Implementing Path-Dependent GADT Reasoning for Scala 3

Yichen Xu[1,*] Aleksander Boruch-Gruszecki[2] Lionel Parreaux[3]
[1]Beijing University of Posts and Telecommunications
[2]École polytechnique fédérale de Lausanne
[3]Hong Kong University of Science and Technology
linyxus@bupt.edu.cn
aleksander.boruch-gruszecki@epfl.ch
parreaux@cse.ust.hk

Abstract

Generalized Algebraic Data Types (GADT) are a popular programming language feature allowing advanced type-level properties to be encoded in the data types of a program. While Scala does not have direct support for them, GADT definitions can be encoded through Scala class hierarchies. Moreover, the Scala 3 compiler recently augmented its pattern matching capabilities to reason about such class hierarchies, making GADT-based programming practical in Scala. However, the current implementation can only reason about *type parameters*, but Scala's type system also features *singleton types* and *abstract type members* (collectively known as *path-dependent types*), about which GADT-style reasoning is also useful and important. In this paper, we show how we extended the existing constraint-based GADT reasoning of the Scala 3 compiler to also consider path-dependent types, making Scala's support for GADT programming complete and also bringing Scala closer to its formal foundations.

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

Keywords: Generalized algebraic data types, Type members, Path-dependent types, Singleton types, Scala

1 Introduction

Functional programming languages traditionally define the data structures they operate on through Algebraic Data Types (ADTs), which are essentially named sums of product types — i.e., sets of possible data constructors, each containing its own data fields. Generalized Algebraic Data Types (GADTs) [Xi et al. 2003] extend the notion of plain ADTs by allowing each data constructor to refine the type of the data type being defined, thereby encapsulating additional information about the types involved in the construction of this data. This extra type information is then retrieved during pattern matching, which requires some special reasoning capabilities from the type checker.

On the other hand, Scala is an object-oriented programming language at heart, whereby data types are defined through classes, rather than (G)ADTs. Thankfully, Scala's expressive class definitions can model a *generalization* of GADTs, allowing the encoding of not only closed single-level data types, but also open and multi-level ones as well. Listing 1 shows how to define a typical closed single-level GADT in Scala — an Expr data type, parameterized by some type A, defined as one of three constructors: integer literals and addition expressions, which only make sense when A is Int, and pairs, which similarly restrict A to the case where it is a tuple of two other types B and C. Since the types B and C do not appear in the parent Expr type, we say that they are *existentially quantified*: if we know that an Expr[T] value is constructed with the Pair constructor, then there *exist* some types a and b such that $T = (a, b)$. The use of a plus '+' sign in front of A additionally specifies that Expr is *covariant* in this type parameter, meaning that Expr[A] is a *subtype* of Expr[B] if A is itself a subtype of B.

```
sealed abstract class Expr[+A]
case class IntLit(i: Int)                 extends Expr[Int]
case class Add(l: Expr[Int], r: Expr[Int]) extends Expr[Int]
case class Pair[B, C](a: Expr[B], b: Expr[C])
                                          extends Expr[(B, C)]
```

Listing 1. Scala Definition of the Expr class using GADT.

```
def eval[T](e: Expr[T]): T = e match {
  case IntLit(i)    => i
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Pair(a, b)   => (eval(a), eval(b))
}
```

Listing 2. Scala definition of the eval function pattern matching on Expr defined in Listing 1.

Modelling GADTs through object-oriented class hierarchies is not enough to make them as useful as in languages where GADTs are supported natively — one also need to implement a pattern matching construct which can leverage the typing information uncovered while deconstructing the corresponding GADT values. Consider Listing 2 as a concrete example, which shows how to evaluates an expression of type Expr[T], for any T. This code will not compile without GADT reasoning, as the compiler will throw an type mismatch error on the IntLit case, saying that we supply an Int while a T is expected. Thankfully, GADT reasoning allows the compiler to derive the constraint T :> Int (i.e., T is a *supertype* of Int) allowing the case body to type check.

*The work is done during his internship at LAMP, EPFL.

The task of implementing GADT reasoning for Scala is complicated by Scala's support for advanced type system features, including subtyping, variance [Castagna 1995], and refinements [Cook et al. 1989], which fundamentally differ from existing languages with support for GADTs. Boruch-Gruszecki [2017] proposed a GADT-aware algorithm for verifying the exhaustiveness of GADT pattern matching and later added support for GADT reasoning to the Scala 3 compiler, which enabled many usual GADT patterns to type check in Scala 3. Parreaux et al. [2019] showed that looking at the problem through the lens of Scala's formal foundation, the Dependent Object Type (DOT) calculus [Amin et al. 2016] (and its pDOT extension [Rapoport and Lhoták 2019]), provided a way of deriving sound GADT reasoning principles for Scala, which Waśko [2020] later elaborated. The core of this idea is to use type members to represent the existential types that arise from GADTs and to use bounds on these type members and path-dependent types to represent and leverage the additional information attached to them.

A stable path p is a path made of "stable" values, such as function parameters and immutable object fields (for instance $p = param.field_1.field_2$). A path-dependent type is a type of the form p.type or $p.X$ — the former is a *singleton type* representing the type of the specific value of p, and the latter represents the specific type X that "lives" in p, where X is declared as a possibly abstract type member in one of the parent types of p. Since they depend on the runtime value of variables, path-dependent types can be viewed as a form of *dependent types*. Boruch-Gruszecki's original implementation of GADT reasoning for Scala 3 was limited to dealing with type parameters, leaving general path-dependent types out of the picture. This was unfortunate for two reasons: first, because path-dependent types provide a very general and powerful way of abstracting over types in Scala, and leaving them out makes Scala's support for GADTs patently incomplete; second, because as explained above our current understanding of GADT reasoning in Scala is *based* on the theory of path-dependent types, so not handling them in Scala 3 seems like a major missed opportunity.

