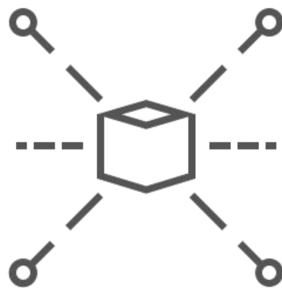
INPUT | OUTPUT

# PLUTUS

# Recursive Datatypes in System F-omega

# Why are we doing this?

- Plutus Core = System F-omega
- System F-omega has no datatypes
- We want datatypes
- So we need to encode them



What can you do with a datatype value?

# Pattern match on it!

# Abstracting pattern matching

Start with a simple pattern match

```
case m of
   Just x -> f x
   Nothing -> g
```

Type parameter

Abstract the scrutinee

```
Λ(a :: *) -> λ(m : Maybe a) ->
  case m of
    Just x -> f x
    Nothing -> g
```

Abstract the case branches

```
Λ(a :: *) -> λ(m : Maybe a) ->
  Λ(r :: *) -> λ(f : a -> r) (g : r) ->
    case m of
      Just x -> f x
      Nothing -> g
```

Result type

# An old friend

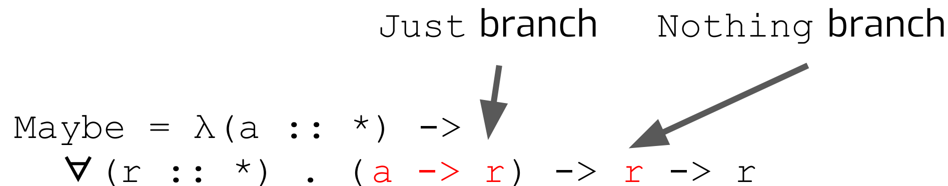
```
maybe :: b -> (a -> b) -> Maybe a -> b
```
# Source

The `maybe` function takes a default value, a function, and a `Maybe` value. If the `Maybe` value is `Nothing`, the function returns the default value. Otherwise, it applies the function to the value inside the `Just` and returns the result.

⊞ **Examples**

# Scott encoding

Identify the datatype with its eliminator

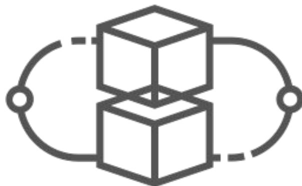



The constructors pick the appropriate branch

```
Just = Λ(a :: *) -> λ(arg : a) ->
  Λ(r :: *) ->
    λ(branch_Just :: a -> r)
     (branch_Nothing :: r) ->
       branch_Just arg
```

Pattern matching is application

```
(Just {Int} 1) {Int} (+1) 0 == 2
```



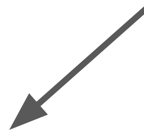
But what about recursive types?

# We need lists!

# Naive encoding is not enough

Can't we just do the same encoding?

Rats!

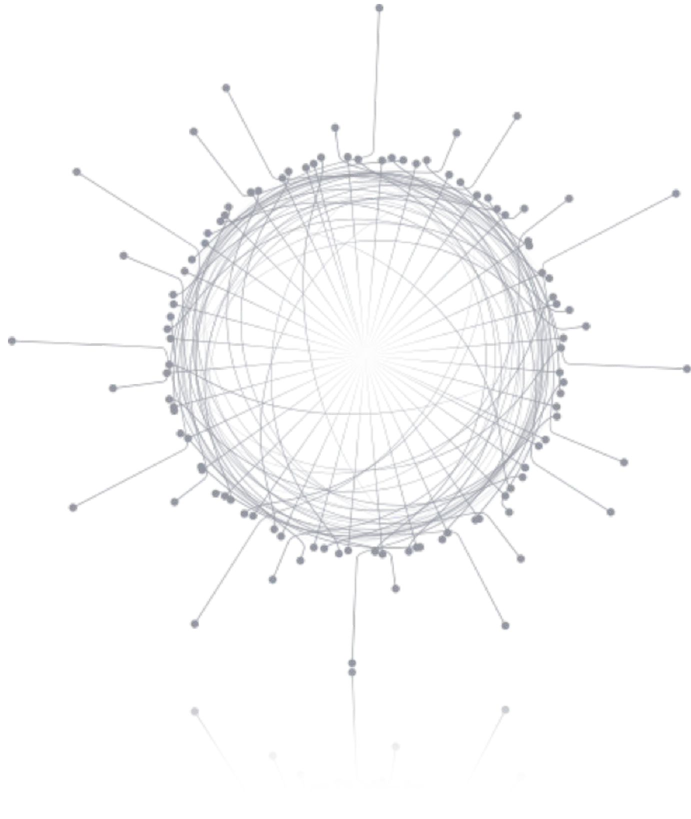


```
List = λ(a :: *) ->
  ∀(r :: *) . r -> (a -> List a -> r) -> r
```

Can't use the type in its own definition, need a recursion/fixpoint operator at the type level.

