Functional Programming

GADT: Generalized Algebraic DataType

Dr. Gabriel Radanne

Albert-Ludwigs-Universität Freiburg, Germany

WS 2024/25

Interpreters, again

Language definition

```haskell
data Term = I Integer
          | Add Term Term
            deriving (Eq, Show)
```

Interpreters, again

Evaluation

```haskell
eval :: Term -> Integer
eval (I n) = n
eval (Add t u) = eval t + eval u
```

A language with multiple types

Language definition

```haskell
data Term = I Integer
          | B Bool
          | Add Term Term
          | Eql Term Term
            deriving (Eq, Show)
```

A language with multiple types

Evaluation

```
type Value = Int Integer | Bool Bool
           deriving Show

eval :: Term -> Value
eval (I n) = Int n
eval (B b) = Bool b
eval (Add t t') = case (eval t, eval t') of
                  (Int i, Int i2) -> Int (i + i2)
eval (Eql t t') = case (eval t, eval t') of
                  (Int i, Int i2) -> Bool (i == i2)
                  (Bool i, Bool i2) -> Bool (i == i2)
```

Issues with the interpreter

- The interpreter can fail because of type mismatch.
- We need to deal with failures manually (for instance by making the interpreter monadic).
- The more values we have in the language, the more complicated it becomes.
- The Haskell type system does not help us.

Issues with the interpreter

- The interpreter can fail because of type mismatch.
- We need to deal with failures manually (for instance by making the interpreter monadic).
- The more values we have in the language, the more complicated it becomes.
- The Haskell type system does not help us.

Issues with the interpreter

- The interpreter can fail because of type mismatch.
- We need to deal with failures manually (for instance by making the interpreter monadic).
- The more values we have in the language, the more complicated it becomes.
- The Haskell type system does not help us.

Issues with the interpreter

- The interpreter can fail because of type mismatch.
- We need to deal with failures manually (for instance by making the interpreter monadic).
- The more values we have in the language, the more complicated it becomes.
- The Haskell type system does not help us.

GADTs to the rescue!

Algebraic Data Type

```haskell
data Maybe a =
    Nothing
  | Just a
```

GADTs to the rescue!

Generalized Algebraic Data Type

```haskell
{-# LANGUAGE GADTs #-}

data Maybe a where
  Nothing :: Maybe a
  Just :: a -> Maybe a
```

- Now we also specify the return type of the data constructors!
- We cannot change the type constructor **Maybe**, but its arguments may vary

GADTs to the rescue!

Generalized Algebraic Data Type

```haskell
{-# LANGUAGE GADTs #-}

data Maybe a where
  Nothing :: Maybe a
  Just :: a -> Maybe a
```

- Now we also specify the return type of the data constructors!
- We cannot change the type constructor **Maybe**, but its arguments may vary

Language definition without GADTs

```haskell
data Term =
    I Integer
  | B Bool
  | Add Term Term
  | Eql Term Term
```

Language definition with GADT syntax

```haskell
data Term where
  I :: Integer -> Term
  B :: Bool -> Term
  Add :: Term -> Term -> Term
  Eql :: Term -> Term -> Term
```

Language definition with GADTs

```haskell
data Term a where
  I :: Integer -> Term Integer
  B :: Bool -> Term Bool
  Add :: Term (?) -> Term (?) -> Term (?)
  Eql :: Term (?) -> Term (?) -> Term (?)
```

Language definition with GADTs

```
data Term a where
  I :: Integer -> Term Integer
  B :: Bool -> Term Bool
  Add :: Term Integer -> Term Integer -> Term Integer
  Eql :: Term (?) -> Term (?) -> Term (?)
```

Language definition with GADTs

```haskell
data Term a where
  I :: Integer -> Term Integer
  B :: Bool -> Term Bool
  Add :: Term Integer -> Term Integer -> Term Integer
  Eql :: (Eq x) => Term x -> Term x -> Term Bool
```

- Read the last line as "the exists some type `x` such that `x` is an instance of **`Eq`** and the two arguments have the **same** type `Term x`."

Evaluation for GADTs

```haskell
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors **`Int`** and **`Bool`**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors **Int** and **Bool**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors `Int` and `Bool`.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors **`Int`** and **`Bool`**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```haskell
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors **`Int`** and **`Bool`**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors **Int** and **Bool**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called "tag-less", because it does not require type tags like the data constructors **Int** and **Bool**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

Evaluation for GADTs

```haskell
eval :: Term a -> a -- This type annotation is mandatory
eval (I i) = i
eval (B b) = b
eval (Add t t') = eval t + eval t'
eval (Eql t t') = eval t == eval t'
```

- This kind of interpreter is called “tag-less”, because it does not require type tags like the data constructors **`Int`** and **`Bool`**.
- Pattern matching specializes the type a according to the return type of the constructor.
- The corresponding right hand side is checked against this specialization of type a.

What about functions?

We want to add functions to our language.

First try

```haskell
data FExp a where
  Var :: FExp a
  Lam :: FExp b -> FExp (a -> b)
  App :: FExp (a -> b) -> FExp a -> FExp b
```

This doesn't work: not enough type information for variables and lambdas.

What about functions?

We want to add functions to our language.

First try

```
data FExp a where
  Var :: FExp a
  Lam :: FExp b -> FExp (a -> b)
  App :: FExp (a -> b) -> FExp a -> FExp b
```

This doesn't work: not enough type information for variables and lambdas.

What about functions?

We want to add functions to our language.

First try

```haskell
data FExp a where
  Var :: FExp a
  Lam :: FExp b -> FExp (a -> b)
  App :: FExp (a -> b) -> FExp a -> FExp b
```

This doesn't work: not enough type information for variables and lambdas.

Existential Types

In the definition of `App`, `a` is present in the argument types, but not in the result. Such a type variable stands for an *existential type*:
For each use of `App` there is some type `a` so that the types of the subterms work out.