It is possible to infer GADT-style constraints for type members and singleton types based on the typing information that arises from pattern matching, but since this functionality is currently missing from the compiler, many dependently-typed programming patterns that are provably sound cannot be type-checked as is. Listing 3 gives such an example, which requires the combined use of GADT reasoning on both type parameters and type members. In this example, we refine the type of the Expr data type to also include some size information to record the number of nodes in the expression tree, encoded through a Size *type member*.[1] The

[1]This encoding is convenient as it does not "pollute" Expr with additional type parameters. Users who care about the sizes of manipulated expressions may use 'sized' refinements, as in 'Expr[A] sized N', i.e., sized[Expr[A], N], which is a shorthand for the type refinement Expr[A] & {type Sized = N}, and users who are not interested in size information may simply use Expr[A], whereby the expression's size is existentially quantified.

```
trait Expr[+X] { type Size <: Int }
case class IntLit(x: Int) extends Expr[Int] { type Size = 1 }
case class Add[N1 <: Int, N2 <: Int](
  e1: Expr[Int] sized N1,
  e2: Expr[Int] sized N2
) extends Expr[Int] { type Size = N1 + N2 + 1 }
```

(a) Definition of a sized variant of Expr.

```
def swap[X](tree: Expr[X]): Expr[X] sized tree.Size =
  tree match {
    case IntLit(x) => IntLit(x)
    case Add(l, r) => Add(swap(r), swap(l))
  }
```

(b) Definition of the swap function on sized Exprs.

```
type sized[X, N <: Int] = X & { type Size = N }
```

(c) Definition of the sized shorthand operator.

Listing 3. A piece of Scala code using both type parameter and type members GADT reasoning. It is sound but can not type check in the current version of Dotty. Note that the example uses the compile time integer operation type '+' introduced by Scala 3 in package scala.compiletime.ops.int.

pattern match shown in Sublisting 1b is sound but cannot type check without path-dependent GADT reasoning. To see this, consider the IntLit case, for which the compiler infers that X = Int via type parameter GADT reasoning, and should additionally know that tree.Size = 1 via type member reasoning, allowing IntLit(x) to be recognized as a value of type Expr[X] sized tree.Size. To work over this limitation, one is currently required to painstakingly add many path-dependent type annotations to the program, following the approach described by [Parreaux et al. 2019], which results in needless boilerplate and obscures the meaning of the program. On the theory side, Scala type parameters themselves are modeled in DOT through type members, and DOT allows GADT reasoning based on information implied by constraints over type members [Parreaux et al. 2019; Waśko 2020]. Thus, supporting GADT reasoning for type members will bring the compiler closer to its formal foundations.

In this paper, we propose an implementation of GADT reasoning for type members in Dotty.[2] We lay out the current status of GADTs in Dotty, present our implementation for path-dependent GADTs, briefly discuss its soundness, and suggest future work in the same vein. It turns out that we can reuse much of the existing logic for GADT reasoning for type parameters in our implementation. The specific contribution of this paper are the following:

- We present a way to implement relatively efficient path-dependent GADT reasoning in Dotty, with an in-depth description of the GADT reasoning framework implemented in the compiler.

[2]Dotty is the internal name of the new Scala 3 compiler.

- We explain the interplay between path-dependent types and GADT reasoning in Scala from both the practical and the theoretical perspectives.

2 Background

In this section, we briefly introduce the background knowledge, to get the readers familiar with the topic. We will start with the definition and informal discussion of general GADTs, followed by the implementation of GADTs in Scala, and finally introduce the basic ideas of Scala's type members.

Generalized Algebraic Data Types. Generalized Algebraic Data Types [Xi et al. 2003] is an extension to parameterized ADTs that enables the data constructors to specialize type parameters of the classes, and makes it possible to discover constraints over type parameters when performing pattern matching. These constraints allow the compilation of code snippets that will not compile before without GADT reasoning. As shown before, Listing 2 gives an example where inferred GADT constraints are crucial in the compilation of the code. Additionally, GADTs enable *existential quantification*. To see an example of this, if we have a TupleTag of type Tag t defined in Listing 4, we know that there *exists* two types a and b, such that $t = (a, b)$.

```
trait Tag[T]
case class TupleTag[A, B]() extends Tag[(A, B)]
```

Listing 4. Definition of Tag using GADT. Existential quantification is involved in the TupleTag constructor.

Implementing GADTs in Scala. Implementing Generalized Algebraic Data Types is much more complicated than that in other languages (like Haskell and OCaml) because of the sophisticated class system it supports. Listing 5 illustrates a pitfall brought by the variance of class type parameters [Giarrusso 2013; Parreaux et al. 2019]. Counterintuitively, the inequality $A <: B$ does not hold in the body because the class type parameters are defined to have variance and can be further refined. For instance, it is possible to set A = Any and B = Nothing and create new Identity[X] & Func[Nothing, Any] to satisfy the match case. If we wrongly assume that $A <: B$ in this example, we will have Any <: Nothing and end up with a soundness hole.

```
trait Func[-A, +B]
class Identity[X] extends Func[X, X]

def foo[A, B](func: Func[A, B]) = func match {
  case _: Identity[c] =>
    val b: B = ??? : A
      // error! We can not assume A <: B here
}
```

Listing 5. Code illustrating a pitfall in Scala GADT reasoning due to type parameter variance.

In essence, Scala GADT constraints are inferred based on the *cohabitation* of types [Parreaux et al. 2019]. Formally, type A and B are cohabitated iff there exists a value x that can be ascribed to both types. The existence of such a value $x : A \,\&\, B$ can be viewed as proof of cohabitation. For example, in the IntLit case of Listing 2, we can safely assume that e is of type Expr[T] & IntLit, implying that Expr[T] and IntLit are cohabitated, and from this, we can infer the GADT constraint T :> Int. Previous work formalizes GADT reasoning in DOT by modeling type parameters and inheritance with type members [Parreaux et al. 2019]. Following this work, there are further attempts to formalize and analyze this topic under the framework of pDOT [Rapoport and Lhoták 2019; Waśko 2020].

