The Table Monad in Haskell

Alexander Vandenbroucke,
Tom Schrijvers & Frank Piessens

aka Fixing Non-determinism

Alexander Vandenbroucke,
Tom Schrijvers & Frank Piessens

Non-determinism

1 ⸮ 2

Non-determinism

$1 \ ? \ 2 \longrightarrow \{1, 2\}$

Non-determinism

```
1 + (1 ? 2)
```

Non-determinism

`1 + (1 ? 2)` → {2, 3}

Non-determinism

```
(1 ? 2) + (1 ? 2)  ⟶
```

Non-determinism

`(1 ? 2) + (1 ? 2)` $\longrightarrow$ $\{2, 3, 4\}$

Non-determinism in Haskell

```
1 ? 2
```

Non-determinism in Haskell

```
1 ? 2
```

```
S.singleton 1
`S.union`
S.singleton 2
```

Non-determinism in Haskell

```
1 + (1 ? 2)
```

→

```
S.map (1+)
 (S.singleton 1
  `S.union`
  S.singleton 2)
```

Non-determinism in Haskell

`(1 ? 2) + (1 ? 2)` → too tedious

Non-determinism in Haskell

`(1 ? 2) + (1 ? 2)` → too tedious

⇒ use monadic model

Non-determinism + Recursion

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

Non-determinism + Recursion

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

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

Non-determinism + Recursion

swap (x,y) = (y,x)
f = (1,2) ? swap f

```
f = S.singleton (1,2)
    `S.union`
    S.map swap f
```

```
ghci> f
```

Non-determinism + Recursion

swap (x,y) = (y,x)
f = (1,2) ? swap f

```
f = S.singleton (1,2)
    `S.union`
    S.map swap f
```

```
ghci> f
S.fromList <diverges>
```

Non-determinism + Recursion

swap (x,y) = (y,x)
f = (1,2) ? swap f

```
f = S.singleton (1,2)
    `S.union`
    S.map swap f
```

```
ghci> f
S.fromList <diverges>
```

use monadic model with correct recursive semantics

i.e. correct fixpoint semantics

Monadic Model

Monadic Model

f = (1,2) ? swap f → Monad

A brief detour: Open Recursion

```
type Open s = s -> s
```

A brief detour: Open Recursion

f = (1,2) ? swap f →

```haskell
type Open s = s -> s
```

```haskell
f0 :: Open (() -> Set (Int, Int))
f0 rec () = S.singleton (1,2)
            `S.union`
            S.map swap (rec ())
```

A brief detour: Open Recursion

```haskell
type Open s = s -> s
```

f = (1,2) ? swap f

```haskell
f0 :: Open (() -> Set (Int, Int))
f0 rec () = S.singleton (1,2)
            `S.union`
            S.map swap (rec ())
```

the recursive call becomes an argument of type
`() -> Set (Int, Int)`

the actual argument

A brief detour: Open Recursion

```haskell
f = S.singleton (1,2)
    `S.union`
    S.map swap f
```

```haskell
f0 :: Open (() -> Set (Int, Int))
f0 open () = S.singleton (1,2)
             `S.union`
             S.map swap (open ())
```

