Functional Programming

2022-2023

Sjaak Smetsers

Algebraic datatypes and type classes

Lecture 4

Outline

- New datatypes
- Product and sum datatypes
- Parametric datatypes
- Recursive datatypes
- Type classes
- Summary

New datatypes

- we've seen `type` synonyms for existing types
- we've also seen enumerations as new `data`types
- `data` is much more general than this
  - product and sum datatypes
  - parametric datatypes
  - recursive datatypes

Product datatypes

- constructors of enumerated types are constants (Mon); constructors may be functions too (i.e. can have arguments)
- e.g. people with names and ages

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

- then P `:: Name ⟶ Age ⟶ Person`
- such *constructor functions* **do not** simplify: they are in (head) normal form; moreover, they can be used in pattern-matching

```
showPerson :: Person ⟶ String
showPerson (P n a) = "Name: " ++ n ++ ", Age: " ++ show a
```

Sum datatypes

- datatypes can have *multiple variants*

```
data Suit = Spades | Hearts | Diamonds | Clubs
data Rank = Faceless Integer | Jack | Queen | King
data Card = Card Rank Suit | Joker
```

- so a Rank is *either* of the form `Faceless n` for some `n`, or a constant `Jack`, `Queen`, or `King`
- The name `Card` is used both for a type and for a constructor

Parametric (polymorphic) datatypes

- datatypes may be *parametric/ polymorphic*
- then constructors are *polymorphic functions*

```
data Maybe a = Nothing | Just a
```

- e.g. `Just 13 :: Maybe Int`
- so `Nothing :: Maybe a`, `Just :: a → Maybe a`
- useful for indicating failure

```
head' :: [a] → Maybe a
head' []     = Nothing
head' (x:_) = Just x
```

Recursive datatypes

- datatypes may be recursive too
- e.g. arithmetic expressions
- e.g. lists
- e.g. binary trees
- e.g. general trees

Example 1: natural numbers

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

- A value of type `Nat` is either `Zero`, or of the form `Succ n` where `n :: Nat`. That is, `Nat` contains the following infinite sequence of values:

```
Zero
Succ Zero
Succ (Succ Zero)
…
```

- We can think of values of type `Nat` as natural numbers, where `Zero` represents `0`, and `Succ` represents the successor function `(+ 1)`
- For example, the value `Succ (Succ (Succ Zero))` represents `(+ 1) ((+ 1) ((+ 1) 0)) = 3`

Conversions between Nat and Int

- Using recursion, it is easy to define functions that convert between values of type `Nat` and `Int`

```
nat2int :: Nat ⟶ Int
nat2int Zero     = 0
nat2int (Succ n) = 1 + nat2int n

int2nat :: Int ⟶ Nat
int2nat 0 = Zero
int2nat n = Succ (int2nat (n-1))
```

`Nat` design pattern

- remember: every datatype comes with a pattern of definition
- task: define a function `f :: Nat ⟶ S`
- step 1: solve the problem for `Zero`

  ```
  f Zero = …
  ```

- step 2: assume that you already have the solution for `n` at hand, *extend* the intermediate solution to a solution for `Succ n`

  ```
  f Zero     = …
  f (Succ n) = … n … f n …
  ```

  you have to program only a step

`Int` design pattern (ignoring negative values)

- task: define a function `f :: Int ⟶ S`
- step 1: solve the problem for `0`

```
f 0 = …
```

- step 2: assume that you already have the solution for `(n-1)` at hand, *extend* the intermediate solution to a solution for `n`

```
f 0  = …
f n  = … n … f (n-1) …
```

Addition

- Two naturals can be added by converting them to integers, adding, and then converting back:

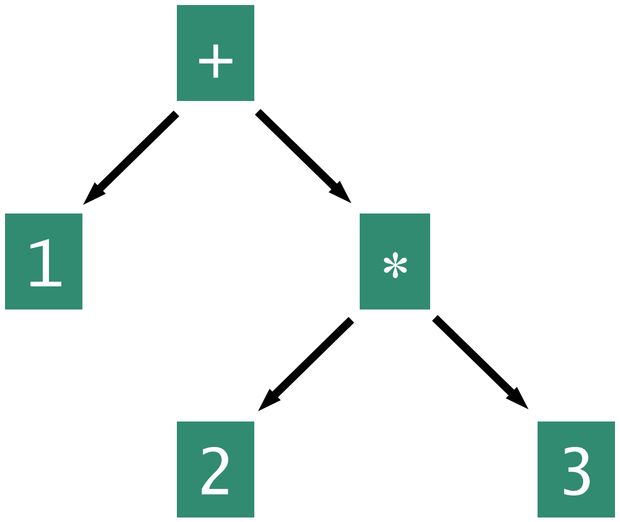
```
add :: Nat → Nat → Nat
add m n = int2nat (nat2int m + nat2int n)
```

- However, using recursion the function `add` can be defined without the need for conversions:

```
add Zero     n = n
add (Succ m) n = Succ (add m n)
```

Example 2: Arithmetic expressions

- Consider a simple form of expressions built up from integers using addition and multiplication.



