Xavier Van de Woestyne

Yet Another introduction to GADTs

Generalized Algebraic Data Types

Scala io

Xavier Van de Woestyne

- https://xvw.lol
- @vdwxv
- @xvw@merveilles.town
- github.com/xvw
- **LambdaNantes**

Yet Another introduction to *GADTs*

Generalized Algebraic Data Types

Scala io

Xavier Van de Woestyne

- https://xvw.lol
- @vdwxv
- @xvw@merveilles.town
- github.com/xvw
- **LambdaNantes**

DISCLAIMER

Yet Another introduction to *GADTs*

Generalized Algebraic Data Types

Scala io

I am essentially an **OCaml** developer

Xavier Van de Woestyne

- https://xvw.lol
- @vdwxv
- @xvw@merveilles.town
- github.com/xvw
- **LambdaNantes**

DISCLAIMER

Yet Another introduction to *GADTs*

Generalized Algebraic Data Types

Scala io

I am essentially an OCaml developer

This talk is my first experience with Scala

Xavier Van de Woestyne

- https://xvw.lol
- @vdwxv
- @xvw@merveilles.town
- github.com/xvw
- **LambdaNantes**

DISCLAIMER

Yet Another introduction to *GADTs*

Generalized Algebraic Data Types

Xavier Van de Woestyne

- https://xvw.lol
- @vdwxv
- @xvw@merveilles.town
- github.com/xvw
- **LambdaNantes**

I am essentially an OCaml developer

This talk is my first experience with Scala

Very cool language btw

DISCLAIMER

Yet Another introduction to *GADTs*

Generalized Algebraic Data Types

Scala io

Xavier Van de Woestyne

- https://xvw.lol
- @vdwxv
- @xvw@merveilles.town
- github.com/xvw
- **LambdaNantes**

I am essentially an OCaml developer

This talk is my first experience with Scala

Very cool language btw

DISCLAIMER

yes, **why**? As a **keynote**?

Yet Another introduction to GADTs

Generalized Algebraic Data Types

Scala io

Because it's quite a story!

Because it's quite a story!

A quest of a litmus case

Betrayals

Because it's quite a story!

A quest of a litmus case

Betrayals

Introduced by example
since Scala 2.x

Because it's **quite** a story!

A quest of a litmus case

Nov 16, 2011: Paul Chiusano's gist

Betrayals

Introduced by example
since Scala 2.x

Because it's **quite** a story!

A quest of a litmus case

Nov 16, 2011: Paul Chiusano's gist

Betrayals

Introduced by example since Scala 2.x

Because it's **quite** a story!

A quest of a litmus case

A huge amount of work

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019.
Towards improved GADT reasoning in Scala.

Nov 16, 2011: Paul Chiusano's gist

Betrayals

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.

A huge amount of work

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. **2019**.
Towards improved GADT reasoning in Scala.

Nov 16, 2011: Paul Chiusano's gist

Betrayals

Introduced by example since Scala 2.x

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 ... encodings of GADTs, demonstrating ... between, on one hand, the (sub)ty... that formally underlie GADT ... 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.

So, let's try to:

- understand why
- trying with an other approach

story!

A quest of a litmus case

A huge amount of work

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019.
Towards improved GADT reasoning in Scala.

A plan!

why they're useful

meh. Why "poor"

Reminder: Algebraic types → GADTs: a first definition → The poor typed AST example

and ADT vs AT

A plan!

why they're useful

meh. Why "poor"

Reminder: Algebraic types → GADTs: a first definition → The poor typed AST example

and ADT vs AT

The quintessential GADT ← Types equalities/constraints ← More about litmus cases

A plan!

Reminder: Algebraic types
and ADT vs AT

GADTs: a first definition
why they're useful

The poor typed AST example
meh. Why "poor"

More about litmus cases

Types equalities/constraints

The quintessential GADT

Finding the litmus case
Some usage of Eq[A, B]

More about existentials

Type-level lists

A plan!

Reminder: Algebraic types

and ADT vs AT

GADTs: a first definition

why they're useful

The poor typed AST example

meh. Why "poor"

More about litmus cases

Types equalities/constraints

The quintessential GADT

Finding the litmus case

Some usage of Eq[A, B]

More about existentials

Type-level lists

A practical example

Bi-directional URL definition

Conclusions

Algebraic types

Reuniting broken fragments

Algebra: الجبر, *al-jabr*

Algebraic types

Reuniting broken fragments

As in common Algebra,
we can **build new types on top of types and operators**
and applying **equational reasoning**

Algebra: **الجبر**, ***al-jabr***

Algebraic types

Reuniting broken fragments

As in common Algebra,
we can **build new types on top of types and operators**
and applying **equational reasoning**

Algebra: الجبر, ***al-jabr***

Algebraic types

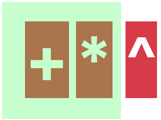
+ ***** **^**

Reuniting broken fragments

Algebra: الجبر, ***al-jabr***

As in common Algebra,
we can **build new types on top of types and operators**
and applying **equational reasoning**

Algebraic types



Are Algebraic **Data** Types
Describe Data 😉

Reuniting broken fragments

As in common Algebra,
we can **build new types on top of types and operators**
and applying **equational reasoning**

Algebra: الجبر, ***al-jabr***

Algebraic types

\+ * ^

Are Algebraic **Data** Types
Describe Data 😉

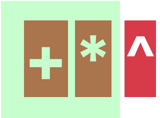
Have no particular name
Describe behaviour on data

Algebraic types

Algebra: الجبر, ***al-jabr***

Reuniting broken fragments

As in common Algebra,
we can **build new types on top of types and operators**
and applying **equational reasoning**



Are Algebraic **Data** Types
Describe Data 😉

Since the subject of the presentation
is **GADT**, we will focus on **ADTs**

Have no particular name
Describe behaviour on data

equational reasoning can be used to estimate cardinality and more computational algebra, but this is not at all what the presentation is about

And it is, in fact, not very interesting except for DDD.

