Pattern matching exhaustiveness for GADTs

George Karachalias

School of Electrical and Computer Engineering
National Technical University of Athens

December 27, 2013

Today we are going to talk about:

Today we are going to talk about:

- Algebraic Data Types

Today we are going to talk about:

- Algebraic Data Types
- Generalized Algebraic Data Types

Today we are going to talk about:

- Algebraic Data Types
- Generalized Algebraic Data Types
- The Glasgow Haskell Compiler

Today we are going to talk about:

- Algebraic Data Types
- Generalized Algebraic Data Types
- The Glasgow Haskell Compiler
- Exhaustiveness of GADT Matches

What is an Algebraic Data Type?

What is an Algebraic Data Type?

An ordered pair, consisting of:

What is an Algebraic Data Type?

An ordered pair, consisting of:

- A *Type Constructor*, i.e. a type-level function that results to the type we are defining.

What is an Algebraic Data Type?

An ordered pair, consisting of:

- A *Type Constructor*, i.e. a type-level function that results to the type we are defining.
- A set of *Data Constructors*, i.e. a set of value-level functions that have as result values of the declared type.

What is an Algebraic Data Type?

An ordered pair, consisting of:

- A *Type Constructor*, i.e. a type-level function that results to the type we are defining.
- A set of *Data Constructors*, i.e. a set of value-level functions that have as result values of the declared type.

```
data TypeConstructor  type_variables_list
   = DataConstructor1 type_parameter_list1
   | DataConstructor2 type_parameter_list2
   | ...
   | DataConstructorN type_parameter_listN
```

Some ADT Examples

Some ADT Examples

1. Enumeration Types

```
data Bool = True | False
```

Some ADT Examples

1. Enumeration Types

```
data Bool = True | False
```

2. Wrapper Types

```
data Height = Height Float
data Width  = Width  Float
```

Some ADT Examples

1. Enumeration Types

```
data Bool = True | False
```

2. Wrapper Types

```
data Height = Height Float
data Width  = Width  Float
```

3. Polymorphism

```
data Tuple a b = MkTuple a b
```

Some ADT Examples

1. Enumeration Types

```
data Bool = True | False
```

2. Wrapper Types

```
data Height = Height Float
data Width  = Width  Float
```

3. Polymorphism

```
data Tuple a b = MkTuple a b
```

4. Recursive Types

```
data List a = Nil | Cons a (List a)
```

A Larger Example

Suppose we want to create an interpreter for a simply typed expression language with integers, booleans and pairs. For this purpose, we are going to need:

Suppose we want to create an interpreter for a simply typed expression language with integers, booleans and pairs. For this purpose, we are going to need:

- A data type to represent terms/expressions

Suppose we want to create an interpreter for a simply typed expression language with integers, booleans and pairs. For this purpose, we are going to need:

- A data type to represent terms/expressions
- A simple lexer/parser

Suppose we want to create an interpreter for a simply typed expression language with integers, booleans and pairs. For this purpose, we are going to need:

- A data type to represent terms/expressions
- A simple lexer/parser
- An evaluating function

Representing Terms

For starters, let us concern ourselves only with the data type for the representation of terms.

Representing Terms

For starters, let us concern ourselves only with the data type for the representation of terms.
A possible definition would look like the following:

Representing Terms

For starters, let us concern ourselves only with the data type for the representation of terms.
A possible definition would look like the following:

```
data Term
    = Lit Int
    | Inc Term
    | IsZ Term
    | If Term Term Term
    | Pair Term Term
    | Fst Term
    | Snd Term
```

Representing Terms

Or, with explicit typing:

Representing Terms

Or, with explicit typing:

```
data Term where
  Lit  :: Int  -> Term
  Inc  :: Term -> Term
  IsZ  :: Term -> Term
  If   :: Term -> Term -> Term -> Term
  Pair :: Term -> Term -> Term
  Fst  :: Term -> Term
  Snd  :: Term -> Term
```

Evaluation of Terms

The evaluation of a term can result to an integer, a boolean, or a pair of two values:

Evaluation of Terms

The evaluation of a term can result to an integer, a boolean, or a pair of two values:

```
eval (Lit 7)                  = 7     -- :: Int
eval (IsZ (Lit 7))            = False -- :: Bool
eval (Pair (Lit 6) (Lit 7)) = (6,7) -- :: (Int, Int)
```

Evaluation of Terms

The evaluation of a term can result to an integer, a boolean, or a pair of two values:

```
eval (Lit 7)                 = 7     -- :: Int
eval (IsZ (Lit 7))           = False -- :: Bool
eval (Pair (Lit 6) (Lit 7)) = (6,7) -- :: (Int, Int)
```

Hence, we have to define one more ADT for values:

Evaluation of Terms

The evaluation of a term can result to an integer, a boolean, or a pair of two values:

```
eval (Lit 7)                 = 7     -- :: Int
eval (IsZ (Lit 7))           = False -- :: Bool
eval (Pair (Lit 6) (Lit 7)) = (6,7) -- :: (Int, Int)
```

Hence, we have to define one more ADT for values:

```
data Value
   = VI Int          -- Integer Value
   | VB Bool         -- Boolean Value
   | VP Value Value -- Pair    Value
```

Evaluation of Terms

Now we can write the `eval` function:

Evaluation of Terms

Now we can write the `eval` function:

```
eval :: Term -> Value
eval (Lit x) = VI x
eval (Inc t)
  | VI x <- eval t = VI (x+1)
  | otherwise = error "Inc: Not an Int"
eval (IsZ t)
  | VI x <- eval t = VB (x==0)
  | otherwise = error "IsZ: Not a Bool"
eval (If t x y)
  | VB b <- eval t = if b then eval x
                          else eval y
  | otherwise = error "If: Not a Bool"
```

Evaluation of Terms

..and the rest of it:

```
eval (Pair x y) = VP (eval x) (eval y)
eval (Fst t)
  | VP v1 _ <- eval t = v1
  | otherwise = error "Fst: Not a Pair"
eval (Snd t)
  | VP _ v2 <- eval t = v2
  | otherwise = error "Snd: Not a Pair"
```

Issues

Issues

1. **The ADT does not enforce type-checking**

Issues

1. **The ADT does not enforce type-checking**
   Nothing prevents the formation of ill-typed terms like the following:

```
IsZ (IsZ (Lit 42))
Fst (Lit 5)
```

Issues

1. **The ADT does not enforce type-checking**
   Nothing prevents the formation of ill-typed terms like the following:

```
IsZ (IsZ (Lit 42))
Fst (Lit 5)
```

2. **Evaluator's run-time checks**

Issues

1. **The ADT does not enforce type-checking**
   Nothing prevents the formation of ill-typed terms like the following:

```
IsZ (IsZ (Lit 42))
Fst (Lit 5)
```

2. **Evaluator's run-time checks**
   Due to (1), we have to manually check that the recursive calls to `eval` return the expected type of value (hence the guards).

Issues

1. **The ADT does not enforce type-checking**
   Nothing prevents the formation of ill-typed terms like the following:

```
IsZ (IsZ (Lit 42))
Fst (Lit 5)
```

2. **Evaluator's run-time checks**
   Due to (1), we have to manually check that the recursive calls to `eval` return the expected type of value (hence the guards).

   - Tiresome for the programmer

Issues

1. **The ADT does not enforce type-checking**
   Nothing prevents the formation of ill-typed terms like the following:

```
IsZ (IsZ (Lit 42))
Fst (Lit 5)
```

2. **Evaluator's run-time checks**
   Due to (1), we have to manually check that the recursive calls to `eval` return the expected type of value (hence the guards).
   - Tiresome for the programmer
   - Additional overhead

How can we make our solution more elegant?

How can we make our solution more elegant?