Type members. As an essential language feature of Scala [Amin et al. 2014; Odersky et al. 2016], Scala objects can contain type members apart from fields and methods. Type members can be abstract in the base classes and be instantiated or refined in derived subclasses. Importantly, all type members of a class must be instantiated when creating an instance of that class. Therefore, the type member of an inhabited class can be viewed as proof for the subtyping relation between its lower and upper bounds. To make use of the type members, there should exist a way to refer to them from runtime objects, which can be viewed as one form of dependent type. Scala uses *path-dependent types* to reference type members from values, which is in the form of $x_1.x_2.\cdots.T$. Formally, it is defined as a chain of selections from an immutable variable, selecting immutable fields along the way and ending with a type member [Amin et al. 2014; Rapoport and Lhoták 2019]. Type members of local variables that are of extensible classes should be considered *abstract*, which means that GADT constraints can be inferred for them based on pattern-matching just like what we do for type parameters. As a missing part in the current implementation of GADT in Dotty, implementing GADT constraint inference for path-dependent types can benefit real-world use cases and bring the compiler closer to its formalism.

3 Current Implementation of GADT

In this section, we give an introduction to the implementation of GADT constraint inference in the current compiler by answering two design questions: (1) how constraints for type members are stored in the compiler and (2) how GADT constraints are inferred, which account for data structures and program logic of the implementation respectively. Note that the current implementation will only derive GADT bounds for type parameters, but not type members. Our contribution to implementing GADT reasoning for type members will be presented in Section 4 in detail.

3.1 Storing Constraints for Type Parameters

In Dotty, GADT constraints are stored in a class named GadtConstraint, where two parts of information are stored: (1)

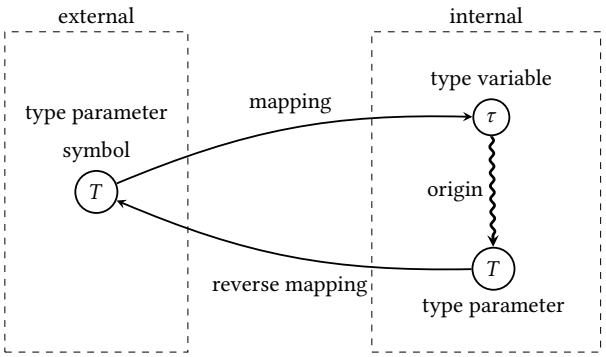

Figure 1. The mappings between constrained type parameters and their internal representations in GadtConstraint.

the mapping between type parameters and internal representations, and (2) derived GADT constraints over the internal representations in an OrderingConstraint.

GadtConstraint is intended to map the constrained type parameters into the internal representations used by OrderingConstraint, and rely on OrderingConstraint to store and manage the constraints. OrderingConstraint is an immutable data structure that can record and manage constraints for type parameters, but it is intended to be used for type inference and cannot directly store GADT constraints. Therefore, we implement GadtConstraint to adapt the functionalities provided by OrderingConstraint. GadtConstraint also makes use of the additional functionalities provided by ConstraintHandling, which keeps an instance of OrderingConstraint in a mutable field. For brevity, we will not present ConstraintHandling – it merely defines "surface" methods which require mutating the current constraint. In the following part, we first describe the structure of mappings used in GadtConstraint, which serve as *bridges* between the GADT-constrained external type parameters and the internal representations for constraint handling. Then, we explain the logic for the addition and retrieval of GADT bounds utilizing stored mappings.

3.1.1 Mappings. The internal representations of the external constrained type parameters are the internal type parameters in the OrderingConstraint and its associated type variables. Type variables are a data structure for constraint inference, and they can be viewed as mutable trackers of type parameters in the OrderingConstraint. The GadtConstraint class will store the two-way mappings between the external type parameters and their internal representations. As illustrated in Figure 1, there will be two data members in the GadtConstraint to store the mappings: mapping and reverseMapping. The signature for mapping is SimpleIdentityMap[Symbol, TypeVar][3] that maps the symbol of the

[3]SimpleIdentityMap is a linear map based on instance identity (using the eq method in Scala).

```
sealed abstract class GadtConstraint {
  def addBound(sym: Symbol, bound: Type, isUpper: Boolean)(using
      Context): Boolean

  def bounds(sym: Symbol)(using Context): TypeBounds

  def isLess(sym1: Symbol, sym2: Symbol)(using Context): Boolean

  def addToConstraint(syms: List[Symbol])(using Context):
      Boolean
}
```

Listing 6. GadtConstraint interfaces for adding and retrieving constraints, and registering type parameters.

constrained type parameter to the internal type variable; reverseMapping is of type SimpleIdentityMap[TypeParamRef, Symbol] that maps the reference to an internal type parameters back to the external symbols.

3.1.2 Adding and retrieving GADT bounds. As listed in Listing 6, GadtConstraint provides interfaces for adding and retrieving GADT bounds. For adding bounds, the addBound function will first try to transform the type parameter referenced by the symbol or in the bounds to their internal representations, and then call the constraint handling logic. For retrieving bounds, bounds will return all GADT-inferred bounds for the type parameter, but the constraints of the ordering between other GADT-constrained type parameters will not be included, since including this information will require subtype checking, and may thus result in infinite loops interacting with TypeComparer. In cases where this information is necessary, isLess is used to retrieve the ordering constraints between two constrained type parameters. Both functions will first try to transform the external symbols into the internal representations, and then query bounds for them.

A key implementation concern here is that we must ensure to transform all constrained type parameters into their internal representations when adding and querying constraints for them, and transform the internal type variables back to the external types to prevent the leaking of internal representations.

3.1.3 Type parameter registration. Before we can actually record and retrieve constraints for type parameters, we explicitly tell the GadtConstraint which type parameters are constrainable by *registering* them. The addToConstraint method can register type parameters into the constrainer. Specifically, the method will create internal representations for the type parameters to be registered, and update the mappings to record the relationship between external types and internal variables. Additionally, it will also properly handle the inter-dependency between registered type parameters. To see a concrete example, consider we are registering a type parameter A with bounds $A <: T$, where T is a type parameter that is being registered together with A, or already being in the GADT constraint handler. Then the T should be

```
trait Container[+T]
case class IntList() extends Container[List[Int]]

def foo[T](e: Container[List[T]]): T = e match {
  case IntList() => 0
}
```

Listing 7. Scala code illustrating GADT constraint inference.

substituted to its internal representation when we are storing bounds for A internally.

