GADTs

GADTs in Haskell

ADT vs GADT

Algebraic Datatype

```
Data List a
 = Nil
| Cons a (List a)

Data Tree a b =
 Tip a
| Node (Tree a b) b
| Fork (Tree a b) (Tree a b)
```

Note that types than can be expressed as an ADT always have the identical range types on all their constructors

Generalized Algebraic Datatype

```
Data List a where
  Nil:: List a
Cons:: a -> List a -> List a

Data Tree a b  where
 Tip a : Tree a b
 Node:    Tree a b ->
          b ->
          Tree a b
 Fork::   Tree a b ->
          Tree a b ->
          Tree a b
```

GADTs relax the range restriction

```
data Even n where
  Base:: Even Z
  NE:: Odd n -> Even (S n)

data Rep:: * -> * where
 Int:: Rep Int
 Char:: Rep Char
 Float:: Rep Float
 Bool :: Rep Bool
 Pair:: Rep a -> Rep b -> Rep (a,b)
 List :: Rep a -> Rep [a]
```

The range is always the type being defined, (Even & Rep) but that type's arguments can vary. We call an argument that varies and index.

Examples

- Length indexed lists
- Balanced Trees
  - Redblack trees
  - 2-threes trees
  - AVL trees
- Representation types
- Well-typed terms
  - Terms as typing judgements
  - Tagless interpreters
  - Subject reduction
  - Type inference
- Witnesses
  - Odd and Even
  - Well formed join and cross product in relational algebra
- Well structured paths in trees
- Units of measure (inches, centimeters, etc)
- Provable Equality
- Proof carrying code

Length indexed lists

Note we introduce uninhabited types that have the structure of the Natural numbers to serve as indexes to LList

```
data Z
data S n

data LList :: * -> * -> * where
   LNil:: LList a Z
   LCons:: a -> LList a n -> LList a (S n)
```

Note the range type is always LList
But they differ in the indexes.

Promotion

- As of GHC version 7 GHC allows indexes to be ordinary datatypes.
- One says the datatype is promoted to the type level.
- This is very usefull, as it enforces a typing system on the indexes.
  - For example does (LList Int String) make any sense. The index is supposed to be drawn from Z and S

Length indexed lists again

An ordinary algebraic datatype

```
data Nat = Zero | Succ Nat
```

The type Nat is promoted to a Kind

```
data Vec:: * -> Nat ->  * where
  Nil :: Vec a Zero
  Cons:: a -> Vec a n -> Vec a (Succ n)
```

The index is always drawn from well typed values of type Nat, so values of type Nat are promoted to Types

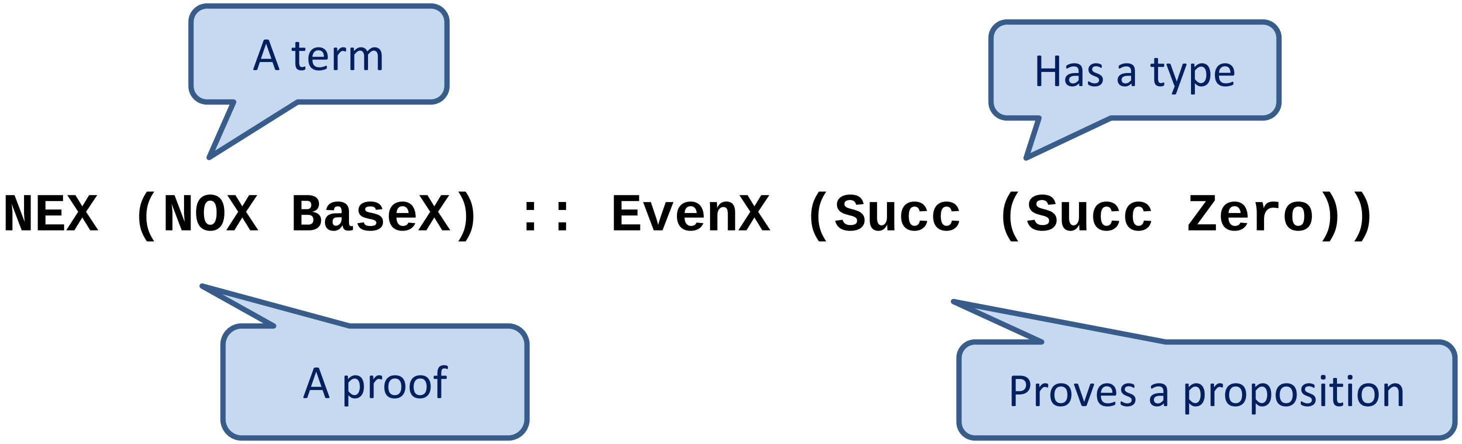
GADTs as proof objects

```
data EvenX:: Nat -> * where
  BaseX:: EvenX Zero
  NEX:: OddX n -> EvenX (Succ n)

data OddX:: Nat -> * where
  NOX:: EvenX n -> OddX (Succ n)
```

