

Algebraic Data Types*

Andreas Lochbihler

Department of Computer Science
ETH Zurich

*Thanks to David Basin for slide material

The problem

- Until now, data modeling with
 - Base types:** Int, Bool, Char, Double, ...
 - Compound types:** tuples, lists, functions, ...
 - Type synonyms:** `type Complex = (Double, Double)`
- **Example:** modeling dates, e.g., months
 - ▶ Using strings: "January", "February", ..., "December"
 - ▶ Using integers: 1, 2, ..., 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

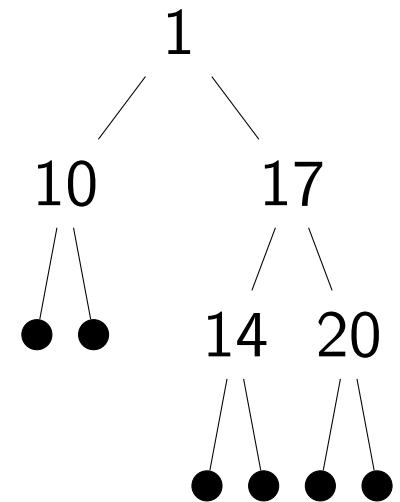
- Assembler or C: model using **pointers**
- Alternative: model using **functions**
 - ▶ Tree addresses (position) as path address

$[], [1], [2, 1, 2], \dots$

- ▶ 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

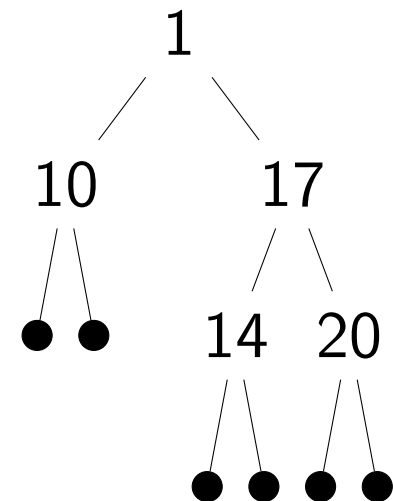
- Declare new types tailored to the objects being modeled.
- For months, we declare the type `Month` with elements

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: $\text{Season} = \{\text{Spring}, \text{Summer}, \text{Fall}, \text{Winter}\}$
- 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 Double | Rectangle Double Double
```

- 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 -> Double
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

$$\begin{array}{lcl} \text{data } T & = & \text{Constr}_1 \ T_{11} \ \dots \ T_{1k_1} \\ & | & \text{Constr}_2 \ T_{21} \ \dots \ T_{2k_2} \\ & & \vdots \\ & | & \text{Constr}_n \ T_{n1} \ \dots \ T_{nk_n} \end{array}$$

- 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

$$2 \quad Lit\ 2$$

$$2 + 3 \quad Add\ (Lit\ 2)\ (Lit\ 3)$$

$$2 + (3 - 1) \quad Add\ (Lit\ 2)\ (Sub\ (Lit\ 3)\ (Lit\ 1))$$

- 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++")"
```

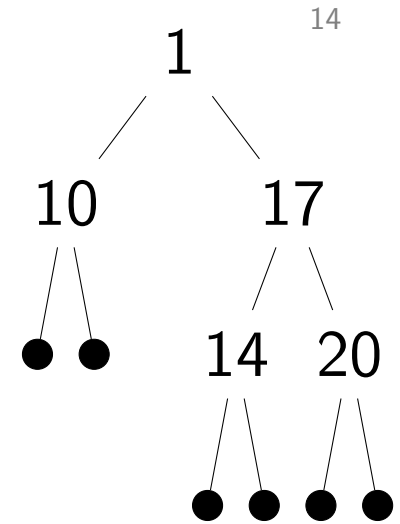
```
? eval (Add (Lit 2) (Sub (Lit 3) (Lit 1)))      -- 2 + (3 - 1)
4 :: Int
```

```
? 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

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


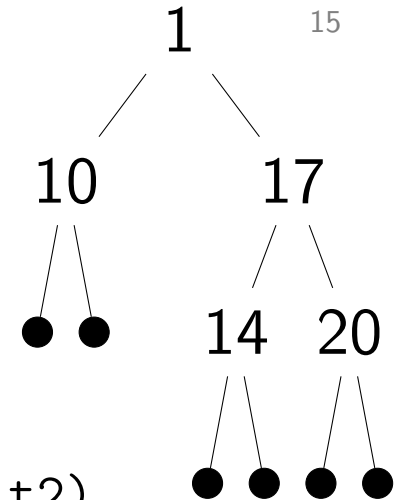
- Haskell data type

```
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 $\text{treeSum } t = 62$

```
treeSum :: ITree -> Int
```

```
treeSum Leaf = 0
```

```
treeSum (Node n t1 t2) = n + (treeSum t1) + (treeSum t2)
```

