Towards Improved GADT Reasoning in Scala

Lionel Parreaux
EPFL

Aleksander Boruch-Gruszecki
EPFL

Paolo G. Giarrusso
Delft University of Technology

Abstract

Generalized algebraic data types (GADT) have been notoriously difficult to implement correctly in Scala. Both major Scala compilers, Scalac and Dotty, are currently known to have type soundness holes related to them. In particular, covariant GADTs have exposed paradoxes due to Scala's inheritance model. We informally explore foundations for GADTs within Scala's core type system, to guide a principled understanding and implementation of GADTs in Scala.

CCS Concepts • **Software and its engineering → Data types and structures**; **Classes and objects**;

Keywords Generalized algebraic data types, Scala, DOT

ACM Reference Format:
Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019. Towards Improved GADT Reasoning in Scala. In *Tenth ACM SIGPLAN Scala Symposium (Scala '19), July 17, 2019, London, United Kingdom.* ACM, New York, NY, USA, 5 pages. https://doi.org/10.1145/3337932.3338813

1 Introduction

Generalized algebraic data types (GADT) were proposed to encode expressive invariants through types [4, 11, 21]. For instance, Figure 1 defines a GADT to represent well-typed terms of simply-typed λ-calculus, similarly to Rompf and Odersky [18]. `Expr` is a GADT because each of its cases extends `Expr` with different type arguments. The `eval` function maps each value of type `Expr[A]` into a value of type `A`.

Why does type checking the `eval` function require special reasoning? First, in all but the `Var` case, the type of the scrutinee `e` is refined from `Expr[A]` to a more precise type. For example, `Lit` extends `Expr[Int]`, so if `e` matches `Lit(n)`, we can deduce that `A = Int`. This allows `n`, which has type `Int`, to agree with `eval`'s expected return type `A`. Second, in addition to refining `A`, the `Fun` and `App` cases uncover existential types (unknown types that do not appear in the function's signature); in the `App` case, which matches against patterns of type `App[X,A] <: Expr[A]`, type `X` is unknown, but has to be treated consistently as it appears in the two extracted subexpressions `fun` and `arg`. In the `Fun` case, we have to use a type pattern `f: Fun[a,b]` to bind the uncovered existential type `a` so we can use it in a required type annotation.

```
enum Expr[A] {
  case Var[A](a: A) extends Expr[A]
  case Lit(n: Int) extends Expr[Int]
  case Plus(lhs: Expr[Int], rhs: Expr[Int])
         extends Expr[Int]
  case Fun[A, B](fun: Expr[A] => Expr[B])
         extends Expr[A => B]
  case App[A, B](fun: Expr[A => B], arg: Expr[A])
         extends Expr[B]
}
def eval[A](e: Expr[A]): A = e match {
  case Var(x) => x  case Lit(n) => n
  case Plus(a,b) => eval(a) + eval(b)
  case f: Fun[a,b] => (x: a) => eval(f.fun(Var(x)))
  case App(fun,arg) => eval(fun)(eval(arg))
}
```

Figure 1. GADT in Scala, using Dotty's new `enum` syntax (https://dotty.epfl.ch/docs/reference/enums/enums.html).

Today, this specific example already works well in Dotty, the future Scala 3 compiler. However, there are several lingering unresolved issues with GADTs in Scala:

Subtle Soundness Issues. Scala GADTs have been plagued with type soundness issues. The scalac compiler uses approximate reasoning that easily leads to runtime crashes [15], while Dotty GADTs are still unsound, despite some recent substantial improvements, and are subject to ongoing work.[1]

Declaration-Site Variance. Scala supports declaration-site variance, a convenient way of defining subtyping relationships between parameterized types. For instance, in Figure 1 we could make `Expr` covariant to encode the λ calculus *with subtyping*. However, Dotty then rejects our definition of `eval` as ill-typed, and requires adding unsafe casts. It is unclear whether definitions like `eval` actually are unsafe, or whether Dotty is overly conservative; indeed, sound typing rules for pattern matching on open GADTs is an open problem [7].[2]

Scala '19, July 17, 2019, London, United Kingdom

ACM ISBN 978-1-4503-6824-7/19/07…$15.00
https://doi.org/10.1145/3337932.3338813

[1] See pull requests #5736 and #6398 at https://github.com/lampepfl/dotty.
[2] Scherer and Rémy [19] did consider the problem of GADTs with subtyping, but in the context of OCaml, where they made simplifying assumptions that are not necessary in Scala, such as the use of type equality constraints only, as opposed to more precise subtyping constraints.

