Algebraic Data Type: an essential concept for safe and compositional code

Sol Kim

December 6, 2017

Abstract

Many of advanced type systems have been introduced to practical languages. An algebraic data type is one of the fundamental materials for most of the advanced type systems. Still, an algebraic data type can be used to provide more safety and compositionality as itself, not as a part of the advanced type systems. A classic, but ironically a radical example, that shows advantages and essence of an algebraic data type, is inductively defined primitive data types. An algebraic data type can provide the safe and compositional programming development environments not only while building programs from a scratch but also while refactoring large-scale programs. On the other hand, non-algebraic data type languages—most of the conventional industrial languages such as C, C++, Java—use tricks and language-dependent methods to obtain similar environments of an algebraic data type languages, but most of those highly depend on developers' skills. Fortunately, there are interesting research and practical examples that try to infuse advantages and essence of an algebraic data type to non-algebraic data type languages. Likewise, various programming patterns, tricks have been introduced for conventional industrial languages to obtain safety and compositionality, in the meanwhile, modern languages embrace an algebraic data type as a primitive feature. Undoubtedly an algebraic data type is an essential concept for safe and compositional code.

Keywords: algebraic data type, programming language

Advanced type systems can guarantee more safe and compositional programming environments than conventional type systems. As well known, a type safety is one of the safety baselines that guarantees programs run safely. Statically typed languages validate type safety during compile time, and programs which passed a type checker never crash by type errors while running. Unfortunately, type checkers are inevitably not enough to cover every case of programs but some advanced type systems can cover more safety than conventional type systems. Proof assistants, which is one of the extreme examples of advanced type systems, even can represent and prove specifications of programs formally, and the languages are general and expressive enough to encode every computational logic.

Many of advanced type systems have been introduced to practical languages. Generalized algebraic data types (GADTs) have been introduced several practical functional languages as a primitive data type[3, 5]. GADTs make languages possible to express more invariants of code than conventional data types. The data type can encode invariants into types that are used to be implemented as runtime code to satisfy specifications of code, and the invariants are guaranteed by a type checker even before running it by definition of a type system. A typical example of uses of the data type is implementing DSL and its evaluator with robust type safety. Dependent types are another example of an advanced data type. Coq and Idris are purely functional languages with dependent types[2, 4], and the data type even can be defined *depending* on values. Values can be a part of types, and a type checker will check type safety with the types while compiling it. More detailed specifications of code could be defined as types using the data types, such as types that include list's length, types that include tree's shape, types that include whether a number is even or odd, and others. Thus, logic which are typed with dependent types could be *certified* by a type checker, and the certified logic never go wrong after compiled. Some of newly introduced advanced type systems have a specific purpose than the traditional general purpose type systems such as GADTs, dependent types. Rust has a distinctive type system that checks latent errors, which can be happened in parallel and concurrency programming environments such as race conditions, during compile time[7].

An algebraic data type is one of the fundamental materials for most of the advanced type systems. Crucial parts of type theory have been based on a correspondence between types and mathematical logic[9]. Logical disjunction ($\vee$) and logical conjunction ($\wedge$) are the most basic materials of mathematical logic. Naturally, in the sense of correspondence between types and logic, types must have something that is dual of the two logical operators, and an algebraic data type satisfies the role. A sum type (`A + B`) and product type (`A × B`) of the data type is the dual of logical disjunction and logical conjunction respectively[10]. Besides the duality, more connections between types and mathematical logic have been discovered and introduced to languages. As consequence of observing and evolving type systems through the lens of mathematical logic, type systems have been able to be improved to right directions; safe and compositional programming environments with the mathematical basis. But, basically, to stack connections between types and mathematical logic from the most primary correspondence—$\vee, \wedge$ and $+, \times$—to a higher dimension of correspondences, an algebraic data type is an essential feature.