- What type does NEX (NOX BaseX) have?

The Curry-Howard isomorphism

- The Curry-Howard isomorphism says that two things have exactly the same structure.
  - A term has a type
  - A proof proves a proposition



A term

Has a type

`NEX (NOX BaseX) :: EvenX (Succ (Succ Zero))`

A proof

Proves a proposition

Note that there is no term with type: **`EvenX (Succ Zero)`**

Proofs and witnesses

- GADTs make the Curry-Howard isomorphism useful in Haskell.

- Sometime we say a term “witnesses” a property. I.e the term **`NEX (NOX BaseX)`** witnesses that 2 is even.

- We use GADTs has indexes to show that some things are not possible.

Paths and Trees

```
data Shape = Tp | Nd | Fk Shape Shape

data Path :: Shape -> * where
  Here:: Path Nd
  Left:: Path x  -> Path (Fk x y)
  Right:: Path y -> Path (Fk x y)
```

Note there are no paths with index Tp

```
data Tree:: Shape -> * -> * where
  Tip :: Tree Tp a
  Node:: a -> Tree Nd a
  Fork :: Tree x a -> Tree y a -> Tree (Fk x y) a
```

Well formed paths

```
find:: Eq a => a -> Tree sh a -> [Path sh]
find n Tip = []
find n (Node m) =
  if n==m then [Here] else []
find n (Fork x y) =
 (map Left (find n x)) ++
 (map Right (find n y))
```

Using a path. No possibility of failure

```
extract:: Eq a => Tree sh a -> Path sh -> a
extract (Node n) (Here) = n
extract (Fork l r) (Left p) = extract l p
extract (Fork l r) (Right p) = extract r p

-- No other cases are possible,
-- Since there are no Paths with index Tp
```

Balanced Trees

- Balanced trees are used as binary search mechanisms.
- They support log(n) time searches for trees that have n elements
- They rely on the trees being balanced.
  - Usually saying that all paths from root to leaf have roughly the same length
- Indexes make perfect tools to represent these invariants.

Red Black Trees

- A red-black tree is a binary search tree with the following additional invariants:
  - Each node is colored either red or black
  - The root is black
  - The leaves are black
  - Each Red node has Black children
  - for all internal nodes, each path from that node to a descendant leaf contains the same number of black nodes.

- We can encode these invariants by thinking of each internal node as having two attributes: a color and a black-height.

Red Black Tree as a GADT

```
data Color = Red | Black

data SubTree :: Color -> Nat -> * where
   LeafRB :: SubTree Black Zero
   RNode :: SubTree Black n -> Int ->
            SubTree Black n -> SubTree Red n
   BNode :: SubTree cL m -> Int ->
            SubTree cR m ->
            SubTree Black (Succ m)

data RBTree where
  Root:: (forall n. (SubTree Black n)) -> RBTree
```