- Depth $\text{depth } t = 3$

```
depth :: ITree -> Int
```

```
depth Leaf = 0
```

```
depth (Node n t1 t2) = 1 + max (depth t1) (depth t2)
```

- How often does an element occur? $\text{occurs } t \ 10 = 1$

```
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)))
```

Algebraic data types from the Prelude

Standard types are defined as algebraic data types

- Booleans are an enumeration

```
data Bool = False | True
  deriving (Eq, Ord, Enum, Read, Show, Bounded)
```

- Lists

```
data [a] = [] | a : [a] deriving (Eq, Ord)
```

- Optional values

```
data Maybe a = Nothing | Just a
  deriving (Eq, Ord, Read, Show)
```

- Disjoint union type

```
data Either a b = Left a | Right b
  deriving (Eq, Ord, Read, Show)
```

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`

<code>foldr f e E</code>	<code>= e</code>	<code>---</code>	<code>E = []</code>
<code>foldr f e (C x xs)</code>	<code>= f x (foldr f e xs)</code>	<code>---</code>	<code>C = (:) </code>

- 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`

<code>treeFold f e Leaf</code>	<code>= e</code>
<code>treeFold f e (Node x l r)</code>	<code>= f x (treeFold f e l) (treeFold f e r)</code>

What is computed?

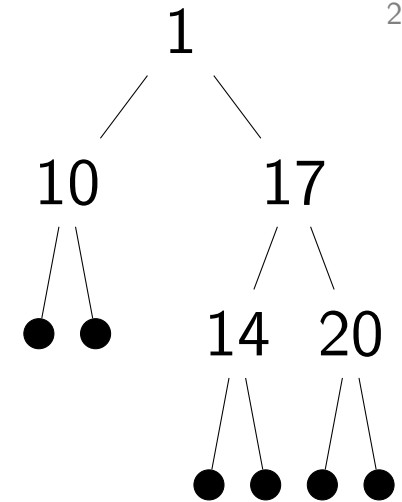
```
t = Node 1 (Node 10 Leaf Leaf)
          (Node 17 (Node 14 Leaf Leaf)
                   (Node 20 Leaf Leaf))
```

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

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

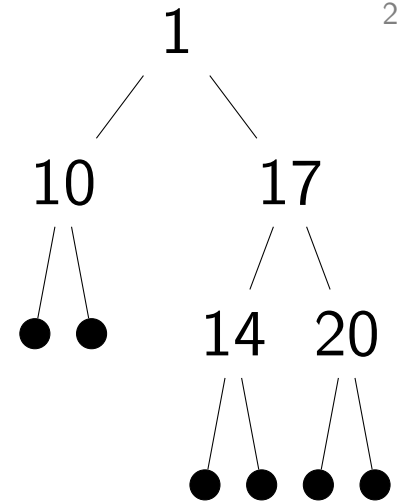
```
? treeFold (\_ l r -> 1 + max l r) 0 t
3 :: Int
```

```
? 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

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 ...
```

3. Fix the components (arguments)

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

Development #2: functions

- Main function: carry out transformation

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

- Base cases

```
transform [] [] = []  
transform xs [] = map (\_ -> Delete) xs  
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      ]
```

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
          deriving (Eq,Show)

transform [] [] = []
transform xs [] = map (\_ -> Delete) xs
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, Delete]  
(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)
```

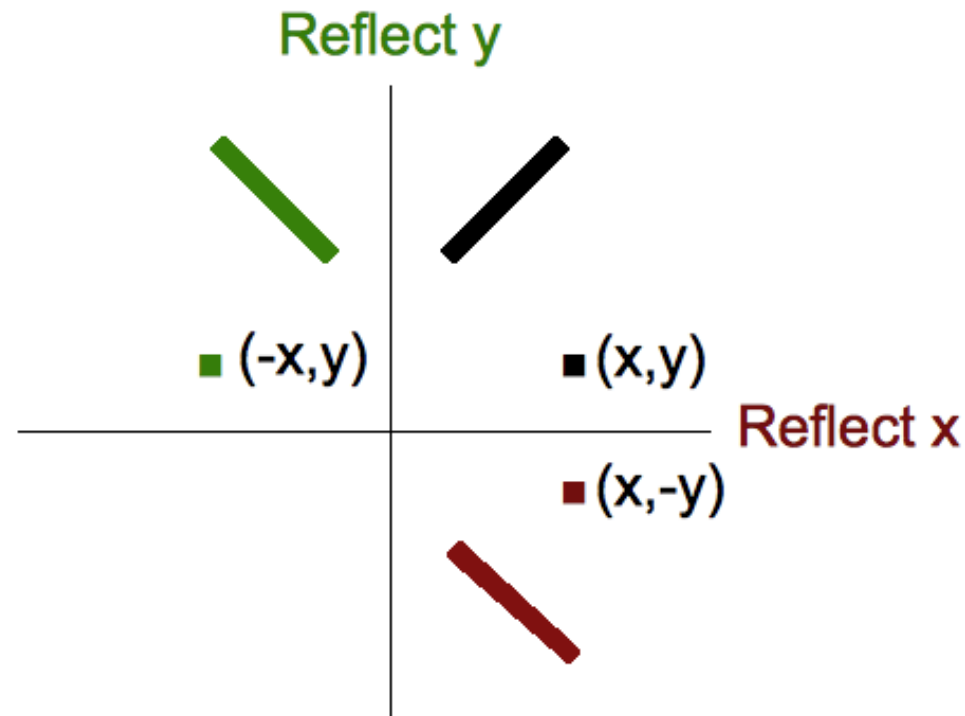
```
? transform "12345678" "87654321"
```

