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

Micael Loberg

December 12, 2017

This page intentionally left blank

Abstract

TODO

This page intentionally left blank

Contents

1. Background

1.1 Encore

Encore is an object-oriented programming language developed at Uppsala University blah blah **[Kiko: Don't forget to reference [brandauer:2015:POM]]** TODO

1.2 Algebraic Data Types

[Kiko: I think it may be better if you switch the approach: functional languages use ADTs to describe... Potentially, you should reference the original paper, which seems to be "A history of Haskell: being lazy with class", according to Wikipedia] 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."*

[Kiko: What are sums of no products? You are defining sum of products but what about just sums?] **Sums** of products can be seen as the choice of two or more variants of a data structure, **[Kiko: instead of "," , this is either a "." or a ";"]** both producs and sums can be seen in the following example of a polymorphic binary tree **[Kiko: I would add here: Listing 1.2 reference]**. 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.

[Kiko: use verbatim or other font for Tree, Nil, t, etc since you are referring to code.]

```
1 data Tree t = Nil
2             | Branch t (Tree t) (Tree t)
```

Listing 1.1: Binary tree definition in Haskell

1.2.1 ADTs In Functional Languages

[Kiko: I found this paper, which seems relevant: https://dl.acm.org/citation.cfm?doid=1094811.1
"Generalized algebraic data types and object-oriented programming"]
[Kiko: I find this introduction a bit abrupt. Maybe ADTs started in the functional setting to solve a problem and you can continue from there.] In most functional languages there are no objects, so the primary method of defining composite data is with Algebraic Data Types.

[Kiko: What do you mean? This first sentence is not clear.] 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. **[Kiko: Maybe I am mistaken but Animal¡t¿ has subclasses Cat¡t¿ and Dog¡t¿. Can a polymorphic class be considered sum type? I am not saying that it is, I am asking.]**

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.

1.2.2 ADTs In Imperative/OO Languages

1.2.2.1 Scala

[Kiko: In general, you are using the "
" too often. I don't think you should be using it, at all.]

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 **[Kiko: Missing reference here. the reader doesn't know what Variant types are and she cannot find them without a reference]**. A binary tree in Scala could have the following definition:

```
sealed trait Tree[+T]
case object Nil extends Tree[Nothing]
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 **[Kiko: Can you assume that the reader knows what is a trait or do you want to write a one line sentence that explains it?]**. **[Kiko: In Scala]** 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 **[Kiko: Great if you could add references to this concepts]**. Case classes can also be used in pattern matching:

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

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 **[Kiko: use ref{your-listing}]**. Here Tree is the new type, Nil and Branch could be seen as constructors for the type Tree. **[Kiko: use verbatim font for Nil, Branch, etc.]**

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.

1.2.2.2 ABS

[Kiko: You should add the reference to ABS language]

The Abstract Behavioral Specification language (ABS) is another language that features ADTs. It is a language that is interesting to look at because just like Encore it is an Object Oriented language that uses the Actor model as its concurrency model. When implementing ADTs in Encore, some design decisions have to be made and looking at another language that were faced with the same design decisions might give some insight of what a good design could be. One such example is that of mutability. In ABS all values held by an ADT is immutable, the reasoning behind this is to make them safe to pass around between different actors and that it makes it easier to reason about the code[lista ut ur

referenser fungerar milo]

A linked list containing Integers can be defined as

```
1 data List = Nil
2           | Node(Int value, List tail);
```

Listing 1.4: Linked list in ABS

Names for constructors and their arguments can optionally be omitted, an example is the following implementation of the Bool datatype

```
1 data Bool = True | False;
```

Listing 1.5: Actual definition of built-in type Bool

Test Kiko 1.5

```
1 data Bool = True | False;
```

Listing 1.6: Actual definition of built-in type Bool

If data constructor arguments have names - which needs to be a valid identifier, it defines a function that, when passed a value expressed with the given constructor, return the argument **[Kiko: It's not clear to me what you mean with this sentence above (first sentence)]**. The name of an accessor function must be unique in the module it is defined in. It is an error to have multiple accessor functions with the same name, or to have a function definition with the same name as an accessor function.

```
1 data List = Nil
2           | Node(Int head, List tail);
3 {
4   List list = List(1, List(2, Nil));
5   Int head = head(list);
6 }
```

Listing 1.7: Accessor funtion in ABS, variable head on line 5 is assigned 1

[Kiko: Overall: I think that you have done a nice job introducing the most important elements but, I get the feeling that the reading is not quite fluent yet. I think it's nice how you defined ADTs, and then put them in the functional and OO context. However, some parts introduce ideas a bit abruptly, IMO, and you should take some time to re-structure the text and put things in context. I believe you should also introduce one line sentence for concepts that you take for granted, e.g. traits, which is not something that all OOP languages have.] [Kiko: Grammar-wise, be careful when you talk in singular and plural: e.g. ADTs is in no way equivalent with classes – it should be – ADTs are in no way equivalent with classes. I have seen this mistake more than 5 times in chapter 1, and it happens from the

singular to plural case and vice versa. Use ”is” when you talk about a single thing and ”are” when you talk about many things :)]

2. Encoding ADTs in Encore

2.0.1 Encoding scheme

In Encore it is possible to emulate the behaviour of ADTs using a scheme of traits and classes.[**gustavL**]
An implementation of a linked list in Encore could consist of the two classes `Node` and `Last`, both of which implements a trait `LinkedList`

Classes can implement destructor methods **[Kiko: what is a destructor method?]** that exposes the internal structure of an object which can then be used in pattern matching. If the trait `LinkedList` requires the classes who implements the trait to also implement destructor methods, an object of type `LinkedList` can be used in pattern matching