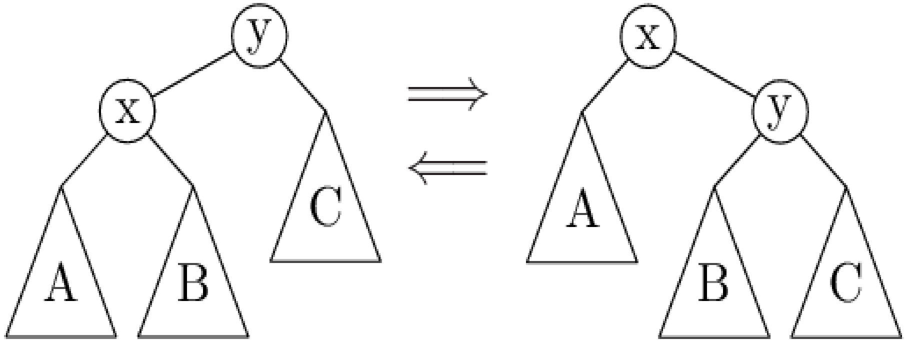
AVL Trees

- In an AVL tree, the heights of the two child sub trees of any node differ by at most one;

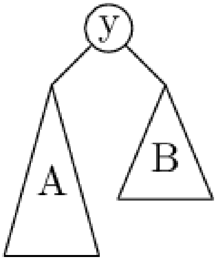
```
data Balance:: Nat -> Nat -> Nat -> * where
  Same :: Balance n n n
  Less :: Balance n (Succ n) (Succ n)
  More :: Balance (Succ n) n (Succ n)

data Avl:: Nat -> * where
  TipA:: Avl Zero
  NodeA:: Balance i j k -> Avl i -> Int ->
          Avl j -> Avl (Succ k)

data AVL = forall h. AVL (Avl h)
```

A witness type that witnesses only the legal height differences.

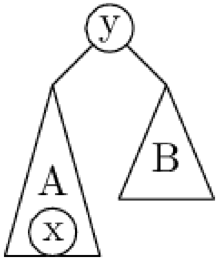
Balancing Constructors. The algorithms for insertion and deletion each follow the same basic pattern: First do the insertion (or deletion) as you would for any other binary search tree. Then re-balance any subtree that became unbalanced in the process. The tool used for re-balancing is tree rotation, which is best described visually.

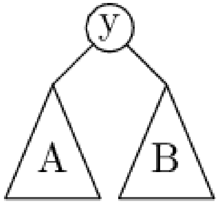


INPUT OUTPUT

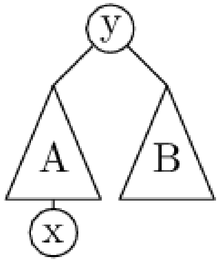


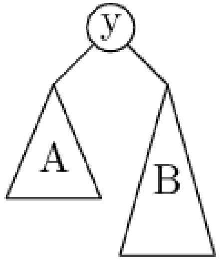
Post-insertion height is the same.
Keep the same `Balance`



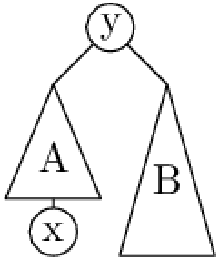


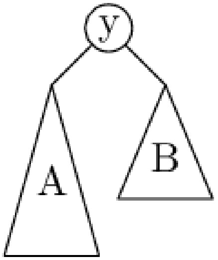
Height increases.
Change `Balance` from `Same` to `More`