Still, an algebraic data type can be used to provide more safety and compositionality as itself, not as a part of the advanced type systems. A sum type and product type is general and expressive enough to composite complex data structures. It means that languages, which provide an algebraic data type as a primitive data type, easily obtain compositionality and its type safety. Induction, also, well fits with an algebraic data type, because base cases and inductive cases of an induction can be defined through an algebraic data type. A combined way of composing data structures using the two basic mathematical materials is general and expressive enough to represent highly complex data structures. Even the method can express

```
type 'a list = Nil | Cons of 'a * 'a list

let (::) : 'a → 'a list → 'a list = fun x l → Cons(x, l)

let rec zip : 'a list → 'b list → ('a * 'b) list = fun l1 l2 →
  match l1, l2 with
  | Cons(x, l1'), Cons(y, l2') → (x, y)::(zip l1' l2')
  | Nil, Nil | Nil, _ | _, Nil → Nil
```

Figure 1: Inductively defined natural number using algebraic data type, and a simple add function of it.

from natural numbers and its operators to general purpose languages[8].

A classic, but ironically a radical example, that shows advantages and essence of an algebraic data type, is inductively defined primitive data types. An inductively defined `list` is a classic example of an inductively defined data structure using an algebraic data type.

```
type 'a list = Nil | Cons of 'a * 'a list
```

A `zip` function in figure 1 shows that how to handle algebraically and inductively defined data types. The function consists of destructions and constructions of terms based on the type `list` definitions. A program logic, which handles algebraically and inductively defined data types, should handle every case of data types. A pattern matching operator, like in figure 1, will check that destruction logic cover every case of data type during compile time. Therefore, most of the logic has to be straightforwardly implemented to follow its type definitions when destructing terms. Furthermore, specifications of functions are naturally represented and exposed by destruction and construction logic. The function `zip` zips two lists until an opposite list isn't an empty list, and the specification is defined by destructing logic without control flow statements such as `if-else`. It means that the specification is guaranteed by a type checker even before running it, unlike runtime code. Likewise, even language isn't a proof assistant, well-designed data structures and its functions based on an algebraic data type make us possible to build more safe and compositional code.

An algebraic data type can provide the safe and compositional development environments not only while building programs from a scratch but also while refactoring large-scale programs. Modifying already defined data structures is a complicated and dangerous work for large-scale programs. Tracking influenced points by changes of data structures is an extremely difficult work for large-scale programs. But, because most of the logic that handles an algebraic data type consist of destruction and construction logic based on data types, every influenced point is relatively straightforwardly exposed by logic itself. Furthermore, every destruction and construction logic is checked by a type checker while compiling code, so even if developers miss some points that are affected by changes, a compiler will notice the points missed by developers.

```
interface SumOfAB { }
class A implements SumOfAB { }
class B implements SumOfAB { }
class C { }
class ProductOfABandC {
    public SumOfAB ab;
    public C c;
}
```

Figure 2: A Java version of a sum type and product type

On the other hand, non-algebraic data type languages—most of the conventional industrial languages such as C, C++, Java—use tricks and language-dependent methods to obtain similar environments of an algebraic data type languages, but most of those highly depend on developers' skills. Obviously, because a concept of an algebraic data type is simple, non-algebraic data type languages can easily encode the data type. Figure 2 shows a trivial way of expressing an algebraic data type using Java. An *interface-implementation* hierarchy seems well covering a sum type of an algebraic data type, but uses of the encoded algebraic data type is not, strictly saying, type-safe. As we have seen in figure 1, pattern matching operators of functional languages can destruct a sum type term into its cases, but in case of most of the non-algebraic data type languages don't have those kinds of operators. For example, in Java, a type of term is defined by an `interface`, than the term should be re-assigned into its actual type when developers want to use the term as the actual type. It is unavoidable to use runtime-checked cast like `instanceof` in Java, but type checkers don't handle the runtime-checked cast operators during compile time, in other words, the method is not type-safe. As an alternative method against runtime-checked cast operators, developers have encoded data structures that originated in algebraically defined data structures mostly in functional languages. A following algebraically defined `option` type is a trivial data structure in functional languages.

```
type 'a option = None | Some of 'a
```

A Java standard library provides a class `Optional` to mimic the `option` type in functional languages.

```
public class Optional<T> {
    private T value;
    public boolean isPresent() ...
    public T get() ...
    public T orElse(T other) ...
    /* and other auxiliary functions */
}
```

The class has various helper functions that handle `Optional` terms, but many of the helper functions even consist of unstraightforward implementation details to satisfy specifications.

```
trait SumType
case class A() extends SumType
case class B() extends SumType
object SumType {
  def exampleFun(sumData: SumType): Any =
    sumData match {
      case (a: A) => // a variable a is casted as a type A
      case (b: B) => // a variable b is casted as a type B
    }
}
```

Figure 3: A Scala (non-Dotty) version of a sum type

A problem of those kinds of workarounds is that every virtue, i.e. type safety, compositionality, complexity, depends on developer's level of understanding about design patterns, tricks, and language-dependent features.

