Algebraic Data Types

David Basin

Department of Computer Science
ETH Zurich

The problem

- Until now, data modeling with

  Base types: Int, Bool, Char, Float, . . .

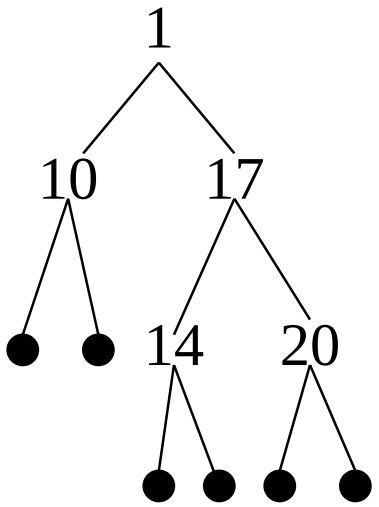
  Compound types: tuples, lists, functions, . . .

  Type synonyms: `type Complex = (Float,Float)`

- **Example**: modeling dates, e.g., months
  - Using strings: `"January"`, `"February"`, . . . , `"December"`
  - Using integers: $1, 2, \ldots, 12$

- Many possibilities (like in assembler). Not particularly abstract.
  - Also error-prone. What does 0 represent? Or $1 + 2$?
  - Analogous to problem of modeling years.

Another example: modeling trees



- Assembler or C: model using **pointers**
- Alternative: model using **functions**
  - Tree addresses (position) as path address

$$[\,], [1], [2, 1, 2], \ldots$$

  - Function application produces value associated with node

$$f\,[\,] = 1,\ f\,[1] = 10,\ f\,[2] = 17,\ f\,[2, 1] = 14,\ f\,[2, 2] = 20$$

- Data type is a derived notion, not first-class
  - Low-level coding
  - Can be improperly used

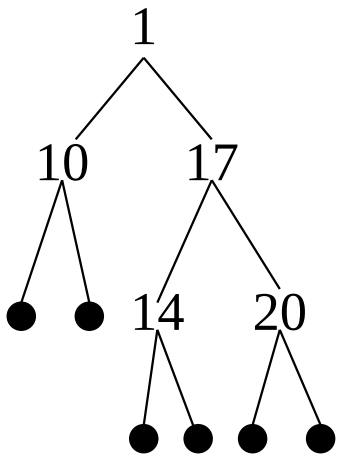
Solution: algebraic data types

- Declare new types tailored to the objects being modeled.
- For months, we declare the type `Month` with elements

  `January, February, ..., December`

  These are new **data constructors**.
- For trees, declare type `Tree` with elements like

```
Node 1 (Node 10 Leaf Leaf)
       (Node 17 (Node 14 Leaf Leaf)
                (Node 20 Leaf Leaf))
```



Enumeration types (disjoint unions)

```
data Season = Spring | Summer | Fall | Winter
data Month = January | February | March | April | May | June | July |
             August | September | October | November | December
```

- Syntax
  - Starts with keyword `data`
  - Names different (uniquely named) constructors
  - First letter of each constructor must be upper-case
- Defines a set: $\texttt{Season} = \{\texttt{Spring}, \texttt{Summer}, \texttt{Fall}, \texttt{Winter}\}$
- Functions can be written using pattern matching

```
whichSeason :: Month -> Season

whichSeason January = Winter
whichSeason February = Winter
whichSeason March = Spring
...
```

Product types

```
data People = Person Name Age
type Name = String
type Age = Int
```

- An element of type `People` consists of a name `n` and an age `a`, e.g.,

```
Person "Uncle George" 85
Person "Levi Jeans"  501          -- Nonsense but type correct
```

- Constructors are functions

```
? :type Person
Person :: Name -> Age -> People
```

- Functions may be defined by pattern matching

```
showPerson :: People -> String
showPerson (Person n a) = n ++ " who is " ++ show a ++ " years old"

? showPerson (Person "Uncle George" 85)
"Uncle George who is 85 years old" :: [Char]
```

Product types versus tuples

- Alternative to products are tuples

```
data People  = Person Name Age
type People' = (Name, Age)
```

- Advantage of product types
  - Conceptual: new, self-contained type
  - Objects are labeled and hence types are unambiguous
- Disadvantages
  - Longer definitions
  - Many polymorphic functions no longer applicable (*fst*, *zip*, . . .)