To address these problems and to gain confidence in the soundness of GADTs in Scala, we believe necessary to justify them in terms of Scala's core foundations, which have been formalized as the Dependent Object Types calculus (DOT) [1]. This short paper makes the following contributions:

- Drawing from the `Expr` motivating example, which we believe to be quite representative, we informally sketch how to encode closed GADTs, first in full Scala, and then in a core Scala subset which can be mapped to DOT. We show that `eval` actually is safe with a covariant `Expr`, and thus argue Scala should improve its support for closed variant GADTs (Section 2).
- We consider the more general case of open GADTs, and sketch a minimal extension to core Scala that allows encoding them. We explain the mismatch between such encoding and Scala's treatment of type parameters, and following that insight we propose improvements to Scala usability which would allow sound pattern matching on variant open GADTs (Section 3).

Our examples and our encodings are available in full at https://github.com/Blaisorblade/scala19_gadt_code.

2 Closed GADTs in Scala Core and DOT

In this section, we focus on encoding *closed* GADTs. By "closed," we mean those GADTs which can be defined using the new Dotty **`enum`** syntax. Slightly more generally, we mean a flat class or trait hierarchy where (1) there is a single, sealed parent; (2) each implementing case is **`final`**; (3) each case extends the parent exactly once.

2.1 Encoding of ADTs and Pattern Matching

It is well-known that structurally-recursive pattern-matching on sealed hierarchies of data types can be emulated using fold functions. This technique allows encoding *algebraic data types* in System F through the Church/Böhm-Berarducci encoding [3, 10]; however, recursion that is not structural becomes awkward to expres and inefficient [12]. On the other hand, in a setting with general recursion and recursive types, such as Scala (and its foundation DOT), we can instead express pattern matching through the Scott/Parigot encoding, which supports unrestricted recursion. In object-oriented languages, this encoding is equivalent to using *external* visitors [9]: one simply defines a "visitor" method in the superclass, which will be implemented by each subclass of the hierarchy. For instance, we can encode the `Option` data type as follows:

```
abstract class Option[+A] { def visit[R]
  (_Some: Some[A] => R, _None: None.type => R): R }
class Some[+A](a: A) extends Option[A] {
  def visit[R](_Some: Some[A] => R,
    _None: None.type => R): R = _Some(this) }
object None extends Option[Nothing] {
  def visit[R](_Some: Some[Nothing] => R,
    _None: None.type => R): R = _None(this) }
```

2.2 GADTs and Object-Oriented Languages

As we have seen, GADTs are essentially ADTs with both existential types and type (in)equality[3] proofs to be uncovered via pattern matching [4, 19, 21]. It is also well-understood that GADT-like type hierarchies can be defined in object-oriented languages [5, 11]. The attentive reader will have guessed where we are going with this. We will encode GADTs in Scala using visitor methods. But first, we need to determine how we will encode existentials and subtyping proofs.

2.3 Existentials and Subtyping Proofs

In Scala, the primary way of representing existential types is via abstract type members, which are denoted using path-dependent types. However, an alternative encoding of existential types is to use higher-rank polymorphism [3], which we will use in our first encoding approach.