# Fix it up!

- Recursive types are fixpoint types
  - `Nat = fix (λNat -> ∀r . r -> (Nat -> r) -> r)`
- Isomorphism between fixpoint type and its one-step unfolding witnessed by `wrap` and `unwrap` terms
  - `data Fix f = Wrap (f (Fix f))`
  - `wrap :: f (Fix f) -> Fix f`
  - `unwrap :: Fix f -> f (Fix f)`
  - Alternative: equirecursive types (much harder theory)
- Actually we use:
  - `ifix :: ((k -> *) -> (k -> *)) -> (k -> *)`

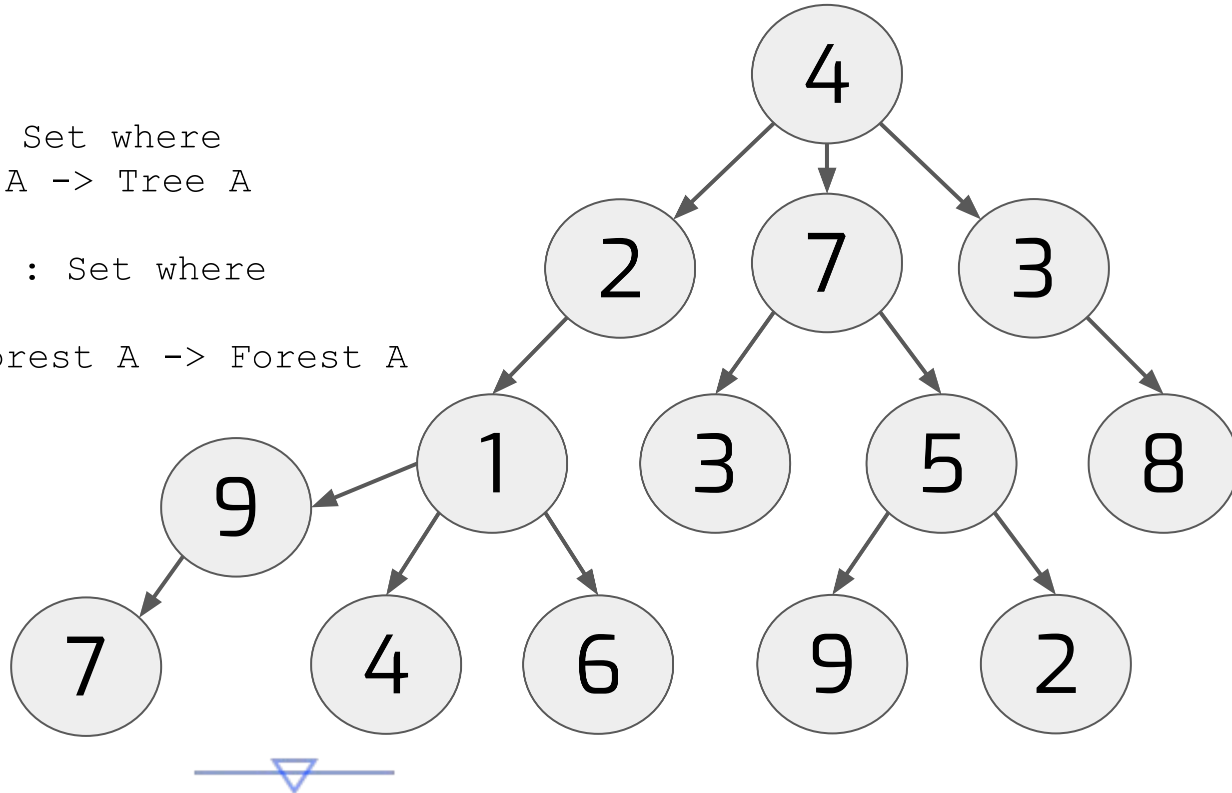


# Agda time!

# Direct Tree & Forest in Agda

```
mutual
  data Tree (A : Set) : Set where
    node : A -> Forest A -> Tree A

  data Forest (A : Set) : Set where
    nil  : Forest A
    cons : Tree A -> Forest A -> Forest A
```