Representation

- using recursion, a suitable new type to represent such expressions can be declared by:

```
data Expr = Lit Integer | Add Expr Expr | Mul Expr Expr
```

- an arithmetic expressions is either a literal, or two expressions added together, or two multiplied
- e.g. the expression on the previous slide would be represented as follows:

```
Add (Lit 1) (Mul (Lit 2) (Lit 3))
```

- constructor names may be (infix) *operators* (starting with `':'`)

```
infixl 6 :+:
infixl 7 :*:
data Expr = Lit Integer    -- a literal
          | Expr :+: Expr -- addition
          | Expr :*: Expr -- multiplication
```

Constructing expressions

- constructing expressions

```
expr1, expr2 :: Expr
expr1 = (Lit 4 :*: Lit 7) :+: (Lit 11)
expr2 = (Lit 4 :+: Lit 7) :*: (Lit 11)
```

- note the difference between *syntax*

```
>>> Lit 4 :+: Lit 7 :*: Lit 11
Lit 4 :+: Lit 7 :*: Lit 11
```

- and *semantics*

```
>>> 4 + 7 * 11
81
```

`Expr` design pattern (I)

- recursive definitions by pattern-matching

```
evaluate :: Expr ⟶ Integer
evaluate (Lit i)      = i
evaluate (e1 :+: e2) = evaluate e1 + evaluate e2
evaluate (e1 :∗: e2) = evaluate e1 ∗ evaluate e2
```

- the evaluator essentially replaces syntax (`:+:` and `:∗:`) by semantics (+ and ∗)

`Expr` design pattern (II)

- remember: every datatype comes with a pattern of definition
- task: define a function `f :: Expr ⟶ S`
- step 1: solve the problem for literals

```
f (Lit n) = … n …
```

- step 2: solve the problem for addition, assume that you already have the solution for `x` and `y` at hand, *extend* the intermediate solutions to a solution for `x :+: y`

```
f (Lit n) = … n …
f (x :+: y) = … x … y … f x … f y …
```

- step 3: do the same for `x :*: y`

```
f (Lit n) = … n …
f (x :+: y) = … x … y … f x … f y …
f (x :*: y) = … x … y … f x … f y …
```

Lists

- built-in type of lists is not special (has only special syntax)
- equivalent user-defined datatype
  ```
  data List a = Nil | Cons a (List a)
  ```
- e.g. `[1,2,3]` or `1:2:3:[]` corresponds to `Cons 1 (Cons 2 (Cons 3 Nil))`
- recursive definitions by pattern-matching
  ```
  mapList :: (a ⟶ b) ⟶ (List a ⟶ List b)
  mapList _f Nil          = Nil
  mapList f (Cons x xs) = Cons (f x) (mapList f xs)
  ```

List design pattern

- remember: every datatype comes with a pattern of definition
- task: define a function `f :: List P ⟶ S`
- step 1: solve the problem for the empty list

```
f Nil          = …
```

- step 2: solve the problem for non-empty lists;assume that you already have the solution for xs at hand; extend the intermediate solution to a solution for `Cons x xs`

```
f Nil          = …
f (Cons x xs) = … x … xs … f xs …
```

  you have to program only a step
- put on your problem-solving glasses

Binary trees

- externally-labelled binary trees (*leaf trees*)

```
data Btree a = Tip a | Bin (Btree a) (Btree a)
```

- e.g. `Bin (Tip 1) (Bin (Tip 2) (Tip 3))`
- e.g. size (number of elements)

```
size :: Btree a ⟶ Int
size (Tip _)     = 1
size (Bin t u)   = size t + size u
```

Binary search trees

- internally-labelled binary trees (*search trees*)

```
data STree a = Nil | Node (STree a) a (STree a)
```

- e.g. `Node Nil 1 (Node 3 (Node Nil 2 Nil) Nil)`
- e.g. size (number of elements)

```
size :: STree a ⟶ Int
size Nil            = 0
size (Node t _ u) = size t + 1 + size u
```

Binary search trees -- cont

- finding an element

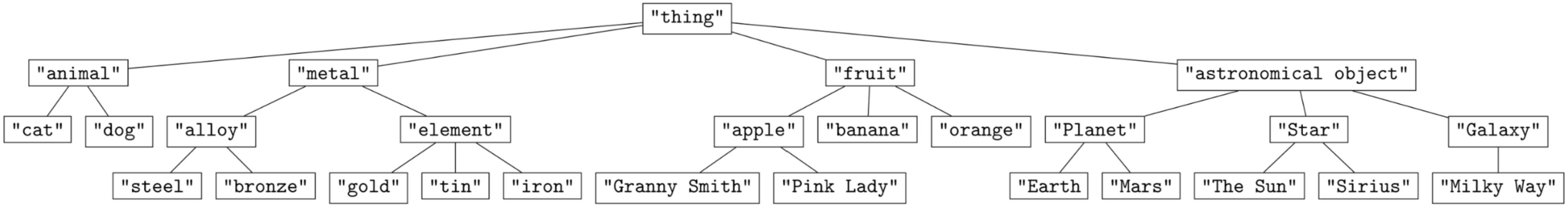
```
contains :: (Ord a) ⇒ STree a ⟶ a ⟶ Bool
contains Nil _ = False
contains (Node l v r) x
  | x < v     = contains l x
  | x > v     = contains r x
  | otherwise = True
```