Enumeration and product types

- They can be combined

```
data Shape = Circle Float | Rectangle Float Float
```

- Two kinds of shapes
  - Circle with radius, e.g. `Circle 3.0`
  - Rectangle with two sides, e.g. `Rectangle 45.9 87.6`

- Functions again definable by pattern matching

```
area :: Shape -> Float
area (Circle r)      = pi * r * r
area (Rectangle h w) = h * w
```

Integration with classes

- No default functions like == or `show`

```
data Foo = D1 | D2 | D3

? D1 == D2
ERROR: No instance for (Eq Foo)
```

- Class instances can be explicitly created

```
instance Eq Foo where
  D1 == D1 = True
  D2 == D2 = True
  D3 == D3 = True
  _  == _  = False

? D1 == D2
False :: Bool
```

Integration with classes (cont.)

- In some cases, class instances can be automatically **derived**

```
data Foo = D1 | D2 | D3
           deriving (Eq, Ord, Enum, Show)
? D1
D1 :: Foo

? [D1 .. D3]
[D1, D2, D3] :: [Foo]

? D2 < D3
True :: Bool
```

See "Haskell Report" for further details

General definition

$$
\begin{array}{rcl}
\textit{data}\ T & = & Constr_1\ T_{11}\ \ldots\ T_{1k_1} \\
& | & Constr_2\ T_{21}\ \ldots\ T_{2k_2} \\
& & \vdots \\
& | & Constr_n\ T_{n1}\ \ldots\ T_{nk_n}
\end{array}
$$

- T_{ij} are types, possibly also containing T (i.e., recursion allowed)
- T can have type variables as arguments (polymorphism)

Let's look more closely at these extensions

Recursive types

- Sets of objects are often recursively defined

$$Expr \ ::= \ Int \ \mid \ Expr + Expr \ \mid \ Expr - Expr$$

- Formalized as a recursive data type

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
```

- Bijection between elements of $Expr$ and data-type elements

$$\begin{array}{rl} 2 & \mathit{Lit}\ 2 \\ 2+3 & \mathit{Add}\ (\mathit{Lit}\ 2)\ (\mathit{Lit}\ 3) \\ 2+(3-1) & \mathit{Add}\ (\mathit{Lit}\ 2)\ (\mathit{Sub}\ (\mathit{Lit}\ 3)\ (\mathit{Lit}\ 1)) \end{array}$$

- Recursion: `Expr` is recursive. Hence so are functions over `Expr`

Recursive functions over data types

- **Example**: Interpreter for arithmetic expressions

```
data Expr = Lit Int | Add Expr Expr | Sub Expr Expr
            deriving (Show, Eq)
```

- Evaluator

  Lit n**:** n
  Add e_1 e_2**:** value of e_1 + value of e_2
  Sub e_1 e_2**:** value of e_1 − value of e_2

- Program

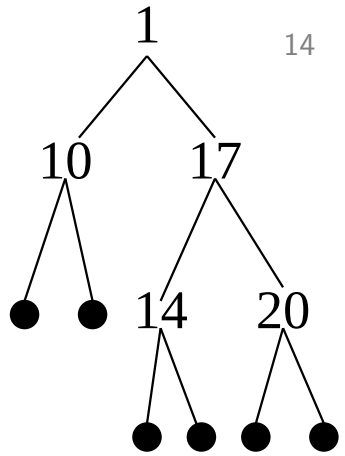
```
eval :: Expr -> Int

eval (Lit n) = n
eval (Add e1 e2) = (eval e1) + (eval e2)
eval (Sub e1 e2) = (eval e1) - (eval e2)
```

Arithmetic expressions (cont.)

- Other functions are written similarly

```
showExpr (Lit n)     = show n
showExpr (Add e1 e2) = "("++showExpr e1++"+"++showExpr e2 ++")"
showExpr (Sub e1 e2) = "("++showExpr e1++"-"++showExpr e2 ++")"

? eval (Add (Lit 2) (Sub (Lit 3) (Lit 1)))      -- 2 + (3 - 1)
4 :: Int

? show (Add (Lit 2) (Sub (Lit 3) (Lit 1)))
"Add (Lit 2) (Sub (Lit 3) (Lit 1))" :: [Char]

? showExpr (Add (Lit 2) (Sub (Lit 3) (Lit 1)))
"(2+(3-1))" :: [Char]
```

Trees (with internal integer nodes)



- Grammar

$$ITree ::= Leaf \mid Node\ Int\ ITree\ ITree$$

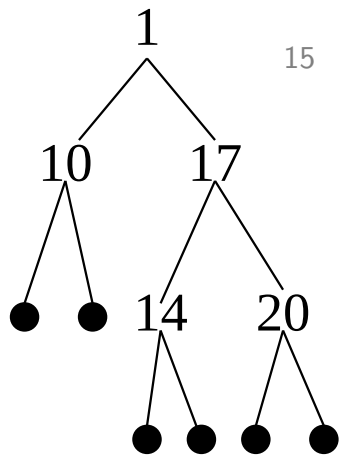
- Haskell data type

