

IMPLEMENTING PATH-DEPENDENT GADT REASONING FOR SCALA 3

SCALA SYMPOSIUM '21

Yichen Xu

linyus@bupt.edu.cn

Joint Work with Aleksander Boruch-Gruszecki and Lionel Parreaux

Beijing University of Posts and Telecommunications

[†] Work done during the internship at LAMP, EPFL

- 1 Preamble
- 2 Path-Dependent GADT Reasoning
- 3 Discussion
- 4 Conclusion

PREAMBLE

What are GADTs. GADTs enables the encapsulation of **additional type information** in ADTs, along with the ability of **utilizing the encapsulated type information** when performing pattern matching.

Example. Definition of **type-safe** embedded algebraic expression GADT.

```
enum Expr[A]:  
  case LitInt(i: Int) extends Expr[Int]  
  case Add(e1: Expr[Int], e2: Expr[Int]) extends Expr[Int]  
  case Tuple[X, Y](x: Expr[X], y: Expr[Y]) extends Expr[(X, Y)]
```

```
enum Expr[A]:  
  case LitInt(i: Int) extends Expr[Int]  
  case Add(e1: Expr[Int], e2: Expr[Int]) extends Expr[Int]  
  case Tuple[X, Y](x: Expr[X], y: Expr[Y]) extends Expr[(X, Y)]
```

- Additional GADT type information: the `LitInt` constructor asserts that the created data is an `Expr[Int]`.
- This ensures stronger type safety by disallowing expressions like `Add(Tuple(LitInt(1), LitInt(2)), LitInt(3))`.

```
def eval[T](e: Expr[T]): T = e match
  case LitInt(i) => i
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Tuple(x, y) => (eval(x), eval(y))
```

- The other part of GADT's power: utilizing the additional type information in the pattern matching.
- The LitInt case: GADT constraint $\text{Int} = T$ allows the compilation of the code.
- GADT enables **existential types**.
The Tuple case: there **exists** α, β , such that $(\alpha, \beta) = T$.

- Scala has a sophisticated class system: possible to define **open GADT with complex inheritance hierarchy**.
- Scala's type system has advanced features: e.g. **variance and subtyping**.

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

def foo[A, B](func: Func[A, B]) = func match
  case _: Identity[c] =>
    (??? : A) : B
    // error. A <: c <: B does NOT hold here.
```

Counter-example: new Identity[X] & Func[Nothing, Any].

- The essence of GADT in Scala: **extracting *necessary constraints* from *cohabitation*.**
- *Cohabitation*. For any types S and T ,

S and T are cohabitated $\Leftrightarrow \exists x, x : S \ \& \ T$

```
def eval[T](e: Expr[T]): T = e match
  case LitInt(i) => i
  case Add(e1, e2) => eval(e1) + eval(e2)
  case Tuple(x, y) => (eval(x), eval(y))
```

- The LitInt case: $e : \text{Expr}[T] \ \& \ \text{LitInt}$
 $\Rightarrow e : \text{Expr}[T] \ \& \ \text{Expr}[\text{Int}]$
 \Rightarrow **Expr[T] and Expr[Int] are cohabited**
- From the cohabitation, we extract the GADT constraint
 $T = \text{Int}$.

- Dotty¹ has brought better GADT reasoning to Scala [2].
- However, existing implementation only support GADT reasoning for type parameters, lacking GADT reasoning for *path-dependent types*: we can have GADT constraints for a type parameter T , but not a path-dependent type $p.T$.

¹Also known as the Scala 3 compiler.

The missing of path-dependent GADT reasoning is unfortunate for two reasons:

- Current understanding of GADT reasoning in Scala is based upon the theory of path-dependent types [2, 3].
- Path-dependent types is a general and powerful way of abstracting over types in Scala. Leaving out GADT reasoning for them makes the implementation patently incomplete.

An example that relies on both type parameter and path-dependent GADT to compile.

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