Reuniting broken fragments

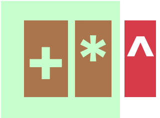
As in common Algebra,
we can **build new types on top of types and operators**
and applying **equational reasoning**

Algebra: الجبر, *al-jabr*

Algebraic types

Since the subject of the presentation
is **GADT**, we will focus on **ADTs**

Are Algebraic **Data** Types
Describe Data 😉



Have no particular name
Describe behaviour on data

Product types

Describes the conjunction of several types (their **Cartesian product**).

```
case class Human(
  firstName: String,
  lastName:  String,
  age:       Int
)
```

Product types

Describes the conjunction of several types (their **Cartesian product**).

They can be **recursive.**

```
case class Human(
  firstName: String,
  lastName:  String,
  age:       Int,
  children:  List[Human]
)
```

Product types

Describes the conjunction of several types (their **Cartesian product**).

They can be **recursive.**

They can introduce **Type Parameters** *(parametric polymorphism)*, sometimes introducing **variance** markers for expressing subtyping relations.

```
case class Human(
  firstName: String,
  lastName:  String,
  age:       Int,
  children:  List[Human]
)
```

type parameters
(generics)

```
case class Prod[+A, +B](fst: A, snd: B)
```

Product types

Describes the conjunction of several types (their **Cartesian product**).

They can be **recursive.**

They can introduce **Type Parameters** *(parametric polymorphism)*, sometimes introducing **variance** markers for expressing subtyping relations.

```scala
case class Human(
  firstName: String,
  lastName:  String,
  age:       Int,
  children:  List[Human]
)
```

type parameters
(generics)

```scala
case class Prod[+A, +B](fst: A, snd: B)
```

Is, in fact, **the most minimal product type**

Product types

Describes the conjunction of several types (their **Cartesian product**).

They can be **recursive.**

They can introduce **Type Parameters** *(parametric polymorphism)*, sometimes introducing **variance** markers for expressing subtyping relations.

```scala
case class Human(
  firstName: String,
  lastName:  String,
  age:       Int,
  children:  List[Human]
)
```

type parameters
(generics)

```scala
case class Prod[+A, +B](fst: A, snd: B)
```

Is, in fact, **the most minimal product type**

```scala
val quad = Prod(1, Prod(2, Prod(3, 4)))
```

Sum types

Describes the disjunction of several types (their **Disjoint union**).

```
enum Bool:
  case True  extends Bool
  case False extends Bool
```

Sum types

Describes the disjunction of several types (their **Disjoint union**).

```
enum Bool:
  case True  extends Bool
  case False extends Bool
```

Refutable but:
highlights encoding via **subtyping** and **sealing**.

Sum types

Describes the disjunction of several types (their **Disjoint union**).

```
enum Bool:
  case True  extends Bool
  case False extends Bool
```

Refutable but:
highlights encoding via **subtyping** and **sealing**.

This sometimes requires **annotation** or **unification tricks**.

Sum types

Describes the disjunction of several types (their **Disjoint union**).

As for Product, they can be **recursive** and introducing **generics** (and **variance** markers)

```
enum Bool:
  case True  extends Bool
  case False extends Bool

enum MList[+A]:
  case Nil
  case Cons(x: A, xs: MList[A])
```

Sum types

Describes the disjunction of several types (their **Disjoint union**).

As for Product, they can be **recursive** and introducing **generics** (and **variance** markers)

And as for Product, there is a minimal Sum type **(Either)**

```
enum Bool:
  case True  extends Bool
  case False extends Bool

enum MList[+A]:
  case Nil
  case Cons(x: A, xs: MList[A])

enum Sum[+A, +B]:
  case Left(x: A)
  case Right(x: B)
```

Sum types

Describes the disjunction of several types (their **Disjoint union**).

As for Product, they can be **recursive** and introducing **generics** (and **variance** markers)

And as for Product, there is a minimal Sum type **(Either)**

```
enum Bool:
  case True  extends Bool
  case False extends Bool

enum MList[+A]:
  case Nil
  case Cons(x: A, xs: MList[A])

enum Sum[+A, +B]:
  case Left(x: A)
  case Right(x: B)
```

```
type Triple = Sum[Int, Sum[Double, String]]
val a : Triple = Sum.Left(1)
val b : Triple = Sum.Right(Sum.Left(1.0))
val c : Triple = Sum.Right(Sum.Right("1"))
```

Sum types

Describes the disjunction of several types (their **Disjoint union**).

As for Product, they can be **recursive** and introducing **generics** (and **variance** markers)

And as for Product, there is a minimal Sum type **(Either)**

minimal exponential type

type Arr[-A, +B] = A ⇒ B

```
enum Bool:
  case True  extends Bool
  case False extends Bool

enum MList[+A]:
 case Nil
 case Cons(x: A, xs: MList[A])

enum Sum[+A, +B]:
 case Left(x: A)
 case Right(x: B)
```

to conclude on

Algebraic types

They can express Model/Domain

to conclude on

Algebraic types

They can express Model/Domain

Paired with Pattern matching they
are expressive for defining Data Structures

to conclude on

Algebraic types

They can express Model/Domain

Paired with Pattern matching they
are expressive for defining Data Structures

to conclude on

Algebraic types

They have minimal representations

They can express Model/Domain

Paired with Pattern matching they are expressive for defining Data Structures

to conclude on

Algebraic types

They have minimal representations

Sum types relies on **subtyping** and **sealing**

They can express Model/Domain

Paired with Pattern matching they
are expressive for defining Data Structures

to conclude on

Algebraic types

They have minimal representations

Let's play with that

Sum types relies on **subtyping** and **sealing**

```
enum StringOrInt:
 case SString(x: String)
 case SInt(x: Int)
```

Let's be more explicit

```
enum StringOrInt:
  case SString(x: String) extends StringOrInt
  case SInt(x: Int)       extends StringOrInt
```

Let's add a type parameter, just for fun

```
enum StringOrInt:
  case SString(x: String) extends StringOrInt
  case SInt(x: Int)       extends StringOrInt
```

Does not compile **String and Int needs a Type Parameter**

```
enum StringOrInt[A]:
  case SString(x: String) extends StringOrInt[A]
  case SInt(x: Int)       extends StringOrInt[A]
```

Let's fix the type parameter

```
enum StringOrInt[A]:
  case SString(x: String) extends StringOrInt[String]
  case SInt(x: Int)       extends StringOrInt[Int]
```

