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

Noam Zeilberger

Ecole Polytechnique

6 September 2023

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!

[1] Such 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:

$$\textbf{data}\ Bool = False \mid True$$

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:

$$
\begin{aligned}
&\mathit{not} :: \mathit{Bool} \to \mathit{Bool}\\
&\mathit{not}\ \mathit{False} = \mathit{True}\\
&\mathit{not}\ \mathit{True} = \mathit{False}
\end{aligned}
$$

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

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

By definition, we have:

$$
\begin{aligned}
&\mathit{not}\ (\mathit{not}\ \mathit{False}) = \mathit{not}\ \mathit{True} = \mathit{False}\\
&\mathit{not}\ (\mathit{not}\ \mathit{True}) = \mathit{not}\ \mathit{False} = \mathit{True}
\end{aligned}
$$

QED!

Example: conjunction

Definition #1:

$$\begin{aligned}&both :: Bool \to Bool \to Bool\\&both\ False\ False = False\\&both\ False\ True = False\\&both\ True\ False = False\\&both\ True\ True\ = True\end{aligned}$$

Definition #2 (version in Prelude):

$$\begin{aligned}&(\&\&) :: Bool \to Bool \to Bool\\&False\ \&\&\ \_ = False\\&True\ \&\&\ b = b\end{aligned}$$

Example: conjunction

Observing the difference between v1 and v2...

```
$ ghci DataCode
*Main> let loop = loop
*Main> both False loop
^CInterrupted.
*Main> False && loop
False
```

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*.

OTOH, *loop* describes a computation that never terminates.

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:

$$\textbf{data}\ Either\ a\ b = Left\ a \mid Right\ b$$

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

This definition automatically introduces two (value) constructors:

$$Left :: a \rightarrow Either\ a\ b$$
$$Right :: b \rightarrow Either\ a\ b$$

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 $f :: a \to c$ and $g :: b \to c$ are two functions, then we can define a single function

$$
\begin{aligned}
&h :: Either\ a\ b \to c \\
&h\ (Left\ x) = f\ x \\
&h\ (Right\ y) = g\ y
\end{aligned}
$$

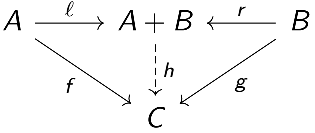
For example, an integer coercion routine:

$$
\begin{aligned}
&asInt :: Either\ Bool\ Int \to Int \\
&asInt\ (Left\ False) = 0 \\
&asInt\ (Left\ True) = 1 \\
&asInt\ (Right\ n) \quad = n
\end{aligned}
$$

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 $\ell : A \to A + B$ and $r : B \to A + B$, such that for for any pair of arrows $f : A \to C$ and $g : B \to C$ there exists a unique $h : A + B \to C$ such that $f = h \circ \ell$ and $g = h \circ r$:



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

$$fst :: (a, b) \to a$$
$$snd :: (a, b) \to b$$

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!

$$\textbf{data}\ \mathit{Both}\ a\ b = \mathit{Pair}\ a\ b$$

Define the projections by pattern-matching:

$$
\begin{aligned}
&\mathit{projLeft} :: \mathit{Both}\ a\ b \to a\\
&\mathit{projLeft}\ (\mathit{Pair}\ u\ v) = u\\
&\mathit{projRight} :: \mathit{Both}\ a\ b \to b\\
&\mathit{projRight}\ (\mathit{Pair}\ u\ v) = v
\end{aligned}
$$

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 $f :: A \to B$ and $g :: B \to A$ such that $g\ (f\ x) = x$ for all $x :: A$, and $f\ (g\ y) = y$ for all $y :: B$.

$$A \underset{g}{\overset{f}{\rightleftarrows}} B$$

Distributivity of products over sums

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

$$\begin{aligned}
&f :: \mathit{Both}\ a\ (\mathit{Either}\ b\ c) \rightarrow \mathit{Either}\ (\mathit{Both}\ a\ b)\ (\mathit{Both}\ a\ c)\\
&f\ (\mathit{Pair}\ x\ (\mathit{Left}\ y)) = \mathit{Left}\ (\mathit{Pair}\ x\ y)\\
&f\ (\mathit{Pair}\ x\ (\mathit{Right}\ y)) = \mathit{Right}\ (\mathit{Pair}\ x\ y)
\end{aligned}$$

$$\begin{aligned}
&g :: \mathit{Either}\ (\mathit{Both}\ a\ b)\ (\mathit{Both}\ a\ c) \rightarrow \mathit{Both}\ a\ (\mathit{Either}\ b\ c)\\
&g\ (\mathit{Left}\ (\mathit{Pair}\ x\ y)) = \mathit{Pair}\ x\ (\mathit{Left}\ y)\\
&g\ (\mathit{Right}\ (\mathit{Pair}\ x\ y)) = \mathit{Pair}\ x\ (\mathit{Right}\ y)
\end{aligned}$$