```
data ITree = Leaf | Node Int ITree ITree
             deriving (Eq, Show)
```

- Example tree t

```
Node 1 (Node 10 Leaf Leaf)
       (Node 17 (Node 14 Leaf Leaf)
                (Node 20 Leaf Leaf))
```

Functions over trees



- Sum of values `treeSum t = 62`

```
treeSum :: ITree -> Int
treeSum Leaf            = 0
treeSum (Node n t1 t2) = n + (treeSum t1) + (treeSum t2)
```

- Depth `depth t = 3`

```
depth :: ITree -> Int
depth Leaf            = 0
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

- How often does an element occur? `occurs t 10 = 1`

```
occurs :: ITree -> Int -> Int
occurs Leaf p = 0
occurs (Node n t1 t2) p
    | n == p    = 1 + rest
    | otherwise = rest
    where rest = occurs t1 p + occurs t2 p
```

Polymorphic algebraic types

- Examples have monomorphic types

```
data ITree = Leaf | Node Int ITree ITree
```

  In general, types may include type variables

- Example: `data Pair t = MkPair t t` has elements like:

```
MkPair 2 3 :: Pair Int
MkPair [] [2,3] :: Pair [Int]
MkPair [] [] :: Pair [t]
```

- Functions can now also be polymorphic

```
equalPair :: Eq t => Pair t -> Bool
equalPair (MkPair x y) = (x == y)
```

Trees

- Definition with type parameters

```
data Tree t = Leaf | Node t (Tree t) (Tree t)
              deriving (Eq,Ord,Show)

? Node 2 Leaf Leaf
Node 2 Leaf Leaf :: Tree Int

? Node True (Node False Leaf Leaf) (Node True Leaf Leaf)
Node True (Node False Leaf Leaf) (Node True Leaf Leaf) :: Tree Bool
```

- Same definitions. Now types are more general

```
occurs :: Eq t => Tree t -> t -> Int
occurs Leaf p = 0
occurs (Node n t1 t2) p
  | n == p      = 1 + rest
  | otherwise   = rest
  where rest = occurs t1 p + occurs t2 p
```

Polymorphic algebraic types — questions

- Have you seen this type before?

```
data L t = E | C t (L t)
           deriving (Eq,Ord,Show)
```

- Observe that

```
E :: L t
C :: t -> L t -> L t
```

- What is the type of the following function?

```
f y E       = False
f y (C x l) = x == y || f y l
```

- What is the result?

```
? f 3 (C 2 (C 3 (C 4 E)))
```

- So standard types (like Bool, Lists $[a]$, etc.) can be defined as algebraic data types

Higher-order programming with data types

- You have done this already! For example, `map` for lists

```
map f E = E                                  --- E = []
map f (C x xs) = C (f x) (map f xs)          --- C = (:)
```

- Analogous program on trees

```
mapTree :: (t -> u) -> Tree t -> Tree u
mapTree f Leaf            = Leaf
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1) (mapTree f t2)

? mapTree (+2)
          (Node 7 (Node 20 Leaf Leaf) (Node 1 Leaf Leaf))
Node 9 (Node 22 Leaf Leaf) (Node 3 Leaf Leaf) :: Tree Int

? mapTree not
         (Node True (Node False Leaf Leaf) (Node True Leaf Leaf))
Node False (Node True Leaf Leaf) (Node False Leaf Leaf) :: Tree Bool
```

From foldr to treeFold

- Recall `foldr`

```
foldr :: (a -> b -> b) -> b -> [a] -> b

foldr f e E          = e                          --- E = []
foldr f e (C x xs)   = f x (foldr f e xs)         --- C = (:)
```

- Procedure for `foldr f e l`

  In the list l, C is replaced with f and E with e

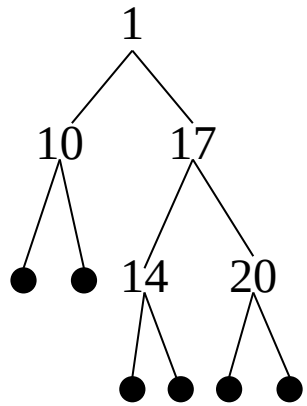
- Procedure for `treeFold f e t`

  In the tree t, `Node` is replaced with f and `Leaf` with e

- Definition

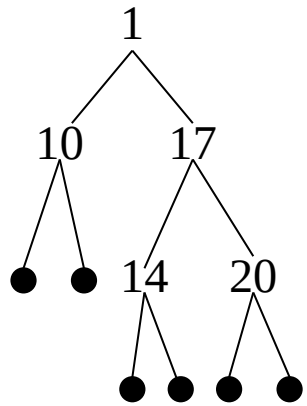
```
treeFold:: (a -> b -> b -> b) -> b -> Tree a -> b

treeFold f e Leaf          = e
treeFold f e (Node x l r) = f x (treeFold f e l) (treeFold f e r)
```

