The ABCs of ADTs

Algebraic Data Types

Justin Lubin

January 18, 2017

Asynchronous Anonymous @ UChicago

Overview

A. What is an ***algebraic data type***?

B. Why are algebraic data types ***useful***?

C. Why are algebraic data types ***cool***?

What is an ***algebraic data type***?

What is a type?

Data = ones & zeros

Types = **intent**

A ***type*** is a ***label*** on a piece of data that describes a ***set of values*** that it may take on.

Some basic types

- **Boolean**

  `True, False`

- **Character**

  `‘A’, ‘B’, ..., ‘Z’, ‘a’, ‘b’, ..., ‘z’`

- **Integer**

  `..., -3, -2, -1, 0, 1, 2, 3, ...`

Untyped vs. typed programs

```
fn add(x, y) {
  return x + y;
}
```

```
>>> add(3, 2)
5

>>> add(true, false)
```

Runtime error!

```
fn int add(int x, int y) {
  return x + y;
}
```

```
>>> add(3, 2)
5

>>> add(true, false)
```

Compile-time error!

Untyped vs. typed programs

```
fn and(x, y) {
  return x * y;
}
```

```
fn bool and(bool x, bool y) {
  return x * y;
}
```

Compile-time error!

```
>>> add(true, false)
```

Runtime error!

What is an algebraic data type (ADT)?

Algebraic data type = a type defined by **type constructors** in terms of *data constructors*.

```
type Boolean = False | True

type Character = ‘A’ | ‘B’ | ... | ‘Z’ | ‘a’ | ‘b’ | ... | ‘z’

type Integer = ... | -3 | -2 | -1 | 0 | 1 | 2 | 3 | ...


type HttpError = NotFound | InternalServerError | ...
```

Fancier type constructors

```
type Shape
  = Rectangle Float Float Float Float
  | Circle Float Float Float

area : Shape -> Float
area shape =
  case shape of
    Rectangle x y width height ->
      width * height

    Circle x y radius ->
      π * radius * radius
```

```
>>> r = Rectangle 0 0 10 5
r : Shape
>>> area r
50.0


>>> c = Circle 2 4 10
c : Shape
>>> area c
314.159265
```

Quick aside: type aliases

```
type Shape
  = Rectangle Float Float Float Float
  | Circle Float Float Float
```

Quick aside: type aliases

```
type alias Position = Float
type alias Width = Float
type alias Height = Float
type alias Radius = Float

type Shape
  = Rectangle Position Position Width Height
  | Circle Position Position Radius
```

Quick aside: type aliases

```
type alias Position = Double
type alias Width = Double
type alias Height = Double
type alias Radius = Double

type Shape
  = Rectangle Position Position Width Height
  | Circle Position Position Radius
```

Case study: `find`

Case study: `find` (ideal situation)

```
>>> x = find 19 [10, 13, 19, 44]
2

>>> y = find 10 [10, 13, 19, 44]
0

>>> absoluteValue (y - x)
2
```

Case study: `find` (problematic situation)

```
>>> x = find 19 [10, 13, 19, 44]
2

>>> y = find 876 [10, 13, 19, 44]
-1

>>> “The distance is:” ++ toString (absoluteValue (y - x))
The distance is: 3
```

Case study: `find` (problematic situation)

```
>>> x = find 19 [10, 13, 19, 44]
2

>>> y = find 876 [10, 13, 19, 44]
null

>>> “The distance is:” ++ toString (absoluteValue (y - x))
```

Runtime error! *Null pointer exception (or something)!*

Case study: `find` (with algebraic data types)

```
>>> x = find 19 [10, 13, 19, 44]
Just 2

>>> y = find 876 [10, 13, 19, 44]
Nothing

>>> “The distance is:” ++ toString (absoluteValue (y - x))
Compile-time error! Can’t perform subtraction on type Maybe Int.
```

Case study: `find` (with algebraic data types)

```
>>> x = find 19 [10, 13, 19, 44]
Just 2

>>> y = find 876 [10, 13, 19, 44]
Nothing

>>> case (x, y) of
      (Just index1, Just index2) ->
        “The distance is:” ++ toString (absoluteValue (index2 - index1))
      _ ->
        “I can’t find the distance between these...”
```

Maybe type constructor

```
type Maybe a
  = Just a
  | Nothing
```

Maybe type constructor

```
type Maybe a
  = Just a
  | Nothing
```

Maybe type constructor

```
type Maybe stuff
  = Just stuff
  | Nothing
```

Maybe is *not* a type! (Maybe a is)

- **Maybe** Bool

  Just True, *Just* False, ..., *Nothing*

- **Maybe** Character

  Just ‘A’, *Just* ‘q’, ..., *Nothing*

- **Maybe** Integer

  Just (-1), *Just* 14, *Just* 0, ..., *Nothing*

- **Maybe**

  ???