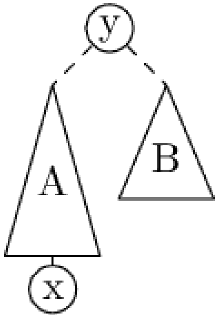


Height increases.
Change `Balance` from `Less` to `Same`





Height increases.
Rebalance with `rotr`

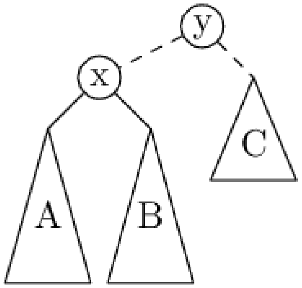


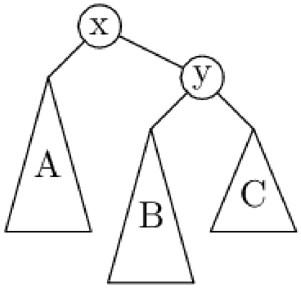
Insertion

```
insert :: Int -> AVL -> AVL
insert x (AVL t) = case ins x t of L t -> AVL t; R t -> AVL t

ins :: Int -> Avl n -> (Avl n + Avl (Succ n))
ins x TipA = R(NodeA Same TipA x TipA)
ins x (NodeA bal lc y rc)
  | x == y = L(NodeA bal lc y rc)
  | x < y  = case ins x lc of
               L lc -> L(NodeA bal lc y rc)
               R lc ->
                 case bal of
                   Same -> R(NodeA More lc y rc)
                   Less -> L(NodeA Same lc y rc)
                   More -> rotr lc y rc -- rebalance
  | x > y  = case ins x rc of
               L rc -> L(NodeA bal lc y rc)
               R rc -> case bal of
                         Same -> R(NodeA Less lc y rc)
                         More -> L(NodeA Same lc y rc)
                         Less -> rotl lc y rc -- rebalance
```

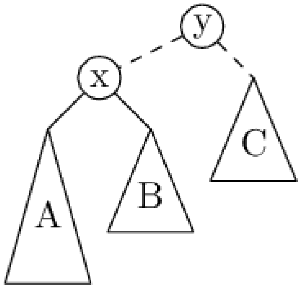
Note the rotations in red

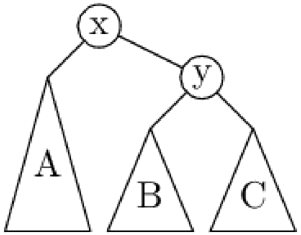
```
rotr (Node Same a x b) y c     = R(Node Less a x (Node More b y c))
```



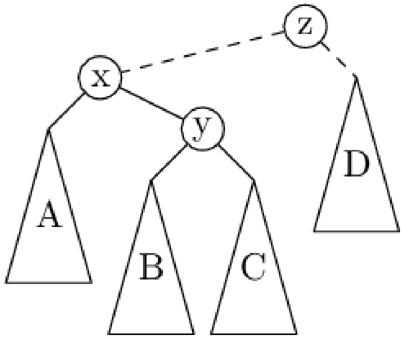


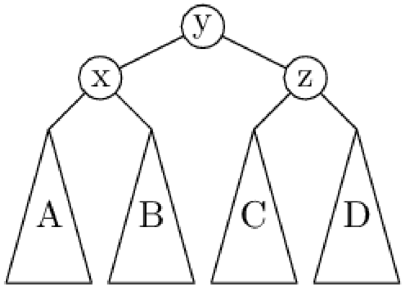
```
rotr (Node More a x b) y c     = L(Node Same a x (Node Same b y c))
```



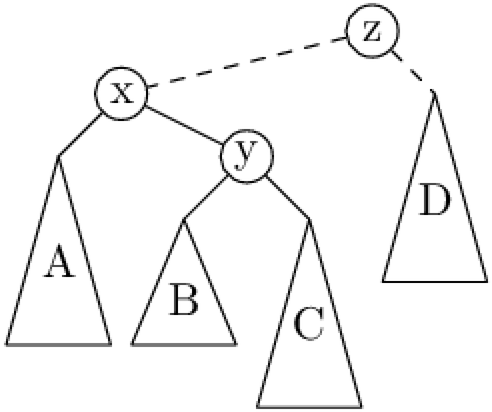


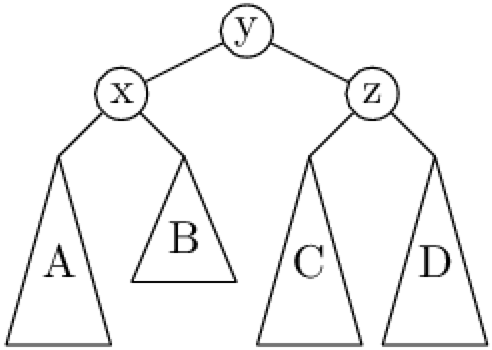
```
rotr (Node Less a x                 L(Node Same (Node Same a x b) y
            (Node Same b y c)) z d =              (Node Same c z d) )
```