```
App :: FExp (a -> b) -> FExp a -> FExp b
```

Demo!

Existential Types

In the definition of `App`, a is present in the argument types, but not in the result. Such a type variable stands for an *existential type*:
For each use of `App` there is some type a so that the types of the subterms work out.

```
App :: FExp (a -> b) -> FExp a -> FExp b
```

Demo!

Back to functions!

Type definition

```
data FExp e a where
  App :: FExp e (a -> b) -> FExp e a -> FExp e b
  Lam :: FExp (a, e) b -> FExp e (a -> b)
  Var :: Nat e a -> FExp e a

data Nat e a where
  Zero :: Nat (a, b) a
  Succ :: Nat e a -> Nat (b, e) a
```

Demo!

Back to functions!

Type definition

```
data FExp e a where
  App :: FExp e (a -> b) -> FExp e a -> FExp e b
  Lam :: FExp (a, e) b -> FExp e (a -> b)
  Var :: Nat e a -> FExp e a

data Nat e a where
  Zero :: Nat (a, b) a
  Succ :: Nat e a -> Nat (b, e) a
```

Demo!

Back to functions!

Type definition

```
data FExp e a where
  App :: FExp e (a -> b) -> FExp e a -> FExp e b
  Lam :: FExp (a, e) b -> FExp e (a -> b)
  Var :: Nat e a -> FExp e a

data Nat e a where
  Zero :: Nat (a, b) a
  Succ :: Nat e a -> Nat (b, e) a
```

Demo!

Back to functions!

Type definition

```
data FExp e a where
  App :: FExp e (a -> b) -> FExp e a -> FExp e b
  Lam :: FExp (a, e) b -> FExp e (a -> b)
  Var :: Nat e a -> FExp e a

data Nat e a where
  Zero :: Nat (a, b) a
  Succ :: Nat e a -> Nat (b, e) a
```

Demo!

Origin of GADTs

- Comparatively recent extension to Haskell's type system.
  Invented by 3 different groups:
  - Augustsson & Petersson (1994): Silly Type Families
  - Cheney & Hinze (2003): First-Class Phantom Types.
  - Xi, Chen & Chen (2003): Guarded Recursive Datatype Constructors.
- Type *checking* is decidable.
- Type *inference* is undecidable.
- Pattern matching is more complicated.

Origin of GADTs

- Comparatively recent extension to Haskell's type system.
  Invented by 3 different groups:
  - Augustsson & Petersson (1994): Silly Type Families
  - Cheney & Hinze (2003): First-Class Phantom Types.
  - Xi, Chen & Chen (2003): Guarded Recursive Datatype Constructors.
- Type *checking* is decidable.
- Type *inference* is undecidable.
- Pattern matching is more complicated.

Origin of GADTs

- Comparatively recent extension to Haskell's type system.
  Invented by 3 different groups:
  - Augustsson & Petersson (1994): Silly Type Families
  - Cheney & Hinze (2003): First-Class Phantom Types.
  - Xi, Chen & Chen (2003): Guarded Recursive Datatype Constructors.
- Type *checking* is decidable.
- Type *inference* is undecidable.
- Pattern matching is more complicated.

Origin of GADTs

- Comparatively recent extension to Haskell's type system.
  Invented by 3 different groups:
  - Augustsson & Petersson (1994): Silly Type Families
  - Cheney & Hinze (2003): First-Class Phantom Types.
  - Xi, Chen & Chen (2003): Guarded Recursive Datatype Constructors.
- Type *checking* is decidable.
- Type *inference* is undecidable.
- Pattern matching is more complicated.

Wrapping up

- GADTs can express extra properties in types:
    -
- We leverage Haskell's type system.
- GADTs do not solve *all* the problems. For example, you can try to write a function of type

```haskell
parse :: String -> Expr a
```

GADTs can be combined with other Haskell features such as type classes and type families.

- GADTs become very complex when the domain grows!

Wrapping up

- GADTs can express extra properties in types:
- We leverage Haskell's type system.
- GADTs do not solve *all* the problems. For example, you can try to write a function of type

```haskell
parse :: String -> Expr a
```

- GADTs can be combined with other Haskell features such as type classes and type families.
- GADTs become very complex when the domain grows!

Wrapping up

- GADTs can express extra properties in types:
  -
- We leverage Haskell's type system.
- GADTs do not solve *all* the problems. For example, you can try to write a function of type

```
parse :: String -> Expr a
```

GADTs can be combined with other Haskell features such as type classes and type families.

- GADTs become very complex when the domain grows!

Wrapping up

- GADTs can express extra properties in types:
  -
- We leverage Haskell's type system.
- GADTs do not solve *all* the problems. For example, you can try to write a function of type

```
parse :: String -> Expr a
```

GADTs can be combined with other Haskell features such as type classes and type families.

- GADTs become very complex when the domain grows!

Wrapping up

- GADTs can express extra properties in types:
  -
- We leverage Haskell's type system.
- GADTs do not solve *all* the problems. For example, you can try to write a function of type

```haskell
parse :: String -> Expr a
```

- GADTs can be combined with other Haskell features such as type classes and type families.
- GADTs become very complex when the domain grows!

Wrapping up

- GADTs can express extra properties in types:
  -
- We leverage Haskell's type system.
- GADTs do not solve *all* the problems. For example, you can try to write a function of type

```
parse :: String -> Expr a
```

  GADTs can be combined with other Haskell features such as type classes and type families.
- GADTs become very complex when the domain grows!