Maybe is *not* a type! (Maybe <u>a</u> is)

- `Maybe Bool`

  Just **True**, *Just* **False**, ..., ***Nothing***

- `Maybe Character`

  Just **'A'**, *Just* **'q'**, ..., ***Nothing***

- `Maybe Integer`

  Just **(-1)**, *Just* **14**, *Just* **0**, ..., ***Nothing***

- `Maybe`

  ???

Case study: `find`

```
find : Int -> List Int -> Maybe Int
```

— or —

```
find :  a  -> List  a  -> Maybe Int
```

List type constructor

```
type List a
  = EmptyList
  | Cons a (List a)
```

`List` type constructor

```
type List a
  = EmptyList
  | Cons a (List a)
```

List type constructor

```
type List a
  = EmptyList
  | Cons a (List a)

>>> []
EmptyList

>>> [1, 2, 3]
Cons 1 (Cons 2 (Cons 3 EmptyList))
```

List type constructor

```
type List a
  = EmptyList
  | Cons a (List a)

>>> []
EmptyList

>>> [1, 2, 3]
Cons 1 (Cons 2 (Cons 3 EmptyList))
```

List type constructor

```
type List a
  = EmptyList
  | Cons a (List a)

>>> []
EmptyList

>>> [1, 2, 3]
Cons 1 (Cons 2 (Cons 3 EmptyList))          -- a = Int
```

List type constructor

```
type List a
  = EmptyList
  | Cons a (List a)

>>> []
EmptyList

>>> [1, 2, 3]
Cons 1 (Cons 2 (Cons 3 EmptyList))          -- a = Int
```

List type constructor

```
type List a
  = EmptyList
  | Cons a (List a)

>>> []
EmptyList

>>> [1, 2, 3]
Cons 1 (Cons 2 (Cons 3 EmptyList))          -- a = Int
```

Quick aside: strings

```
type alias String = List Character

>>> “Hello”
[‘H’, ‘e’, ‘l’, ‘l’, ‘o’]
-- Cons ‘H’ (Cons ‘e’ (Cons ‘l’ (Cons ‘l’ (Cons ‘o’ EmptyList))))

>>> find ‘e’ “Hello”
Just 1
```

List is *not* a type! (`List` a is)

- **`List`** `Bool`

  `[], [True], [False], [True, False, True], ...`

- **`List`** `Character`

  `[], [‘a’], [‘b’, ‘c’], [‘d’, ‘d’, ‘d’, ‘e’], ...`

- **`List`** `Integer`

  `[], [1], [-1, 0, 1], [3, 3, 3, 3, 3], [4], ...`

- **`List`**

  `???`

Quick aside: composing types (order matters!)

- `List (Maybe Bool)`

```
[],
[Just False],
[Just True, Nothing],
[Nothing], ...
```

- `Maybe (List Bool)`

```
Nothing,
Just [True, False],
Just [True],
Just [False],
Just [False, False, False], ...
```

Example: binary trees

```
type BinaryTree a
  = Leaf a
  | Node (BinaryTree a) a (BinaryTree a)
```

Example: pairs

```
type Pair a b
  = P a b
```

```
>>> a = (P 103 "hi")
(P 103 "hi") : Pair Integer String
```

Commonly denoted:

```
>>> b = (103, "hi")
(103, "hi") : (Integer, String)
```

Final example: list zipper

```
type alias Zipper a
  = (List a, List a)
```

```
>>> a = ([1, 2, 3, 4], [])
([1, 2, 3, 4], [])
>>> b = next a
([2, 3, 4], [1])
>>> c = next b
([3, 4], [2, 1])
>>> d = next c
([4], [3, 2, 1])
>>> e = prev d
([3, 4], [2, 1])
>>> focus e
3
```

Why are algebraic data types ***useful***?

Benefits of ADTs

- \+ Provide an incredibly general mechanism to describe types
- \+ Allow us to express types like `Maybe` a
  - Eliminate an entire class of bugs: null pointer exceptions
- \+ Promote composability of types (code reuse = good)
- \+ Urge us to fully consider our problem domain before coding
- – Can be too general/abstract to understand easily “in the wild”
  - To avoid incomprehensible code: choose the simplest possible abstraction needed for the problem at hand

So... who has ‘em?

- Elm
- Haskell
- Kotlin
- ML (OCaml, Standard ML, ...)
- Nim
- Rust
- Scala
- Swift
- Typescript

More: https://en.wikipedia.org/wiki/Algebraic_data_type#Programming_languages_with_algebraic_data_types

That’s great and all, but…

Why are algebraic data types ***cool***?