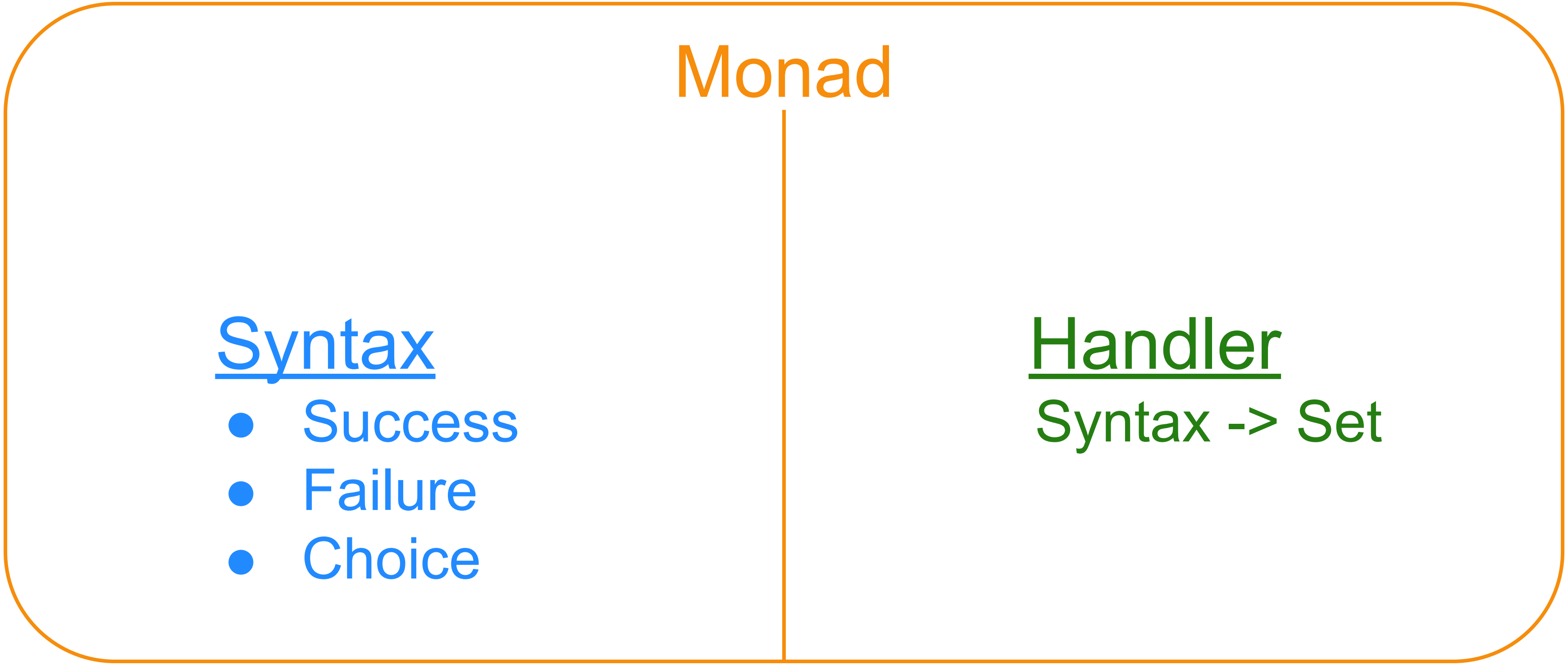
```haskell
fix :: (a -> a) -> a
fix f = f (fix f)
```

```haskell
f = fix f0 ()
```

Monadic Model

f = (1,2) ? swap f → Monad

Monadic Model: Effect Handlers



Monadic Model: Syntax

```
data ND arg res a
  = Success a
  | Fail
  | Or (ND arg res a) (ND arg res a)
  | Rec arg (res -> ND arg res a)
```

Monadic Model: Syntax

```
data ND arg res a
  = Success a
  | Fail
  | Or (ND arg res a) (ND arg res a)
  | Rec arg (res -> ND arg res a)
```

```
rec :: arg -> ND arg res res
rec i = Rec i Success
```

Monadic Model: Syntax

```
data ND arg res a
  = Success a
  | Fail
  | Or (ND arg res a) (ND arg res a)
  | Rec arg (res -> ND arg res a)
```

```
rec :: arg -> ND arg res res
rec i = Rec i Success
```

```
instance Monad (ND arg res) where
  return = Success
  Success a >>= f = f a
  Fail      >>= f = Fail
  Or l r    >>= f = Or (l >>= f) (r >>= f)
  Rec i k   >>= f = Rec i (\o -> k o >>= f)
```

Monadic Model: Handler (1/2)

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

open recursive function

argument to regular function

Monadic Model: Handler (1/2)

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

least fixed point operator

```
lfp :: Eq a => a -> (a -> a) -> a
lfp a0 f = let a1 = f a0
           in if a0 == a1 then
                 a1
              else
                 lfp a1 f
```

Monadic Model: Handler (1/2)