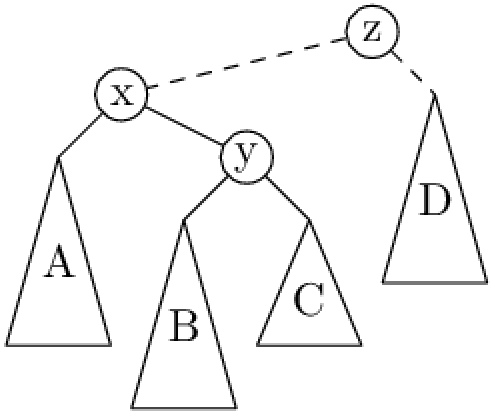


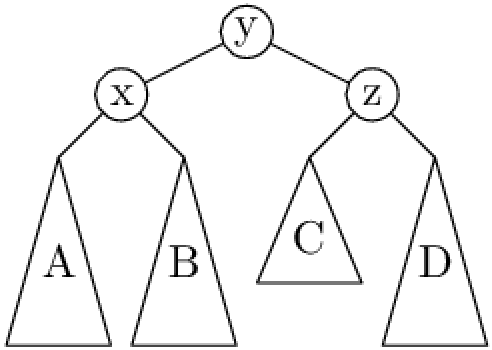
```
rotr (Node Less a x                     L(Node Same (Node More a x b) y
           (Node Less b y c)) z d   =               (Node Same c z d) )
```





```
rotr (Node Less a x                     L(Node Same (Node Same a x b) y
           (Node More b y c)) z d   =               (Node Less c z d) )
```





Example code

The rest is in the accompanying Haskell file

```
data (+) a b = L a | R b

rotr :: Avl(Succ(Succ n)) -> Int -> Avl n ->
           (Avl(Succ(Succ n))+Avl(Succ(Succ(Succ n))))
-- rotr Tip u a = unreachable
rotr (NodeA Same b v c) u a = R(NodeA Less b v (NodeA More c u a))
rotr (NodeA More b v c) u a = L(NodeA Same b v (NodeA Same c u a))
-- rotr (NodeA Less b v TipA) u a = unreachable
rotr (NodeA Less b v (NodeA Same x m y)) u a =
      L(NodeA Same (NodeA Same b v x) m (NodeA Same y u a))
rotr (NodeA Less b v (NodeA Less x m y)) u a =
      L(NodeA Same (NodeA More b v x) m (NodeA Same y u a))
rotr (NodeA Less b v (NodeA More x m y)) u a =
     L(NodeA Same (NodeA Same b v x) m (NodeA Less y u a))
```

At the top level

- Insertion may make the height of the tree grow.
- Hide the height of the tree as an existentially quantified index.

```
data AVL = forall h. AVL (Avl h)
```

2-3-Tree

- a tree where every node with children (internal node) has either two children (2-node) and one data element or three children (3-nodes) and two data elements. Nodes on the outside of the tree (leaf nodes) have no children and zero, one or two data elements

```
data Tree23:: Nat -> * -> * where
  Three :: (Tree23 n a) -> a ->
           (Tree23 n a) -> a ->
           (Tree23 (Succ n) a)
  Two:: (Tree23 n a) -> a ->
        (Tree23 n a) -> (Tree23 (Succ n) a)
  Leaf1 :: a -> (Tree23 Zero a)
  Leaf2 :: a -> a -> (Tree23 Zero a)
  Leaf0 :: (Tree23 Zero a)
```

Witnessing equality

- Sometimes we need to prove that two types are equal.
- We need a type that represents this proposition.
- We call this a witness, since legal terms with this type only witness that the two types are the same.
- This is sometimes called provable equality (since it is possible to write a function that returns an element of this type).

```
data Equal:: k -> k -> * where
  Refl :: Equal x x
```

Representation types

```
data Rep:: * -> * where
 Int:: Rep Int
 Char:: Rep Char
 Float:: Rep Float
 Bool :: Rep Bool
 Pair:: Rep a -> Rep b -> Rep (a,b)
 List :: Rep a -> Rep [a]
```

- Some times this is called a Universe, since it witnesses only those types representable.

