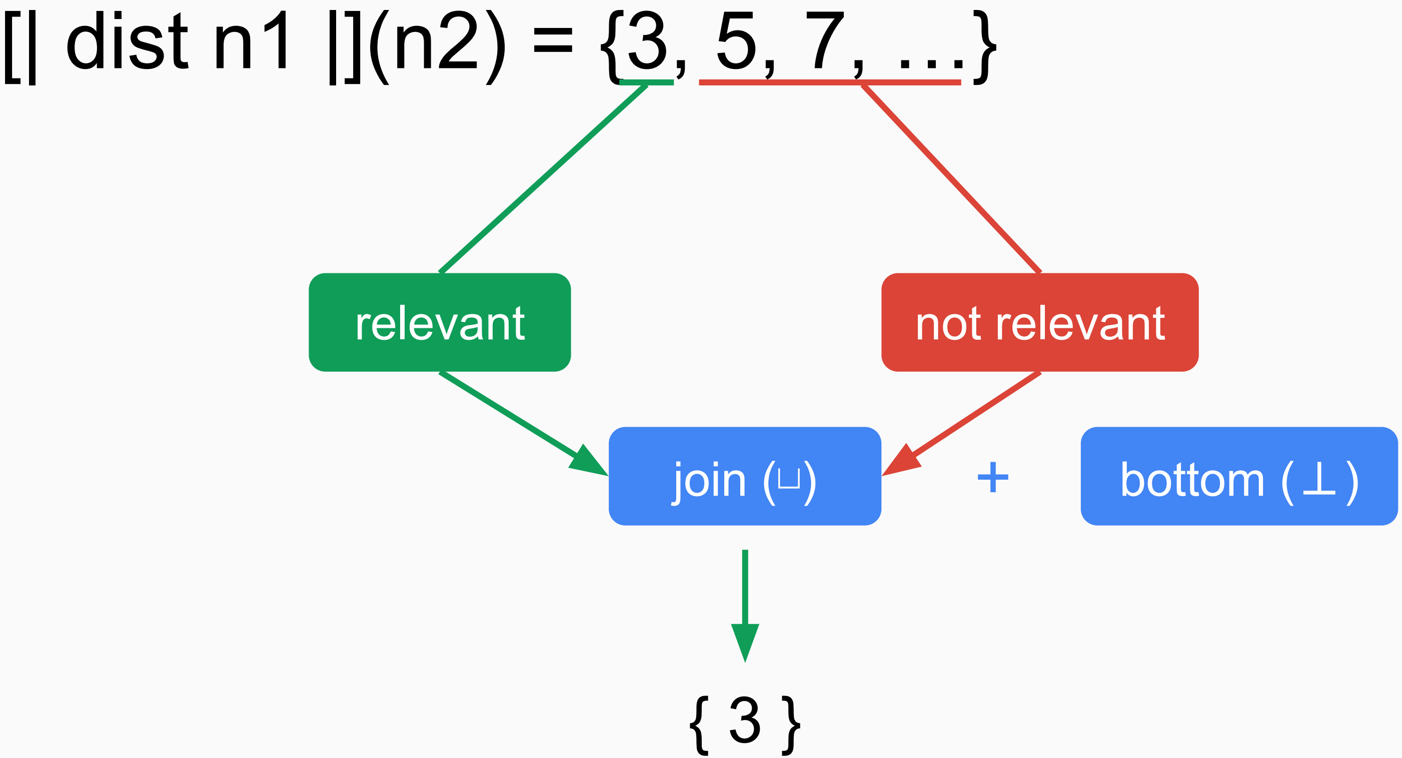
not relevant

Distance in a cyclic graph



[| dist n1 |](n2) = {3, 5, 7, …}

Distance in a cyclic graph



$$[\!|\ \text{dist n1}\ |\!](\text{n2}) = \{\underbrace{3}_{\text{relevant}}, \underbrace{5, 7, \ldots}_{\text{not relevant}}\}$$

relevant

not relevant

join (⊔) + bottom (⊥)

{ 3 }

(Complete) Lattice

```
class Lattice l where
  join   :: l -> l -> l
  bottom :: l
```

Distance lattice

```
data Dist = InfDist | Dist Int deriving Eq
```

```
instance Lattice Dist where
  bottom = InfDist
  join InfDist x = x
  join x InfDist = x
  join (Dist x) (Dist y) = Dist (min x y)
```

```
instance Num Dist where
  ...
```

Semantics $[|\ .\ |]_L$

$$[|\ .\ |]_L : \overbrace{(I -> NDRec\ I\ L\ L)}^{\text{function that builds a syntax tree}} -> \underbrace{(I -> L)}_{\text{function computing a lattice of outcomes}}$$