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:

$$\textbf{data}\ \mathit{Unit} = U$$

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

We can define it like so:

$$\textbf{data}\ \mathit{Zero}$$

Some more valid type isomorphisms

$$\mathit{Either}\ a\ (\mathit{Either}\ b\ c) \cong \mathit{Either}\ (\mathit{Either}\ a\ b)\ c \tag{1}$$

$$\mathit{Either}\ a\ b \cong \mathit{Either}\ b\ a \tag{2}$$

$$\mathit{Both}\ a\ (\mathit{Both}\ b\ c) \cong \mathit{Both}\ (\mathit{Both}\ a\ b)\ c \tag{3}$$

$$\mathit{Both}\ a\ b \cong \mathit{Both}\ b\ a \tag{4}$$

$$\mathit{Both}\ \mathit{Unit}\ a \cong a \tag{5}$$

$$\mathit{Either}\ \mathit{Zero}\ a \cong a \tag{6}$$

(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:

$$\textbf{data}\ [a] = [\,] \mid a : [a]$$

(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:

$$\textbf{data}\ \mathit{List}\ a = \mathit{Nil} \mid \mathit{Cons}\ a\ (\mathit{List}\ a)$$

introducing the following constructors:

$$\begin{aligned} &\mathit{Nil} :: \mathit{List}\ a \\ &\mathit{Cons} :: a \to \mathit{List}\ a \to \mathit{List}\ a \end{aligned}$$

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

Example: concatenation

Concatenation defined by pattern-matching and recursion:

$$
\begin{aligned}
&(+\!\!+) :: [a] \to [a] \to [a] \\
&[\,] +\!\!+ ys = ys \\
&(x : xs) +\!\!+ ys = x : (xs +\!\!+ ys)
\end{aligned}
$$

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]

$$\textbf{data}\ \mathit{Maybe}\ a = \mathit{Nothing} \mid \mathit{Just}\ a$$

Observe that $\mathit{Maybe}\ a \cong \mathit{Either}\ ()\ a$.

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

[2] Also 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"

$$
\begin{aligned}
&\mathit{lookup} :: \mathit{Eq}\ a \Rightarrow a \rightarrow [(a, b)] \rightarrow \mathit{Maybe}\ b \\
&\mathit{lookup}\ k\ [\,] \quad\ \ = \mathit{Nothing} \\
&\mathit{lookup}\ k\ ((k', v) : \mathit{kvs}) \\
&\quad\ \mid k \equiv k' \quad\ \ = \mathit{Just}\ v \\
&\quad\ \mid \mathit{otherwise} = \mathit{lookup}\ k\ \mathit{kvs}
\end{aligned}
$$

Example: *elemIndex*

"try to find the index of an element within a list"

$$
\begin{array}{l}
\mathit{elemIndex} :: \mathit{Eq}\ a \Rightarrow a \rightarrow [a] \rightarrow \mathit{Maybe}\ \mathit{Int} \\
\mathit{elemIndex}\ x\ [\,] = \mathit{Nothing} \\
\mathit{elemIndex}\ x\ (x' : xs) \\
\quad \mid x \equiv x' \qquad = \mathit{Just}\ 0 \\
\quad \mid \mathit{otherwise}\ = \mathbf{case}\ \mathit{elemIndex}\ x\ xs\ \mathbf{of} \\
\qquad\qquad\qquad\quad \mathit{Nothing} \rightarrow \mathit{Nothing} \\
\qquad\qquad\qquad\quad \mathit{Just}\ i \rightarrow \mathit{Just}\ (i + 1)
\end{array}
$$

Algebraic interlude

The following type isomorphism is valid:

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

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:

$$reverse :: [a] \to [a]$$
$$reverse\ [\,] = [\,]$$
$$reverse\ (x : xs) = reverse\ xs +\!\!+ [x]$$

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:

$$
\begin{aligned}
&\mathit{revacc} :: [a] \to [a] \to [a] \\
&\mathit{revacc}\ [\,]\ \mathit{ys} = \mathit{ys} \\
&\mathit{revacc}\ (x : \mathit{xs})\ \mathit{ys} = \mathit{revacc}\ \mathit{xs}\ (x : \mathit{ys})
\end{aligned}
$$

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*:

$$\mathit{reverse}\ \mathit{xs} = \mathit{revacc}\ \mathit{xs}\ [\,]$$

Example #1: list-reversal using an accumulator

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

$$\begin{aligned}
&\textit{reverse}\ [1,2,3,4] \\
&= \textit{revacc}\ [1,2,3,4]\ [\,] \\
&= \textit{revacc}\ [2,3,4]\ [1] \\
&= \textit{revacc}\ [3,4]\ [2,1] \\
&= \textit{revacc}\ [4]\ [3,2,1] \\
&= \textit{revacc}\ [\,]\ [4,3,2,1] \\
&= [4,3,2,1]
\end{aligned}$$

Example #2: Fibonnaci numbers (horrible version)

