Functional and Logic Programming

Lecture 9 — Algebraic data types

Pedro Vasconcelos
DCC/FCUP

2025

Type synonyms

We can define a new name for an existing type using a type synonym declaration.

Example from the Prelude:

```
type String = [Char]
```

Type synonyms (cont.)

Type synonym declarations are used to improve program readability.

Example: in a 2D game we might define

```
type Pos = (Int,Int)     -- x, y coordinates
type Ship = Pos
type Obstacles = [Pos]   -- list of positions
```

Then we can write

```
colides :: Ship → Obstacles → Bool
```

instead of

```
colides :: (Int,Int) → [(Int,Int)] → Bool
```

Type synonyms (cont.)

Type synonym declarations can also have parameters.

Example: association lists of key-value pairs.

```
type Assoc k v = [(k,v)]

ages :: Assoc String Int
ages = [("Anne", 21), ("Bob", 18), ("Carol", 19)]

emails :: Assoc String String
emails = [("Anne", "anne@domain.org"),
          ("Bob", "bob@example.biz"),
          ("Carol", "carol@bigtech.com")]
```

Type synonyms (cont.)

Type synonyms may be used in other synonym declarations.

```
type Pos = (Int,Int)
type Ship = Pos              -- OK
type Obstacles = [Pos]       -- OK
```

But they cannot be used recursively:

```
type Stream = (Int,Stream)   -- ERROR
```

For the later, we need to declare a new data type.

Declaring new data types

We can define new data types using a `data` declaration.

Example:

```
data Season = Winter | Spring | Summer | Autumn
```

defines a new type named `Season` with four possible values (Winter, Spring, Summer, Autumn).

Declaring new data types (cont.)

- The `data` declares enumerates all possible cases for the new type
- `Winter`, `Spring`, etc. are called the constructors of `Season`
- Constructors must be unique (i.e. you can't use the same name constructor in two distinct types or two alternatives of the same type)
- Names of types and constructors must start with an upper-case letter

Declaring new data types (cont.)

We can use this new type just like pre-defined language types.

```
next :: Season → Season
next Winter = Spring
next Spring  = Summer
next Summer  = Autumn
next Autumn  = Winter

isWarm :: Season → Bool
isWarm Summer = True
isWarm Autumn = True
isWarm _      =  False

allSeasons :: [Season]
allSeasons = iterate next Winter
-- [Winter, Spring, Summer, Autumn, Winter ...
```

Constructors with parameters

The constructors can may also take any number of parameters.

Example:

```
data Shape = Circ Float         -- radius
           | Rect Float Float   -- width, height

square :: Float → Shape
square h = Rect h h

area :: Shape → Float
area (Circ radius)         = pi*radius^2
area (Rect width height) = width*height
```

Constructors with parameters (cont.)

- Constructors may take any number of parameters
- The parameters may be of distinct types
- We use constructors both to build values and as patterns to decompose values

```
Circ 1.0 :: Shape
Rect 2.0 3.0 :: Shape

area :: Shape → Float
area (Circ radius)        = pi*radius^2
area (Rect width height) = width*height

-- alternative, using case
area shape = case shape of
  Circ radius         → pi*radius^2
  Rect width height → width*radius
```

New data types with parameters

Data declarations can also have type parameters.

Example (from the Prelude):

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

`Maybe` is useful to represent an optional value or result:

```
safeHead :: [a] → Maybe a
safeHead []     = Nothing
safeHead (x:xs) = Just x
```

`Maybe` is often used instead of a nullable value (which does not exist in Haskell).

Algebraic data types

- Data declarations allow us to build new types from simpler ones using disjunctions and conjunctions
- These behave like sums and products, hence the designation of algebraic
- Pairs are the special case of a binary product:

```
data Pair a b = MkPair a b
  -- isomorphic to (a,b)
```

- `Either` is the special case of a binary sum:

```
data Either a b = Left a | Right b
  -- defined in the Prelude
```

Typeclasses

- By default, new types do not implement the type classes `Eq`, `Ord` or `Show`
- We can ask for automatic definitions using a `deriving` clause in the declaration:

```
data Shape = Circ Float
           | Rect Float Float
            deriving (Eq, Show)
```

- The derived operations are implemented structurally:

```
> show (Circ 2)
"Circ 2.0"
> Rect 1 (1+1) == Rect 1 2
True
```

Recursive types

Unlike type synonyms, data declarations may be recursive.