The type parameter registration is triggered in Typer. When we are typing a method or class definition with a type parameter list, we will register the type parameter introduced by them.

3.2 Inferring GADT Constraints

With the utilities for storing and handling GADT constraints for type parameters, we are now concerned with the way to actually inference the constraints. In Dotty, the class named PatternTypeConstrainer implements the GADT constraints inference, relying on the subtyping comparison logic implemented in TypeComparer. The inference logic is called when we are typing the pattern match, with the scrutinee and pattern types supplied in the arguments. The inference is done in two main steps: (1) firstly, since the GADT reasoning is actually based on the cohabitation of the scrutinee and pattern type, we will find out the subtyping relations necessary for the cohabitation condition to hold; (2) then, we call TypeComparer on these subtyping relations to get the constraints.

Listing 7 gives a concrete example for GADT constraint inference. Firstly, to satisfy the cohabitation condition between Container[List[T]] and IntList, the relation List[Int] <: List[T] must hold, since IntList is a case class [Odersky et al. 2016] that is impossible to extend and the type parameter is covariant [Castagna 1995]. Then we call TypeComparer on this subtyping relation, and will end up with the constraint T :> Int being inferred and recorded.

4 Implementing Path-Dependent GADT

In this section, we present our implementation of GADT reasoning for path-dependent types. We will start with an overview of use cases with path-dependent GADT, followed by an explanation of the work we do to enable the recording and handling of GADT constraints for type members. Then, we present a primitive implementation for recording equalities between singleton types based on pattern match. Finally, we discuss the soundness of the implementation and prospect future work. Built upon the current GADT reasoning framework, the implementation for type members extensively reuses the existing codebase for type parameter GADT.

```
trait Tag { type T >: Int }
def f(e: { type T }): e.T = e match {
  case e1: Tag => 0
}
```

(a) Deriving bound between type members.

```
trait Tag { type T }
def f(p: Tag, m: Expr[p.T]): p.T = m match {
  case IntLit(i) => i
}
```

(b) Constraining path-dependent types.

```
trait IntTag { type T >: IntExpr }
def f[X](e: { type T <: Expr[X] }): X = e match {
  case e1: IntTag => 0
}
```

(c) Type members as subtyping proofs.

Listing 8. Scala code illustrating the use cases of path-dependent GADT reasoning.

4.1 Use Cases

GADT constraints of path-dependent types are useful in many real-world scenarios, and the lack of them will prevent a certain number of sound Scala codes from compiling. Here we list a series of key use cases of path-dependent GADT reasoning.

Case 1. Deriving bounds between type members. Based on the pattern match, GADT constraints can be inferred for type members from the cohabitation of the scrutinee and pattern type. Consider the example given in Listing 2a. Since, in the case body, we actually have e <: Tag & $\{T\}$, which implies e <: $\{T$ <: Int$\}$, we can derive the GADT bound $e.T$:> Int for the path-dependent type $e.T$.

Case 2. Constraining path-dependent types. Since the type members in extensible classes can be further refined in subclasses, the path-dependent types referencing type members from these classes can be viewed as *constrainable*. In other words, we can inference GADT constraints for them just like what we do for type parameters. Listing 2b gives such an example. The path-dependent type $p.T$ is constrainable, and we can derive the GADT constraint $p.T$:> Int for it.

Case 3. Type members as subtyping proofs. Additionally, the type members of an inhabited type can serve as subtyping proofs [Amin et al. 2016, 2014]. Therefore, we can derive more GADT constraints based on the subtyping relation. Inside the case body of Listing 2c, we know that the structural type IntTag & $\{T$ <: Expr[X]$\}$, which can be simplified as $\{T$:> IntExpr <: Expr[X]$\}$, is inhabited. Therefore, the type member T in the structural type can serve as proof for IntExpr <: Expr[X], leading to the GADT constraint X :> Int.

A real-world example. To better see the real-world benefits brought by path-dependent GADT reasoning, we provide an example in Listing 9. The example defines a type class TupOf to encode the shape and element type of a homogeneous (i.e., uniformly-typed) tuple. In its essence, a type class instance p of TupOf[T, A] for type T <: Tuple

```scala
abstract class TupOf[T, +A] {
  type Size <: Int
}
```

(a) Definition of the TupOf type class.

```scala
object TupOf {
  given Empty: TupOf[EmptyTuple, Nothing] with
    type Size = 0
  final given Cons[A, T <: Tuple, N <: Int]
    (using p: T TupOf A sized N): TupOf[A *: T, A] with
      val p0: T TupOf A sized N = p
      type Size = S[N]
}
```

(b) Implicit derivation for the TupOf type class.

```scala
enum Vec[N <: Int, +A] {
  case VecNil                          extends Vec[0, Nothing]
  case VecCons[N0 <: Int, A](head: A, tail: Vec[N0, A])
                                       extends Vec[S[N0], A]
}
```

(c) Definition of the Vec class.

```scala
def tupToVec[A, T <: Tuple](xs: T)(using p: T TupOf A): Vec[p.
     Size, A] =
  p match {
    case _: TupOf.Empty.type => VecNil
    case p1: TupOf.Cons[a, t, n] =>
      VecCons(xs.head, tupToVec(xs.tail)(using p1.p0))
  }
```

(d) Definition of the tupToVec function.

