

[CSE301 / Lecture 1]
First-order data types and pattern-matching

Noam Zeilberger

Ecole Polytechnique

4 September 2024

What is a data type?¹

A type defined by a (finite) collection of **constructors**, each of which can take any number of arguments of different types.

Since the values of a data type have a limited set of possible patterns, functions can be defined by **pattern-matching**.

This lecture: lots of examples!

¹Such types are sometimes called *algebraic data types*, since they obey laws similar to the algebraic laws for sums and products, as we will see.

First example: the booleans

Defined in the Haskell Prelude as follows:

```
data Bool = False | True
```

This definition says that *Bool* is a data type with two constructors taking no arguments:

```
False :: Bool
```

```
True  :: Bool
```

Moreover, these are the *only* ways to build a value of type *Bool*.

Example: negation

Define negation by pattern-matching:

$$\text{not} :: \text{Bool} \rightarrow \text{Bool}$$
$$\text{not False} = \text{True}$$
$$\text{not True} = \text{False}$$

An example “theorem” we can prove using this definition is that *not* is an involution, i.e., that $\text{not} (\text{not } x) = x$ for all $x :: \text{Bool}$.

Indeed, it suffices to consider $x = \text{False}$ and $x = \text{True}$.

By definition, we have:

$$\text{not} (\text{not False}) = \text{not True} = \text{False}$$
$$\text{not} (\text{not True}) = \text{not False} = \text{True}$$

QED!

Example: conjunction

Definition #1:

both :: Bool → Bool → Bool

both False False = False

both False True = False

both True False = False

both True True = True

Definition #2 (version in Prelude):

(&&) :: Bool → Bool → Bool

False && _ = False

True && b = b

Example: conjunction

Observing the difference between v1 and v2...

```
$ ghci DataCode
```

```
GHCi, version 9.10.1: https://www.haskell.org/ghc/ :? for
```

```
[1 of 1] Compiling Main                ( DataCode.hs, interpre
```

```
Ok, one module loaded.
```

```
ghci> :set +s
```

```
ghci> both False (length [1..109] == 109)
```

```
False
```

```
(8.74 secs, 72,000,072,856 bytes)
```

```
ghci> False && (length [1..109] == 109)
```

```
False
```

```
(0.01 secs, 70,752 bytes)
```

Values vs. expressions

A **value** of a given data type is built using one of its constructors.

An **expression** describes a *computation* of a value.

For example, *not False* is an expression evaluating to *True*.

`(length [1..109] == 109)` is an expression that also eventually evaluates to *True*, but after a long time.

It is also possible to write expressions that never return values.

```
ghci> loop x = loop (x+1)
```

```
ghci> loop 0
```

```
    C-c C-cInterrupted.
```

Sums and products

Besides defining particular types like *Bool*, data declarations also provide a way of combining one or more types to form a new type.

Two important instances are called **sum types** and **product types**.

Sum types

Definition in Prelude:

data *Either* *a b* = *Left* *a* | *Right* *b*

Here, *Either* is called a **type constructor**.

This definition automatically introduces two (value) constructors:

Left :: *a* → *Either* *a b*

Right :: *b* → *Either* *a b*

In set-theoretic terms, the set of values of a sum type may be considered as a disjoint union $\{Left\ x \mid x :: a\} \cup \{Right\ y \mid y :: b\}$.

Definition by cases

In general, if $f :: a \rightarrow c$ and $g :: b \rightarrow c$ are two functions, then we can define a single function

$$h :: \text{Either } a \ b \rightarrow c$$
$$h \ (\text{Left } x) = f \ x$$
$$h \ (\text{Right } y) = g \ y$$

For example, an integer coercion routine:

$$\text{asInt} :: \text{Either } \text{Bool } \text{Int} \rightarrow \text{Int}$$
$$\text{asInt} \ (\text{Left } \text{False}) = 0$$
$$\text{asInt} \ (\text{Left } \text{True}) = 1$$
$$\text{asInt} \ (\text{Right } n) = n$$

Sum types \approx coproducts in category theory

A “category” is roughly a set of objects and a set of arrows between objects, which can be composed (like paths in a graph).