Let's fix the type parameter

```scala
enum StringOrInt[A]:
  case SString(x: String) extends StringOrInt[String]
  case SInt(x: Int)       extends StringOrInt[Int]
```

```scala
def eval[A](x: StringOrInt[A]) : A =
 x match
   case StringOrInt.SString(x) => x
   case StringOrInt.SInt(x)    => x
```

Let's fix the type parameter

We no longer rely on **subtyping** to describe tags. We use a concrete type

```
enum StringOrInt[A]:
  case SString(x: String) extends StringOrInt[String]
  case SInt(x: Int)       extends StringOrInt[Int]
```

```
def eval[A](x: StringOrInt[A]) : A =
 x match
   case StringOrInt.SString(x) => x
   case StringOrInt.SInt(x)    => x
```

Constructors of our sum are no more **Surjective** in **[A]**

Let's fix the type parameter

We no longer rely on **subtyping** to describe tags. We use a concrete type

```
enum StringOrInt[A]:
  case SString(x: String) extends StringOrInt[String]
  case SInt(x: Int)       extends StringOrInt[Int]
```

```
def eval[A](x: StringOrInt[A]) : A =
 x match
   case StringOrInt.SString(x) => x
   case StringOrInt.SInt(x)    => x
```

Constructors of our sum are no more **Surjective** in **[A]**

Let's fix the type parameter

We no longer rely on **subtyping** to describe tags. We use a concrete type

```
enum StringOrInt[A]:
 case SString(x: String) extends StringOrInt[String]
 case SInt(x: Int)       extends StringOrInt[Int]
```

Since we are no more surjective, we can have **partial pattern matching**

```
def getInt(x: StringOrInt[Int]) =
 x match
   case StringOrInt.SInt(x) => x
```

```
def eval[A](x: StringOrInt[A]) : A =
 x match
   case StringOrInt.SString(x) => x
   case StringOrInt.SInt(x)    => x
```

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

We've just seen how

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

which often involves polymorphic recursions

We've just seen how

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

which often involves polymorphic recursions

We've just seen how

*"A Generalized Algebraic Data Type **is a sum type that allows its constructors to be non-surjective** on one or more of its type parameters and **introduces local type-equality constraints in pattern-matching branches,** making the expression of existential types trivial"*

This was the part forgotten in many GADT definitions

which often involves polymorphic recursions

We've just seen how

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

This was the part forgotten in many GADT definitions

We'll see why later in the presentation, but it's a consequence of introducing local type equality constraints.

which often involves polymorphic recursions

We've just seen how

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

This was the part forgotten in many GADT definitions

We'll see why later in the presentation, but it's a consequence of introducing local type equality constraints.

Let's see why using a poor example

We have a little arithmetic AST

```
enum AST:
  case I(x: Int)
  case Add(l: AST, R: AST)
  case Mul(l: AST, R: AST)
```

```
def eval(ast: AST) : Int =
  import AST.*
  ast match
    case I(x)        => x
    case Add(l, r) => eval(l) + eval(r)
    case Mul(l, r) => eval(l) * eval(r)
```

We have a little arithmetic AST

```
enum AST:
  case I(x: Int)
  case Add(l: AST, R: AST)
  case Mul(l: AST, R: AST)
```

```
def eval(ast: AST) : Int =
  import AST.*
  ast match
    case I(x)      => x
    case Add(l, r) => eval(l) + eval(r)
    case Mul(l, r) => eval(l) * eval(r)
```

Let's add some Boolean/Condition support

We have a little arithmetic AST

```
enum AST:
  case I(x: Int)
  case Add(l: AST, R: AST)
  case Mul(l: AST, R: AST)
```

```
def eval(ast: AST) : Int =
  import AST.*
  ast match
    case I(x)         => x
    case Add(l, r) => eval(l) + eval(r)
    case Mul(l, r) => eval(l) * eval(r)
```

Let's add some Boolean/Condition support

Haskell Doc has a beautiful elaboration about the implementation, progressively, but since the example of the AST is broken:
let's get straight to the point

Introducing Boolean support

```
enum AST[A]:
  case I(x: Int)                                        extends AST[Int]
  case B(x: Boolean)                                    extends AST[Boolean]
  case Add(l: AST[Int], r: AST[Int])                    extends AST[Int]
  case Mul(l: AST[Int], r: AST[Int])                    extends AST[Int]
  case Equal(l: AST[A], r: AST[A])                      extends AST[Boolean]
  case Cond(c: AST[Boolean], t: AST[A], f:AST[A]) extends AST[A]
```

Introducing Boolean support

We use a witness, for the GADT

```
enum AST[A]:
  case I(x: Int)                                       extends AST[Int]
  case B(x: Boolean)                                   extends AST[Boolean]
  case Add(l: AST[Int], r: AST[Int])                   extends AST[Int]
  case Mul(l: AST[Int], r: AST[Int])                   extends AST[Int]
  case Equal(l: AST[A], r: AST[A])                     extends AST[Boolean]
  case Cond(c: AST[Boolean], t: AST[A], f:AST[A]) extends AST[A]
```

Introducing Boolean support

We use a witness, for the GADT

We fix the return type of every constructor

```
enum AST[A]:
  case I(x: Int)                                     extends AST[Int]
  case B(x: Boolean)                                 extends AST[Boolean]
  case Add(l: AST[Int], r: AST[Int])                 extends AST[Int]
  case Mul(l: AST[Int], r: AST[Int])                 extends AST[Int]
  case Equal(l: AST[A], r: AST[A])                   extends AST[Boolean]
  case Cond(c: AST[Boolean], t: AST[A], f:AST[A]) extends AST[A]
```

Introducing Boolean support

We use a witness, for the GADT

We fix the return type of every constructor

Normal forms of AST can be constrained

```
enum AST[A]:
  case I(x: Int)                                        extends AST[Int]
  case B(x: Boolean)                                    extends AST[Boolean]
  case Add(l: AST[Int], r: AST[Int])                    extends AST[Int]
  case Mul(l: AST[Int], r: AST[Int])                    extends AST[Int]
  case Equal(l: AST[A], r: AST[A])                      extends AST[Boolean]
  case Cond(c: AST[Boolean], t: AST[A], f:AST[A])       extends AST[A]
```