What is computed?




```
t = Node 1 (Node 10 Leaf Leaf)
           (Node 17 (Node 14 Leaf Leaf)
                    (Node 20 Leaf Leaf))

? treeFold (\x l r -> x + l + r) 0 t
62 :: Int

? treeFold (\_ l r -> 1 + l + r) 0 t
5 :: Int

? treeFold (\_ l r -> 1 + max l r) 0 t
3 :: Int

? treeFold (\x l r -> Node x r l) Leaf t
Node 1 (Node 17 (Node 20 Leaf Leaf) (Node 14 Leaf Leaf))
       (Node 10 Leaf Leaf) :: Tree Int
```

From trees to lists




```
t = Node 1 (Node 10 Leaf Leaf)
           (Node 17 (Node 14 Leaf Leaf)
                    (Node 20 Leaf Leaf))

preorder t = treeFold (\x l r -> [x] ++ l ++ r) [] t

inorder t = treeFold (\x l r -> l ++ [x] ++ r) [] t

postorder t = treeFold (\x l r -> l ++ r ++ [x]) [] t

? preorder t
[1, 10, 17, 14, 20] :: [Int]

? inorder t
[10, 1, 14, 17, 20] :: [Int]

? postorder t
[10, 14, 20, 17, 1] :: [Int]
```

Case study: editing distance

- **Motivation**: compute the minimal number of changes needed to transform one string into another
  - Practical problem, e.g., updating display
  - "diff" programs

- **Goal**: find "cheapest" sequence of editing steps using operations

  Change a character
  Copy a character without change
  Delete a character
  Insert a character
  Kill rest of string, i.e., delete to the end

  Assume unit price for all operations except `copy`, which is for free.

Example: "Help" is not far from "Hello"

```
? transform "help" "hello"
[Copy, Copy, Copy, Insert 'l', Change 'o']

? transform "hello" "help"
[Copy, Copy, Copy, Delete, Change 'p']
```

In both cases, the cost is 2

Development #1: data types

1. Identify types, e.g., for editing

```
data Edit = ...
```

2. Identify kinds of data

   Each corresponds to a constructor

```
data Edit =  Change ...  |  Copy ...     |
             Delete ...  |  Insert ...   |  Kill ...
```

3. Fix the components (arguments)

```
data Edit =  Change Char |  Copy         |
             Delete      |  Insert Char  |  Kill
             deriving (Eq, Show)
```

Development #2: functions

- Main function: carry out transformation

```
transform :: String -> String -> [Edit]
```

- Base cases

```
transform [] [] = []
transform xs [] = [Kill]
transform [] ys = map Insert ys
```

- General case: choose between the operations

```
transform (x:xs) (y:ys)
  | x == y       = Copy : transform xs ys
  | otherwise    = best [ Delete   : transform xs     (y:ys) ,
                          Insert y : transform (x:xs) ys     ,
                          Change y : transform xs     ys     ]
```

N.B. `Kill` is not necessary. Why?

Development (cont.)

- Define the auxiliary function `best`

```
best :: [[Edit]] -> [Edit]

best [x] = x
best (x:xs)
  | cost x <= cost x' = x
  | otherwise         = x'
  where x' = best xs
```

- Formalize unit price for all operations except `copy`

```
cost :: [Edit] -> Int
cost = length . filter (/=Copy)
```

The entire program

```
data Edit = Change Char | Copy | Delete | Insert Char | Kill
            deriving (Eq,Show)

transform [] [] = []
transform xs [] = [Kill]
transform [] ys = map Insert ys
transform (x:xs) (y:ys)
  | x == y       = Copy : transform xs ys
  | otherwise    = best [ Delete   : transform xs     (y:ys) ,
                          Insert y : transform (x:xs) ys     ,
                          Change y : transform xs     ys     ]

best [x] = x
best (x:xs)
  | cost x <= cost x' = x
  | otherwise         = x'
  where x' = best xs

cost = length . filter (/=Copy)
```