Generic Programming

```
eq:: Rep a -> a -> a -> Bool
eq Int  x y  = x==y
eq Char x y  = x==y
eq Float x y = x==y
eq Bool  x y = x==y
eq (Pair t1 t2)(a,b) (c,d) =
  (eq t1 a c) &&  (eq t2 b d)
eq (List t) xs ys
   | not(length xs == length ys) = False
   | otherwise = and (zipWith (eq t) xs ys)
```

Using provable equality

- We need a program to inspect two Rep types (at runtime) to possible produce a proof that the types that they represent are the same.

```
test:: Rep a -> Rep b -> Maybe(Equal a b)
```

- This is sort of like an equality test, but reflects in its type that the two types are really equal.

Code for test

```
test:: Rep a -> Rep b -> Maybe(Equal a b)
test Int Int = Just Refl
test Char Char = Just Refl
test Float Float = Just Refl
test Bool Bool = Just Refl
test (Pair x y) (Pair m n) =
  do { Refl <- test x m
     ; Refl <- test y n
     ; Just Refl }
test (List x) (List y) =
  do { Refl <- test x y
     ; Just Refl}
test _ _ = Nothing
```

When we pattern match against Refl, the compiler know that the two types are statically equal in the scope of the match.

Well typed terms

```
data Exp:: * -> * where
  IntE :: Int -> Exp Int
  CharE:: Char -> Exp Char
  PairE :: Exp a -> Exp b -> Exp (a,b)
  VarE:: String -> Rep t -> Exp t
  LamE:: String -> Rep t ->
         Exp s -> Exp (t -> s)
  ApplyE:: Exp (a -> b) ->
           Exp a -> Exp b
  FstE:: Exp(a,b) -> Exp a
  SndE:: Exp(a,b) -> Exp b
```

Typing judgements

- Well typed terms have the exact same structure as a typing judgment.
- Consider the constructor ApplyE

```
ApplyE:: Exp (a -> b) -> Exp a -> Exp b
```

Compare it to the typing judgement

$$\frac{f : a \to b \qquad x : a}{f \quad x : b}$$

Tagless interpreter

- An interpreter gives a value to a term.
- Usually we need to invent a value datatype like

```
data Value
  = IntV Int
  | FunV (Value -> Value)
  | PairV Value Value
```

- We also need to store Values in an environment
  - `data Env = E [(String,Value)]`

Valueless environments

```
data Env where
  Empty :: Env
  Extend :: String -> Rep t -> t ->
            Env -> Env
```

Note that the type variable “t” is existentially quantified.

Given a pattern (`Extend var rep t more`) There is not much we can do with t, since we do not know its type.

Tagless interpreter

```
eval:: Exp t -> Env -> t
eval (IntE n) env = n
eval (CharE c) env = c
eval (PairE x y) env = (eval x env, eval y env)
eval (VarE s t) Empty =
   error ("Variable not found: "++s)
eval (LamE nm t body) env =
   (\ v -> eval body (Extend nm t v env))
eval (ApplyE f x) env = (eval f env) (eval x env)
eval (FstE x) env = fst(eval x env)
eval (SndE x) env = snd(eval x env)
eval (v@(VarE s t1)) (Extend nm t2 value more)
  | s==nm = case test t1 t2 of
              Just Refl -> value
              Nothing -> error "types don't match"
  | otherwise = eval v more
```

Units of measure

```
data TempUnit = Fahrenheit
              | Celsius
              | Kelvin

data Degree:: TempUnit -> * where
 F:: Float -> Degree Fahrenheit
 C:: Float -> Degree Celsius
 K:: Float -> Degree Kelvin

add:: Degree u -> Degree u -> Degree u
add (F x) (F y) = F(x+y)
add (C x) (C y) = C(x+y)
add (K x) (K y) = K(x+y)
```

