The Curious case of Pattern-Match Coverage Checking

Ryan Scott
Indiana University

MuniHac 2018

What is pattern-match coverage checking?

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
```

```
λ> f Nothing
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
```

```
λ> f Nothing
0
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
```

```
λ> f Nothing
0

λ> g Nothing
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
```

```
λ> f Nothing
0

λ> g Nothing
*** Exception: MuniHac.hs:9:1-14: Non-
exhaustive patterns in function g
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
g Nothing  = 0
g Nothing  = 1
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
g Nothing  = 0
g Nothing  = 1
```

```
λ> g Nothing
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
g Nothing  = 0
g Nothing  = 1
```

```
λ> g Nothing
0
```

```haskell
f :: Maybe Int -> Int
f (Just x) = x
f Nothing  = 0
```

```haskell
g :: Maybe Int -> Int
g (Just x) = x
g Nothing  = 0
g Nothing  = 1
```

```
λ> g Nothing
0
```

Pattern-match coverage checking

Checks that a function's patterns satisfy two properties:

Exhaustivity
(it has no
incomplete patterns)

```
g1 :: Maybe Int -> Int
g1 (Just x) = x
```

Non-redundancy
(it has no
overlapping patterns)

```
g2 :: Maybe Int -> Int
g2 (Just x) = x
g2 Nothing  = 0
g2 Nothing  = 1
```

Enable `-Wall`!

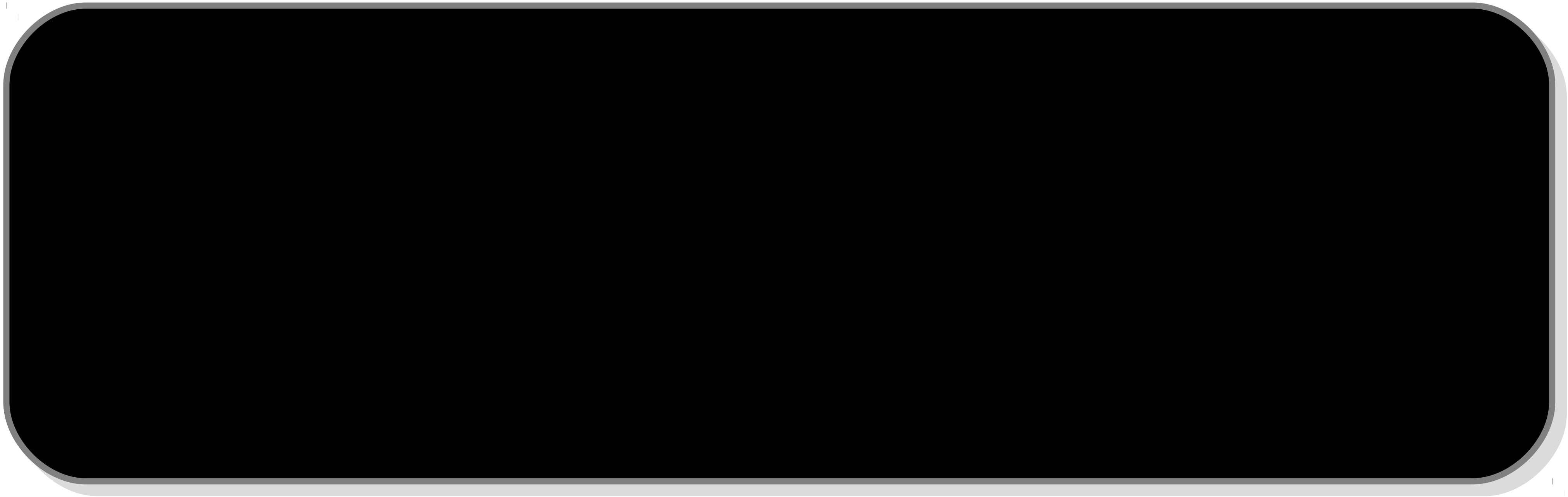
```
-Wincomplete-patterns
-Woverlapping-patterns
```

```haskell
g1 :: Maybe Int -> Int
g1 (Just x) = x
```

```haskell
g1 :: Maybe Int -> Int
g1 (Just x) = x
```

```
warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘g1’:
    Patterns not matched: Nothing
   |
   | g1 (Just x) = x
   | ^^^^^^^^^^^^^^^
```

```haskell
g2 :: Maybe Int -> Int
g2 (Just x) = x
g2 Nothing  = 0
g2 Nothing  = 1
```



```haskell
g2 :: Maybe Int -> Int
g2 (Just x) = x
g2 Nothing  = 0
g2 Nothing  = 1
```

```
warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In an equation for ‘g2’:
    g2 Nothing = ...
   |
   | g2 Nothing  = 1
   | ^^^^^^^^^^^^^^^
```