Moreover, Scala has first-class subtyping proofs thanks to bounded abstract type members. For instance, getting hold of an object `ev` of type `ev: { type Ev >: S <: T }` is equivalent to having a proof, evidence, or *witness* that `S <: T`. Though DOT does *not* require explicit usage of `ev` to leverage such proof, that is known to make type checking in DOT undecidable [13, 17]. In practice, Scala users normally have to explicitly apply these proofs; for example, if one wishes to "upcast" a value `s` of type `S` to a type `T` where `S <: T` does not hold syntactically, one has to write `s: ev.Ev` (a *type ascription*). We can make this approach more convenient by defining the following data type, used for manipulating subtyping proofs, which also doubles as an implicit conversion:

```
import scala.language.implicitConversions
abstract class <:< [-A,+B] extends Conversion[A,B]{
  type Ev >: A <: B; def apply(a: A): B = a: Ev }
implicit def Refl[A]: A <:< A = new { type Ev = A }
```

This is the same as the data type of the same name defined in the standard library, except that we have an additional `Ev` type member, which can be leveraged when the `apply` function is not enough (for example, we can convert a list `ls: List[S]` into a `List[T]` given some `ev: S <:< T` at no runtime cost, by writing `ls: List[ev.Ev]`).

2.4 Closed GADT Encoding in Scala

Figure 2 shows the encoding of *a covariant version* of `Expr` in Scala — that is, we show how to encode pattern matching on covariant GADTs using other mechanisms. Thanks to this encoding, the soundness of closed GADTs with declaration-site variance reduces to the soundness of the rest of Scala. While the encoding is elegant and intuitive, it is not fully satisfactory since full Scala is known to still have soundness holes (see e.g., [2]). Therefore, we now propose an encoding into Core Scala, a Scala subset we describe next.

[3] These are *subtyping* proofs, in the case of languages with subtyping.

2.5 Core Scala and DOT

We define Core Scala as the specific subset of Scala that can be translated into DOT in a straightforward manner. Since we make use of singleton types, which are not supported in DOT, we target in particular the pDOT dialtect of DOT [16], which soundly extends DOT with singleton types (and with paths, a feature we do not use).

DOT and pDOT do not directly support classes, but there are several examples in the literature on how to encode them [1, 8, 16]. Essentially, a class is represented as an abstract type whose upper bound specifies the class API, along with some constructors for building instances of the class.

Building on top of non-generic classes, we translate generic classes to non-generic classes using abstract type members: each class type parameter is turned into an abstract type member of the class, and applications of the class' type constructor to some arguments are represented as refinements of the class type (with type intersections). We refer to the literature for more concrete examples [1, 14, 16].

2.6 Closed GADT Encoding in Core Scala

Figure 3 shows an encoding of covariant `Expr` in Core Scala. The parameterized type alias `Expr[+A0]` is not definable in DOT but can be inlined at its call sites before translation. Indeed, we have encoded by hand the full Figure 3 in pDOT syntax following the class encoding mentioned earlier. We have not mechanically verified that this code can be type-checked in (p)DOT, due to the lack of an implementation.

```
abstract class Expr[+A] { def visit[R](
    Lit: given (Int <:< A) => Lit => R,
    Plus: given (Int <:< A) => Plus => R,
    App: [B] => App[B,A] => R,
    Fun: [B,C] => given ((B => C) <:< A) =>
        Fun[B,C] => R,
    Var: Var[A] => R ): R }
final case class Lit(n:Int) extends Expr[Int]{ s =>
  override def visit[R](
    Lit: given (Int <:< Int) => Lit & s.type => R,
    Plus... /* as in Expr */) = Lit.apply(this) }
// other cases similarly defined...
def eval[A](e: Expr[A]): A = e.visit[A](
  Lit = l => l.n,
  Plus = p => eval(p.lhs) + eval(p.rhs),
  App = [B] => a => eval(a.fun).apply(eval(a.arg)),
  Fun = [B,C] => f => ((x: B) =>
      eval[C](f.fun(Var(x)))),
  Var = v => v.a)
```

Figure 2. An encoding of the closed, covariant GADT in Figure 1. This code currently (2019.06.06) compiles in Dotty, and leverages polymorphic function types (see pull request at https://github.com/lampepfl/dotty/pull/4672), which use the `[X] => F[X]` syntax, analogous to system F's Λ/∀ binders.