Listing 9. A real-world code example involving type classes and dependent programming. It relies on path-dependent GADT reasoning to pass type checking. The definition in Listing 2c uses the enum[4] syntax, and the example operates on the Tuple[5] class, both introduced in Scala 3.

can serve as proof that any tuple of this type is of size p.Size and all its elements are of type A. Thanks to Scala's implicits system [Odersky et al. 2016] (whose syntax was recast in terms of the 'given' and 'using' keywords in Scala 3), we can automatically derive type class instances for tuples. Vec is a dependently-typed vector with its length encoded as a type parameter. The tupToVec function pattern matches on the TupOf instance to extract information about the tuple's shape and element type, and then utilizes this information to convert the tuple into a Vec. For instance, tupToVec(0, 1, 2), which is syntax sugar for tupToVec(0 *: 1 *: 2 *: EmptyTuple), where *: and EmptyTuple are the inductive tuple constructors, results in a vector typed as Vec[Int, 3]. Apart from type parameter GADT reasoning, GADT constraints of path-dependent types are crucial for the compilation of the example. Specifically, in the TupleOf.Empty case, we can derive the GADT constraint that p.Size = TupOf.Empty.Size = 0. This allows us to supply a VecNil : Vec[0, Nothing] for return type Vec[p.Size, A]. This example illustrates how path-dependent GADT reasoning can benefit type reasoning in real world scenarios, showing that our work improves the experience of dependently-typed programming in Scala.

[2] https://dotty.epfl.ch/docs/reference/enums/enums.html
[3] https://www.scala-lang.org/2021/02/26/tuples-bring-generic-programming-to-scala-3.html

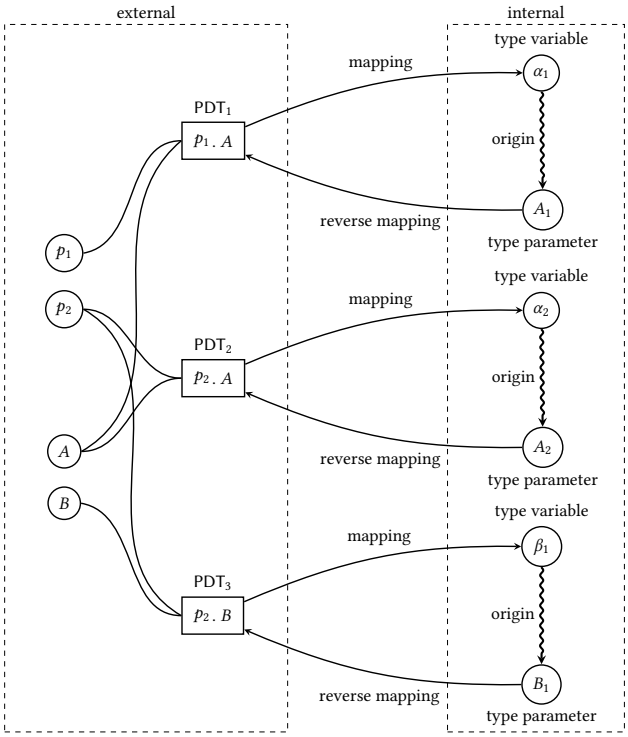

Figure 2. The mappings between constrained path-dependent types and their internal representations in GadtConstraint.

4.2 Extending GADT to Path-Dependent Types

4.2.1 Storing and managing GADT constraints. The storage of GADT constraints for path-dependent types is implemented by extending GadtConstraint built for type parameters. Recap that we maintain mappings between the symbols of constrained type parameters and their internal representation in GadtConstraint. For path-dependent types, only recording the symbol of the type member will not be enough, since a path-dependent type is not only determined by its type member symbol, but also the prefix. Therefore, to properly maintain the mapping between path-dependent types and internal type variables, we will map the prefix together with its type member symbol to the internal data structures.

Specifically, since the path-dependent types are represented as a TypeRef(prefix: Type, designator: Designator) in Dotty, we use a mapping of type SimpleIdentityMap[SingletonType, SimpleIdentityMap[Symbol, TypeVar]] to map the path dependent types to their type variables. Here we use the SingletonType to represent the prefix, since, in order to form a valid path-dependent type, the prefix must be a singleton. Additionally, it is always possible to query the type member symbol using the designator, so we stick to the Symbol representation to ensure the uniqueness and easy manipulation. On the reverse side, we use the mapping type

SimpleIdentityMap[TypeParamRef, TypeRef] to map the internal type parameters back to their corresponding path-dependent type outside.

With the mappings maintained between the path-dependent types and the internal representations, we can implement the methods for constraint recording and retrieval similarly for path-dependent types.

4.2.2 Constraint Inference.

Path-dependent GADT reasoning can be classified into two main cases. Firstly, we would like to inference GADT constraints based on the type members of the scrutinee and the pattern, as in Listing 2a and 2c. Since the pattern and scrutinee will inhabit the same intersection type in the case body, we assume that the shared type members of the pattern and scrutinee types are the same type as each other and then derive constraints based on the subtyping relations. Specifically, in Listing 2a, for $e : \{T\}$ and $e_1 : \{T <: \mathsf{Int}\}$, we extract constraints from the subtyping relations $e.T <: e_1.T$ and $e_1.T <: e.T$, getting $e.T <: \mathsf{Int}$ as a GADT bound.

Secondly, we would like to constrain path-dependent types when no pattern matching are performed on their path but GADT constraints can be inferred, as in Listing 2b. This can be achieved with existing GADT reasoning logic for type parameters, as long as the path-dependent types are recognized by the GadtConstraint as constrainable and have internal representations created for them. We ensure this by implementing the *just-in-time* type registration scheme in GadtConstraint, which will be explained in detail in the following section.

4.3 Registering Path-Dependent Types

Unlike type parameters, whose registration can be done when the compiler is typing a definition with a type parameter list, the registration of path-dependent types can happen in more cases and can involve subtle issues. We implement two path-dependent type registering schemes: (1) registering when pattern matching and (2) just-in-time registration.

Registering when pattern matching. When doing pattern matching, we will scan the type member list of the scrutinee and pattern type, and create internal representations for the corresponding path-dependent types. For example, when constraining the pattern match in Listing 2a, we will register $e.T$ and $e_1.T$. Note that we will also trigger the constraint inference logic based on the subtyping relations between path-dependent types we have just registered.