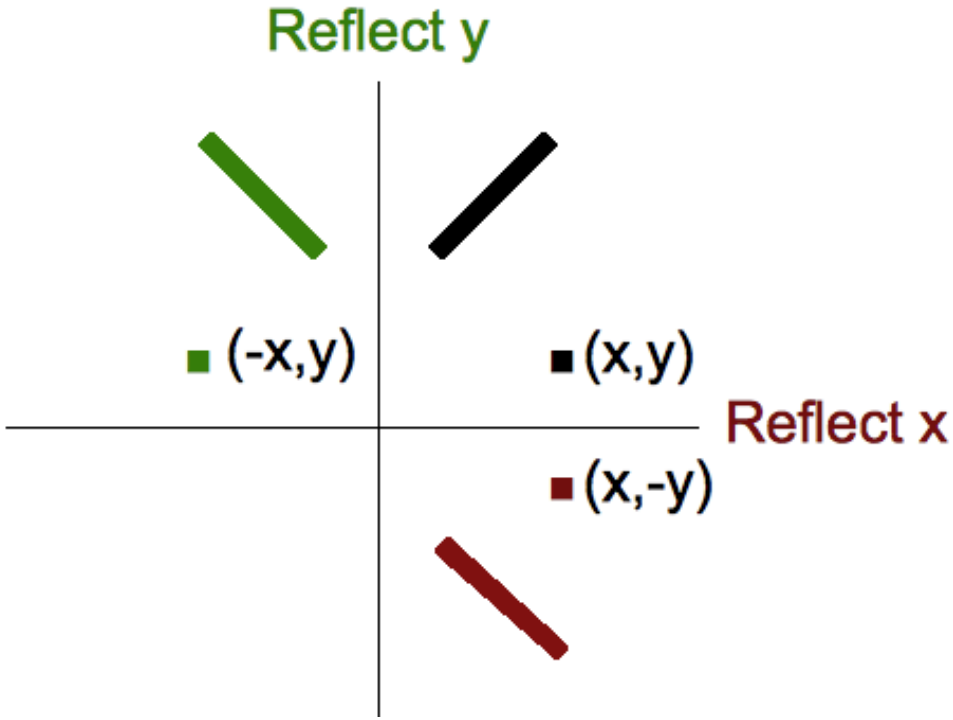
Transform — examples

```
? :set +s

? transform "fish" "chips"
[Insert 'c', Change 'h', Copy, Insert 'p', Copy, Kill]
(0.02 secs, 3803456 bytes)

? transform "1234" "4321"
[Delete, Change '4', Copy, Insert '2', Change '1']
(0.01 secs, 3051880 bytes)

? transform "123456" "654321"
[Delete, Change '6', Change '5', Copy, Insert '3', Change '2',
 Change '1']
(0.04 secs, 5990368 bytes)

? transform "12345678" "87654321"
[Delete, Change '8', Change '7', Change '6', Copy, Insert '4',
 Change '3', Change '2', Change '1']
(1.00 secs, 84024700 bytes)
```

Does `transform` always terminate? Run-time complexity?

Algebraic types and type classes

Goal: define hierarchy of movable objects



Support standard objects like points, lines, ...
and operations like reflection and (simple 180-degree) rotation.

Algebraic types and type classes (cont.)

```
data Vector = Vector Float Float          -- x & y offset

class Movable t where
  move      :: Vector -> t -> t           -- Translation
  reflectX  :: t -> t                     -- Reflection
  reflectY  :: t -> t
  rotate180 :: t -> t                     -- Rotation
  rotate180 = reflectX . reflectY
```

Instance: point

```
data Point = Pt Float Float
             deriving Show

instance Movable Point where
  move (Vector v1 v2) (Pt c1 c2) = Pt (c1+v1) (c2+v2)
  reflectX (Pt c1 c2)            = Pt c1 (-c2)
  reflectY (Pt c1 c2)            = Pt (-c1) c2
  rotate180 (Pt c1 c2)           = Pt (-c1) (-c2)
```

Types and classes (cont.)

- Figures are also movable

```
data Figure = Line Point Point | Circle Point Float
              deriving Show

instance Movable Figure where
  move v (Line p1 p2)   = Line (move v p1) (move v p2)
  move v (Circle p r)   = Circle (move v p) r

  reflectX (Line p1 p2) = Line (reflectX p1) (reflectX p2)
  reflectX (Circle p r) = Circle (reflectX p) r

  reflectY (Line p1 p2) = Line (reflectY p1) (reflectY p2)
  reflectY (Circle p r) = Circle (reflectY p) r
```

- Lists of movable objects are also movable

```
instance Movable t => Movable [t] where
  move v   = map (move v)
  reflectX = map reflectX
  reflectY = map reflectY
```

Algebraic types and classes

- Algebraic types are "first class" citizens

  Fully compatible with polymorphism and type classes

- Programs are simple to read and understand

  only `move` rather than `movePoint`, etc.