Can translate the standard recurrence to a recursive definition:

$$\begin{aligned}
&\mathit{fib} :: \mathit{Integer} \rightarrow \mathit{Integer} \\
&\mathit{fib}\ n \\
&\quad \mid n \equiv 0 = 0 \\
&\quad \mid n \equiv 1 = 1 \\
&\quad \mid n \geqslant 2 = \mathit{fib}\ (n - 1) + \mathit{fib}\ (n - 2)
\end{aligned}$$

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: Fibonnaci 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:

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

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
!752096896232398733224711616429964409065331879382989696499
(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., $\mathit{revacc}\ \mathit{xs}\ \mathit{ys} = \mathit{reverse}\ \mathit{xs} +\!\!+\ \mathit{ys}$.

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., $\mathit{fibacc}\ n\ 2\ 1$ is the *n*th "Lucas number".)

Trees

Trees give another important example of a data type.

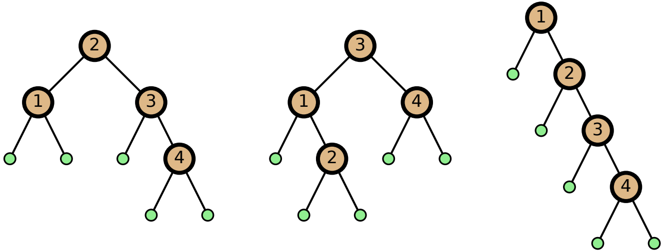
There are many different kinds of "trees". For concreteness, we'll consider binary trees with labelled nodes:

$$\textbf{data}\ BinTree\ a = Leaf \mid Node\ a\ (BinTree\ a)\ (BinTree\ a)$$

Again to be clear, this means that:

$$Leaf :: BinTree\ a$$
$$Node :: a \rightarrow BinTree\ a \rightarrow BinTree\ a \rightarrow BinTree\ a$$

Some example trees



are represented as the following values:

$$
\begin{aligned}
&\mathit{t1}, \mathit{t2}, \mathit{t3} :: \mathit{BinTree\ Int} \\
&\mathit{t1} = \mathit{Node}\ 2\ (\mathit{Node}\ 1\ \mathit{Leaf}\ \mathit{Leaf})\ (\mathit{Node}\ 3\ \mathit{Leaf}\ (\mathit{Node}\ 4\ \mathit{Leaf}\ \mathit{Leaf})) \\
&\mathit{t2} = \mathit{Node}\ 3\ (\mathit{Node}\ 1\ \mathit{Leaf}\ (\mathit{Node}\ 2\ \mathit{Leaf}\ \mathit{Leaf}))\ (\mathit{Node}\ 4\ \mathit{Leaf}\ \mathit{Leaf}) \\
&\mathit{t3} = \mathit{Node}\ 1\ \mathit{Leaf}\ (\mathit{Node}\ 2\ \mathit{Leaf}\ (\mathit{Node}\ 3\ \mathit{Leaf}\ (\mathit{Node}\ 4\ \mathit{Leaf}\ \mathit{Leaf})))
\end{aligned}
$$

Example: computing statistics of trees

$$
\begin{aligned}
&\mathit{nodes} :: \mathit{BinTree}\ a \to \mathit{Int}\\
&\mathit{nodes}\ \mathit{Leaf} = 0\\
&\mathit{nodes}\ (\mathit{Node}\ \_\ \mathit{tL}\ \mathit{tR}) = 1 + \mathit{nodes}\ \mathit{tL} + \mathit{nodes}\ \mathit{tR}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{leaves} :: \mathit{BinTree}\ a \to \mathit{Int}\\
&\mathit{leaves}\ \mathit{Leaf} = 1\\
&\mathit{leaves}\ (\mathit{Node}\ \_\ \mathit{tL}\ \mathit{tR}) = \mathit{leaves}\ \mathit{tL} + \mathit{leaves}\ \mathit{tR}
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{height} :: \mathit{BinTree}\ a \to \mathit{Int}\\
&\mathit{height}\ \mathit{Leaf} = 0\\
&\mathit{height}\ (\mathit{Node}\ \_\ \mathit{tL}\ \mathit{tR}) = 1 + \mathit{max}\ (\mathit{height}\ \mathit{tL})\ (\mathit{height}\ \mathit{tR})
\end{aligned}
$$

Example: reflecting a tree

$$
\begin{aligned}
&\mathit{mirror} :: \mathit{BinTree}\ a \rightarrow \mathit{BinTree}\ a \\
&\mathit{mirror}\ \mathit{Leaf} = \mathit{Leaf} \\
&\mathit{mirror}\ (\mathit{Node}\ x\ \mathit{tL}\ \mathit{tR}) = \mathit{Node}\ x\ (\mathit{mirror}\ \mathit{tR})\ (\mathit{mirror}\ \mathit{tL})
\end{aligned}
$$

Structural induction over binary trees

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

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

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

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