With the expressive power of

Generalized Algebraic Data Types

1) Each data constructor may return a different instantiation of the abstract type:

1) Each data constructor may return a different instantiation of the abstract type:

```
data Term a where
 Lit  :: Int -> Term Int
 Inc  :: Term Int -> Term Int
 IsZ  :: Term Int -> Term Bool
 If   :: Term Bool -> Term a -> Term a -> Term a
 Pair :: Term a -> Term b -> Term (a,b)
 Fst  :: Term (a,b) -> Term a
 Snd  :: Term (a,b) -> Term b
```

2) Alternatively, all data constructors have the same return type, but their type may quantify over constraints (*qualified types*):

2) Alternatively, all data constructors have the same return type, but their type may quantify over constraints (*qualified types*):

```
data Term a where
 Lit  :: forall a. (a~Int)  => Int -> Term a
 Inc  :: forall a. (a~Int)  => Term Int -> Term a
 IsZ  :: forall a. (a~Bool) => Term Int -> Term a
 If   :: forall a. Term Bool -> Term a -> Term a -> Term a
 Pair :: forall a b c. (a~(b,c)) => Term b -> Term c -> Term a
 Fst  :: forall a b. (a~b)  => Term (b,c) -> Term a
 Snd  :: forall a c. (a~c)  => Term (b,c) -> Term a
```

Now, the implementation of the evaluating function is absolutely straightforward and its type trivial:

Now, the implementation of the evaluating function is absolutely straightforward and its type trivial:

```
eval :: Term a -> a
eval (Lit i)    = i
eval (Inc t)    = eval t + 1
eval (IsZ t)    = eval t == 0
eval (If t a b) = if eval t then eval a else eval b
eval (Pair a b) = (eval a, eval b)
eval (Fst t)    = fst (eval t)
eval (Snd t)    = snd (eval t)
```

One More Example: Vectors

```
data Vec a n where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)
```

One More Example: Vectors

```
data Vec a n where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)

vhead :: Vec a (Succ n) -> a
vhead (VCons x _) = x
```

One More Example: Vectors

```
data Vec a n where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)

vhead :: Vec a (Succ n) -> a
vhead (VCons x _) = x


vmap :: (a -> b) -> Vec a n -> Vec b n
vmap f VNil          = VNil
vmap f (VCons x xs) = VCons (f x) (vmap f xs)
```

```
data Vec a n where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)

vhead :: Vec a (Succ n) -> a
vhead (VCons x _) = x


vmap :: (a -> b) -> Vec a n -> Vec b n
vmap f VNil          = VNil
vmap f (VCons x xs) = VCons (f x) (vmap f xs)
```

```haskell
data Vec a n where
  VNil  :: Vec a Zero
  VCons :: a -> Vec a n -> Vec a (Succ n)

vhead :: Vec a (Succ n) -> a
vhead (VCons x _) = x
vhead VNil        = error "Inaccessible Code!"

vmap :: (a -> b) -> Vec a n -> Vec b n
vmap f VNil          = VNil
vmap f (VCons x xs) = VCons (f x) (vmap f xs)
```

A tedious situation

A tedious situation

```
vzip :: Vec a n -> Vec b n -> Vec (a,b) n
vzip VNil         VNil         = VNil
vzip (VCons x xs) (VCons y ys) = VCons (x,y) (vzip xs ys)
```

A tedious situation

```
vzip :: Vec a n -> Vec b n -> Vec (a,b) n
vzip VNil         VNil         = VNil
vzip (VCons x xs) (VCons y ys) = VCons (x,y) (vzip xs ys)
```

ghc complains with the following warning:

```
    Warning: Pattern match(es) are non-exhaustive
             In an equation for `vzip':
                 Patterns not matched:
                     VNil (VCons _ _)
                     (VCons _ _) VNil
```

A tedious situation

```
vzip :: Vec a n -> Vec b n -> Vec (a,b) n
vzip VNil         VNil         = VNil
vzip (VCons x xs) (VCons y ys) = VCons (x,y) (vzip xs ys)
vzip VNil         (VCons _ _ ) = error "Inaccessible Code!"
vzip (VCons _ _ ) VNil         = error "Inaccessible Code!"
```