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

least fixed point operator

initial state

step function

lookup `arg0` on final state

Monadic Model: Handler (1/2)

```haskell
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
  -- s0 :: Map arg (Set res)
  s0 = M.singleton arg0 S.empty
```

Monadic Model: Handler (1/2)

```haskell
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
  -- s0 :: Map arg (Set res)
  s0 = M.singleton arg0 S.empty

  -- step :: Map arg (Set res) -> Map arg (Set res)
  step m = foldr (\k -> go k (open rec k)) m (M.keys m)
```

Monadic Model: Handler (1/2)

```haskell
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
  -- s0 :: Map arg (Set res)
  s0 = M.singleton arg0 S.empty

  -- step :: Map arg (Set res) -> Map arg (Set res)
  step m = foldr (\k -> go k (open rec k)) m (M.keys m)
```

```haskell
open          :: Open (arg -> ND arg res res)
open rec      :: arg -> ND arg res res
open rec k    :: ND arg res res
```

Monadic Model: Handler (1/2)

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
  -- s0 :: Map arg (Set res)
  s0 = M.singleton arg0 S.empty

  -- step :: Map arg (Set res) -> Map arg (Set res)
  step m = foldr (\k -> go k (open rec k)) m (M.keys m)

  -- go :: arg -> ND arg res res
  --      -> Map arg (Set res) -> Map arg (Set res)
  <to be continued>
```

Monadic Model: Handler (2/2)

```
-- go :: arg -> ND arg res res
--    -> Map arg (Set res) -> Map arg (Set res)
```

Monadic Model: Handler (2/2)

```
-- go :: arg -> ND arg res res
--     -> Map arg (Set res) -> Map arg (Set res)

go a (Success x) m = M.insertWith S.union a (S.singleton x) m
```

Monadic Model: Handler (2/2)

```
-- go :: arg -> ND arg res res
--     -> Map arg (Set res) -> Map arg (Set res)

go a (Success x) m = M.insertWith S.union a (S.singleton x) m

go a Fail m = m
```

Monadic Model: Handler (2/2)

```
-- go :: arg -> ND arg res res
--    -> Map arg (Set res) -> Map arg (Set res)

go a (Success x) m = M.insertWith S.union a (S.singleton x) m

go a Fail m = m

go a (Or l r) m = go a r (go a l m)
```

Monadic Model: Handler (2/2)

```haskell
-- go :: arg -> ND arg res res
--    -> Map arg (Set res) -> Map arg (Set res)

go a (Success x) m = M.insertWith S.union a (S.singleton x) m

go a Fail m = m

go a (Or l r) m = go a r (go a l m)

go a (Rec b cont) m = case M.lookup b m of
  Nothing -> M.insert b S.empty m
  Just s  -> foldr (go a . cont) m (S.elems s)
```

Monadic Model: Example

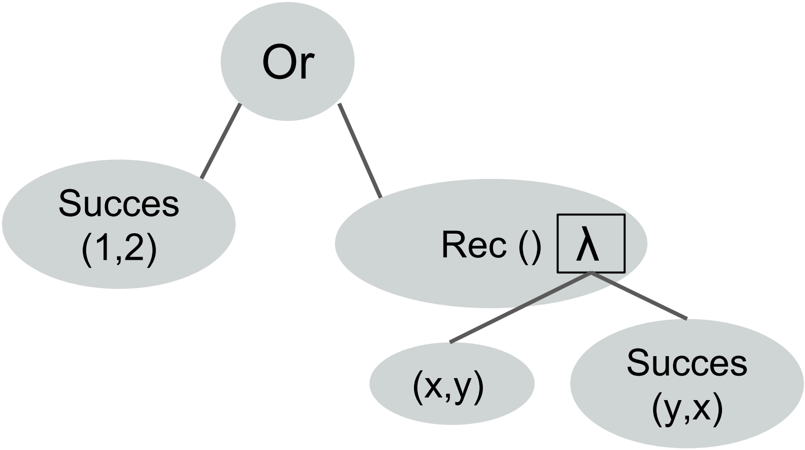
```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

```
ghci> runND f0 ()
```