```
[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 Double Double           -- 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 Double Double
              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 Double
  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 $\{Leaf, Node\ 0\ Leaf\ Leaf, \dots\}$

- Algebraic here means the smallest set S , where

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

- Intuition: set S is built in steps

- ▶ $Leaf \in S$ and

- ▶ $(Node\ x\ t_1\ t_2) \in S$, 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 with some old friends

Natural numbers

- 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

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

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

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
```

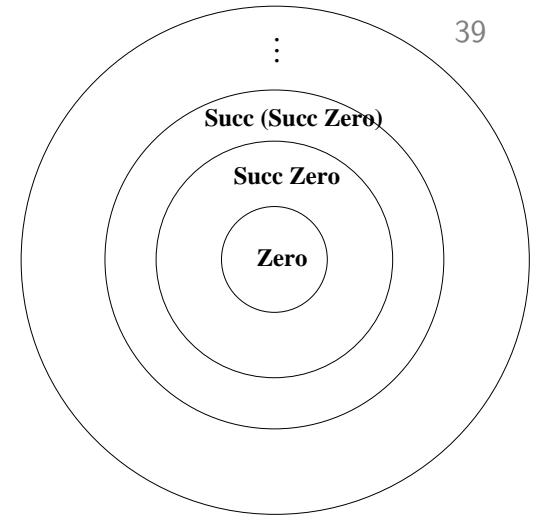
```
? plus (Succ Zero) (Succ Zero)
Succ (Succ Zero)
```

```
? 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
```

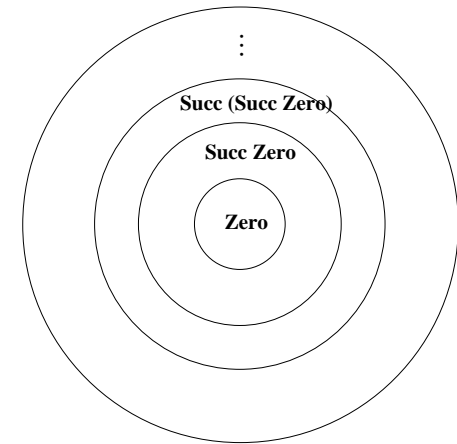


- Natural numbers are (isomorphic to) the set

$$\mathit{Nat} = \{ \mathit{Zero}, \mathit{Succ} \ \mathit{Zero}, \mathit{Succ} \ (\mathit{Succ} \ \mathit{Zero}), \dots \}$$

- Data type provides two rules for constructing members of Nat
 - 0) $\mathit{Zero} \in \mathit{Nat}$
 - 1) $\mathit{Succ} \ x \in \mathit{Nat}$, if $x \in \mathit{Nat}$
- 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)

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

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

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

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

$$\frac{\Gamma \vdash P(Nil) \quad \Gamma, P(xs) \vdash P(Cons\ x\ xs) \quad \begin{array}{l} x, xs \text{ not free in } \Gamma, \\ x \text{ not free in } P \end{array}}{\Gamma \vdash \forall xs \in L\ t. P(xs)}$$

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

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

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?

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

Auxiliary propositions

- **Lemma:** $\text{inorder Leaf} = []$

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

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

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

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

Auxiliary propositions (cont.)

- **Lemma:** $\text{map } f (l ++ r) = (\text{map } f l) ++ (\text{map } f r)$

Proof: Let $P(l) \equiv \text{map } f (l ++ r) = (\text{map } f l) ++ (\text{map } f r)$.

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

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

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

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

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

Correctness proof

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

Proof: Let $P(s) \equiv \text{map } f (\text{inorder } s) = \text{inorder } (\text{mapTree } f s)$.

We show $\forall s \in \text{Tree } t. P(s)$ by induction.

Base case: Show $P(\text{Leaf})$.

$$\text{map } f (\text{inorder } \text{Leaf}) = \text{map } f [] = [] = \text{inorder } \text{Leaf} = \text{inorder } (\text{mapTree } f \text{ Leaf})$$

Step case: Let $a \in t$ and $l, r \in \text{Tree } t$ be arbitrary.

Assume $P(l)$ and $P(r)$. Show $P(\text{Node } a l r)$.

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

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?

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

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

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

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

- Formalized as induction rule

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

(*) a, s, s_1, s_2 not free in Γ or P

Conclusion — algebraic types

- Algebraic types provide new 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