```
1 trait LinkedList[t]
2   require def Node() : Maybe[(t, LinkedList)]
3   require def Last() : Maybe[(t)]
4 end
```

Listing 2.1: trait LinkedList that requires classes to implement destructor methods

An object of type `Node` will implement both of the extractor methods `Node()` and `Last` **[Kiko: or Last() with parenthesis?]**. The method `Node()` will return a `Just` **[Kiko: what is a Just? maybe better to phrase it as: will return an option type ... [maybe add reference to option types if you think that it is not clear what they are.]]**with a tuple consisting of the internal objects to be exposed, in this case `value` and `next` while the method `Last()` will return `Nothing`
Similarly an object of type `Last` will also implement both extractor methods `Node()` and `Last()`. Though here the method `Node()` will return `Nothing` while `Last()` returns `Just(value)`. **[Kiko: I didn't know that you were referencing a listing. You should mention that somewhere. To reference a listing, just add the label="my-label" inside the [] of the listing and reference in the normal style.]**

```
1 read class Node[t] : List[t](value, next)
2   val value : t
3   val next : List[t]
4
```

```
5   def init(value : t, next : List[t]) : unit
6     this.value = value
7     this.next = next
8   end
9
10  def Last() : Maybe[t]
11    Nothing
12  end
13
14  def Node() : Maybe[(t, List[t])]
15    Just((this.value, this.next))
16  end
17 end
18
19 read class Last[t] : List[t](value)
20   val value : t
21
22   def init(value : t) : unit
23     this.value = value
24   end
25
26   def Last() : Maybe[t]
27     Just(this.value)
28   end
29
30   def Node() : Maybe[(t, List[t])]
31     Nothing
32   end
33 end
```

Listing 2.2: Implementation of Node and Last classes

To make the classes `Node` and `Last` look more like branches of an ADT and hide the fact that they are classes, we also add constructor functions to create the objects for us.

```
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
```

Listing 2.3: Constructor functions for Node and Last

A LinkedList containing the values $[1, 2, 3]$ can now be created like this

```
1 var list = Node(1, Node(2, Last(3)))
```

Listing 2.4: Creation of list containing three elements

The variable `list` can now be used in pattern matching

```
1 fun listLength[t](list : List[t]) : int
2   match list with
3     case Node(value, next) => 1 + listLength(next)
4     case Last(value) => 1
5   end
```

```
6 end
```

Listing 2.5: Function that uses pattern matching to calculate the length of a list

Now we have something that looks and behaves a lot like an ADT in for example Haskell. We have functions that creates the objects for us and hides the fact that we're creating an object from a class and we can perform pattern matching on the structure of of the object.

2.0.2 Problems with encoding

In the previous section we have seen how we in Encore can emulate the behavior of ADTs by using a scheme of traits, classes, methods and pattern matching. Though this is far from perfect.

Perhaps the most noticeable problem is the difference in amount of lines of code needed to create something that behaves like an ADT in Encore compared to languages that has ADTs as actual types.

Another problem is the cost of doing pattern matching on classes. Consider the extractor method `Node()` on line 30 in listing 2.2.
The purpose of the method is to tell if a given `List` is of type `Node` or not. The value it returns is a `Maybe` type, holding a tuple that contains the values a `Node` wants to expose. Memory for both the Maybe and the Tuple types needs to be allocated for and the fields from the Node needs to be copied over to the tuple. All of this adds a considerable amount to the runtime of an application that does a lot of pattern matching.

Both of these problems could be fixed by extending Encore and adding ADTs to the language. A more compact syntax and a better optimized method of performing pattern matching could be designed.

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

```
24 read class Last[t] : List[t](value)
25   val value : t
26
27   def init(value : t) : unit
28     this.value = value
29   end
30
31   def Last() : Maybe[t]
32     Just(this.value)
33   end
34
35   def Node() : Maybe[(t, List[t])]
36     Nothing
37   end
38 end
39
40 fun Node[t](value : t, next : List[t]) : List[t]
41   new Node[t](value, next)
42 end
43
44 fun Last[t](value : t) : List[t]
45   new Last[t](value)
46 end
47
```

Listing 4.2: Desugared linked list

The trait generated on lines $X - YY$ requires that the classes implements extractor methods, one for each branch of the ADT. The extractor methods are used in pattern matching and their purpose will be discussed in the next chapter. In this case the extractor methods are `Node()` and `Last()`.

Every branch in the ADT will be transformed into a class containing the fields contained in the branch, a constructor method and extractor methods for all the branches of the ADT.

A creator function for each branch will also be created. These are used as a syntactic sugar to create instances of the ADTs branches and hide the fact that they are implemented as classes.

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

7. References

Bibliography

[1] Gustav Lundin *Pattern Matching in Encore*. http://www.diva-portal.org/smash/get/diva2:930151/FULLTEXT01.pdf

Appendices