The “coproduct” of two objects A and B in a category is an object $A + B$ equipped with arrows $\ell : A \rightarrow A + B$ and $r : B \rightarrow A + B$, such that for any pair of arrows $f : A \rightarrow C$ and $g : B \rightarrow C$ there exists a unique $h : A + B \rightarrow C$ such that $f = h \circ \ell$ and $g = h \circ r$:

$$\begin{array}{ccccc} A & \xrightarrow{\ell} & A + B & \xleftarrow{r} & B \\ & \searrow f & \downarrow h & \swarrow g & \\ & & C & & \end{array}$$

Product types

Whereas sum types describe values that can take multiple forms, product types describe values that contain multiple components.

Haskell has built-in product types, written (a, b) where a and b are types. A value of type (a, b) is a pair (u, v) , where $u :: a$ and $v :: b$. (This kind of overloading is common in Haskell...get used to it!)

Also, there are built-in projection functions

$$\begin{aligned}fst &:: (a, b) \rightarrow a \\snd &:: (a, b) \rightarrow b\end{aligned}$$

satisfying $fst\ (u, v) = u$ and $snd\ (u, v) = v$.

Product types as a data type

But we could have also defined product types for ourselves!

data *Both* *a b = Pair a b*

Define the projections by pattern-matching:

projLeft :: *Both a b* → *a*

projLeft (*Pair u v*) = *u*

projRight :: *Both a b* → *b*

projRight (*Pair u v*) = *v*

The two versions of product types are *isomorphic*.

Type isomorphism $A \cong B$

Informally: “ A and B are essentially equivalent”.

A bit more precisely: “we can convert values of type A into values of type B , and vice versa, in a reversible way.”

Formally: there are a pair of functions $f :: A \rightarrow B$ and $g :: B \rightarrow A$ such that $g (f x) = x$ for all $x :: A$, and $f (g y) = y$ for all $y :: B$.

$$A \begin{array}{c} \xrightarrow{f} \\ \xleftarrow{g} \end{array} B$$

Distributivity of products over sums

$$\text{Both } a \text{ (Either } b \text{ } c) \cong \text{Either (Both } a \text{ } b) \text{ (Both } a \text{ } c)$$

$$f :: \text{Both } a \text{ (Either } b \text{ } c) \rightarrow \text{Either (Both } a \text{ } b) \text{ (Both } a \text{ } c)$$

$$f \text{ (Pair } x \text{ (Left } y)) = \text{Left (Pair } x \text{ } y)$$

$$f \text{ (Pair } x \text{ (Right } y)) = \text{Right (Pair } x \text{ } y)$$

$$g :: \text{Either (Both } a \text{ } b) \text{ (Both } a \text{ } c) \rightarrow \text{Both } a \text{ (Either } b \text{ } c)$$

$$g \text{ (Left (Pair } x \text{ } y)) = \text{Pair } x \text{ (Left } y)$$

$$g \text{ (Right (Pair } x \text{ } y)) = \text{Pair } x \text{ (Right } y)$$

Corresponds to the algebraic law $a(b + c) = ab + ac$!

Nullary sums and products

Sum types and product types also come in nullary version.

Nullary product is called the **unit type**, written `()` in Haskell.

But we can also define it as a data type:

```
data Unit = U
```

Nullary sum is called the **zero type** (or void type).