Fortunately, there are interesting research and practical examples that try to infuse advantages and essence of an algebraic data type to non-algebraic data type languages. Kennedy and Russo [6] show how to express GADTs using object oriented language features. Despite a method in the research can't avoid using redundant runtime-checked casts, the research provides an appropriate way to express GADTs using generics, subclassing, and other object orient languages' type system features. Scala before Dotty haven't supported an algebraic data type explicitly, but the language is capable of encoding the data type using *interface-implementation* hierarchies, and a pattern matching operator of the language. Figure 3 shows how Scala code destructs and casts safely without runtime-checked cast operators. Unlike Java and other object oriented languages, Scala provides a pattern matching operator as a native feature. The pattern matching operator of Scala also checks whether code covers every possible case of the target object during compile time, hence the method is definitely safer than runtime-checked cast logic.

Likewise, various programming patterns, tricks have been introduced for conventional industrial languages to obtain safety and compositionality, in the meanwhile, modern languages embrace an algebraic data type as a primitive feature. A certain strong point of newer languages is that the languages are outcomes of improvement from older languages. Interestingly, famous modern industrial languages such as Rust, Swift, and Scala embrace an algebraic data type as a primitive data type. In the case of Scala, after eight years of verification works for type soundness of the language's type system[1], they introduced a new compiler called Dotty and the compiler provides a sum type as a primitive data type. As consequence of the remarkable work, a sum type of Scala is come out to the world with verified type soundness. Besides the language, Rust, Swift, and others have a sum type and a pattern matching operator. Those languages are evidence that an algebraic data type and pattern matching operator are now essential features for languages. On the other hand, in case of a product type, some of the languages provide the type as a native type, i.e. `tuple`, but some of the languages, especially object oriented languages, don't. But, in case of object oriented languages, a product type is relatively easy to express than a sum type using basic programming language materials.

Undoubtedly an algebraic data type is an essential concept for safe and compositional code. An expression power of an algebraic data type is strong enough to encode most of data structures from the most primitive ones to complex ones. The concept of the data type is based on basic mathematical materials, thus, the concept is relatively simple and intuitive than complex programming patterns, tricks. The concept has been used very long time as a basic material to obtain benefits that are mathematically proven in a specific area such as proof assistants and functional languages. Conventional industrial languages hadn't savored the concept, but eventually, with various implementation details the data type has been infused into the conventional languages. In case of modern languages, they have introduced an algebraic data type as a primitive feature of languages. Likewise, the data type is already an essential feature to build safe and compositional code.

References

[1] Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The essence of dependent object types. In *A List of Successes That Can Change the World*. Springer, 249–272.

[2] Edwin Brady. 2013. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming* 23, 5 (2013), 552–593.

[3] James Cheney and Ralf Hinze. 2002. A lightweight implementation of generics and dynamics. In *Proceedings of the 2002 ACM SIGPLAN workshop on Haskell*. ACM, 90–104.

[4] Adam Chlipala. 2013. *Certified Programming with Dependent Types: A Pragmatic Introduction to the Coq Proof Assistant*. MIT Press.

[5] Jacques Garrigue and JL Normand. 2011. Adding GADTs to OCaml: the direct approach. In *Workshop on ML*.

[6] Andrew Kennedy and Claudio V Russo. 2005. Generalized algebraic data types and object-oriented programming. *ACM SIGPLAN Notices* 40, 10 (2005), 21–40.

[7] Nicholas D Matsakis and Felix S Klock II. 2014. The rust language. In *ACM SIGAda Ada Letters*, Vol. 34. ACM, 103–104.

[8] Christine Paulin-Mohring. 2015. Introduction to the calculus of inductive constructions. (2015).

[9] Morten Heine Sørensen and Pawel Urzyczyn. 2006. *Lectures on the Curry-Howard isomorphism*. Vol. 149. Elsevier.

[10] Philip Wadler. 2015. Propositions as types. *Commun. ACM* 58, 12 (2015), 75–84.