Introducing Boolean support

We use a witness, for the GADT

We fix the return type of every constructor

Normal forms of AST can be constrained

```
enum AST[A]:
  case I(x: Int)                                               extends AST[Int]
  case B(x: Boolean)                                           extends AST[Boolean]
  case Add(l: AST[Int], r: AST[Int])                           extends AST[Int]
  case Mul(l: AST[Int], r: AST[Int])                           extends AST[Int]
  case Equal(l: AST[A], r: AST[A])                             extends AST[Boolean]
  case Cond(c: AST[Boolean], t: AST[A], f:AST[A])  extends AST[A]
```

You can keep polymorphic constructors

Introducing Boolean support

We use a witness, for the GADT

We fix the return type of every constructor

Normal forms of AST can be constrained

```
enum AST[A]:
  case I(x: Int)                                           extends AST[Int]
  case B(x: Boolean)                                       extends AST[Boolean]
  case Add(l: AST[Int], r: AST[Int])                       extends AST[Int]
  case Mul(l: AST[Int], r: AST[Int])                       extends AST[Int]
  case Equal(l: AST[A], r: AST[A])                         extends AST[Boolean]
  case Cond(c: AST[Boolean], t: AST[A], f:AST[A])          extends AST[A]
```

You can keep polymorphic constructors

Invalid ASTs can no longer be built

Interpreting AST using polymorphic recursion

```scala
def eval[A](ast: AST[A]) : A =
    import AST.*
    ast match
      case I(x)          => x
      case B(x)          => x
      case Add(l, r)     => eval(l) + eval(r)
      case Mul(l, r)     => eval(l) * eval(r)
      case Equal(l, r)   => eval(l) == eval(r)
      case Cond(c, t, f) => if(eval(c)) then  eval(t) else eval(f)
```

Interpreting AST using polymorphic recursion

As you can see, GADTs allow you to express static invariants and, as far as possible, make code that's **correct by construction!**

```
def eval[A](ast: AST[A]) : A =
    import AST.*
    ast match
      case I(x)           => x
      case B(x)           => x
      case Add(l, r)      => eval(l) + eval(r)
      case Mul(l, r)      => eval(l) * eval(r)
      case Equal(l, r)    => eval(l) == eval(r)
      case Cond(c, t, f) => if(eval(c)) then  eval(t) else eval(f)
```

Interpreting AST using polymorphic recursion

```
def eval[A](ast: AST[A]) : A =
    import AST.*
    ast match
      case I(x)          => x
      case B(x)          => x
      case Add(l, r)     => eval(l) + eval(r)
      case Mul(l, r)     => eval(l) * eval(r)
      case Equal(l, r)   => eval(l) == eval(r)
      case Cond(c, t, f) => if(eval(c)) then  eval(t) else eval(f)
```

As you can see, GADTs allow you to express static invariants and, as far as possible, make code that's correct by construction!

So what's the problem with this example?
(that worked in Scala 2.x**)**

It only covers this part of the definition

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

It only covers this part of the definition

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

And the examples taking advantage of the second part didn't work (in Scala 2), hence the weakness of the example.

It only covers this part of the definition

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

The Litmus case was wrong

And the examples taking advantage of the second part didn't work (in Scala 2), hence the weakness of the example.

And it was already described in this excellent paper

Andrew Kennedy, Claudio Russo. 2006.
Generalized Algebraic Data Types and Object-Oriented Programming.

And it was already described in this excellent paper

Andrew Kennedy, Claudio Russo. 2006.
Generalized Algebraic Data Types and Object-Oriented Programming.

So implementing the typed interpreter/AST does not guarantee that the language supports GADTs.

And it was already described in this excellent paper

Andrew Kennedy, Claudio Russo. 2006.
Generalized Algebraic Data Types and Object-Oriented Programming.

So implementing the typed interpreter/AST does not guarantee that the language supports GADTs.

GADTS: algebraic types whose constructors introduce existential types and use type equality constraints
OBJECTS: classes whose methods universally quantify over types, and use subtyping constraints

Both enable statically typed AST implementation

But if both approaches allow the **same** encodings, with **incredibly** similar usage, what's the problem?

Some Haskell/OCaml examples are not **transposable**

But if both approaches allow the **same** encodings, with **incredibly** similar usage, what's the problem?

Some Haskell/OCaml examples are not **transposable**

But if both approaches allow the **same** encodings, with **incredibly** similar usage, what's the problem?

Naming things correctly facilitates their understanding, evolution and maintenance

(ahem "typeclasses")

Some Haskell/OCaml examples are not **transposable**

But if both approaches allow the **same** encodings, with **incredibly** similar usage, what's the problem?

Naming things correctly facilitates their understanding, evolution and maintenance

(ahem "typeclasses")

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019.
Towards improved GADT reasoning in Scala.

Some Haskell/OCaml examples are not **transposable**

But if both approaches allow the **same** encodings, with **incredibly** similar usage, what's the problem?

Naming things correctly facilitates their understanding, evolution and maintenance

(ahem "typeclasses")

Yes, Scala 2.x **didn't** support GADTs properly

Lionel Parreaux, Aleksander Boruch-Gruszecki, and Paolo G. Giarrusso. 2019.
Towards improved GADT reasoning in Scala.

One of the easiest ways to prove that a language is **Turing-Complete** is to implement a **Brainfuck interpreter**, a very minimalist language that is also Turing-Complete.

One of the easiest ways to prove that a language is **Turing-Complete** is to implement a **Brainfuck interpreter**, a very minimalist language that is also Turing-Complete.

It is a perfect Litmus case for the Turing-Completude

Can we find a Litmus case for GADTs?

One of the easiest ways to prove that a language is **Turing-Complete** is to implement a **Brainfuck interpreter**, a very minimalist language that is also Turing-Complete.

It is a perfect Litmus case for the Turing-Completude

As the local equality constraint was not used, we define it via a GADT
(or an indexed type if it is not supported)

Can we find a Litmus case for GADTs?

One of the easiest ways to prove that a language is **Turing-Complete** is to implement a **Brainfuck interpreter**, a very minimalist language that is also Turing-Complete.