N-way zip

```
zipN 1 (+1) [1,2,3]  → [2,3,4]

zipN 2 (+) [1,2,3] [4,5,6]
    → [5,7,9]

zipN 3 (\ x y z -> (x+z,y))[2,3]
        [5,1] [6,8]
    → [(8,5),(11,1)]
```

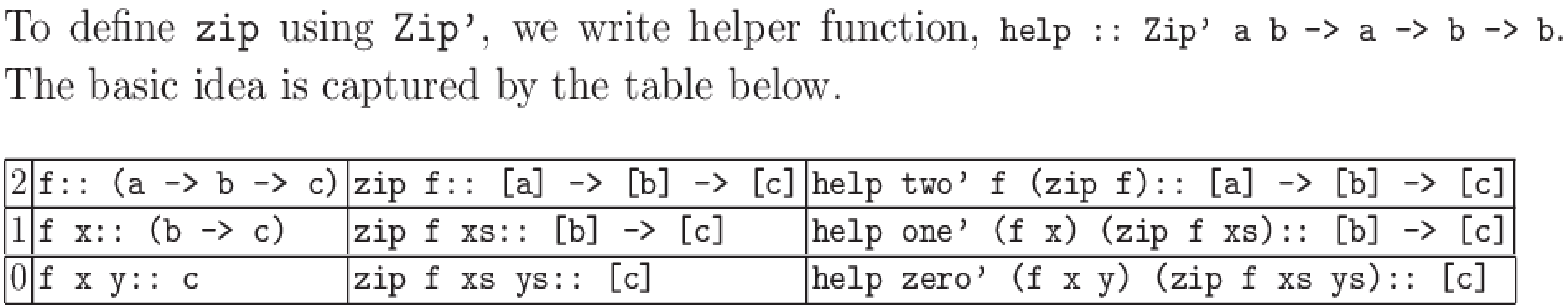
The Natural Numbers with strange types

```
data Zip :: * -> * -> * where
  Z:: Zip a [a]
  S:: Zip b c ->  Zip (a -> b) ([a] -> c)
```

```
Z :: Zip a [a]

(S Z) :: Zip (a -> b) ([a] -> [b])

(S (S Z)) ::
   Zip (a -> a1 -> b) ([a] -> [a1] -> [b])

(S(S (S Z))) ::
     Zip (a -> a1 -> a2 -> b)
         ([a] -> [a1] -> [a2] -> [b])
```

Why these types.

```
f :: (a -> b -> c -> d)
(f x) :: (b -> c -> d)
(f x y) :: (c -> d)
(f x y z) :: d

(zip f) :: ([a] -> [b] -> [c] -> [d])
(zip f xs) :: ([b] -> [c] -> [d])
(zip f xs ys) :: ([c] -> [d])
(zip f xs ys za) :: [d]
```



To define `zip` using `Zip’`, we write helper function, `help :: Zip’ a b -> a -> b -> b`. The basic idea is captured by the table below.

2	`f:: (a -> b -> c)`	`zip f:: [a] -> [b] -> [c]`	`help two’ f (zip f):: [a] -> [b] -> [c]`
1	`f x:: (b -> c)`	`zip f xs:: [b] -> [c]`	`help one’ (f x) (zip f xs):: [b] -> [c]`
0	`f x y:: c`	`zip f xs ys:: [c]`	`help zero’ (f x y) (zip f xs ys):: [c]`

```
zero’ = Z
one’ = S Z
two’ = S (S Z)

help:: Zip a b -> a -> b -> b
help Z x xs = x:xs
help (S n) f rcall =
 (\ ys -> case ys of
           (z:zs) -> help n (f z) (rcall zs)
           other -> skip n)
```

Code

```
skip:: Zip a b -> b
skip Z = []
skip (S n) = \ ys -> skip n

zipN:: Zip a b -> a -> b
zipN Z = \ n -> [n]
zipN (n@(S m)) =
  let zip f = help n f (\ x -> zip f x)
  in zip
```