Just-in-time registration. Since in Listing 2b, the path in $p.T$ is not pattern matched on, the internal representation of $p.T$ will not get created by the previous scheme. However, it is still possible to record GADT constraints for them. To this end, we propose the just-in-time (JIT) scheme for creating internal type variables. For a better understanding of the JIT scheme, we first describe the way to infer GADT bounds in TypeComparer. Internally, the comparer will examine deeply into the subtyping relation to verify it. When the comparer finds it possible to add bounds for the compared types, it checks whether the GADT inferencing is enabled, and the compared type is constrainable in GadtConstraint. If so, the comparer will call the GadtConstraint to record bounds for the type. Normally, when GadtConstraint is asked whether an unseen path-dependent type is abstract, it will return false to prevent the bounds from being recorded. However, with the JIT scheme enabled, we will check whether the path-dependent type can be constrained, create internal representations for it *just-in-time*, and return true if possible. For example, in Listing 2b, when the comparer asks whether $p.T$ is registered, GadtConstraint will create type variables for $p.T$ and allow the GADT bounds to be recorded for it. The implementation of the JIT scheme will make it possible to record GADT bounds for path-dependent types in almost all possible cases. However, it will bring a subtle issue when pattern-matching on union types.

```
trait Tag { type T }
trait T1
trait T2 extends T1
def f(p: Tag, e: { type T = p.T }): p.T = e match {
  case _: ({ type T <: T1 } | { type T <: T2 }) => new T2 {}
}
```

Listing 10. Scala code snippet illustrating the issue of union types.

Issue of the JIT scheme with union types. Listing 10 illustrates the issue of pattern-matching on union types, in which the scrutinee $e : T$ is matched against a union type $\{T <: T_1\} \mid \{T <: T_2\}$. To extract GADT constraints from the pattern matching, the compiler will branch into two components in the union type, derive GADT bounds in each of the directions, and comparer the constraints in each way to find the more general one. However, with the JIT scheme enabled, the type $p.T$ will be registered *after* branching into each direction, producing two different internal representations for the same $p.T$ type in the two derived constraints. The current compiler implementation can not properly handle the case where two different internal type variables track the same external type, thus preventing Listing 10 from compiling. Therefore, we rewrite the constraint comparison logic to properly handle the case brought by the JIT scheme, by discovering these sibling type variables referring to the same type through inspecting the mappings.

Unbound patterns. Another issue will happen when *unbound* patterns are involved. For example, in the pattern matching of Listing 11, we will end up unifying $e.T$ to the type member S in the pattern being a supertype of Int. Ideally, when comparing Int to $e.T$ in the body, we will first retrieve the bound that $e.T$ is equal to the type member S in the pattern, and then find out that S is a supertype of Int, making the code to pass the type checking. However, since the pattern is unbound, it is impossible to refer to its

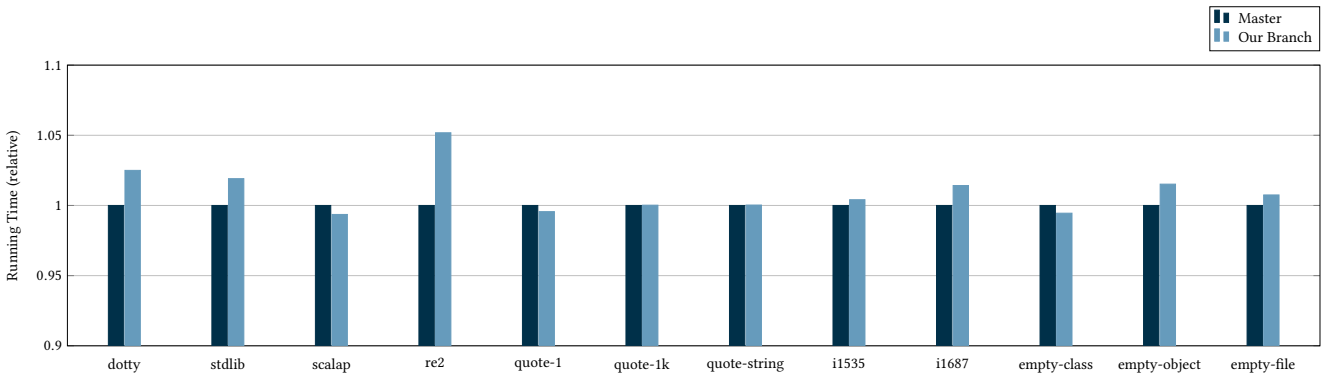

Figure 3. Performance impact of our implementation. It shows the running time of all 12 tests in the Dotty benchmark[6] on master and our branch respectively. Running time value is relative to that on the master.

```
trait Tag { type S >: Int; type T = S }
def f(e: { type T }): e.T = e match {
  case e1: Tag => 0
}
```

Listing 11. Scala code snippet illustrating the issue of unbound patterns.

type member S outside the GADT constraint handler. The naive implementation will simply leak the internal representation of S, which will not be recognized by the comparer, thus preventing the code from compiling. To solve this issue, we create a skolem to represent the unbound pattern value, and map the internal variables of the pattern's type members onto the skolem type. This will make it possible to reference the type members even if the pattern is not bound to a symbol.

4.4 Recording Equality between Singletons

Apart from deriving constraints for the type members of the scrutinee and pattern, it is also safe to assume that the scrutinee and the pattern are referring to the same value in the case body, implying that their singleton types are equal. Listing 12 gives an example of GADT constraint inference over singleton types. In this snippet, a is matched against b, then c, implying that the singleton types from a, b and c are all equal. In other words, the three symbols refer to the same value in the case body. Similar reasoning applies for x and y.

```
a match {
  case b: X1 =>
    a match {
      case c: X2 =>
        x match {
          case y: X3 =>
            // Infer that a.type == b.type == c.type
            //   and x.type == y.type
            val t0: b.type = c } } }
```

Listing 12. Inferring GADT constraints on singleton types.