One central insight of this encoding is that we do not need separate `<:<` witnesses, nor do we need polymorphic function types. This is because (1) we now use an abstract type `A` to represent the type parameter of the same name, and `A` is now visible from the outside; (2) `A` is *refined* in each subclass of `Expr` (see the definition of **`type`** `A` in class `Fun`); and (3) in the visitor method, we intersect the types of the extracted objects with the self-type `s` of the current instance. Thus, following normal DOT rules for type intersections, we are able to define `eval` by simply using the abstract type `A` itself as subtyping proof. Interestingly, to represent GADTs in Core Scala, we did not need to add any of the typical mechanisms commonly used to type check GADTs, such as special type equality proofs and coercions [6, 20].

2.7 Summary

We argue that Scala should handle closed GADTs well, irrelevant of variance. Indeed, we have shown that they can be encoded in a straightforward way, using a representative `Expr` example. Of course, one should properly develop a general formal explanation of this process — here we have merely tried to give an intuition about what that process could be. We reserve that formalization for future work.

3 Open GADTs

We now show an encoding of "open" GADTs — GADTs that are not sealed or do not have a flat hierarchy. These are useful because they can be extended with new constructors in a modular way, providing a solution to the expression problem [18]. Our previous encoding does not readily generalize to open GADTs, as the `visit` method received handlers for a

```
type Expr[+A0] = ExprBase { type A <: A0 }
abstract class ExprBase { s => type A
  def visit[R]( Lit: Lit & s.type => R,
    Plus:Plus & s.type => R, App:App & s.type => R,
    Fun: Fun & s.type => R,
    Var: Var & s.type => R  ): R }
abstract class Fun extends ExprBase {
  type B; type C; type A = B => C
  val fun: Expr[B] => Expr[C]
  override def visit[R](...) = Fun(this) }
// other cases similarly defined...
def eval[A](e: Expr[A]): A = e.visit[A](
  Lit = l => l.n: l.A,
  Plus = p => (eval(p.lhs) + eval(p.rhs)): p.A,
  App = a => eval(a.fun).apply(eval(a.arg)): a.A,
  Fun = f => ((x: f.B) =>
      eval[f.C](f.fun(Var(x)))): f.A,
  Var = v => v.a: v.A)
```

Figure 3. An encoding of the closed GADT in Figure 1 in Core Scala. Currently (2019.06.06) compiles in Dotty.

fixed list of constructors, while open GADTs would require different lists of constructors for different extensions.

3.1 Class Instance Matching

To achieve our new encoding, we assume Scala Core and (p)DOT are extended with a primitive runtime-class instance matching mechanism, which mirrors type matches in Scala:

```
s match { case x_1: C_1 => t1; case x_2: C_2 ... }
```

This construct branches on the runtime class of `s`, comparing it with classes `C1`, `C2`, etc. and evaluating the corresponding branch `t1`, `t2`, etc. To keep the extension simple, we only allow matching against simple class names, not arbitrary types, avoiding the complexities of Scala's type matching syntax. Since Scala Core has no class type parameters, we also avoid soundness problems due to the erasure of type parameters. In each branch, we bind `x_i` to the scrutinee, with type `C_i & s.type` (similarly to Figure 3). In terms of operational semantics, we must tag class instances with their class at run time, as is done in Java runtime systems. Formalizing this extension is out of scope for this paper.

With this extension, encoding *an open version* of `Expr` becomes as simple as dropping the `visit` method from the previous encoding of Figure 3, using type matches instead:

```
def eval[A](e: Expr[A]): A = e match {
  case l: Lit => l.n: l.A; /* other cases... */ }
```

3.2 Understanding an Old Paradox

Giarrusso [7] first noticed that certain desirable typing rules on covariant open GADTs are in fact unsound. Given:

```
trait Expr[+A]; class Const[+A] extends Expr[A]
```

the type checker would assume that if `e: Expr[A]` and `e` is an instance of `Const`, then `e: Const[A]`. But this assumption is false. One can extend covariant types like `Expr` multiple times with different type arguments. We can then break our assumption and define object `Unsound`, which extends `Expr[Int]` and is an instance of `Const`, but not of `Const[Int]`:

```
object Unsound extends Const[Any] with Expr[Int]
```

Viewing type parameters as type members not only makes the problem obvious, but also suggests how to make the assumption true without allowing definitions such as `Unsound`. The classes from the paradox would be translated as follows:

```
type Expr[+A] = ExprBase { type A$0 <: A }
type Const[+A] = ConstBase { type A$1 <: A }
trait ExprBase { type A$0 }
class ConstBase extends ExprBase {
  type A$0 <: A$1; type A$1 }
```

Notice that while translating the type parameters of each class, it is crucial to pick *different* type member names, so they do not conflict with each other. The subtyping relationship arising from inheritance of a *variant* base class is expressed via refinements of these abstract type members.