- Reusability

  `Instance` for `Movable [t]` is polymorphic with respect to lists of movable objects

What actually are algebraic data types?

```
data Tree a = Leaf | Node a (Tree a) (Tree a)
```

- A data type defines a set of terms for each type instance

  E.g., `Tree Int` corresponds to $\{\mathit{Leaf}\ ,\ \mathit{Node}\ 0\ \mathit{Leaf}\ \mathit{Leaf}\ ,\ \ldots\}$

- Algebraic here means the smallest set S, where

  $$\mathit{Leaf}\ \in S \quad \textit{and} \quad x \in a \wedge t_1 \in S \wedge t_2 \in S\ \Rightarrow\ (\mathit{Node}\ x\ t_1\ t_2) \in S$$

- Intuition: set S is built in steps
  - $\mathit{Leaf}\ \in S$ and
  - $(\mathit{Node}\ x\ t_1\ t_2) \in S$, where t_1 and t_2 in S in earlier steps
- What are sound reasoning principles for such types?

Correctness for Algebraic Data Types

Let's start first with some old friends

Natural numbers

- Theorem to prove $\forall n \in \mathcal{N}.\, P(n)$

  Base case: Show $P(0)$.
  Step case: Let $n \in \mathcal{N}$ be arbitrary. Assume $P(n)$.
  Show $P(n+1)$.

- Alternative formulation as natural deduction proof rule

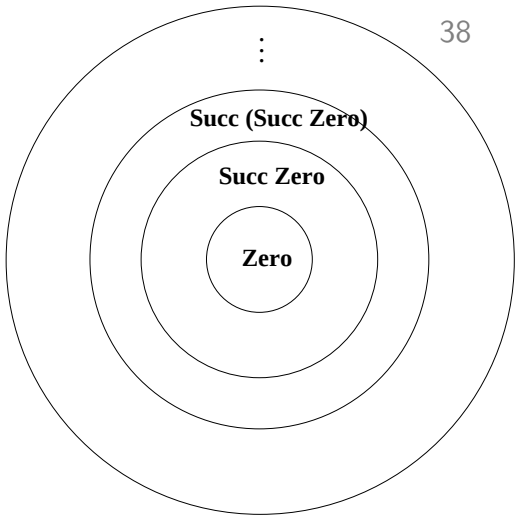
$$\frac{\Gamma \vdash P(0) \quad \Gamma, P(n) \vdash P(n+1)}{\Gamma \vdash \forall n \in \mathcal{N}.\, P(n)} \; n \text{ not free in } \Gamma$$

- $\forall n \in \mathcal{N}.\, P(n)$ holds as $P(0)$, $P(1)$, $P(2)$, . . .

Numbers as a data type

```
data Nat = Zero | Succ Nat
           deriving (Eq, Ord, Show)

plus Zero     y = y
plus (Succ x) y = Succ (plus x y)  --- Normally built-in primitives
                                   --- that use machine arithmetic!
times Zero     _ = Zero
times (Succ x) y = plus (times x y) y


?  plus (Succ Zero) (Succ Zero)
Succ (Succ Zero)

? times (Succ (Succ Zero)) (Succ (Succ Zero))
Succ (Succ (Succ (Succ Zero)))

? Succ Zero < Succ (Succ Zero)     --- How does ghc compute this?
True
```

Induction over the natural numbers




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

- Natural numbers are (isomorphic to) the set

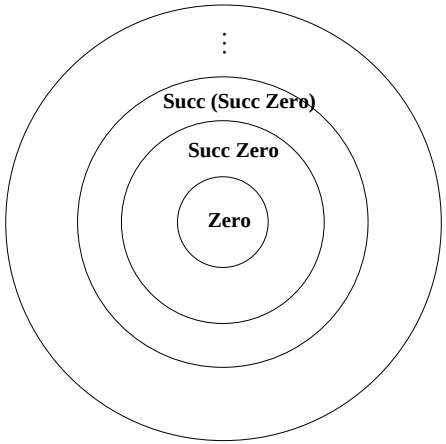
$$Nat = \{Zero, Succ\ Zero, Succ\ (Succ\ Zero), \ldots\}$$

- Data type provides two rules for constructing members of *Nat*

0) $Zero \in Nat$
1) $Succ\ x \in Nat$, if $x \in Nat$

- Elements added step by step

Structural induction




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

- Induction over the structure of terms

  Equivalent to induction over the individual steps
  (also depth of terms)

$$\frac{\Gamma \vdash P(\mathit{Zero}) \qquad \Gamma,\ P(n) \vdash P(\mathit{Succ}\ n)}{\Gamma \vdash \forall n \in \mathit{Nat}.\, P(n)}\ n \text{ not free in } \Gamma$$

- Sufficient to show $P(\mathit{Zero})$, $P(\mathit{Succ}\ \mathit{Zero})$, . . .
- We can prove theorems like

$$\forall x \in \mathit{Nat}.\, \mathit{plus}\ x\ (\mathit{plus}\ y\ z) = \mathit{plus}\ (\mathit{plus}\ x\ y)\ z$$