```haskell
g2 :: Maybe Int -> Int
g2 (Just x) = x
g2 Nothing  = 0
```

~~g2 Nothing = 1~~

```
warning: [-Woverlapping-patterns]
    Pattern match is redundant
    In an equation for ‘g2’:
    g2 Nothing = ...
   |
   | g2 Nothing  = 1
   | ^^^^^^^^^^^^^^^
```

Conclusions

- Enable `-Wall`
- Enable `-Wall`
- Enable `-Wall`
- Seriously, why aren't you using `-Wall` yet
- Enable `-Wall`

The End

Is coverage checking really that simple?

From a first glance, coverage-checked functions seem to obey the Golden Rule of Pattern Matching:

Is coverage checking really that simple?

From a first glance, coverage-checked functions seem to obey the Golden Rule of Pattern Matching:

An exhaustive and non-redundant function will match on every possible combination of constructors exactly **once** in its definition.

```
foo :: Maybe a -> ...
```

```haskell
foo :: Maybe a -> ...
foo (Just _) = ...
foo Nothing  = ...
```

```haskell
foo :: Maybe a -> ...
foo (Just _) = ...
foo Nothing  = ...
```

```haskell
bar :: Maybe a -> Maybe b -> ...
```

```haskell
foo :: Maybe a -> ...
foo (Just _) = ...
foo Nothing  = ...
```

```haskell
bar :: Maybe a -> Maybe b -> ...
bar (Just _) (Just _) = ...
bar (Just _) Nothing  = ...
bar Nothing  (Just _) = ...
bar Nothing  Nothing  = ...
```

The awkward bits

Haskell has a number of features that complicate coverage checking:

- **GADTs**
- **Guards**
- **Laziness**
- **Strictness annotations** (new?)

GADTs

(**G**eneralized **A**bstract **D**ata **T**ype**s**)

```
data Exp a where
  EInt    :: Int  -> Exp Int
  EBool   :: Bool -> Exp Bool
  EIsZero :: Exp Int -> Exp Bool
  EAdd    :: Exp Int -> Exp Int -> Exp Int
  EIf     :: Exp Bool -> Exp a -> Exp a -> Exp a
```

```haskell
data Exp a where
  EInt    :: Int  -> Exp Int
  EBool   :: Bool -> Exp Bool
  EIsZero :: Exp Int -> Exp Bool
  EAdd    :: Exp Int -> Exp Int -> Exp Int
  EIf     :: Exp Bool -> Exp a -> Exp a -> Exp a
```

```haskell
eval :: Exp a -> a
```

```haskell
data Exp a where
  EInt    :: Int  -> Exp Int
  EBool   :: Bool -> Exp Bool
  EIsZero :: Exp Int -> Exp Bool
  EAdd    :: Exp Int -> Exp Int -> Exp Int
  EIf     :: Exp Bool -> Exp a -> Exp a -> Exp a
```

```
eval :: Exp a -> a
eval (EInt i)      = i
```

```
data Exp a where
  EInt    :: Int  -> Exp Int
  EBool   :: Bool -> Exp Bool
  EIsZero :: Exp Int -> Exp Bool
  EAdd    :: Exp Int -> Exp Int -> Exp Int
  EIf     :: Exp Bool -> Exp a -> Exp a -> Exp a
```

```haskell
eval :: Exp a -> a
eval (EInt i)       = i
eval (EBool b)      = b
```

```haskell
data Exp a where
  EInt    :: Int  -> Exp Int
  EBool   :: Bool -> Exp Bool
  EIsZero :: Exp Int -> Exp Bool
  EAdd    :: Exp Int -> Exp Int -> Exp Int
  EIf     :: Exp Bool -> Exp a -> Exp a -> Exp a
```

```haskell
eval :: Exp a -> a
eval (EInt i)       = i
eval (EBool b)      = b
eval (EIsZero e)    = eval e == 0
eval (EAdd e1 e2)   = eval e1 + eval e2
eval (EIf b t f)    = if eval b
                        then eval t
                        else eval f
```

```haskell
data Exp a where
  EInt    :: Int  -> Exp Int
  EBool   :: Bool -> Exp Bool
  EIsZero :: Exp Int -> Exp Bool
  EAdd    :: Exp Int -> Exp Int -> Exp Int
  EIf     :: Exp Bool -> Exp a -> Exp a -> Exp a
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
getInt :: T Int -> Int
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
getInt :: T Int -> Int
getInt (TInt i) = i
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
getInt :: T Int -> Int
getInt (TInt i) = i
getInt (TBool _) = ???
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
getInt :: T Int -> Int
getInt (TInt i) = i
getInt (TBool _) = ???
```

```
Couldn't match type ‘Int’ with ‘Bool’
Inaccessible code in
  a pattern with constructor
    TBool :: Bool -> T Bool,
  in an equation for ‘getInt’
```

