Algebraic data types in Encore: reconciling objection-oriented and functional programming
Uppsala University

Micael Loberg

November 16, 2017

This page intentionally left blank

Abstract

TODO

This page intentionally left blank

Contents

1. Introduction

Encore is an object-oriented programming language developed at Uppsala University blah blah

1.1 Background

1.1.1 encore

TODO

1.1.2 Algebraic Data Types

Algebraic data types (ADTs) are used to define composite data. Data structures can in type theory be described in terms of products, sums and recursive types. This leads to an algebra for describing data structures (hence Algebraic Data Types). Such data types are common in functional languages, such as ML or Haskell.

Robert Harper describes **Products** of types in the book *'Practical Foundations for Programming Languages'* as the following:

*"The binary **product** of two types consists of ordered pairs of values, one from each type in the order specified. The associated eliminatory forms are projections, which select the first and second component of a pair. The nullary product, or unit, type consists solely of the unique "null tuple" of no values, and has no associated eliminatory form."*

Sums of products can be seen as the choice of two or more variants of a data structure, both producs and sums can be seen in the following example of a polymorphic binary tree. Here Tree is the name of the type and Nil and Branch are constructors that are used to create new instances of type Tree and t is a type variable. The type Tree is also recursive as it is defined in terms of itself.

```
1 data Tree t = Nil
```

```
2             | Branch t (Tree t) (Tree t)
```

Listing 1.1: Binary tree definition in Haskell

1.1.2.1 ADTs In Functional Languages

In most functional languages there are no objects, so the primary method of defining composite data is with Algebraic Data Types.

Though ADTs is in no way equivalent with classes. Some of the major differences is mutability. Instance variables in a class is generally mutable while values in an ADT is not. Another distinction is that Algebraic Data Types allow for both sums and products while classes only allows for products.

ADTs also expose their insides to the world in a way that classes do not. This allows the programmer to perform pattern matching on the structure of an Algebraic Data Type. Pattern matching on the structure of types is also a feature that is mostly found in Functional languages and is critical to work with ADTs an effective manner.

1.1.2.2 ADTs In Imperative/OO Languages

While Algebraic Data Types is mostly found in functional languages there are examples of imperative languages that features them. Perhaps most notably in Scala.

Algebraic Data Types in Scala comes in the form of Variant types. A binary tree in Scala could have the following definition:

```
1 sealed trait Tree[+T]
2 case object Nil extends Tree[Nothing]
3 case class Branch[T](value:T, left:Tree[T], right:Tree[T]) extends
      Tree[T]
```

Listing 1.2: ADT definition in Scala

Noteworthy here is that the ADT is defined in terms of a Trait and classes extending the trait. A sealed trait is a special kind of trait that can only be extended in the same file as it is defined. A case class is mostly a normal class with a few differences such as its being immutable. Case classes can also be used in pattern matching:

```
1 tree match {
2     case Branch(value, left, right) => Foo()
3     case Nil()  => Bar()
4 }
```

Listing 1.3: Pattern matching on an ADT in Scala

One can easily see the similarities of this implementation and the one in listing 1.1. Here Tree is the new type, Nil and Branch could be seen as constructors for the type Tree.

By using the scheme of traits and classes to create something that looks and behaves a lot like an ADT Scala have created objects and are both sum and product types, and the fact that case classes can be used in destructured pattern matching allows for allows for programming patterns normally found only in functional languages.

2. Problem – encoding is bad

TODO

3. Design

TODO

3.0.1 Syntax

The syntax for ADTs have gone through a number of iterations before we settled on the final one described below. It is inspired by the syntax Scale use for its Variant types, but modified to fit in with the rest of the Encore language.