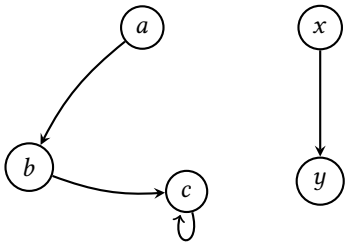

Figure 4. Visualization of the disjoint set data structure for recording singleton equalities in Listing 12.

In our implementation, we take a simple yet efficient approach to record the equalities between singleton types. Specifically, since the constraints are all equalities, which means that the singleton types will form a family of equivalent classes after a series of pattern matching, we store the equalities in the disjoint set data structure. When constraining a pattern match case whose scrutinee and pattern are all bound to symbols, we record the equality between the two singleton types by merging them in the disjoint set. Then, in the type comparer, we can query the stored singleton equalities when two singleton types are compared. Figure 4 visualizes the state of the disjoint set within the innermost match case. This allows the type comparer to recognize that $c <: b.type$.

5 Experimental Evaluation

In this section, we empirically evaluate our implementation in two aspects: conservativeness and performance.

To verify that our implementation is conservative, in the sense that it neither breaks existing compiling Scala code nor unexpectedly makes unsound code compile, we run all compilation unit tests on the modified compiler. The test suite includes both positive and negative tests and covers a broad range of functionalities and use cases of the compiler.

[4] https://github.com/lampepfl/bench

Our branch is able to pass all of the 8345 unit tests, suggesting that our implementation is conservative.

To investigate the performance impact of our modification, we run the test suites in the Dotty benchmark and inspect the changes in the running times. We report the relative running time before and after our modification in Figure 3. From the figure, we observe that our modification seems to slightly slow down the compiler in certain cases. It brings a rise of running time by 5% in the re test, while in most tests the increase is about 1% ∼ 2%. The drop of performance can be attributed to the additional GADT reasoning logic we implemented for type members and singleton types. The added logic will be triggered each time the compiler is typing a pattern match case, which can contribute negatively to the overall performance. However, it should be noted that we have not yet expended any efforts into optimizing our implementation, since we have been focusing on correctness first. So there is still much room for performance improvements to be explored, possibly making the overhead of path-dependent GADT reasoning close to negligible in the future.

6 Discussion

In previous sections, we lay out the framework of GADT reasoning framework in Dotty, and present our extension to the framework to enable GADT reasoning for path-dependent types and singleton types. Apart from the additional functionality it brings to the compiler that will benefit real-world use cases, our work also brings Dotty closer to the dependent object type system, the theoretical foundations of the compiler. In this section, we wrap up our presentation by informally discussing the soundness of our approach to path-dependent GADT constraint inference, and noting the future work in this direction.

6.1 Relation with Theory

Apart from benefiting real-world usages, our implementation also brings the Dotty compiler closer to its formal foundations.

The essence of GADT reasoning in Scala is to derive constraints from cohabitation. On the formal side, DOT [Amin et al. 2016] already takes into account type information following from cohabitation – that is, from values of types of the form $T \,\&\, U$ which are in scope. More specifically, DOT can leverage information present in the bounds of any values bound in the current typing context. This reasoning crucially uses the DOT type member typing rules, which states that path-dependent types are bounded by the bounds of the corresponding type member, and on subtyping transitivity rule. For instance, given a value x of type x: {A :> S <: T} in scope, then the relationships S <: x.A and x.A <: T hold, and by transitivity so does S <: T.

```
(e1: { type T :> A1 <: B1 }) match {
  case e2: { type T :> A2 <: B2 } => // ...
}
```

Listing 13. A code example for discussing soundness of path-dependent GADT constraint inference.

Ideally, on the practical side, the compiler should also make use of this information found in the bounds of the variables in scope. However, in the general case, because DOT supports complex recursive type specifications (just like Scala), these rules make type checking in DOT undecidable, and an algorithmic version of DOT thus *has to* somehow restrict them [Hu and Lhoták 2019; Nieto 2017]. The current approach taken by the Scala 3 compiler until now was to simply completely disregard any information that could be obtained from the bounds of the variables in scope (making the subtyping relationship of Scala 3 not transitive). In order to leverage such bounds information, users would have to use explicit type annotations, essentially proving transitivity to the compiler every time — for instance, given x: {A :> S <: T} in scope, if one wanted to upcast a value e of type F[S] to type F[T], assuming some covariant type constructor F, one would have to write 'e: F[x.A]'.

What we propose and implement in this paper is a compromise, a sweet spot between full leverage of bounds information (DOT, undecidable) and no leverage of bounds information at all (current Scala 3, to restrictive). Indeed, most cases that are actually useful in the context of GADT-style programming are simple and do not involve recursive types, so we can leverage their bounds without getting into decidability problems. We also choose to only consider bounds information arising specifically from values that are being pattern-matched because this is typically where new typing information comes into play in user programs. Trying to leverage all information present in all variables in scope at any time would probably impose too much of a performance penalty on the compiler, for relatively little gain. We reserve investigating this possible extension as future work.

In a nutshell, our work helps the compiler infer and make use of additional subtyping relationships that demonstrably hold in the formal system, thus bringing the implementation closer to the theory.

6.2 Soundness

We discuss the soundness of the approach adopted in our implementation to derive GADT constraints for path-dependent types. In the pattern matching case given in Listing 13, we will extract GADT bounds from the subtyping relations $e_1.T <: e_2.T$ and $e_2.T <: e_1.T$. More specifically, this will end up doing the following operations:

1. Add orderings between type members: $e_1.T <: e_2.T$ and $e_2.T <: e_1.T$.
2. Further constrain type members with propagted bounds: $e_1.T <: B_2$, $e_1.T :> A_2$, $e_2.T <: B_1$ and $e_2.T :> A_1$.

```
class Module[*[_, _]] {
  sealed trait Box[A]

  case class Box2x2[A, B, C, D](value: (A * B) * (C * D))
      extends Box[(A * B) * (C * D)]

  def unbox2[X, Y](box: Box[X * Y]): (X * Y) =
    box match {
      case Box2x2(v) => v
    }
}
```

Listing 14. A code snippet showing the necessity of storing GADT constraints for concrete types. It is taken from issue #13074[7] in the Dotty repository.