A tedious situation

```
vzip :: Vec a n -> Vec b n -> Vec (a,b) n
vzip VNil         VNil         = VNil
vzip (VCons x xs) (VCons y ys) = VCons (x,y) (vzip xs ys)
vzip VNil         (VCons _ _ ) = error "Inaccessible Code!"
vzip (VCons _ _ ) VNil         = error "Inaccessible Code!"
```

ghc complains with the following error:

```
    Couldn't match type `'Zero' with `Succ n1'
    Inaccessible code in
      a pattern with constructor
        VCons :: forall a (n :: Nat). a -> Vec a n
                                          -> Vec a (Succ n)
```

Suppressing the warning

```
vzip :: Vec a n -> Vec b n -> Vec (a,b) n
vzip VNil         VNil         = VNil
vzip (VCons x xs) (VCons y ys) = VCons (x,y) (vzip xs ys)
vzip _            _            = error "Inaccessible Code!"
```

Identifying the problem

Why missing?

Why missing?

GHC does not take into account local constraints when detecting missing patterns.

Why missing?

GHC does not take into account local constraints when detecting missing patterns.

Recall the definition of vectors:

```
data Vec a n where
  VNil  :: forall a n.   (n~Zero)   => Vec a n
  VCons :: forall a n m. (n~Succ m) => a -> Vec a m -> Vec a n
```

Why missing?

GHC does not take into account local constraints when detecting missing patterns.

Recall the definition of vectors:

```
data Vec a n where
  VNil  :: forall a n.   (n~Zero)   => Vec a n
  VCons :: forall a n m. (n~Succ m) => a -> Vec a m -> Vec a n
```

and the type of function vhead:

```
vhead :: Vec a (Succ n) -> a
```

Why missing?

GHC does not take into account local constraints when detecting missing patterns.

Recall the definition of vectors:

```
data Vec a n where
  VNil  ::                                       Vec a n
  VCons ::                                       a -> Vec a m -> Vec a n
```

and the type of function vhead:

```
vhead :: Vec a (Succ n) -> a
```

Why is it an error then?

Why is it an error then?

The type checker of GHC takes into account the local constraints introduced by data constructors (as it should!).

Why is it an error then?

The type checker of GHC takes into account the local constraints introduced by data constructors (as it should!).

```
data Vec a n where
  VNil  :: forall a n.   (n~Zero)   => Vec a n
  VCons :: forall a n m. (n~Succ m) => a -> Vec a m -> Vec a n

vhead :: Vec a (Succ n) -> a
vhead (VCons x _)       =  x
vhead VNil              =  error "Inaccessible Code!"
```

Why can we overcome it?

Why can we overcome it?

GHC's mechanism for the detection of overlapping patterns is also incomplete.

```
vhead :: Vec a (Succ n) -> a
vhead (VCons x _)       =  x
vhead _                 =  error "Inaccessible Code!"
```

A Solution

Idea

Idea

1. Call the previous mechanism to detect the (possibly more than actual) missing patterns.

Idea

1. Call the previous mechanism to detect the (possibly more than actual) missing patterns.
2. For every missing pattern collect the constraints that would introduce, if it appeared.

Idea

1. Call the previous mechanism to detect the (possibly more than actual) missing patterns.
2. For every missing pattern collect the constraints that would introduce, if it appeared.
3. Call the constraint solver for each set of constraints (taking into consideration the program constraints) and, depending on the result:

Idea

1. Call the previous mechanism to detect the (possibly more than actual) missing patterns.
2. For every missing pattern collect the constraints that would introduce, if it appeared.
3. Call the constraint solver for each set of constraints (taking into consideration the program constraints) and, depending on the result:
    - If the solver fails, the pattern under examination *cannot* really appear (with respect to the context) and we should not issue a warning.

Idea

1. Call the previous mechanism to detect the (possibly more than actual) missing patterns.
2. For every missing pattern collect the constraints that would introduce, if it appeared.
3. Call the constraint solver for each set of constraints (taking into consideration the program constraints) and, depending on the result:
   - If the solver fails, the pattern under examination *cannot* really appear (with respect to the context) and we should not issue a warning.
   - If the solver succeeds, the pattern could appear in the specific context and a warning should be issued, since the pattern is actually missing.