Monadic Model: Example

```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

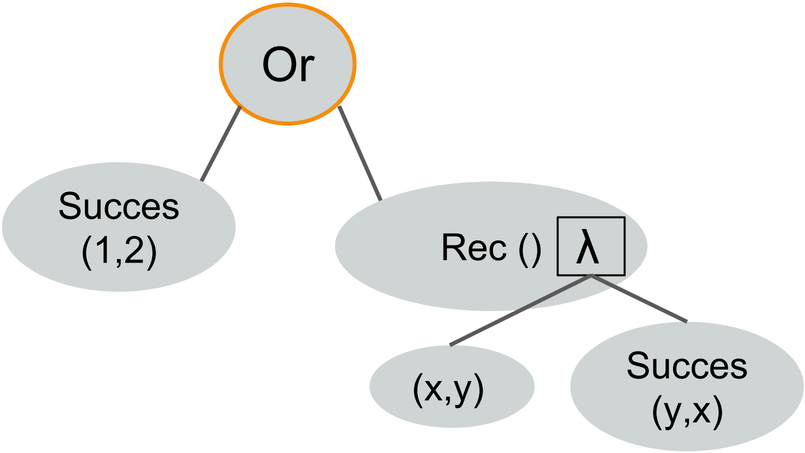
```
ghci> runND f0 ()
```



Monadic Model: Example

```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

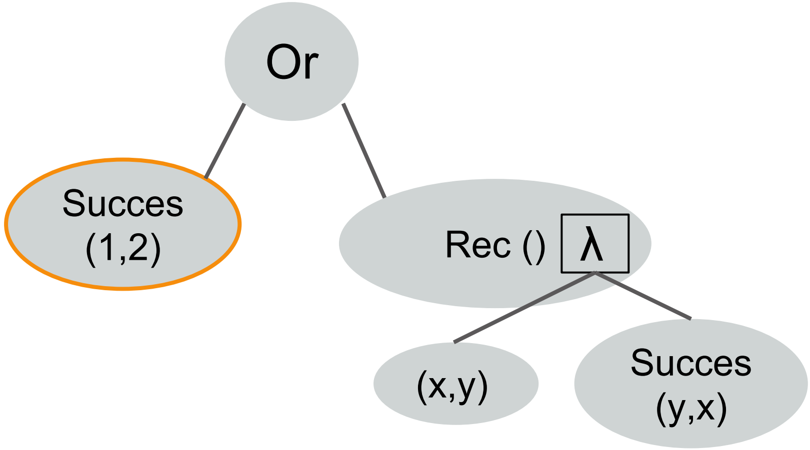
```
ghci> runND f0 ()
```



Monadic Model: Example

```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

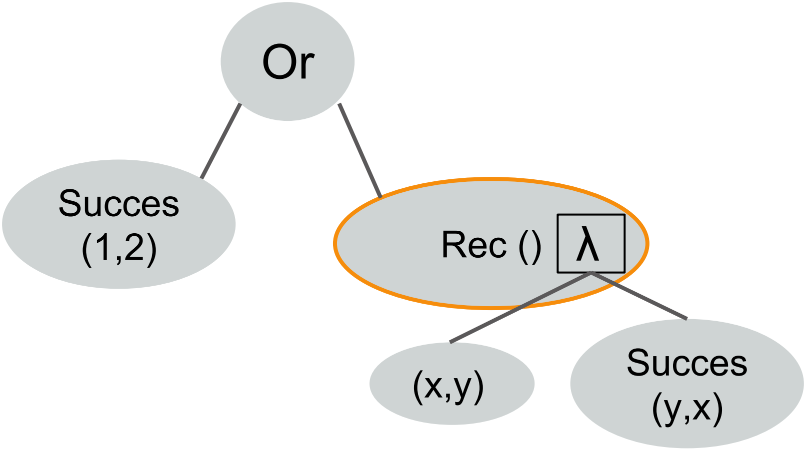
```
ghci> runND f0 ()
fromList [(1,2)]
```



Monadic Model: **Example**