It is a perfect Litmus case for the Turing-Completude

```
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

We define equality between A and B

```
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

We define equality between A and B

```
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

With only 1 constructor Refl

We define equality between A and B

Can only be embodied with two locally equal types

```scala
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

With only 1 constructor Refl

We define equality between A and B

Can only be embodied with two locally equal types

```scala
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

With only 1 constructor Refl

```scala
type Z = Int
val a : Eq[Int, String] = Eq.Refl() // does not compile
val b : Eq[Int, Int] = Eq.Refl()
val c : Eq[Int, Z] = Eq.Refl()
```

We define equality between A and B

Can only be embodied with two locally equal types

```scala
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

With only 1 constructor Refl

```scala
type Z = Int
val a : Eq[Int, String] = Eq.Refl() // does not compile
val b : Eq[Int, Int] = Eq.Refl()
val c : Eq[Int, Z] = Eq.Refl()
```

Which can be translated as "if I can instantiate a `Refl() : Eq[A, B]`, then I have a witness that A and B are locally equal types.

And to ensure equalities, we can apply the Leibniz Substitution Principle to gives some tools

We define equality between A and B

Can only be embodied with two locally equal types

```scala
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

With only 1 constructor Refl

```scala
type Z = Int
val a : Eq[Int, String] = Eq.Refl() // does not compile
val b : Eq[Int, Int] = Eq.Refl()
val c : Eq[Int, Z] = Eq.Refl()
```

Which can be translated as "if I can instantiate a `Refl() : Eq[A, B]`, then I have a witness that A and B are locally equal types.

```
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

```
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```
def symmetry[A, B](
 witness: Eq[A, B]
) : Eq[B, A] = witness match
 case Eq.Refl() => Eq.Refl()
```

```
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```
def symmetry[A, B](
 witness: Eq[A, B]
) : Eq[B, A] = witness match
 case Eq.Refl() => Eq.Refl()
```

Transitivity

```
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

```scala
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
 witness: Eq[A, B]
) : Eq[B, A] = witness match
 case Eq.Refl() => Eq.Refl()
```

Transitivity

```scala
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Free Cast

```scala
def cast[A, B](
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

```scala
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
 witness: Eq[A, B]
) : Eq[B, A] = witness match
 case Eq.Refl() => Eq.Refl()
```

Transitivity

```scala
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Free Cast

```scala
def cast[A, B](
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

Which gives a free-cast that can cross abstraction and boxing (if the cookie has been instantiated in the right place)

```scala
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
  witness: Eq[A, B]
) : Eq[B, A] = witness match
  case Eq.Refl() => Eq.Refl()
```

Transitivity

```scala
def transitivity[A, B, C](
  witnessA: Eq[A, B],
  witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
  case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Free Cast

```scala
def cast[A, B](
  witness: Eq[A, B],
  value: A
) : B = witness match
  case Eq.Refl() => value
```

Injectivity

```scala
def injectivity[T[_], A, B](
  witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
  case Eq.Refl() => Eq.Refl()
```

```scala
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
 witness: Eq[A, B]
) : Eq[B, A] = witness match
 case Eq.Refl() => Eq.Refl()
```

Transitivity

This our Litmus Case!

```scala
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Free Cast

```scala
def cast[A, B](
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

Injectivity

```scala
def injectivity[T[_], A, B](
 witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
 case Eq.Refl() => Eq.Refl()
```

```scala
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
 witness: Eq[A, B]
) : Eq[B, A] = witness match
 case Eq.Refl() => Eq.Refl()
```

Transitivity

This our Litmus Case!

Works since Scala 3!

```scala
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Free Cast

```scala
def cast[A, B](
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

Injectivity

```scala
def injectivity[T[_], A, B](
 witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
 case Eq.Refl() => Eq.Refl()
```

```scala
enum Eq[A, B]:
  case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
  witness: Eq[A, B]
) : Eq[B, A] = witness match
  case Eq.Refl() => Eq.Refl()
```

Transitivity

```scala
def transitivity[A, B, C](
  witnessA: Eq[A, B],
  witnessB: Eq[B, C]
) : Eq[A, C] = (witnessA, witnessB) match
  case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

This our Litmus Case!

Works since Scala 3!

Free Cast

Handling type equalities with injectivity support is very tricky.

So supporting them is a litmus case for GADTs

```scala
def cast[A, B](
  witness: Eq[A, B],
  value: A
) : B = witness match
  case Eq.Refl() => value
```

Injectivity

```scala
def injectivity[T[_], A, B](
  witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
  case Eq.Refl() => Eq.Refl()
```

```scala
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```scala
def symmetry[A, B](
 witness: Eq[A, B]
```

Transitivity

This our Litmus Case!

```scala
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 ...C]
 ...witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Works since Scala 3!

> *Jeremy Yallop, Oleg Kiselyov.* **2010**.
> **First-class modules: hidden power and tantalizing promises.**

Free Cast

```scala
def
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

Handling type equalities with injectivity support is very tricky.

So supporting them is a litmus case for GADTs

Injectivity

```scala
def injectivity[T[_], A, B](
 witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
 case Eq.Refl() => Eq.Refl()
```

```
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Symmetry

```
def symmetry[A, B](
 witness: Eq[A, B]
```

Transitivity

This our Litmus Case!

```
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 ...C]
 ...witnessA, witnessB) match
 case (Eq.Refl(), Eq.Refl()) => Eq.Refl()
```

Works since Scala 3!

Jeremy Yallop, Oleg Kiselyov. **2010**.
First-class modules: hidden power and tantalizing promises.