- inserting an element

```
insert :: (Ord a) ⇒ STree a ⟶ a ⟶ STree a
insert Nil x = Node Nil x Nil
insert (Node l v r) x
  | x < v     = Node (insert l x) v r
  | x > v     = Node l v (insert r x)
  | otherwise = Node l v r
```

General trees

- trees with arbitrary branching (*rose trees*)
  `data Gtree a = Branch a [Gtree a]`
- e.g. `Branch 1 [Branch 2 [],Branch 3 [Branch 4 []],Branch 5 []]`



Rose tree design pattern

- remember: every datatype comes with a pattern of definition
- task: define a function `f :: Gtree P ⟶ S`
- data type consists of a single clause: only 1 step
- step 1: solve the problem for branches; assume that you already have a *list of solutions* of trs at hand; extend the intermediate solutions to a solution for `Branch e trs`

```
f (Branch e trs) = … e … trs … (map f trs) …
```

Game trees

- given available moves `mov :: Pos → [Pos]`, generate game tree

```
gametree :: (Pos → [Pos]) → Pos → Gtree Pos
gametree mov p = Branch p (map (gametree mov) (mov p))
```

Case study: longest domino chain

- Let `pieces` by a set of dominoes. Find the longest possible chain that can be made from these dominoes.
- idea:
  - step 1: create one (possibly gigantic) data structure (i.e. a rose tree) containing *all* admissible chains.
  - step 2: collect the chains of dominoes from this rose tree.

Longest chain: representation

- representation

```
type Piece = (Int,Int)
type Chain = [Piece]
```

- structure of the tree:
  - the root contains number 0 (for simplicity)
  - all pieces that have a 0 at one end will start a new branch; the number at the other end of each piece will be stored in the root of the corresponding subtree
  - this process is repeated for all subtrees

```
type DominoTree = Gtree Int
```

Longest chain: growing and harvesting

- growing a tree

```
growtree :: (Int, [Piece]) ⟶ DominoTree
growtree (m,pcs) = Branch m (map growtree lvs) where
   lvs = [ (if a == m then b else a, delete (a,b) pcs) |
                 (a,b) ← pcs, a == m || b == m ]
```

- picking the chains

```
type Path a = [a]

allpaths :: GTree a ⟶ [Path a]
allpaths (Branch el [])  = [[el]]
allpaths (Branch el trs) = [ el:p | ps ⟵ map allpaths trs, p ⟵ ps ]
```

Overloaded Functions

- a (polymorphic) function is called *overloaded* if its type contains one or more *class constraints*
- *eg* `(+) :: Num a ⇒ a → a → a`
  - for any numeric type `a`, (+) takes two values of type `a` and returns a value of type `a`
- constrained type variables can be instantiated to any types that *satisfy* the constraints
- Haskell has a number of type classes, including `Num`, `Eq`, `Ord`.
- A type class is essentially a set of types.
  - eg the prelude adds instances of `Double`, `Float`, `Int`, `Integer` to `Num`
- You can also add new instances yourself.

Class declarations

- new classes can be declared using the `class` mechanism.
- eg the class `Eq` of equality types is declared in the standard prelude as follows:

```
class Eq a where
    (==), (/=) :: a ⟶ a ⟶ Bool
    x /= y   = not (x == y)
```

- this declaration states that for a type a to be an instance of the class Eq, it must support equality and inequality operators of the specified types.
- default definition has been included the `/=`
  - declaring an instance only requires a definition for `==`

Instance declarations

- the type

```
data Blood = A | B | AB | O
```

- can be made into an equality type as follows:

```
instance Eq Blood where
    A  == A  = True
    B  == B  = True
    AB == AB = True
    O  == O  = True
    _  == _  = False
```

- Haskell can automatically generate trivial instances for some standard classes (`Eq`, `Ord`, `Show`, …); you just need to add a **`deriving`** clause to your data type.
- ie

```
data Blood = A | B | AB | O
    deriving (Eq, Show)
```

More instance declarations

- the type

```
data Gtree a = Branch a [Gtree a]
```

- can be made into an equality type as follows:

```
instance (Eq a) ⟹ Eq (Gtree a) where
    Branch e1 trs1 == Branch e2 trs2 = e1 == e2 && trs1 == trs2
```

Overloading is contagious

- what's the type of this function?

```
triple :: Num a ⇒ a → a
triple x = x + x + x
```

- what's the type of this function?

```
avg :: Fractional a ⇒ a → a → a
avg x y = (x + y) / 2
```

The art of functional programming

- model static aspects of the real world using datatypes
- model dynamic aspects using functions
- don’t shy away from introducing new types