```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

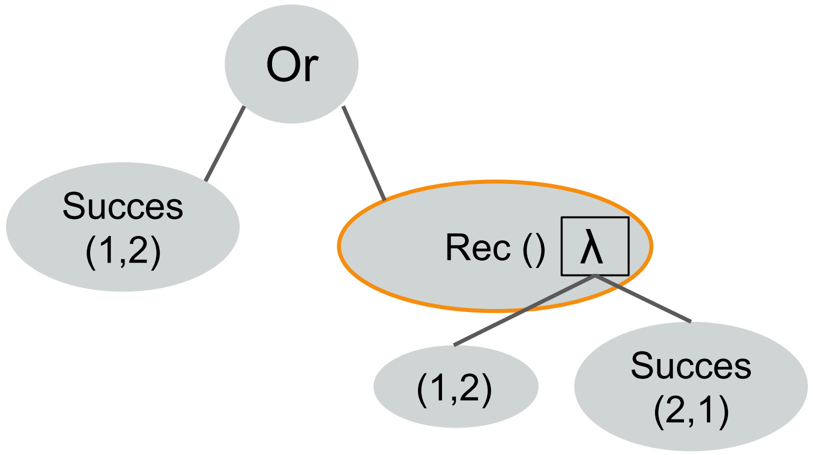
```
ghci> runND f0 ()
fromList [(1,2)
```



Monadic Model: Example

```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

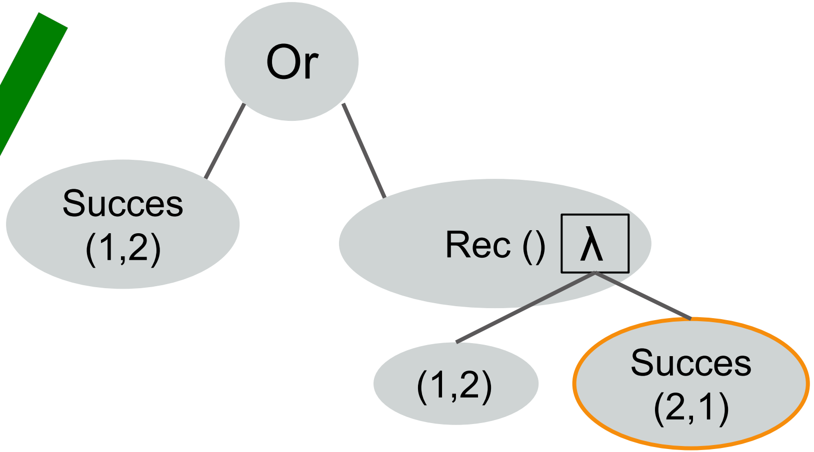
```
ghci> runND f0 ()
fromList [(1,2)]
```



Monadic Model: Example

```
f0 :: Open (() -> ND () (Int, Int) (Int, Int))
f0 open () = return (1,2) `Or` do (x,y) <- open ()
                                  return (y,x)
```

```
ghci> runND f0 ()
fromList [(1,2), (2,1)]
```



Monadic Model: Fixpoint Semantics

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

least fixed point operator

```
lfp :: Eq a => a -> (a -> a) -> a
lfp a0 f = let a1 = f a0
           in if a0 == a1 then
                 a1
              else
                 lfp a1 f
```

Monadic Model: Fixpoint Semantics

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

least fixed point operator

```
lfp :: Eq a => a -> (a -> a) -> a
lfp a0 f = let a1 = f a0
                                 then

                    lfp a1 f
```

well-defined?

Monadic Model: Fixpoint Semantics

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

least fixed point operator

```
lfp :: Eq a => a -> (a -> a) -> a
lfp a0 f = let a1 = f a0
                                 then

                 lfp a1 f
```

slow & stupid

Monadic Model: Fixpoint Semantics

```
runND :: (Ord arg, Ord res)
      => Open (arg -> ND arg res res) -> (arg -> Set res)
runND open arg0 = lfp s0 step ! arg0 where
```

least fixed point operator