Semantics $R[| \, . \, |]_L$

$$R[| \, . \, |]_L : \overbrace{\mathrm{NDRec}\ I\ L\ L}^{\text{syntax tree}} \text{ -> } \underbrace{(I \text{ -> } L)}_{\text{environment}} \text{ -> } \overbrace{L}^{\text{lattice of outcomes}}$$

Semantics $R[| \, . \, |]_L$

$$R[| \text{ Succes } x \; |](s) = \{x\}$$

$$R[| \text{ Fail } |](s) = \varnothing$$

$$R[| \text{ Or } l \; r \; |](s) = R[| \; l \; |](s) \cup R[| \; r \; |](s)$$

$$R[| \text{ Rec } i \; k \; |](s) = \bigcup_{x \in s(i)} R[| \; k(x) \; |](s)$$

Semantics $R[\![\,.\,]\!]_L$

$$R[\![\text{ Succes } x \ ]\!]_L(s) = x$$

$$R[\![\text{ Fail }]\!](s) = \varnothing$$

$$R[\![\text{ Or } l \ r \ ]\!](s) = R[\![\ l \ ]\!](s) \cup R[\![\ r \ ]\!](s)$$

$$R[\![\text{ Rec } i \ k \ ]\!](s) = \bigcup_{x \in s(i)} R[\![\ k(x) \ ]\!](s)$$

Semantics $R[| . |]_L$

$$R[|\ \text{Succes}\ x\ |]_L(s) = x$$

$$R[|\ \text{Fail}\ |]_L(s) = \perp$$

$$R[|\ \text{Or}\ l\ r\ |](s) = R[|\ l\ |](s) \cup R[|\ r\ |](s)$$

$$R[|\ \text{Rec}\ i\ k\ |](s) = \bigcup_{x \in s(i)} R[|\ k(x)\ |](s)$$

Semantics $R[\!|\ .\ |\!]_L$

$$R[\!|\ \text{Succes}\ x\ |\!]_L(s) = x$$

$$R[\!|\ \text{Fail}\ |\!]_L(s) = \bot$$

$$R[\!|\ \text{Or}\ l\ r\ |\!]_L(s) = R[\!|\ l\ |\!]_L(s) \sqcup R[\!|\ r\ |\!]_L(s)$$

$$R[\!|\ \text{Rec}\ i\ k\ |\!](s) = \bigcup_{x \in s(i)} R[\!|\ k(x)\ |\!](s)$$

Semantics $R[\![\,.\,]\!]_L$

$R[\![\text{Succes } x]\!]_L(s) = x$

$R[\![\text{Fail}]\!]_L(s) = \bot$

$R[\![\text{Or } l \; r]\!]_L(s) = R[\![l]\!]_L(s) \sqcup R[\![r]\!]_L(s)$

$R[\![\text{Rec } i \; k]\!]_L(s) = R[\![k(s(i))]\!]_L(s)$

Semantics R[| . |]$_L$

$$[| \; . \; |]_L : (I \to \mathrm{NDRec}\ I\ O\ O) \to (I \to L)$$

$$[| \; f \; |]_L = \mathrm{lfp}(\lambda s.\lambda a.R[| \; f(a) \; |]_L(s))$$

Semantics $R[| \,.\, |]_L$

$$[| \,.\, |]_L : (I \to NDRec\ I\ O\ O) \to (I \to L)$$

$$[| f |]_L = \xcancel{lfp(\lambda s.\lambda a.R[| f(a) |]_L(s))}$$

Semantics $R[| \, . \, |]_L$

$$[| \, . \, |]_L : (I \to NDRec \; I \; O \; O) \to (I \to L)$$

$$[| \, f \, |]_L = \lambda s.\lambda a.R[| \, f(a) \, |]_L(s) \uparrow \infty$$

$$f \uparrow 0 = \bot$$

$$f \uparrow i + 1 = f \uparrow i \sqcup f(f \uparrow i)$$

$$f \uparrow \infty = \bigsqcup\{f \uparrow i \mid i \in N\}$$

Example

$$[|\ \mathrm{dist}\ n_1\ |]_{\mathrm{Dist}}(n_2)$$

$$=$$

$$\mathrm{Dist}\ 3$$

Lattices generalize sets

Sets are lattices too

```
instance Ord a => Lattice (Set a) where
  join   = S.union
  bottom = S.empty
```

Lattices generalize sets

Sets are lattices too

$$R[\![\text{Rec i } k']\!]_L(s) = R[\![\text{Rec i } k]\!](s)$$

where

k’ = fmap unions . traverse k . toList

Dependency Tracking

A smarter least fixed point

```
fib :: Int -> NDRec Int Int Int
fib i | i <= 0     = return 0
      | i == 1     = return 1
      | otherwise = do n1 <- rec (i-1)
                       n2 <- rec (i-2)
                       return (n1 + n2)
```