Does this mean that GADTs are absolutely perfect in Scala? **Nah**

Free Cast

```
def ...
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

Handling type equalities with injectivity support is very tricky.

So supporting them is a litmus case for GADTs

Injectivity

```
def injectivity[T[_], A, B](
 witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
 case Eq.Refl() => Eq.Refl()
```

```scala
enum Eq[A, B]:
 case Refl[A]() extends Eq[A, A]
```

Does Eq[A, B] only serve to prove the partially correct support of GADTs?

This our Litmus Case!

Symmetry

```scala
def symmetry[A, B](
 witness: Eq[A, B]
```

Transitivity

```scala
def transitivity[A, B, C](
 witnessA: Eq[A, B],
 C]
 witnessA, witnessB) match
 case (Eq.Refl(), Eq.Re
```

Works since Scala 3!

Jeremy Yallop, Oleg Kiselyov. **2010**.
First-class modules: hidden power and tantalizing promises.

Does this mean that GADTs are absolutely perfect in Scala? Nah

Free Cast

```scala
def
 witness: Eq[A, B],
 value: A
) : B = witness match
 case Eq.Refl() => value
```

Handling type equalities with injectivity support is very tricky.

So supporting them is a litmus case for GADTs

Injectivity

```scala
def injectivity[T[_], A, B](
 witness: Eq[A, B]
) : Eq[T[A], T[B]] = witness match
 case Eq.Refl() => Eq.Refl()
```

methods that act only if the generics handle
some types

Fixing OOP with guarded methods

methods that act only if the generics handle
some types

Fixing OOP with guarded methods

and without extension methods that
needs to break encapsulation

methods that act only if the generics handle some types

Fixing OOP with guarded methods

and without extension methods that needs to break encapsulation

```
class MList[A](val v: List[A]):
 def sum(witness: Eq[A, Int]) : Int =
   witness match
     case Eq.Refl() =>
       this.v.reduce((x, y) => x + y)

 def flatten[B](witness: Eq[A, List[B]]) : List[B] =
   witness match
     case Eq.Refl() =>
       this.v.flatMap(X => X)
```

guarding sum **using** Eq[A, Int]

methods that act only if the generics handle some types

Fixing OOP with guarded methods

and without extension methods that needs to break encapsulation

```
class MList[A](val v: List[A]):
  def sum(witness: Eq[A, Int]) : Int =
    witness match
      case Eq.Refl() =>
        this.v.reduce((x, y) => x + y)

  def flatten[B](witness: Eq[A, List[B]]) : List[B] =
    witness match
      case Eq.Refl() =>
        this.v.flatMap(X => X)
```

Fixing OOP with guarded methods

methods that act only if the generics handle some types

and without extension methods that needs to break encapsulation

guarding sum **using** Eq[A, Int]

```
class MList[A](val v: List[A]):
 def sum(witness: Eq[A, Int]) : Int =
  witness match
   case Eq.Refl() =>
    this.v.reduce((x, y) => x + y)

 def flatten[B](witness: Eq[A, List[B]]) : List[B] =
  witness match
   case Eq.Refl() =>
    this.v.flatMap(X => X)
```

guarding flatten **using** Eq[A,List[B]]

In fact, Eq[A, B] **is the quintessential GADT. And much like Sum, Prod and Arr, it's** *normally* **sufficient to encode all other GADTs.**

Patricia Johann and Neil Ghani. 2008.
Foundations For Structured Programming With GADTs.

In fact, Eq[A, B] is the quintessential GADT. And much like Sum, Prod and Arr, it's *normally* sufficient to encode all other GADTs.

logic, a GADT is a sum-type where the constructor
is attached to a local type equality constraint
(what is exactly Eq)

Patricia Johann and Neil Ghani. 2008.
Foundations For Structured Programming With GADTs.

In fact, Eq[A, B] is the quintessential GADT. And much like Sum, Prod and Arr, it's *normally* sufficient to encode all other GADTs.

logic, a GADT is a sum-type where the constructor
is attached to a local type equality constraint
(what is exactly Eq)

Patricia Johann and Neil Ghani. 2008.
Foundations For Structured Programming With GADTs.

In fact, Eq[A, B] is the quintessential GADT. And much like Sum, Prod and Arr, it's *normally* sufficient to encode all other GADTs.

GADT

```
enum T[A]:
   case SString() extends T[String]
   case SSInt()   extends T[Int]
```

logic, a GADT is a sum-type where the constructor
is attached to a local type equality constraint
(what is exactly Eq)

Patricia Johann and Neil Ghani. **2008**.
Foundations For Structured Programming With GADTs.

In fact, Eq[A, B] is the quintessential GADT. And much like Sum, Prod and Arr, it's *normally* sufficient to encode all other GADTs.

GADT

```
enum T[A]:
  case SString() extends T[String]
  case SSInt()   extends T[Int]
```

EQ[A, B]

```
enum T[A]:
 case SString(w: Eq[A, String])
 case SInt(   w: Eq[A, Int])
```

logic, a GADT is a sum-type where the constructor
is attached to a local type equality constraint
(what is exactly Eq)

Patricia Johann and Neil Ghani. 2008.
Foundations For Structured Programming With GADTs.

In fact, Eq[A, B] is the quintessential GADT. And much like Sum, Prod and Arr, it's *normally* sufficient to encode all other GADTs.

GADT

```
enum T[A]:
   case SString() extends T[String]
   case SSInt()   extends T[Int]
```

EQ[A, B]

```
enum T[A]:
 case SString(w: Eq[A, String])
 case SInt(   w: Eq[A, Int])
```

Can be used as a GADT

```
def zero[A](tagged: T[A]) : A =
 import T.*
 tagged match
   case SString(Eq.Refl()) => ""
   case SInt(Eq.Refl()) => 0
```

logic, a GADT is a sum-type where the constructor
is attached to a **local type equality constraint**
(what is exactly Eq)

Patricia Johann and Neil Ghani. **2008**.
Foundations For Structured Programming With GADTs.

But it may be my lack of Scala writing skills.

Warming on T.SString(_)

```
def partial(
  tagged: T[Int]
) : Int = tagged match
  case T.SInt(Eq.Refl()) => 0
```

t, Eq[A, B] is the quintessential GADT. And much
um, Prod and Arr, it's *normally* sufficient to encode
her GADTs.

MEH

Can be used as a GADT

```
def zero[A](tagged: T[A]) : A =
 import T.*
 tagged match
   case SString(Eq.Refl()) => ""
   case SInt(Eq.Refl()) => 0
```

GADT

```
enum T[A]:
   case SString() extends T[String]
   case SInt()    extends T[Int]
```

EQ[A, B]