# The encoding trick

"A mutual definition can always be represented as a single inductive family of datatypes indexed by a finite type whose elements label the branches — we might define a family Parity : 𝟚 -> Type with Parity true containing the even numbers and Parity false odd numbers"

— Conor McBride

# Merged Tree & Forest in Agda

```
data TreeForestTag : Set where
  TreeTag ForestTag : TreeForestTag

mutual
  Tree : Set -> Set
  Tree A = TreeForest A TreeTag

  Forest : Set -> Set
  Forest A = TreeForest A ForestTag

  data TreeForest (A : Set) : TreeForestTag -> Set where
    node : A -> Forest A -> Tree A
    nil  : Forest A
    cons : Tree A -> Forest A -> Forest A
```

# Computational Tree & Forest in Agda

```
mutual
  Tree : Set -> Set
  Tree A = TreeForest A TreeTag

  Forest : Set -> Set
  Forest A = TreeForest A ForestTag

  TreeForestF : Set -> TreeForestTag -> Set
  TreeForestF A TreeTag    = A × Forest A
  TreeForestF A ForestTag = ⊤ ⊎ Tree A × Forest A

  data TreeForest (A : Set)  (tag : TreeForestTag)  : Set where
    treeForest : TreeForestF A tag -> TreeForest A tag
```

# IFix

```
data Fix (F : Set -> Set) : Set where
  wrap : F (Fix F) -> Fix F


data IFix {A : Set} (F : (A -> Set) -> A -> Set) (x : A) : Set where
  iwrap : F (IFix F) x -> IFix F x
```

# IFix-ey Tree & Forest in Agda

```
TreeForest : Set -> TreeForestTag -> Set
TreeForest A =
  IFix λ Rec tag ->
    let Tree   = Rec TreeTag
        Forest = Rec ForestTag
    in case tag of λ
      { TreeTag   -> A × Forest
      ; ForestTag -> ⊤ ⊎ Tree × Forest
      }

Tree : Set -> Set
Tree A = TreeForest A TreeTag

Forest : Set -> Set
Forest A = TreeForest A ForestTag
```

# IFix-ey Tree & Forest in Agda

```
TreeForest : Set -> TreeForestTag -> Set
TreeForest A =
  IFix λ Rec tag ->
    let Tree   = Rec TreeTag
        Forest = Rec ForestTag
    in case tag of λ
      { TreeTag   -> A × Forest
      ; ForestTag -> ⊤ ⊎ Tree × Forest
      }

Tree : Set -> Set
Tree A = TreeForest A TreeTag

Forest : Set -> Set
Forest A = TreeForest A ForestTag
```

# IFix-ey Tree & Forest in Agda

```
TreeForest : Set -> TreeForestTag -> Set
TreeForest A =
  IFix λ Rec tag ->
    let Tree   = Rec TreeTag
        Forest = Rec ForestTag
    in tag
      (A × Forest)
      (⊤ ⊎ Tree × Forest)

Tree : Set -> Set
Tree A = TreeForest A TreeTag

Forest : Set -> Set
Forest A = TreeForest A ForestTag
```

```
TreeForestTag : Set₁
TreeForestTag = Set -> Set -> Set

TreeTag : TreeForestTag
TreeTag A B = A

ForestTag : TreeForestTag
ForestTag A B = B
```

# Tree & Forest in System F-omega

```
treeTag   = \(t f :: *) -> t
forestTag = \(t f :: *) -> f

treeForestF =
  \(a :: *) (rec :: (* -> * -> *) -> *) (tag :: * -> * -> *) ->
    all (r :: *). tag
      ((a -> rec forestTag -> r) -> r)
      (r -> (rec treeTag -> rec forestTag -> r) -> r))

treeForest = \(a :: *) (tag :: * -> * -> *) -> ifix (treeForestF a) tag

tree   = \(a :: *) -> treeForest a treeTag
forest = \(a :: *) -> treeForest a forestTag
```