Lists

```
data L t = Nil | Cons t (L t)
```

- Elements in $L\ t$ are built in steps

  0) $\{Nil\}$
  1) $\{Cons\ a\ Nil \in L\ t \mid a \in t\}$
  2) $\{Cons\ b\ (Cons\ a\ Nil) \in L\ t \mid a, b \in t\}$
  ⋮

- $l \in L\ t$ iff l appears in some step of the construction

- Induction

$$\frac{\Gamma \vdash P(Nil) \qquad \Gamma, P(xs) \vdash P(Cons\ x\ xs)}{\Gamma \vdash \forall xs \in L\ t.\, P(xs)}\ x, xs \text{ not free in } \Gamma$$

  i.e., we must prove (i) $P(Nil)$ and (ii) $P(Cons\ x\ xs)$ follows from $P(xs)$, for an arbitrary $x \in t$ and $xs \in L\ t$.

Trees

```
data Tree t = Leaf | Node t (Tree t) (Tree t)
```

- Steps for constructing *Tree* t

  0) $\{$*Leaf* $\}$

  1) $\{$*Node* a *Leaf Leaf* $\in$ *Tree* $t \mid a \in t\}$

  ⋮

  i) Trees in step i are of form *Node* $a\ l\ r$, where $a \in t$ and l and r have been constructed in the previous steps.

- $s \in$ *Tree* t iff s appears in some step of construction

- Induction

$$\frac{\Gamma \vdash P(\textit{Leaf}) \qquad \Gamma, P(l), P(r) \vdash P(\textit{Node}\ a\ l\ r)}{\Gamma \vdash \forall x \in \textit{Tree}\ t.\ P(x)}\ a, l, r \text{ not free in } \Gamma$$

  i.e., we must prove (i) $P($*Leaf* $)$ and (ii) $P($*Node* $a\ l\ r)$ follows from $P(l)$ and $P(r)$, for an arbitrary $a \in t$ and $l, r \in$ *Tree* t.

Example of induction on trees

```
[]     ++ ys = ys
(x:xs) ++ ys = x:(xs ++ ys)

map f []     = []
map f (a:l) = f a : map f l

mapTree f Leaf            = Leaf
mapTree f (Node x t1 t2) = Node (f x) (mapTree f t1) (mapTree f t2)

treeFold f e Leaf          = e
treeFold f e (Node x l r) = f  x (treeFold f e l) (treeFold f e r)

inorder t = treeFold (\x l r -> l ++ [x] ++ r) [] t
```

Does the following hold?

$$\forall s \in \mathit{Tree}\ t.\ \mathit{map}\ f\ (\mathit{inorder}\ s) = \mathit{inorder}\ (\mathit{mapTree}\ f\ s)$$

Auxiliary propositions

- **Lemma:** *inorder Leaf* $= [\,]$

$$\begin{aligned} \textit{inorder Leaf} &= \textit{treeFold}\ (\lambda\, x\, l\, r.\, l +\!\!+ [x] +\!\!+ r)\ [\,]\ \textit{Leaf} \\ &= [\,] \end{aligned}$$

- **Lemma:** *inorder* $(\textit{Node}\ a\ l\ r) = \textit{inorder}\ l +\!\!+ [a] +\!\!+ \textit{inorder}\ r$

  Let $f = \lambda\, x\, l\, r.\, l +\!\!+ [x] +\!\!+ r$. Then

$$\begin{aligned} \textit{inorder}\ (\textit{Node}\ a\ l\ r) &= \textit{treeFold}\ f\ [\,]\ (\textit{Node}\ a\ l\ r) \\ &= f\ a\ (\textit{treeFold}\ f\ [\,]\ l)\ (\textit{treeFold}\ f\ [\,]\ r) \\ &= (\textit{treeFold}\ f\ [\,]\ l) +\!\!+ [a] +\!\!+ (\textit{treeFold}\ f\ [\,]\ r) \\ &= \textit{inorder}\ l +\!\!+ [a] +\!\!+ \textit{inorder}\ r \end{aligned}$$

Auxiliary propositions (cont.)

- **Lemma:** $map\ f\ (l +\!\!+ r) = (map\ f\ l) +\!\!+ (map\ f\ r)$
  Proof: Let $P(l) \equiv map\ f\ (l +\!\!+ r) = (map\ f\ l) +\!\!+ (map\ f\ r)$.

  We show $\forall l \in [a].\ P(l)$ by induction.