```
enum T[A]:
  case SString(w: Eq[A, String])
  case SInt(   w: Eq[A, Int])
```

logic, a GADT is a sum-type where the constructor
is attached to a **local type equality constraint**
(what is exactly Eq)

A very complicated issue.

But it may be my lack of Scala writing skills.

Patricia Johann and Neil Ghani. **2008**.
Foundations For Structured Programming With GADTs.

Warming on T.SString(_)

```
def partial(
  tagged: T[Int]
) : Int = tagged match
  case T.SInt(Eq.Refl()) => 0
```

t, Eq[A, B] is the quintessential GADT. And much um, Prod and Arr, it's *normally* sufficient to encode her GADTs.

GADT

```
enum T[A]:
   case SString() extends T[String]
   case SInt()    extends T[Int]
```

MEH

Can be used as a GADT

```
def zero[A](tagged: T[A]) : A =
 import T.*
 tagged match
   case SString(Eq.Refl()) => ""
   case SInt(Eq.Refl()) => 0
```

EQ[A, B]

```
enum T[A]:
 case SString(w: Eq[A, String])
 case SInt(   w: Eq[A, Int])
```

Jacques Garrigue and Jacques Le Normand. **2015**.
GADTs and exhaustiveness: looking for the impossible.

A very complicated issue.

…e the constructor
…uality constraint
(…hat is exactly Eq)

But it may be my lack of Scala writing skills.

Patricia Johann and Neil Ghani. **2008**.
Foundations For Structured Programming With GADTs.

Warming on T.SString(_)

```
def partial(
  tagged: T[Int]
) : Int = tagged match
  case T.SInt(Eq.Refl()) => 0
```

…t, Eq[A, B] is the **quintessential GADT. And much** …um, Prod and Arr, it's *normally* sufficient **to encode** …her GADTs.

MEH

GADT

```
enum T[A]:
   case SString() extends T[String]
   case SSInt()   extends T[Int]
```

Can be used as a GADT

```
def zero[A](tagged: T[A]) : A =
 import T.*
 tagged match
   case SString(Eq.Refl()) => ""
   case SInt(Eq.Refl()) => 0
```

EQ[A, B]

```
enum T[A]:
 case SString(w: Eq[A, String])
 case SInt(   w: Eq[A, Int])
```

*"A Generalized Algebraic Data Type **is a sum type that allows its constructors to be non-surjective** on one or more of its type parameters and **introduces local type-equality constraints in pattern-matching branches,** making the expression of existential types trivial"*

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

Introducing Local Types Equation is stronger than introducing Local Types (existentials) this is why GADTs come, *de facto* with existentials

"A Generalized Algebraic Data Type ***is a sum type that allows its constructors to be non-surjective*** *on one or more of its type parameters and* ***introduces local type-equality constraints in pattern-matching branches,*** *making the expression of existential types trivial"*

Haskell 's `ExistentialQuantification` **predate GADT introduction**

Introducing Local Types Equation is stronger than introducing Local Types (existentials) this is why GADTs come, *de facto* with existentials

```
enum TList[TY, W]:
 case Nil[W]()                                     extends TList[W, W]
 case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Witness to deal with usage

```
enum TList[TY, W]:
 case Nil[W]()                                      extends TList[W, W]
 case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Witness to deal with usage

Type-level Continuation

```
enum TList[TY, W]:
  case Nil[W]()                                    extends TList[W, W]
  case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Witness to deal with usage

Type-level Continuation

```
enum TList[TY, W]:
 case Nil[W]()                                        extends TList[W, W]
 case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Ensure:

```
TY = X => W
```

Witness to deal with usage

Type-level Continuation

```
enum TList[TY, W]:
  case Nil[W]()                                      extends TList[W, W]
  case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Ensure:

`TY = X => W`

Existential, packed by the continuation

Witness to deal with usage

Type-level Continuation

```
enum TList[TY, W]:
 case Nil[W]()                                    extends TList[W, W]
 case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Ensure:
`TY = X => W`

Existential, packed by the continuation

Track the continuation

```scala
val r : TList[Int => String => String, String] =
  import TList.*
  Cons(1, Cons("foo", Nil()))
```

Witness to deal with usage

Type-level Continuation

```scala
enum TList[TY, W]:
  case Nil[W]()                                       extends TList[W, W]
  case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Ensure:
`TY = X => W`

Existential, packed by the continuation

Track the continuation

```
val r : TList[Int => String => String, String] =
 import TList.*
 Cons(1, Cons("foo", Nil()))
```

W can be fixed by the usage

Witness to deal with usage

Type-level Continuation

```
enum TList[TY, W]:
 case Nil[W]()                                     extends TList[W, W]
 case Cons[A, TY, W](x: A, xs: TList[TY, W]) extends TList[A => TY, W]
```

Ensure:
`TY = X => W`

Existential, packed by the continuation

Track the continuation

```
val r : TList[Int => String => String, String] =
  import TList.*
  Cons(1, Cons("foo", Nil()))
```

Witness to deal with usage

Type-level Continuation

W can be fixed by the usage

```
enum TList[TY, W]:
  case Nil[W]()                                        extends TList[W, W]
  case Cons[A, TY, W](x: A, xs: TList[TY, W])  extends TList[A => TY, W]
```

Ensure:
`TY = X => W`

Existential, packed by the continuation

Track the continuation

Going further

Bidirectional routing

```
val url = genLink("user" ~: string ~: "id" ~: int ~: eop)("u55")(10)
   // Generate : /user/u55/id/10/

val page : Option[(String, Int)]= handleLink(
     "user/u55/id/10/",
     "user" ~: string ~: "id" ~: int ~: eop
   ) (userName => userId => (userName, userId))
   // Generate: Some(("u55", 10))
```

Bidirectional routing

Generate URL from a path

```
val url = genLink("user" ~: string ~: "id" ~: int ~: eop)("u55")(10)
   // Generate : /user/u55/id/10/

val page : Option[Html]= handleLink(
     "user/u55/id/10/",
     "user" ~: string ~: "id" ~: int ~: eop
   ) (userName => userId => renderUserPage(userName, userId))
   // Generate: Some(("u55", 10))
```

Generate controller from a path

Bidirectional routing

Generate URL from a path

```
val url = genLink("user" ~: string ~: "id" ~: int ~: eop)("u55")(10)
    // Generate : /user/u55/id/10/

val page : Option[Html]= handleLink(
      "user/u55/id/10/",
      "user" ~: string ~: "id" ~: int ~: eop
    ) (userName => userId => renderUserPage(userName, userId))
    // Generate: Some(("u55", 10))
```