We can define it like so:

```
data Zero
```


Some more valid type isomorphisms

$$\textit{Either } a \text{ (} \textit{Either } b \text{ } c \text{)} \cong \textit{Either } (\textit{Either } a \text{ } b) \text{ } c \quad (1)$$

$$\textit{Either } a \text{ } b \cong \textit{Either } b \text{ } a \quad (2)$$

$$\textit{Both } a \text{ (} \textit{Both } b \text{ } c \text{)} \cong \textit{Both } (\textit{Both } a \text{ } b) \text{ } c \quad (3)$$

$$\textit{Both } a \text{ } b \cong \textit{Both } b \text{ } a \quad (4)$$

$$\textit{Both Unit } a \cong a \quad (5)$$

$$\textit{Either Zero } a \cong a \quad (6)$$

(Exercise!)

(What are the corresponding algebraic laws?)

Lists as a data type

Lists are ubiquitous in FP (thank you John McCarthy!).

Modulo syntax, **list types** are defined like so:

$$\mathbf{data} \ [a] = [] \mid a : [a]$$

(Though this is unfortunately not valid Haskell syntax.)

Note this is a *recursive* definition!

Lists as a data type

If we want, we can define our own isomorphic version:

data *List a* = *Nil* | *Cons a (List a)*

introducing the following constructors:

Nil :: *List a*

Cons :: *a* → *List a* → *List a*

Easy exercise: $[a] \cong \text{List } a$.

Example: concatenation

Concatenation defined by pattern-matching and recursion:

$$\begin{aligned}(\++) &:: [a] \rightarrow [a] \rightarrow [a] \\ [] \++ \textit{ys} &= \textit{ys} \\ (x : \textit{xs}) \++ \textit{ys} &= x : (\textit{xs} \++ \textit{ys})\end{aligned}$$

Although the definition of $(++)$ is circular, it is well-defined since the first argument always gets smaller.

Logic interlude: the principle of structural induction

Let $P(xs)$ be a property of lists. Suppose that:

1. $P([])$ holds
2. for any element x and list xs , $P(xs)$ implies $P(x : xs)$

Then $P(xs)$ holds for all lists xs .

...Or to be a bit more precise, if

2. for any element $x :: a$ and list $xs :: [a]$, $P(xs)$ implies $P(x : xs)$
- then $P(xs)$ holds for all lists $xs :: [a]$.

Logic interlude: the principle of structural induction

The principle of structural induction is one way to “justify” the definition of $(+)$, taking $P(xs :: [a])$ to be

“for any $ys :: [a]$, there is a $zs :: [a]$ such that $xs ++ ys = zs$ ”.

We can also use structural induction to prove other properties of recursive functions. (See exercises in lecture notes.)

Maybe types

Sometimes we want to run a computation that might fail, but tells us when it fails. In Haskell this is achieved with **maybe types**.²

data *Maybe a* = *Nothing* | *Just a*

Observe that *Maybe a* \cong *Either () a*.

But maybe types are so useful they deserve their own syntax!

²Also known as option types in other languages, such as OCaml.

Example: *lookup*

“try to find the value paired with a key in a list of pairs”

$lookup :: Eq\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow Maybe\ b$

$lookup\ k\ [] = Nothing$

$lookup\ k\ ((k', v) : kvs)$

| $k \equiv k' = Just\ v$

| $otherwise = lookup\ k\ kvs$

Example: *elemIndex*

“try to find the index of an element within a list”

elemIndex :: *Eq a* \Rightarrow *a* \rightarrow [*a*] \rightarrow *Maybe Int*

elemIndex *x* [] = *Nothing*

elemIndex *x* (*x'* : *xs*)

| *x* \equiv *x'* = *Just* 0

| *otherwise* = **case** *elemIndex* *x* *xs* **of**
 Nothing \rightarrow *Nothing*
 Just i \rightarrow *Just* (*i* + 1)

Algebraic interlude

The following type isomorphism is valid:

$$\textit{Both} (\textit{Maybe } a) (\textit{Maybe } b) \cong \textit{Maybe} (\textit{Either} (\textit{Either } a \ b) (\textit{Both } a \ b))$$

What is the algebraic analogue?