With the above definition, the assumption from the paradox is obviously false: given `e: Expr[A]` and `e: ConstBase`, we cannot conclude that `e: Const[A]`. Moreover, the assumption actually becomes true if we use a *different interpretation of inheritance from variant base classes*: if we instead declared `type A$0 = A$1` in `ConstBase`, this would prevent further extending `Expr` with incompatible type arguments, and would allow us to derive the required type equality proofs.

3.3 Solution: Invariant Inheritance

Giarrusso [7] also proposes a solution to the paradox: a syntax for "invariant inheritance," `class Const[+T] extends Expr[=T]`, which forbids definitions such as `Unsound` by simply forbidding further instantiation of `Expr` in children of `Const`. Following our new insights into the paradox, we propose this syntax to instead behave consistently with our encoding of type parameters as type members, translating those type argument marked with `=` to type equalities rather than subtype refinements. This interpretation still forbids definitions like `Unsound` while also allowing further extensions of `Expr`, as long as they conform in their type arguments.

For example, in the code below, `Z` should be able to extend `Expr` via both `Valued` and `Const`, as they are compatible:

```
trait Valued extends Expr[Int] { def v: Int }
object Z extends Const[0] with Valued { def v = 0 }
```

Remark that if we had written `Valued extends Expr[=Int]` above, the code of `Z` would have not compiled, as it would have had conflicting definitions for the parameter to `Expr`.

Interestingly, Dotty already has partial support for invariant inheritance (implemented specifically to counter Giarrusso's paradox in common cases), but it is restricted to case classes and not expressible otherwise. This makes the feature somewhat irregular and "magical" as it is not expressible in terms of regular features, unlike all other case class features.

4 Conclusions and Future Work

GADTs in Scala have historically been poorly understood. In this paper, we showed that they can be explained in terms of simpler features already present in Scala's core type system. We sketched different encodings of GADTs, demonstrating the tight correspondence between, on one hand, the (sub)type proofs and existential types that normally underlie GADT reasoning and, on the other hand, bounded abstract type members and intersection types, which are core to Scala.

It would be desirable to formalize GADT semantics by elaboration into pDOT following our sketches, which we leave for future work. In any case, the insights presented in this paper can already be used to guide future GADT developments in upcoming versions of the Scala compiler.

Acknowledgments

We would like to thank prof. Martin Odersky for his help and for discussing the ideas presented in this paper with us.

References

[1] Nada Amin, Karl Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. 2016. The Essence of Dependent Object Types. In *WadlerFest 2016*, Sam Lindley, Conor McBride, Phil Trinder, and Don Sannella (Eds.). Springer, Edinburgh, UK.

[2] Nada Amin and Ross Tate. 2016. Java and Scala's Type Systems Are Unsound: The Existential Crisis of Null Pointers. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2016)*. ACM, New York, NY, USA, 838–848. https://doi.org/10.1145/2983990.2984004

[3] Corrado Böhm and Alessandro Berarducci. 1985. Automatic synthesis of typed Lambda-programs on term algebras. *Theoretical Computer Science* 39 (Jan. 1985), 135–154. https://doi.org/10.1016/0304-3975(85)90135-5

[4] James Cheney and Ralf Hinze. 2003. *First-class phantom types*. Technical Report. Cornell University.

[5] Burak Emir, Martin Odersky, and John Williams. 2007. Matching Objects with Patterns. In *ECOOP 2007 – Object-Oriented Programming (Lecture Notes in Computer Science)*, Erik Ernst (Ed.). Springer Berlin Heidelberg, 273–298.

[6] Jacques Garrigue and Didier Rémy. 2013. Ambivalent types for principal type inference with GADTs. In *Asian Symposium on Programming Languages and Systems*. Springer, 257–272.