Generate controller from a path

Bidirectional routing

Generate URL from a path

```
val url = genLink("user" ~: string ~: "id" ~: int ~: eop)("u55")(10)
   // Generate : /user/u55/id/10/

val page : Option[Html]= handleLink(
     "user/u55/id/10/",
     "user" ~: string ~: "id" ~: int ~: eop
   ) (userName => userId => renderUserPage(userName, userId))
   // Generate: Some(("u55", 10))
```

Generate controller from a path

Using a technique similar to Typelevel List

Bidirectional routing

```
enum V[T]:
 case String extends V[String]
 case Int    extends V[Int]
 case Bool   extends V[Boolean]
```

```
enum Path[TY, W]:
 case Eop[W]()
     extends Path[W, W]

 case Const(x: String, xs: Path[TY, W])

 case Var[A, TY, W](x: V[A], xs: Path[TY, W])
     extends Path[A => TY, W]

 def ~:[A](x: V[A]) : Path[A => TY, W] = Path.Var(x, this)
 def ~:(x: String) : Path[TY, W]  = Path.Const(x, this)

def eop[W] : Path[W, W] = Path.Eop[W]()
val string = V.String
val int = V.Int
val bool = V.Bool
```

Bidirectional routing

```
enum V[T]:
  case String extends V[String]
  case Int    extends V[Int]
  case Bool   extends V[Boolean]
```

Ensure:

```
TY = X => W
```

```
enum Path[TY, W]:
  case Eop[W]()
      extends Path[W, W]

  case Const(x: String, xs: Path[TY, W])

  case Var[A, TY, W](x: V[A], xs: Path[TY, W])
      extends Path[A => TY, W]

  def ~:[A](x: V[A]) : Path[A => TY, W] = Path.Var(x, this)
  def ~:(x: String) : Path[TY, W]   = Path.Const(x, this)

def eop[W] : Path[W, W] = Path.Eop[W]()
val string = V.String
val int = V.Int
val bool = V.Bool
```

Bidirectional routing

```scala
enum V[T]:
  case String extends V[String]
  case Int    extends V[Int]
  case Bool   extends V[Boolean]
```

Representing type, typelevel

Ensure:
TY = X => W

```scala
enum Path[TY, W]:
 case Eop[W]()
     extends Path[W, W]

 case Const(x: String, xs: Path[TY, W])

 case Var[A, TY, W](x: V[A], xs: Path[TY, W])
     extends Path[A => TY, W]

 def ~:[A](x: V[A]) : Path[A => TY, W] = Path.Var(x, this)
 def ~:(x: String) : Path[TY, W]  = Path.Const(x, this)

def eop[W] : Path[W, W] = Path.Eop[W]()
val string = V.String
val int = V.Int
val bool = V.Bool
```

Bidirectional routing

Ensure:

```
TY = X => W
```

```
enum V[T]:
 case String extends V[String]
 case Int    extends V[Int]
 case Bool   extends V[Boolean]
```

Representing type, typelevel

```
enum Path[TY, W]:
 case Eop[W]()
     extends Path[W, W]

 case Const(x: String, xs: Path[TY, W])

 case Var[A, TY, W](x: V[A], xs: Path[TY, W])
     extends Path[A => TY, W]

 def ~:[A](x: V[A]) : Path[A => TY, W] = Path.Var(x, this)
 def ~:(x: String) : Path[TY, W]  = Path.Const(x, this)

def eop[W] : Path[W, W] = Path.Eop[W]()
val string = V.String
val int = V.Int
val bool = V.Bool
```

Same as our List but constraint by V[T].

Bidirectional routing

Ensure:

```
TY = X => W
```

```
enum V[T]:
 case String extends V[String]
 case Int    extends V[Int]
 case Bool   extends V[Boolean]
```

Representing type, typelevel

Same as our List but constraint by V[T].

```
def genLink[TY]
   (path: Path[TY, String]) : TY = …
```

```
enum Path[TY, W]:
 case Eop[W]()
     extends Path[W, W]

 case Const(x: String, xs: Path[TY, W])

 case Var[A, TY, W](x: V[A], xs: Path[TY, W])
     extends Path[A => TY, W]

 def ~:[A](x: V[A]) : Path[A => TY, W] = Path.Var(x, this)
 def ~:(x: String) : Path[TY, W]   = Path.Const(x, this)

def eop[W] : Path[W, W] = Path.Eop[W]()
val string = V.String
val int = V.Int
val bool = V.Bool
```

A constant does not create Hole

Bidirectional routing

Ensure:

```
TY = X => W
```

```scala
enum V[T]:
  case String extends V[String]
  case Int    extends V[Int]
  case Bool   extends V[Boolean]
```

Representing type, typelevel

```scala
enum Path[TY, W]:
 case Eop[W]()
     extends Path[W, W]

 case Const(x: String, xs: Path[TY, W])

 case Var[A, TY, W](x: V[A], xs: Path[TY, W])
     extends Path[A => TY, W]

 def ~:[A](x: V[A]) : Path[A => TY, W] = Path.Var(x, this)
 def ~:(x: String) : Path[TY, W]  = Path.Const(x, this)

def eop[W] : Path[W, W] = Path.Eop[W]()
val string = V.String
val int = V.Int
val bool = V.Bool
```

A constant does not create Hole

Same as our List but constraint by V[T].

```scala
def genLink[TY]
   (path: Path[TY, String]) : TY = …

def handleLink[TY, W]
    (uri: String, path: Path[TY, W])
    (controller: TY) : Option[W] = …
```

To conclude

*"A Generalized Algebraic Data Type **is a sum type that allows its constructors to be non-surjective** on one or more of its type parameters and **introduces local type-equality constraints in pattern-matching branches,** making the expression of existential types trivial"*

To conclude

*"A Generalized Algebraic Data Type **is a sum type that allows its constructors to be non-surjective** on one or more of its type parameters and **introduces local type-equality constraints in pattern-matching branches,** making the expression of existential types trivial"*

Thank You!