```
lfp :: Eq a => a -> (a -> a) -> a
lfp a0 f = let a1 = f a0
                                     then
                lfp a1 f
```

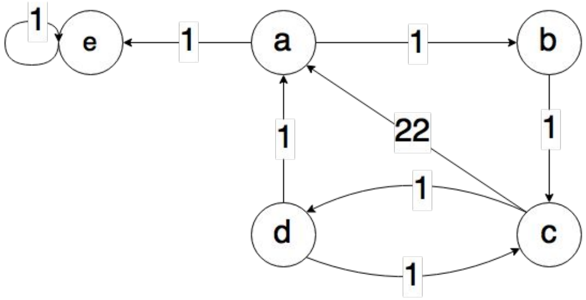
dependency tracking

Dynamic Programming

Reachability

```haskell
data Node = Node { label :: Char, adj :: [(Node, Int)] }
neighbor :: Node -> ND arg res Node
neighbor = foldr Or Fail . map Success . adj
```

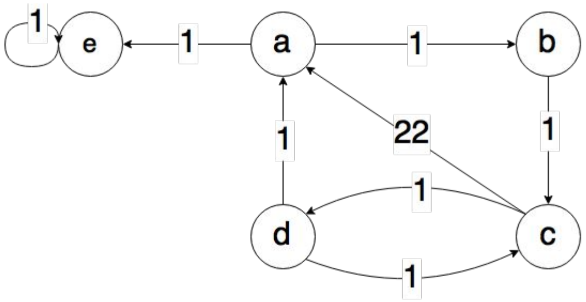
```
ghci>
```



Reachability

```haskell
data Node = Node { label :: Char, adj :: [(Node, Int)] }
neighbor :: Node -> ND arg res Node
neighbor = foldr Or Fail . map Success . adj

reach :: Open (Node -> ND Node Node Node)
reach open n0 = do
  n <- fmap fst (neighbor n0)
  return n0 `Or` return n `Or` open n
```

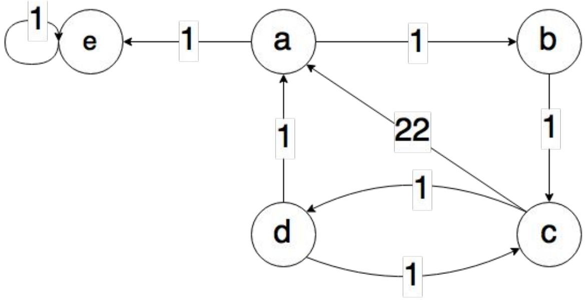
```
ghci> runND reach a
```



Reachability

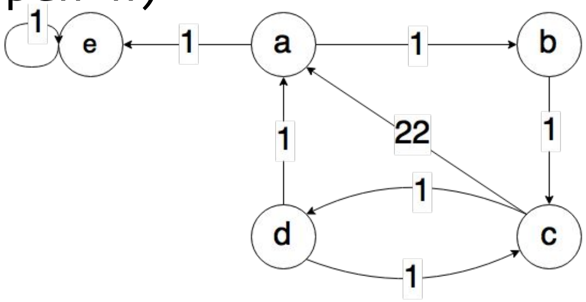
```
data Node = Node { label :: Char, adj :: [(Node, Int)] }
neighbor :: Node -> ND arg res Node
neighbor = foldr Or Fail . map Success . adj

reach :: Open (Node -> ND Node Node Node)
reach open n0 = do
  n <- fmap fst (neighbor n0)
  return n0 `Or` return n `Or` open n
```

```
ghci> runND reach a
fromList [Node a,
Node b, Node c,
Node d, Node e]
```



Shortest Path

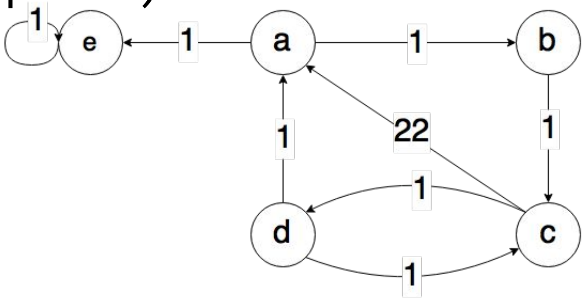
```
data Node = Node { label :: Char, adj :: [(Node, Int)] }
neighbor :: Node -> ND arg res Node
neighbor = foldr Or Fail . map Success . adj

sp :: Char -> Open (Node -> ND Node Int Int)
sp dst open src | dst == label src = return 0
                | otherwise = do (n, d) <- neighbor src
                                 fmap (d +) (open n)
```