```
trait Expr[+X] { type Size }
```

```
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 }
```

```
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))
```

PATH-DEPENDENT GADT REASONING

- Derive constraints between type members.
- Constrain path-dependent types.
- Type members as subtyping proofs.

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

- Desired behavior: deriving constraints between the type members from the scrutinee and pattern.
- The code relies on the constraint `Int <: e.T` to compile.


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

- The (p: Tag) parameter is a *type tag*, which use a path-dependent type to mimic a type parameter.
- Expected behavior: constraining the path-dependent type $p.T$ with GADT constraint $\text{Int} <: p.T$.

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

- In the case body, $e.T := \text{IntExpr} <: \text{Expr}[X]$.
- **Type members from inhabited types can be viewed as proof of subtyping between its lower and upper bounds:**
 $\text{IntExpr} <: \text{Expr}[X]$.
- From the subtyping relation, derive the GADT constraint
 $\text{Int} <: X$.

- We extensively reuse the existing GADT reasoning framework in our implementation.
- *Data structures for constraints:*
 - Previous implementation: only recognizes **type parameter symbols** for constraint management, which is not sufficient for path-dependent types.
 - Our implementation: handle **the path and type member symbols collectively** to support path-dependent types.
- *Type Registration:* extend the type registration scheme to properly register path-dependent types.
- Refer to Section 3 and 4 of our paper for details.

- **Functional tests.** Our branch is able to pass all of the 8345 unit tests, suggesting that our implementation is conservative.
- **Benchmarks.** We run the 10 test suites in the Dotty benchmark and inspect the changes of running time before and after our work.

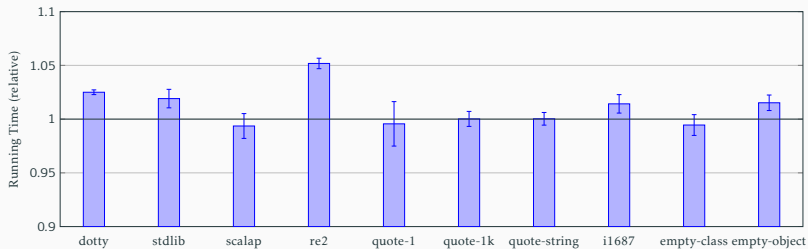


Figure 1: Relative running time before and after our work.

DISCUSSION

```
trait IntTag { type T = Int }  
def f[A](e: { type T <: A }): A = e match  
  case _: IntTag => 0
```

- Our work enables the extraction of constraint $\text{Int} <: A$.
- Although it is not possible to do this with Dotty's current GADT implementation, such reasoning already presents in Dotty's formal system, DOT [1].
- Therefore, our work brings the compiler closer to its formal side.

CONCLUSION




- Unifying the implementation of GADT reasoning for both type parameters and type members.
- Recording GADT constraints for concrete types when we can not break down the types further to extract GADT constraints.
- GADT constraint inference for a wider range of cohabitation.

- As a missing piece of puzzle, path-dependent GADT reasoning is not supported in Dotty's current implementation of GADT reasoning, but it will benefit real-world use cases and bring the compiler closer to its formalism.
- We propose the implementation of path-dependent GADT reasoning. Empirical evaluation shows the efficiency of our implementation, though there is space for improvements.
- We also give a description of the GADT reasoning framework in Dotty to get those who are interested in the technical details of Dotty familiar with related data structures and program logic, and facilitate future development.

- Thank you for listening!
- Slides available at bit.ly/PDGADT-Slides.



Figure 2: Scan me to get the slides.

-  Nada Amin, Samuel Grütter, Martin Odersky, Tiark Rompf, and Sandro Stucki. **The essence of dependent object types.** In *A List of Successes That Can Change the World*, 2016.
-  Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. **Towards improved gadt reasoning in scala.** In *Proceedings of the Tenth ACM SIGPLAN Symposium on Scala, Scala '19*, page 12–16, New York, NY, USA, 2019. Association for Computing Machinery.
-  Radosław Waśko. **Formal foundations for gadts in scala.** 2020.