Introducing accumulators

There may be different ways of writing the same function that differ wildly in terms of resource usage – and understanding these costs is an important part of functional programming.

Example #1: list-reversal (naive version)

An easy recursive definition:

$$\text{reverse} :: [a] \rightarrow [a]$$
$$\text{reverse } [] = []$$
$$\text{reverse } (x : xs) = \text{reverse } xs \mathbin{++} [x]$$

Functionally correct, but $\Theta(n^2)$ time!

Example #1: list-reversal using a stack

There is a simple imperative algorithm for reversing a list in $\Theta(n)$ time, using an auxiliary stack:

1. Initialize the stack to be empty.
 2. While the input list is non-empty, push its head onto the stack, and keep processing its tail.
 3. Once the input list is empty, return the contents of the stack.
- We can turn this imperative solution into a functional program!

Example #1: list-reversal using an accumulator

Define a helper function:

$$\begin{aligned} \text{revacc} &:: [a] \rightarrow [a] \rightarrow [a] \\ \text{revacc } [] \text{ } ys &= ys \\ \text{revacc } (x : xs) \text{ } ys &= \text{revacc } xs \text{ } (x : ys) \end{aligned}$$

The extra parameter *ys* (the “accumulator”) simulates the stack.

The two clauses of the definition correspond to steps (3) and (2) of the algorithm, respectively.

Finally, step (1) is implemented by (re-)defining *reverse*:

$$\text{reverse } xs = \text{revacc } xs []$$

Example #1: list-reversal using an accumulator

It's fun to watch this version in action...

```
reverse [1, 2, 3, 4]
= revacc [1, 2, 3, 4] []
= revacc [2, 3, 4] [1]
= revacc [3, 4] [2, 1]
= revacc [4] [3, 2, 1]
= revacc [] [4, 3, 2, 1]
= [4, 3, 2, 1]
```

Example #2: Fibonnaci numbers (horrible version)

Can translate the standard recurrence to a recursive definition:

fib :: Integer → Integer

fib n

| $n \equiv 0 = 0$

| $n \equiv 1 = 1$

| $n \geq 2 = \text{fib } (n - 1) + \text{fib } (n - 2)$

Mathematically correct, but uses exponential time and space!

Example #2: Fibonnaci numbers (horrible version)

```
*Main> :set +s
*Main> fib 10
55
(0.02 secs, 123,512 bytes)
*Main> fib 20
6765
(0.08 secs, 6,423,944 bytes)
*Main> fib 30
832040
(2.38 secs, 781,578,344 bytes)
*Main> fib 31
1346269
(3.58 secs, 1,264,577,008 bytes)
*Main> fib 32
2178309
(6.05 secs, 2,046,084,072 bytes)
```

Example #2: Fibonacci numbers (fast imperative version)

There is a much more efficient imperative algorithm for computing F_n in linear time, using a pair of auxiliary variables a and b :

- Initialize $a \leftarrow 0$ and $b \leftarrow 1$.
- While $n > 0$, simultaneously update $(a, b) \leftarrow (b, a + b)$, and decrement n .
- Once $n = 0$, return the value of a .

Again, this imperative solution can be transformed almost mechanically into a purely functional one.

Example #2: Fibonnaci numbers (fast functional version)

Define a helper function with two accumulators, and then redefine *fib* as an appropriate call to the helper function:

$$\begin{aligned} & \textit{fibacc} \ n \ a \ b \\ & \quad | \ n \equiv 0 = a \\ & \quad | \ n \geqslant 1 = \textit{fibacc} \ (n - 1) \ b \ (a + b) \\ & \textit{fib} \ n = \textit{fibacc} \ n \ 0 \ 1 \end{aligned}$$

This version is linear time, as it should be!