[7] Paolo G. Giarrusso. 2013. Open GADTs and Declaration-site Variance: A Problem Statement. In *Proceedings of the 4th Workshop on Scala (SCALA '13)*. ACM, New York, NY, USA, 5:1–5:4.

[8] Samuel Gruetter. 2016. Connecting Scala to DOT. *MSc semester project, EPFL* (June 2016). https://github.com/samuelgruetter/dot-calculus/blob/master/doc/Connecting-Scala-to-DOT/report.pdf

[9] Christian Hofer and Klaus Ostermann. 2010. Modular Domain-specific Language Components in Scala. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 83–92. https://doi.org/10.1145/1868294.1868307

[10] Jan Martin Jansen. 2013. Programming in the λ-Calculus: From Church to Scott and Back. In *Essays Dedicated to Rinus Plasmeijer on the Occasion of His 61st Birthday on The Beauty of Functional Code - Volume 8106*. Springer-Verlag, Berlin, Heidelberg, 168–180. https://doi.org/10.1007/978-3-642-40355-2_12

[11] Andrew Kennedy and Claudio V. Russo. 2005. Generalized Algebraic Data Types and Object-oriented Programming. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA '05)*. ACM, New York, NY, USA, 21–40.

[12] Pieter Koopman, Rinus Plasmeijer, and Jan Martin Jansen. 2014. Church Encoding of Data Types Considered Harmful for Implementations: Functional Pearl. In *Proceedings of the 26Nd 2014 International Symposium on Implementation and Application of Functional Languages (IFL '14)*. ACM, New York, NY, USA, 4:1–4:12. https://doi.org/10.1145/2746325.2746330

[13] Abel Nieto. 2017. Towards Algorithmic Typing for DOT (Short Paper). In *Proceedings of the 8th ACM SIGPLAN International Symposium on Scala (SCALA 2017)*. ACM, New York, NY, USA, 2–7. https://doi.org/10.1145/3136000.3136003

[14] Martin Odersky, Guillaume Martres, and Dmitry Petrashko. 2016. Implementing Higher-kinded Types in Dotty. In *Proceedings of the 2016 7th ACM SIGPLAN Symposium on Scala (SCALA 2016)*. ACM, New York, NY, USA, 51–60. https://doi.org/10.1145/2998392.2998400

[15] Lionel Parreaux, Amir Shaikhha, and Christoph E. Koch. 2017. Quoted Staged Rewriting: A Practical Approach to Library-defined Optimizations. In *Proceedings of the 16th ACM SIGPLAN International Conference on Generative Programming: Concepts and Experiences (GPCE 2017)*. ACM, New York, NY, USA, 131–145. https://doi.org/10.1145/3136040.3136043

[16] Marianna Rapoport and Ondřej Lhoták. 2019. A Path To DOT: Formalizing Fully-Path-Dependent Types. (2019). arXiv:cs/1904.07298 http://arxiv.org/abs/1904.07298

[17] Tiark Rompf and Nada Amin. 2015. From F to DOT: Type Soundness Proofs with Definitional Interpreters. (Oct. 2015). https://arxiv.org/abs/1510.05216v2

[18] Tiark Rompf and Martin Odersky. 2010. Lightweight Modular Staging: A Pragmatic Approach to Runtime Code Generation and Compiled DSLs. In *Proceedings of the Ninth International Conference on Generative Programming and Component Engineering (GPCE '10)*. ACM, New York, NY, USA, 127–136. https://doi.org/10.1145/1868294.1868314

[19] Gabriel Scherer and Didier Rémy. 2013. GADTs Meet Subtyping. In *Programming Languages and Systems (Lecture Notes in Computer Science)*, Matthias Felleisen and Philippa Gardner (Eds.). Springer Berlin Heidelberg, 554–573.

[20] Martin Sulzmann, Manuel Chakravarty, Simon Peyton Jones, and Kevin Donnelly. 2011. *System F with type equality coercions*. https://www.microsoft.com/en-us/research/publication/system-f-with-type-equality-coercions/

[21] Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150