```
ghci> runND (sp ‘a’) c
```

Shortest Path

```
data Node = Node { label :: Char, adj :: [(Node, Int)] }
neighbor :: Node -> ND arg res Node
neighbor = foldr Or Fail . map Success . adj

sp :: Char -> Open (Node -> ND Node Int Int)
sp dst open src | dst == label src = return 0
                | otherwise = do (n, d) <- neighbor src
                                 fmap (d +) (open n)
```




```
ghci> runND (sp ‘a’) c
fromList <diverges>
```

Dyn. Prog.: Lattice & Fold

```haskell
-- A bounded join-semilattice
class Ord a => Lattice a where
  join :: a -> a -> a
  bottom :: a

lub :: Lattice a => [a] -> a
lub = foldr join bottom
```

Dyn. Prog.: Lattice & Fold

```haskell
-- A bounded join-semilattice
class Ord a => Lattice a where
  join :: a -> a -> a
  bottom :: a

lub :: Lattice a => [a] -> a
lub = foldr join bottom
```

```haskell
bottom <= a, b <= join a b
```

Dyn. Prog.: Lattice & Fold

```haskell
-- A bounded join-semilattice
class Ord a => Lattice a where
  join :: a -> a -> a
  bottom :: a

lub :: Lattice a => [a] -> a
lub = foldr join bottom
```

```haskell
class Lattice f => Fold f where
  fold :: (f -> b -> b) -> b -> f -> b

toList :: Fold f => f -> [f]
toList = fold (:) []
```

```haskell
bottom <= a, b <= join a b
```

Dyn. Prog.: Lattice & Fold

```haskell
-- A bounded join-semilattice
class Ord a => Lattice a where
  join :: a -> a -> a
  bottom :: a

lub :: Lattice a => [a] -> a
lub = foldr join bottom
```

```haskell
bottom <= a, b <= join a b
```

```haskell
class Lattice f => Fold f where
  fold :: (f -> b -> b) -> b -> f -> b

toList :: Fold f => f -> [f]
toList = fold (:) []
```

```haskell
fold join bottom = id
```

Dyn. Prog.: Handler (1/2)

```
runL :: (Ord arg, Fold res)
     => Open (arg -> ND arg res res) -> (arg -> res)
runL open i0 = lfp step s0 ! i0 where
  -- s0 :: Map i o
  s0 = M.singleton i0 bottom

  -- step :: Map i o -> Map i o
  step m = foldr (\k -> go k (open rec k)) m (M.keys m)

  -- go :: arg -> ND arg res res
  --     -> Map arg res -> Map arg res
  <to be continued>
```

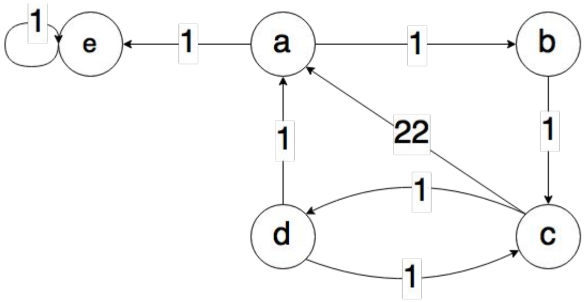
Dyn. Prog.: Handler (2/2)

```
-- go :: arg -> ND arg res res
--    -> Map arg res -> Map arg res

go a (Success x) m = M.insertWith join i x m

go a Fail m = m

go a (Or l r) m = go a r (go a l m)

go a (Rec b cont) m = case M.lookup b m of
  Nothing -> M.insert b bottom m
  Just s  -> fold (go b . cont) m s
```

Dynamic Programming