# Scott-encoded constructors of Tree & Forest

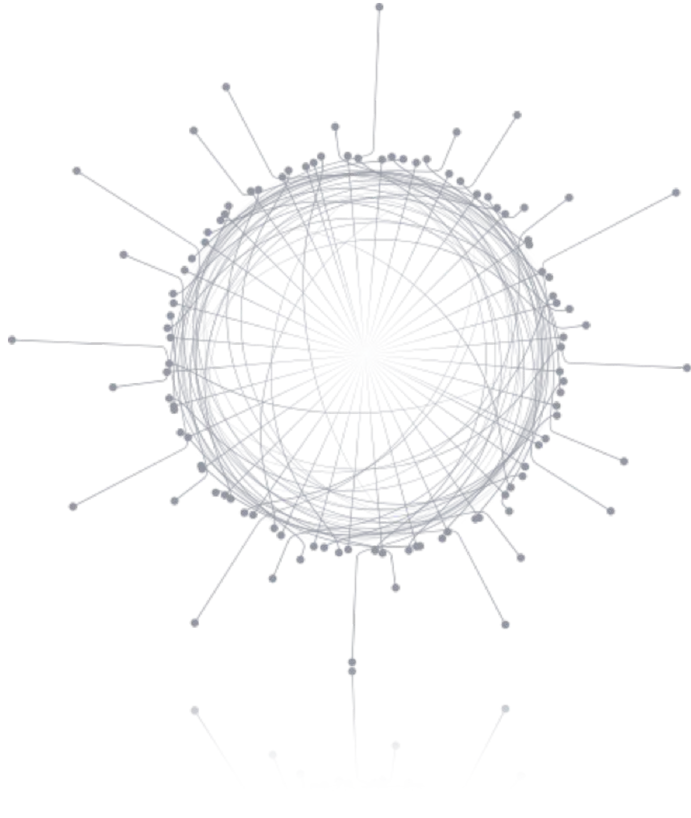
```
node =
  /\(a :: *) -> \(x : a) (fr : forest a) ->
    iwrap (treeForestF a) treeTag
      (/\(r :: *) -> \(f : a -> forest a -> r) -> f x fr)

nil =
  /\(a :: *) ->
    iwrap (treeForestF a) forestTag
      (/\(r :: *) -> \(z : r) (f : tree a -> forest a -> r) -> z)

cons =
  /\(a :: *) -> \(tr : tree a) (fr : forest a) ->
    iwrap (treeForestF a) forestTag
      (/\(r :: *) -> \(z : r) (f : tree a -> forest a -> r) -> f tr fr)
```

# The node constructor in full

```
/\(a :: *) -> \(x : a) -> \(fr : ifix (\(b :: ((* -> (* -> * -> *) -> *) -> *) -> *) -> \(p
:: (* -> (* -> * -> *) -> *) -> *) -> p (\(a :: *) -> \(tag :: * -> * -> *) -> all (r ::
*). tag ((a -> b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) -> f)) ->
r) -> r) (r -> (b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) -> t)) ->
b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) -> f)) -> r) -> r))) (\(r
:: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) -> f))) -> iwrap (\(b :: ((* ->
(* -> * -> *) -> *) -> *) -> *) -> \(p :: (* -> (* -> * -> *) -> *) -> *) -> p (\(a :: *)
-> \(tag :: * -> * -> *) -> all (r :: *). tag ((a -> b (\(r :: * -> (* -> * -> *) -> *) ->
r a (\(t :: *) -> \(f :: *) -> f)) -> r) -> r) (r -> (b (\(r :: * -> (* -> * -> *) -> *) ->
r a (\(t :: *) -> \(f :: *) -> t)) -> b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *)
-> \(f :: *) -> f)) -> r) -> r))) (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f
:: *) -> t)) (/\(r :: *) -> \(f : a -> ifix (\(b :: ((* -> (* -> * -> *) -> *) -> *) -> *)
-> \(p :: (* -> (* -> * -> *) -> *) -> *) -> p (\(a :: *) -> \(tag :: * -> * -> *) -> all
(r :: *). tag ((a -> b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) ->
f)) -> r) -> r) (r -> (b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) ->
t)) -> b (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) -> f)) -> r) ->
r))) (\(r :: * -> (* -> * -> *) -> *) -> r a (\(t :: *) -> \(f :: *) -> f)) -> r) -> f x
fr)
```



# Extra material

# Far too many fixpoints

- What is the kind of fix?
  - `(* -> *) -> *`
    - Nat, (ugly) List ✓
    - (nice) List, mutually recursive types, irregular types ✗
  - `(k -> k) -> k`
    - Nat, List, mutually recursive types, irregular types ✓
    - Wrapping and unwrapping is painfully n-ary
  - `((k -> *) -> (k -> *)) -> (k -> *)`
    - Nat, List, mutually recursive types, irregular types ✓
    - Wrapping and unwrapping need handle only one arg
    - “ifix”