Answer: *math*

- Why are algebraic data types called "algebraic"?
- **Question:** Let N be the number of values of type a. How many values of type `Maybe a` are there?
- **Answer:** $\mathrm{N} + 1$. We have all the values of type a and also *Nothing*.
- **Question:** Is there a systematic way of answering these kinds of questions?
- **Answer:** Yes! ☺

A closer look at the ADT of `Maybe a`

```
type Maybe a
  = Just a
  | Nothing
```

$$\begin{aligned}\mathrm{Size}(\mathbf{Maybe}\ \underline{\mathrm{a}}) &= \mathrm{Size}(\mathit{Just}\ \underline{\mathrm{a}}) + \mathrm{Size}(\mathit{Nothing}) \\ &= \mathrm{N} + 1.\end{aligned}$$

Associated function:

$$M(\mathrm{a}) = a + 1.$$

A closer look at the ADT of `Pair a b`

```
type Pair a b
  = P a b
```

$\text{Size}(\textbf{Pair}\ \underline{\text{a}}\ \underline{\text{b}})$
$\quad = \text{Size}(\underline{\text{a}}) \cdot \text{Size}(\underline{b})$

Associated function:

$P(a, b) = a \cdot b.$

Sum and product types

```
type Maybe a
  = Just a
  | Nothing
```

```
type Pair a b
  = P a b
```

```
type Shape
  = Rectangle Float Float Float Float
  | Circle Float Float Float
```

We can define ***addition*** and ***multiplication*** on ***types***…
what else can we define?

A closer look at the ADT of `List a`

```
type List a
  = EmptyList
  | Cons a (List a)
```

Associated function:

$$L(a) = 1 + a \cdot L(a)$$

$$\Rightarrow\ L(a) - a \cdot L(a) = 1$$

$$\Rightarrow\ L(a) \cdot (1 - a) = 1$$

$$\Rightarrow\ \mathbf{L(a) = 1 / (1 - a)}.$$

A closer look at the ADT of `Zipper a`

```
type alias Zipper a
 = (List a, List a)
```

Associated function:

$$Z(a) = L(a) \cdot L(a)$$

$$\Rightarrow \boldsymbol{Z(a) = L(a)^2}.$$

Subtraction? ***Division***? Do those even make sense?

A better question:

A better question: ***why stop there?***

Calculus on algebraic data types

We know:

- $L(a) = 1 / (1 - a)$
- $Z(a) = L(a)^2$

Let's find $\mathrm{d}L/\mathrm{d}a$:

$$
\begin{aligned}
\boldsymbol{\mathrm{d}L/\mathrm{d}a} &= \mathrm{d}/\mathrm{d}a\,[\,L(a)\,] \\
&= \mathrm{d}/\mathrm{d}a\,[\,1 / (1 - a)\,] \\
&= 1 / (1 - a)^2 \\
&= [\,1 / (1 - a)\,]^2 \\
&= L(a)^2 \\
&= \boldsymbol{Z(a)}
\end{aligned}
$$

The ***derivative*** of a ***list*** is a ***zipper***?! How could this possibly make any sense?!

A shift in perspective

- Derivative = slope at a point…?
  - … really, derivative = ***local information*** at a point
  - With the derivative, we know the slope *locally*, but that doesn't tell us anything about the behavior of a function *globally*.
  - **Contrast:** integration = global information

- **Key point:** zippers tell us ***local*** information about a list by means of a **single, focused** element

- If we run a zipper along the entirety of a list (if we "integrate" the zipper), we get information about the list ***globally***
  - ⭑ Fundamental theorem of calculus! ⭑

Questions we answered

A. What is an ***algebraic data type***?

B. Why are algebraic data types ***useful***?

C. Why are algebraic data types ***cool***?

Interested in more?

- CMSC 16100 includes a lot of content about ADTs and related concepts
- Wikipedia:
  https://en.wikipedia.org/wiki/Algebraic_data_type
- If you want *more* generality:
  https://en.wikipedia.org/wiki/Generalized_algebraic_data_type
- My first introduction to the subject of calculus on ADTs, a *very* well-written series:
  http://chris-taylor.github.io/blog/2013/02/10/the-algebra-of-algebraic-data-types/
- Another good article about calculus on ADTs:
  https://codewords.recurse.com/issues/three/algebra-and-calculus-of-algebraic-data-types

Thank you!

Justin Lubin
justinlubin@uchicago.edu

A big thanks to **Asynchronous Anonymous**, too!