For example, we can define the type of lists as follows:

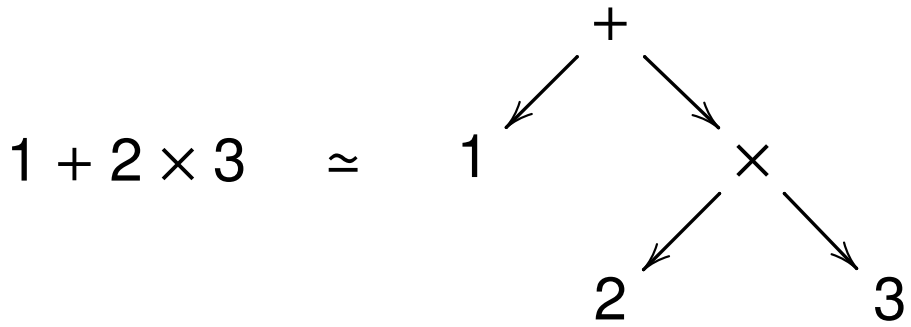
```
data List a
     = Empty                -- empty list []
     | Cons a (List a)      -- add an element (:)
```

This type is isomorphic to `[a]`, i.e. you can convert any value of this type into a value of `[a]` and vice-versa (exercise).

Syntax Trees

We can use a recursive type to represent expressions as a syntax tree: the operators are the nodes in the tree and the constants are the leaves.

Example



Syntax Trees (cont.)

The type of expressions is:

```
data Expr
     = Val Int              -- constant
     | Add Expr Expr        -- add 2 expressions
     | Mult Expr Expr       -- multiply 2 expressions
```

The previous example expression is:

```
Add (Val 1) (Mult (Val 2) (Val 3))
```

Recursion over trees

Because syntax trees are recursive types, it is natural to define functions over syntax trees using pattern matching and recursion.

Evaluation

The evaluation of an expression should produce its value.

Example:

```
eval (Add (Val 1) (Mult (Val 2) (Val 3))) == 7
```

Evaluation (cont.)

We can define `eval` using recursion.

```
eval :: Expr → Int
eval (Val n) = n
eval (Add e1 e2) = eval e1 + eval e2
eval (Mult e1 e2) = eval e1 * eval e2
```

Try this using Haskelite:

```
eval (Add (Val 1) (Mult (Val 2) (Val 3)))
```

Pretty-printing

Pretty-printing transforms a syntax tree into a text string suitable for users.

Example:

```
pretty (Add (Val 1) (Mult (Val 2) (Val 3)))
== "(1+(2*3))"
```

(Note that this is *not* what `show` does.)

Pretty-printing (cont.)

Again we can define `pretty` recursively.

```
pretty :: Expr → String
pretty (Val n)
    | n>=0       = show n
    | otherwise = "(" ++ show n ++ ")"
pretty (Add e1 e2)
  = "(" ++ pretty e1 ++ "+" ++ pretty e2 ++ ")"
pretty (Mult e1 e2)
  = "(" ++ pretty e1 ++ "*" ++ pretty e2 ++ ")"
```

The above definition may introduce redundant parenthesis; removing some of them is left as an exercise.

Parsing

Parsing transforms unstructured text into a syntax tree.

Example:

```
parse "1+2*3"
== (Add (Val 1) (Mult (Val 2) (Val 3)))
```

Parsing (cont.)

- Unlike pretty-printing, parsing is inherently a partial function: not all text strings are valid syntax trees
- Example: `"1++2*("` does not correspond a valid expression
- This means we have to allow for the possibility of failure
- We will dedicate a future lecture to parsing using combinators

Newtype declarations

- Newtype declarations allow us to define a new type that is isomorphic to another existing type
- A newtype must have a single constructor and single field, e.g.:

```
newtype UserID = UID Int
```

- Unlike a type synonym, the newtype is distinct from the old type
- Unlike a data declaration, the newtype has the same runtime representation as the old type
- Unlike a data declaration, the newtype constructor is strict (i.e. not lazy); `UID undefined = undefined`

Record syntax

- We can use a data type with a single constructor to define a custom product type
- We can also use record syntax to define names for each field
- Example: type for user contact info

```
data Contact = Contact
               { name :: String,
               , email :: String
               , address :: String
               }
```

(NB: `Contact` is both a type name and a constructor name.)

Record syntax (cont.)

We can use the field names:

- to create a contact;
- as functions to get the field values;
- to create a new record.

```
>>> let p = Contact { name="Pedro"
                    , address="My home"
                    , email="my@email.com" }
>>> name p
"Pedro"
>>> address p
"My home"
>>> p { email= "new@email.com" }
Contact { name="Pedro", address="My home",
email="new@email.com" }
```