  Base case: Show $P([])$.

$$\begin{aligned} map\ f\ ([\,] +\!\!+ r) &= map\ f\ r \\ &= [\,] +\!\!+ (map\ f\ r) \\ &= (map\ f\ [\,]) +\!\!+ (map\ f\ r) \end{aligned}$$

  Step case: Let $x \in a$ and $l \in [a]$ be arbitrary. Assume $P(l)$. Show $P(x : l)$.

$$\begin{aligned} map\ f\ ((x : l) +\!\!+ r) &= map\ f\ (x : (l +\!\!+ r)) \\ &= f\ x : map\ f\ (l +\!\!+ r) \\ &= f\ x : ((map\ f\ l) +\!\!+ (map\ f\ r)) \\ &= (f\ x : map\ f\ l) +\!\!+ (map\ f\ r) \\ &= (map\ f\ (x : l)) +\!\!+ (map\ f\ r) \end{aligned}$$

Correctness proof

Lemma: $\forall s \in \mathit{Tree}\ \ t.\ \mathit{map}\ f\ (\mathit{inorder}\ s) = \mathit{inorder}\ (\mathit{mapTree}\ f\ s)$

Proof: Let $P(s) \equiv \mathit{map}\ f\ (\mathit{inorder}\ s) = \mathit{inorder}\ (\mathit{mapTree}\ f\ s)$.
We show $\forall s \in \mathit{Tree}\ \ t.\ P(s)$ by induction.

Base case: Show $P(\mathit{Leaf})$.
$\mathit{map}\ f\ (\mathit{inorder}\ \mathit{Leaf}\ ) = \mathit{map}\ f\ [\,] = [\,] = \mathit{inorder}\ \mathit{Leaf} = \mathit{inorder}\ (\mathit{mapTree}\ f\ \mathit{Leaf}\ )$

Step case: Let $a \in t$ and $l, r \in \mathit{Tree}\ \ t$ be arbitrary.
Assume $P(l)$ and $P(r)$. Show $P(\mathit{Node}\ a\ l\ r)$.

$$
\begin{aligned}
& \mathit{map}\ f\ (\mathit{inorder}\ (\mathit{Node}\ a\ l\ r)) \\
= \ & \mathit{map}\ f\ (\mathit{inorder}\ l +\!\!+ [a] +\!\!+ \mathit{inorder}\ r) \\
= \ & (\mathit{map}\ f\ (\mathit{inorder}\ l)) +\!\!+ (\mathit{map}\ f\ [a]) +\!\!+ (\mathit{map}\ f\ (\mathit{inorder}\ r)) \\
= \ & (\mathit{map}\ f\ (\mathit{inorder}\ l)) +\!\!+ [f\ a] +\!\!+ (\mathit{map}\ f\ (\mathit{inorder}\ r)) \\
= \ & (\mathit{inorder}\ (\mathit{mapTree}\ f\ l)) +\!\!+ [f\ a] +\!\!+ (\mathit{inorder}\ (\mathit{mapTree}\ f\ r)) \\
= \ & \mathit{inorder}\ (\mathit{Node}\ (f\ a)\ (\mathit{mapTree}\ f\ l)\ (\mathit{mapTree}\ f\ r)) \\
= \ & \mathit{inorder}\ (\mathit{mapTree}\ f\ (\mathit{Node}\ a\ l\ r))
\end{aligned}
$$

Structural induction — general idea

- Induction based on structure of terms

```
data T t = Leaf t | Node1 (T t) | Node2 t (T t) (T t)
```

- What are the terms in Step 0?

$$\{\mathit{Leaf}\ a \mid a \in t\}$$

- How do we go from step $i-1$ to step i?

$$\{\mathit{Node1}\ s \mid s \in T_{i-1}\} \cup \{\mathit{Node2}\ a\ s_1\ s_2 \mid a \in t \text{ and } s_1, s_2 \in T_{i-1}\}$$

where T_{i-1} contains the elements from the previous steps.

- Formalized as induction rule

$$\frac{\Gamma \vdash P(\mathit{Leaf}\ a) \quad \Gamma, P(s) \vdash P(\mathit{Node1}\ s) \quad \Gamma, P(s_1), P(s_2) \vdash P(\mathit{Node2}\ a\ s_1\ s_2)}{\Gamma \vdash \forall x \in T\ t.\ P(x)}\ (*)$$

(*) a, s, s_1, s_2 not free in Γ

Conclusion — algebraic types

- Algebraic types improve possibilities for modeling
  - No ambiguity: is "2000" a number or a year?
  - Terms directly model objects
- General recursive types combine enumeration and product types
- Lists as example
  - Recursive and polymorphic
  - Many specialized functions can be written using general recursive combinators
- Integrated with classes. Supports development in the large
- Induction is a fundamental reasoning principle