3. Derive GADT constraints based on the subtyping relations: $A_1 <: B_2$ and $A_2 <: B_1$.

It is trivial to see that the constraints from the first and the second operation is necessary from the cohabitation, since $e_1.T$ and $e_2.T$ refer to the same path-dependent type after the pattern matching. Next, we informally show that the two subtyping relations in the third point are necessary from the cohabitation condition, and thus deriving constraints from them are sound, too.

First of all, we point out that, after pattern matching, we will have a variable e of type $\{T :> A_1 <: B_1\} \,\&\, \{T :> A_2 <: B_2\} = \{T :> A_1 \mid A_2 <: B_1 \,\&\, B_2\}$. As stated before, the existence of an instance of a class can serve as subtyping proof between the lower and upper bounds of its type members. Therefore, from the existence of variable e we can know that $A_1 \mid A_2 <: B_1 \,\&\, B_2$, which implies $A_2 <: B_1$ and $A_1 <: B_2$.

6.3 Future work

Here, we further note the future work that can be done in this direction.

Unified GADT reasoning. Currently, despite their shared framework, GADT reasoning for type parameters and type members have their separate data structure and logic. Since type parameter GADT can be modeled with type members [Parreaux et al. 2019; Waśko 2020], it is possible to unify the handling logic for the two families of types, which will not only simplify the codebase, but also improve the consistency between the compiler and the theory.

Recording GADT constraint for concrete types. The JIT scheme for registering path-dependent types introduces a new mechanism where we create entries for types just-in-time when constraints can be inferred for them. Similarly, we can store GADT constraints for concrete types to improve type reasoning. In Listing 14, since we can not assume the injectivity of the * operator, the compiler will end up with no bounds inferred for the type parameters, preventing the code from compiling. However, the code snippet itself is sound, since it is sound to assume $A * B * C * D = X * Y$ even if no bounds can be derived for each of the type parameters. Therefore, it is natural to think about the possibility to store GADT constraints for concrete types when GADT reasoning ceases due to the lack of injectivity. The implementation would be non-trivial because we have to store and retrieve bounds for arbitrary type expressions, but this functionality will benefit type reasoning in real-world code examples.

Constraint inference for a wider range of cohabitation. The essence of GADT reasoning is deriving necessary constraints implied from cohabitation proof. The scrutinee and pattern in the pattern matching expression is just a source of cohabitation. By principle, any instance of an intersection type can be viewed as cohabitation proof, and thus making it possible to derive constraints from the cohabitation. For example, the existence of a value $x : S \,\&\, T$ in the scope can be viewed as proof for the cohabitation between S and T. Therefore, whenever a value of an intersection type is introduced into the scope, we can derive GADT constraints based on the cohabitation just like what we do in pattern matching. There are pros and cons for such constraint inference functionality. It is true that such functionality may lead the compiler to infer a large number of unused bounds, and can cause performance problems. What's more, though it will result in boilerplates, the users can always trigger GADT reasoning on the intersection type by manually pattern matching on the value. On the other hand, it will enable the compiler to discover *all* possible constraints automatically from type cohabitation, and the richer bound information, in many cases, can benefit type reasoning, reduce boilerplates and improve user experience. Furthermore, the GADT reasoning can be generalized and unified to the general framework for constraint derivation from cohabitation, and bring the compiler implementation even closer to the theoretical side.

7 Conclusion

As a missing piece of the puzzle, path-dependent and singleton GADT reasoning is not implemented in the current Dotty compiler but will benefit many real world use cases and shorten the gap between the compiler implementation and the formal foundations. To this end, we propose to implement GADT reasoning for path-dependent and singleton types in Dotty. This paper hopes to present the GADT implementation in Dotty, and get those who are interested in the technical details of Dotty internals familiar with related data structures and program logic, to facilitate future development.

References

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.*

Nada Amin, Tiark Rompf, and Martin Odersky. 2014. Foundations of Path-Dependent Types. *SIGPLAN Not.* 49, 10 (Oct. 2014), 233–249. https://doi.org/10.1145/2714064.2660216

[7] https://github.com/lampepfl/dotty/issues/13074

Aleksander Boruch-Gruszecki. 2017. *Verifying the totality of pattern matching in Scala.* Master's thesis. Wrocław University of Science and Technology.

Giuseppe Castagna. 1995. Covariance and Contravariance: Conflict without a Cause. *ACM Trans. Program. Lang. Syst.* 17, 3 (May 1995), 431–447. https://doi.org/10.1145/203095.203096

William R. Cook, Walter Hill, and Peter S. Canning. 1989. Inheritance is Not Subtyping. In *Proceedings of the 17th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '90)*. Association for Computing Machinery, New York, NY, USA, 125–135. https://doi.org/10.1145/96709.96721

Paolo G. Giarrusso. 2013. Open GADTs and Declaration-Site Variance: A Problem Statement. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. Association for Computing Machinery, New York, NY, USA, Article 5, 4 pages. https://doi.org/10.1145/2489837.2489842

Jason Z. S. Hu and Ondřej Lhoták. 2019. Undecidability of D<: And Its Decidable Fragments. *Proc. ACM Program. Lang.* 4, POPL, Article 9 (Dec. 2019), 30 pages. https://doi.org/10.1145/3371077

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

Martin Odersky, Lex Spoon, and Bill Venners. 2016. *Programming in Scala: Updated for Scala 2.12* (3rd ed.). Artima Incorporation, Sunnyvale, CA, USA.

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019. Towards Improved GADT Reasoning in Scala. In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala* (London, United Kingdom) *(Scala '19)*. Association for Computing Machinery, New York, NY, USA, 12–16. https://doi.org/10.1145/3337932.3338813

Marianna Rapoport and Ondřej Lhoták. 2019. A Path to DOT: Formalizing Fully Path-Dependent Types. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 145 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360571

Radosław Waśko. 2020. Formal foundations for GADTs in Scala.

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* (New Orleans, Louisiana, USA) *(POPL '03)*. Association for Computing Machinery, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150