An adt is defined by using the keyword ́data; followed by an identifier starting with a capital letter.

```
1 data Foo
```

A branch (or constructor?) of an ADT is defined as following:

```
1 case Bar(valueA : t1, valueB : t2) : Foo
```

where `Bar` is the name of the branch, `valueA` and `valueB` are the fields of type t1 and t2, `:Foo` lets the compiler know that `Bar` is a branch of `Foo`. A branch does not have to be defined on the line following the ADT, but anywhere as long as it is in the same file. Both ADTs and its branches can define methods. Methods are defined on a new indented line under the ADT/branch definition. A ADT or branch that have methods defined needs to be closed with the `end` keyword.

```
1 data Foo
2   def Bar() : unit
3     --methodbody
4   end
5 end
```

Listing 3.1: ADT definition with a method

ADTs can also take optional type parameters. Type parameters are defined with a comma separated list within brackets

```
1 data List[t]
2 case Node[t](value : t, next : List[t]) : List[t]
3 case Nil[t]() : List[t]
```

Listing 3.2: Generic linked list implemented with an ADT

To create an instance of a branch you call the constructor functions that is generated for each of the branches. A instance of a `Node` can be created like this:

```
let
  list = Node(1, Nil())
in
  --body
end
```

Listing 3.3: Declaration of a list containing one element

ADTs can be used in pattern matching expressions

```
match list with
  case Node(value, next) => Foo()
  case Nil() => Bar()
end
```

Listing 3.4: Pattern matching on a linked list

3.0.2 Behaviour

As ADTs and their branches gets desugared to read only traits and classes they can do everything traits and classes are capable of. Methods that are declared inside of the ADT declaration ends up in the trait as required methods, and methods declared in the branch end up in the class. It's however worth to note that in the current state of Encore, when you call a constructor function for a branch you will get an object with the type of the trait back. So right now it's not possible to ever call a method on an ADT branch. This can quite easily be solved by adding a few more features to Encore, this I will discuss in chapter 6.

As mentioned above, the classes and traits generated are read only, this means that the values held by an ADT branch are immutable. The main motivation for them being immutable is that it is what I believe most users will expect from an ADT as its a language feature mostly found in functional languages. It also makes them safe to pass around between different actors as no actor is able to modify it.

4. Implementation

4.0.1 Implementation via desugaring

The Encore compiler is written in Haskell and generates C code as output which is then piped into Clang to generate executable code. The Encore compiler has a desugaring phase that can be used to turn the ADT nodes in the Abstract Syntax Tree (AST) into class and trait nodes. Methods that are used as constructors for the different branches will also be created.

The following linked list implemented with an ADT

```
1 data List[t]
2 case Node[t](value : t, next : List[t]) : List[t]
3 case Last[t](value : t) : List[t]
```

Listing 4.1: Linked list before it has been desugared

will after the desugaring phase be transformed to the following trait, classes and methods.

```
1  fun Node[t](value : t, next : List[t]) : List[t]
2    new Node[t](value, next)
3  end
4
5  fun Last[t](value : t) : List[t]
6    new Last[t](value)
7  end
8
9  read trait List[t]
10   require def Last() : Maybe[t]
11   require def Node() : Maybe[(t, List[t])]
12 end
13
14 read class Node[t] : List[t](value, next)
15   val value : t
16   val next : List[t]
17
18   def init(value : t, next : List[t]) : unit
19     this.value = value
20     this.next = next
21   end
22
23   def Last() : Maybe[t]
24     Nothing
```

```
25    end
26
27    def Node() : Maybe[(t, List[t])]
28      Just((this.value, this.next))
29    end
30  end
31
32  read class Last[t] : List[t](value)
33    val value : t
34
35    def init(value : t) : unit
36      this.value = value
37    end
38
39    def Last() : Maybe[t]
40      Just(this.value)
41    end
42
43    def Node() : Maybe[(t, List[t])]
44      Nothing
45    end
46  end
```

Listing 4.2: Desugared linked list

The trait generated on lines $9 - 12$ requires that the classes implements extractor methods, one for each branch of the ADT. The extractor methods are used in pattern matching, in this case they are `Node()` and `Last()`

4.0.2 Pattern matching optimization

TODO

5. Evaluation and discussion

TODO

5.1 Expressive power

TODO

5.1.1 Some cool example

TODO

5.1.2 Performance

TODO

5.1.2.1 Benchmark

TODO

6. Conclusion and Future work

TODO

Appendices