Key Points

Key Points

- The missing patterns that GHC issues warnings for are always a **superset** of the patterns that are actually missing.

Key Points

- The missing patterns that GHC issues warnings for are always a **superset** of the patterns that are actually missing.
- GHC's type checker (via constraint solving) detects inaccessible patterns.

Advantages

Advantages

Simplicity Based on mechanisms that are supported by most (if not all) languages that support ADTs:

1. Contraint Solving
2. Non-Exhaustiveness Check

Advantages

Simplicity Based on mechanisms that are supported by most (if not all) languages that support ADTs:

1. Contraint Solving
2. Non-Exhaustiveness Check

Efficiency Patterns usually introduce only a few constraints.

Advantages

Simplicity Based on mechanisms that are supported by most (if not all) languages that support ADTs:

1. Contraint Solving
2. Non-Exhaustiveness Check

Efficiency Patterns usually introduce only a few constraints.

Consistency Preserves the properties of the type system. We consider a pattern missing, only if the typechecker *allows* us to. If the semantics change, so does the behaviour of our mechanism.

Disadvantage

Disadvantage

Recovery In the cases of GADT constructors that are not actually missing, the constraint solver must fail and recover. Hence, the compilation for programs that make heavy use of GADTs may delay a bit.

Implementation Results

Performance (Part 1)

Performance (Part 1)

- **GHC Build**
  Both about 1 hour and 47 minutes. (22 secs faster)

Performance (Part 1)

- **GHC Build**
  Both about 1 hour and 47 minutes. (22 secs faster)
- **Testsuite Build**
  Both about 4 hours and 58 minutes. (17 secs slower)

Performance (Part 2)

```
data F :: * -> * -> * -> * where
  MkF1  :: Int  -> Int  -> Int  -> F Int  Int  Int
  MkF2  :: Int  -> Int  -> Char -> F Int  Int  Char
  MkF3  :: Int  -> Int  -> Bool -> F Int  Int  Bool
  ...
  MkF27 :: Bool -> Bool -> Bool -> F Bool Bool Bool
```

Performance (Part 2)

```
data F :: * -> * -> * -> * where
  MkF1  :: Int  -> Int  -> Int  -> F Int  Int  Int
  MkF2  :: Int  -> Int  -> Char -> F Int  Int  Char
  MkF3  :: Int  -> Int  -> Bool -> F Int  Int  Bool
  ...
  MkF27 :: Bool -> Bool -> Bool -> F Bool Bool Bool
```

- **Non Exhaustive**

  ```
  func1 :: F a b c -> Int
  ```

Performance (Part 2)

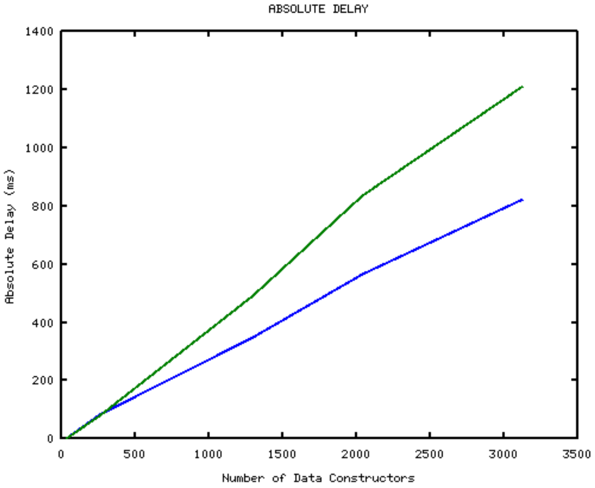
```
data F :: * -> * -> * -> * where
  MkF1  :: Int  -> Int  -> Int  -> F Int  Int  Int
  MkF2  :: Int  -> Int  -> Char -> F Int  Int  Char
  MkF3  :: Int  -> Int  -> Bool -> F Int  Int  Bool
  ...
  MkF27 :: Bool -> Bool -> Bool -> F Bool Bool Bool
```