Example #2: Fibonnaci numbers (fast functional version)

```
*Main> fib n = fibacc n 0 1
```

```
*Main> fib 32
```

```
2178309
```

```
(0.00 secs, 82,288 bytes)
```

```
*Main> fib 100
```

```
354224848179261915075
```

```
(0.01 secs, 114,400 bytes)
```

```
*Main> fib 1000
```

```
4346655768693745643568852767504062580256466051737178040248  
!7520968962323987332247116164299644090653318793829896964992
```

```
(0.01 secs, 637,736 bytes)
```

Accumulators, a bit more conceptually

To solve a particular problem, oftentimes it can be helpful to try to solve a *more general problem*.

Here, *revacc* actually solves the following more general problem: given two lists, compute the reversal of the first list concatenated with the second list, i.e., $\text{revacc } xs \ ys = \text{reverse } xs \ ++ \ ys$.

Similarly, *fibacc* *n a b* computes the *n*th entry of a *generalized Fibonacci sequence*, defined by the same recurrence but with initial values *a* and *b*. (E.g., *fibacc* *n* 2 1 is the *n*th “Lucas number”.)

Trees

Trees give another important example of a data type.

There are many different kinds of “trees”. For concreteness, we'll consider binary trees with labelled nodes:

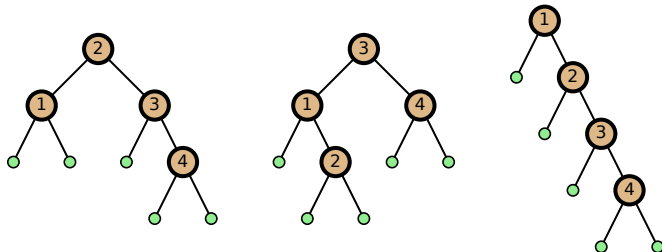
data *BinTree* *a* = *Leaf* | *Node* *a* (*BinTree* *a*) (*BinTree* *a*)

Again to be clear, this means that:

Leaf :: *BinTree* *a*

Node :: *a* → *BinTree* *a* → *BinTree* *a* → *BinTree* *a*

Some example trees



are represented as the following values:

$t1, t2, t3 :: \text{BinTree Int}$

$t1 = \text{Node } 2 (\text{Node } 1 \text{ Leaf Leaf}) (\text{Node } 3 \text{ Leaf } (\text{Node } 4 \text{ Leaf Leaf}))$

$t2 = \text{Node } 3 (\text{Node } 1 \text{ Leaf } (\text{Node } 2 \text{ Leaf Leaf})) (\text{Node } 4 \text{ Leaf Leaf})$

$t3 = \text{Node } 1 \text{ Leaf } (\text{Node } 2 \text{ Leaf } (\text{Node } 3 \text{ Leaf } (\text{Node } 4 \text{ Leaf Leaf})))$

Example: computing statistics of trees

nodes :: *BinTree a* → *Int*

nodes Leaf = 0

nodes (Node _ tL tR) = 1 + *nodes tL* + *nodes tR*

leaves :: *BinTree a* → *Int*

leaves Leaf = 1

leaves (Node _ tL tR) = *leaves tL* + *leaves tR*

height :: *BinTree a* → *Int*

height Leaf = 0

height (Node _ tL tR) = 1 + max (*height tL*) (*height tR*)

Example: reflecting a tree

mirror :: *BinTree a* → *BinTree a*

mirror Leaf = *Leaf*

mirror (Node x tL tR) = *Node x (mirror tR) (mirror tL)*

Structural induction over binary trees

Let $P(t :: \text{BinTree } a)$ be a property of binary trees. Suppose that:

1. $P(\text{Leaf})$ holds
2. for any element $x :: a$ and pair of trees $tL, tR :: \text{BinTree } a$,
 $P(tL)$ and $P(tR)$ implies $P(\text{Node } x \ tL \ tR)$

Then $P(t)$ holds for all binary trees $t :: \text{BinTree } a$.

Exercise: $\text{height}(\text{mirror } t) = \text{height } t$ for all $t :: \text{BinTree } a$.