```haskell
data Dist = InfDist | Dist Int
instance Lattice Dist where
  join InfDist a          = a
  join a InfDist          = a
  join (Dist a) (Dist b) = Dist (min a b)
  bottom = InfDist
```

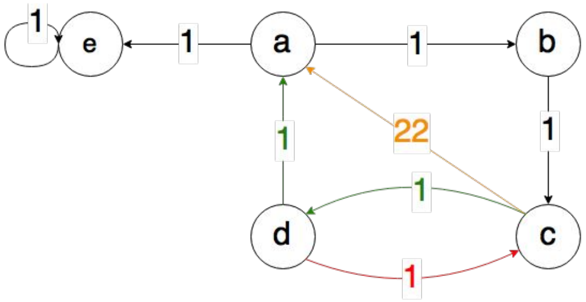
```
>
```



Dynamic Programming

```haskell
data Dist = InfDist | Dist Int
instance Fold Dist where
  fold f b InfDist = b
  fold f b d       = f d b
```

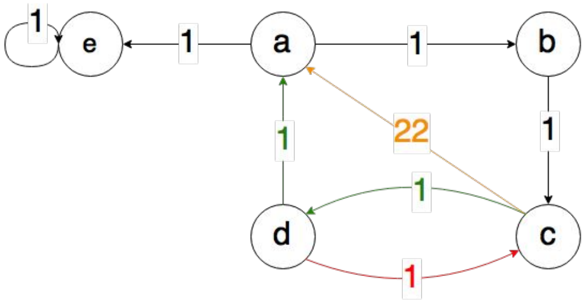
```
>
```



Dynamic Programming

```haskell
instance Num Dist

sp :: Char -> Open (Node -> ND Node Dist Dist)
sp dst open src | dst == label src = return 0
                | otherwise = do (n, d) <- neighbor src
                                 fmap (Dist d +) (open n)
```

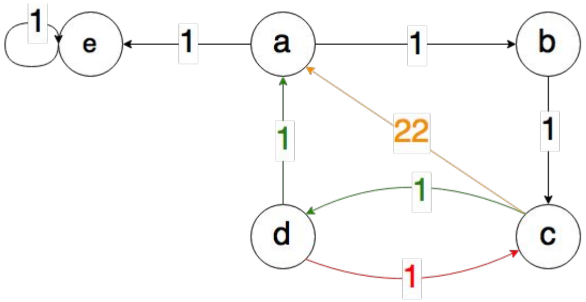
```
> runL (sp 'a') c
```



Dynamic Programming

```
instance Num Dist

sp :: Char -> Open (Node -> ND Node Dist Dist)
sp dst open src | dst == label src = return 0
                | otherwise = do (n, d) <- neighbor src
                                 fmap (Dist d +) (open n)
```

```
> runL (sp ‘a’) c
Dist 2
```



Summary

- recursive non-deterministic computations
- monadic model: effect handlers approach
- handler has least fixed point semantics
- extended to “dynamic programming”
- see https://bitbucket.org/AlexanderV/thesis

Related Work

- Inspiration: tabulation or tabling in Prolog
  XSB-Prolog, B-Prolog, YapProlog

- Dynamic Programming
  - lattice and order answer subsumption (XSB-Prolog)
  - tabling modes (B-Prolog, YapProlog, ALS-Prolog)
    Evaluation order (DFS/BFS) has an influence!

Related Work

- Parsing
    - Memoization of Top-down Parsing, Mark Johnson
    - Constructing functional programs for grammar analysis problems, Johan Jeuring and Doaitse Swierstra
- Language Constructs for Non-well-Founded Computation, Jean-Baptiste Jeannin, Dexter Kozen and Alexandra Silva

Questions

```haskell
let s = S.singleton 1 `S.union` S.singleton 2
in foldMap (\i -> S.map (i+) s) s
```

```haskell
(+) <$> (return 1 `Or` return 2)
    <*> (return 1 `Or` return 2)

do x <- return 1 `Or` return 2
   y <- return 1 `Or` return 2
   return (x + y)
```

```haskell
[x + y | x <- [1,2], y <- [1,2]]
```