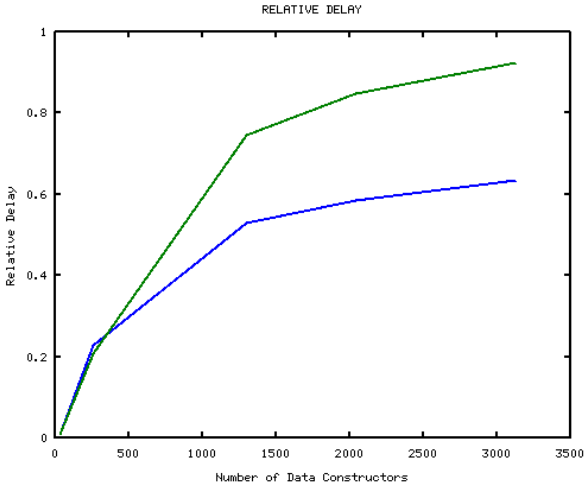
- **Non Exhaustive**

  `func1 :: F a b c -> Int`

- **Exhaustive**

  `func1 :: F a a a -> Int`


ABSOLUTE DELAY
1400
1200
1000
800
600
400
200
0
Absolute Delay (ms)
0
500
1000
1500
2000
2500
3000
3500
Number of Data Constructors


RELATIVE DELAY
Relative Delay
Number of Data Constructors
0
0.2
0.4
0.6
0.8
1
500
1000
1500
2000
2500
3000
3500

Correctness

- **Testsuite Results**

	Adjusted GHC	Vanilla GHC
expected passes	11122	11126
expected failures	141	141
unexpected passes	2	2
unexpected failures	72	68

- **Testsuite Results**

	Adjusted GHC	Vanilla GHC
expected passes	11122	11126
expected failures	141	141
unexpected passes	2	2
unexpected failures	72	68

- **GHC Tickets**
    - ~~#366~~
    - ~~#2006~~
    - #3927
    - #4139 (half)

Other Examples

Case Expressions

```
data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int  Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int  Char

f :: T a a -> Int
f x = case x of      -- should not issue warning
        T1 i j -> i+j
```

Pattern Matching

```
data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int  Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int  Char

f :: T a a -> Int
f (T1 i j) = i+j     -- should not issue warning
```

Let Bindings

```
data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int  Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int  Char

f :: T a a -> Int
f x = let T1 i j = x -- should not issue warning
      in  i+j
```

Where Bindings

```
data T :: * -> * -> * where
  T1 :: Int  -> Int  -> T Int  Int
  T2 :: Char -> Int  -> T Char Int
  T3 :: Int  -> Char -> T Int  Char

f :: T a a -> Int
f x = i+j              -- should not issue warning
  where T1 i j = x
```

Nested Patterns

```
data X :: * -> * -> * where
  X1 :: X Char Char
  X2 :: X Int  Char

data Y :: * -> * -> * where
  Y1 :: Int  -> Char -> Y Int  Char
  Y2 :: Char -> Char -> Y Char Char
  Y3 :: a    -> b    -> Y a    b

fxy :: Y (X a a) (X a a) -> a
fxy value = case value of
              Y3 X1 X1 -> 'a'
```

A Tricky Example

```
data T a where
  T1 :: T Int
  T2 :: T Bool

f1 :: T a -> T a -> Bool
f1 T1 T1 = True
f1 T2 T2 = False

f2 :: T a -> T a -> Bool
f2 T1 (T1 :: T Int ) = True
f2 T2 (T2 :: T Bool) = False
```

Data Kinds & Type Classes

```
data Vec v a where
  Nil :: Vec '[] a
  Vec :: a -> Vec v a -> Vec (() ': v) a

instance Eq a => Eq (Vec v a) where
    Nil        == Nil        = True
    (Vec x xs) == (Vec y ys) = x == y && xs == ys
```

What's Next?

- Forthcoming version of GHC (7.8.1)
- Overlapping Patterns (#595 and at least 8 more tickets)

Questions?