A smarter least fixed point

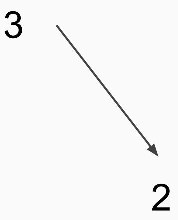
```
fib :: Int -> NDRec Int Int Int
fib i | i <= 0      = return 0
      | i == 1      = return 1
      | otherwise = do n1 <- rec (i-1)
                       n2 <- rec (i-2)
                       return (n1 + n2)
```

3

3:{}

A smarter least fixed point

```
fib :: Int -> NDRec Int Int Int
fib i | i <= 0    = return 0
      | i == 1    = return 1
      | otherwise = do n1 <- rec (i-1)
                       n2 <- rec (i-2)
                       return (n1 + n2)
```



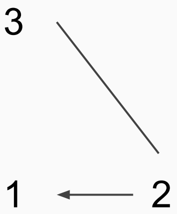
3

2

3:{}
3:{},2:{}

A smarter least fixed point

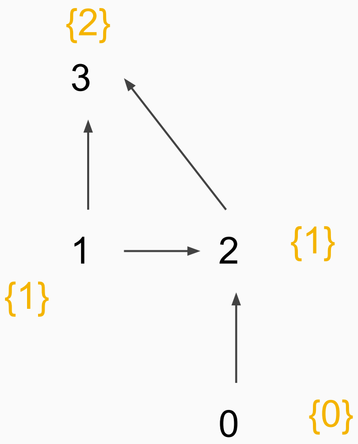
```
fib :: Int -> NDRec Int Int Int
fib i | i <= 0      = return 0
      | i == 1      = return 1
      | otherwise = do n1 <- rec (i-1)
                        n2 <- rec (i-2)
                        return (n1 + n2)
```



3:{}
3:{},2:{}
3:{},2:{},1:{}

A smarter least fixed point

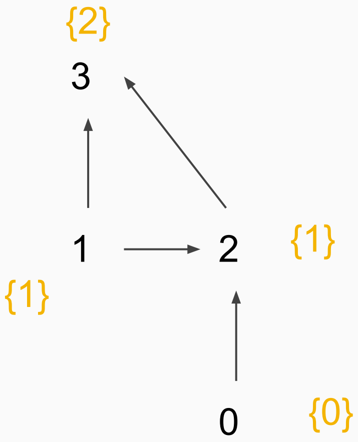
```haskell
fib :: Int -> NDRec Int Int Int
fib i | i <= 0     = return 0
      | i == 1     = return 1
      | otherwise = do n1 <- rec (i-1)
                       n2 <- rec (i-2)
                       return (n1 + n2)
```



```
3:{}
3:{},2:{}
3:{},2:{},1:{}
3:{},2:{},1:{1}
3:{},2:{},1:{1},0:{}
3:{},2:{},1:{1},0:{0}
3:{},2:{1},1:{1},0:{0}
3:{2},2:{1},1:{1},0:{0}
3:{2},2:{1},1:{1},0:{0}
```

A smarter least fixed point

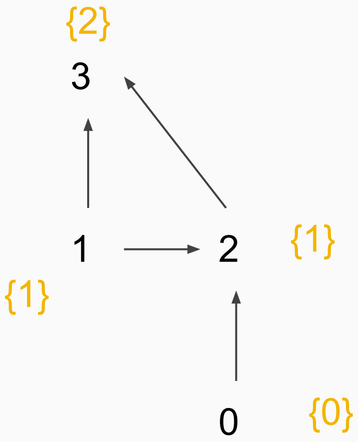
```haskell
fib :: Int -> NDRec Int Int Int
fib i | i <= 0     = return 0
      | i == 1     = return 1
      | otherwise = do n1 <- rec (i-1)
                       n2 <- rec (i-2)
                       return (n1 + n2)
```



```
3:{}
3:{},2:{}
3:{},2:{},1:{}
3:{},2:{},1:{1}
3:{},2:{},1:{1},0:{}
3:{},2:{},1:{1},0:{0}
3:{},2:{1},1:{1},0:{0}
3:{2},2:{1},1:{1},0:{0}
3:{2},2:{1},1:{1},0:{0}
```

A smarter least fixed point

```
fib :: Int -> NDRec Int Int Int
fib i | i <= 0     = return 0
      | i == 1     = return 1
      | otherwise = do n1 <- rec (i-1)
                        n2 <- rec (i-2)
                        return (n1 + n2)
```



3:{}
3:{},2:{}
3:{},2:{},1:{}
3:{},2:{},1:{1}
3:{},2:{},1:{1},0:{}
3:{},2:{},1:{1},0:{0}
3:{},2:{1},1:{1},0:{0}
3:{2},2:{1},1:{1},0:{0}
3:{2},2:{1},1:{1},0:{0}

O(n^2) becomes O(n)

Wrapping Up

Compute Least Fixed Point Explicitly

+

Generalize to Lattices

+

Generalize to Mutual recursion

+

Add Dependency Tracking

Questions?