```
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```
getInt :: T Int -> Int
getInt (TInt i) = i
getInt (TBool _) = ???
```

```
Couldn't match type ‘Int’ with ‘Bool’
Inaccessible code in
  a pattern with constructor
    TBool :: Bool -> T Bool,
  in an equation for ‘getInt’
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
getInt :: T Int -> Int
getInt (TInt i) = i
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
addAnd (TBool _)  (TInt  _)  = ???
addAnd (TInt _)   (TBool _)  = ???
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
addAnd (TBool _)  (TInt  _)  = ???
addAnd (TInt _)   (TBool _)  = ???
```

```
Couldn't match type ‘Bool’ with ‘Int’
  ...
Couldn't match type ‘Int’ with ‘Bool’
  ...
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
```

~~addAnd (TBool _) (TInt _) = ???~~

~~addAnd (TInt _)   (TBool _) = ???~~

```
Couldn't match type ‘Bool’ with ‘Int’
  ...
Couldn't match type ‘Int’ with ‘Bool’
  ...
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
```

```
GHCi, version 7.10.3:
    Pattern match(es) are non-exhaustive
    In an equation for ‘addAnd’:
        Patterns not matched:
            (TInt _) (TBool _)
            (TBool _) (TInt _)
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
```

```haskell
addAnd :: T a -> T a -> a
addAnd (TInt  i1) (TInt  i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
addAnd _ _ = error "GHC is dumb :("
```

```
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TInt 0) (UChar ‘a’)
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TInt 0) (UChar ‘a’)
  Couldn't match type ‘Char’ with ‘Int’
  Expected type: U Int
    Actual type: U Char
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TBool True) (UInt 0)
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TBool True) (UInt 0)
  Couldn't match type ‘Int’ with ‘Bool’
  Expected type: U Bool
    Actual type: U Int
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TBool True) (undefined :: U Bool)
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TBool True) (undefined :: U Bool)
*** Exception: MuniHac.hs:47:1-32: Non-
exhaustive patterns in function tu
```

Laziness

Laziness

$$\bot :: a$$

Laziness

$$\bot :: a$$

```
let x = x
 in x
```

Laziness

$$\bot :: a$$

```
let x = x
 in x
```

```
undefined

error “boom”
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
```

```
λ> tu (TBool True) (⊥ :: U Bool)
*** Exception: MuniHac.hs:47:1-32: Non-
exhaustive patterns in function tu
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
tu (TBool _) _         = 42
```

```
λ> tu (TBool True) (⊥ :: U Bool)
42
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool

data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
tu (TBool _) x         = case x of {}
```

```
λ> tu (TBool True) (⊥ :: U Bool)
⊥
```

```haskell
data T a where
  TInt  :: Int  -> T Int
  TBool :: Bool -> T Bool
data U a where
  UInt  :: Int  -> U Int
  UChar :: Char -> U Char
```

```haskell
{-# LANGUAGE EmptyCase #-}
```

```haskell
tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
tu (TBool _) x         = case x of {}
```

```
λ> tu (TBool True) (⊥ :: U Bool)
⊥
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
GHCi, version 7.10.3:
    Pattern match(es) are overlapped
    In an equation for ‘weird’:
      weird True False = ...
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
λ> weird ⊥ True
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
λ> weird ⊥ True
3
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
λ> weird ⊥ True
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
λ> weird ⊥ True
⊥
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
GHCi, version 7.10.3:
    Pattern match(es) are overlapped

    In an equation for ‘weird’:
      weird True False = ...
```

```haskell
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

```
GHCi, version 8.6.1:
    Pattern match has inaccessible
      right hand side
    In an equation for ‘weird’:
      weird True False = ...
```

Pattern-match coverage checking

Checks that a function's patterns satisfy two properties:

Exhaustivity
(it has no
incomplete patterns)

Non-redundancy
(it has no
overlapping patterns)

Pattern-match coverage checking

Checks that a function's patterns satisfy ~~two~~ ***three*** properties:

Exhaustivity
(it has no
incomplete patterns)

Non-redundancy
(it has no
overlapping patterns)

Reachability
(no clause has an
inaccessible
right-hand side)

```
weird :: Bool -> Bool -> Int
weird _    False = 1
weird True False = 2
weird _    _     = 3
```

Guards

Guards

```haskell
abs :: Int -> Int
abs x | x < 0     = -x
      | otherwise = x
```

Guards

```
abs :: Int -> Int
abs x | x < 0      = -x
      | x >= 0     = x
```

Guards

```haskell
abs :: Int -> Int
abs x | x < 0      = -x
      | x >= 0     = x
```

```
warning: [-Woverlapping-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘abs’:
    Patterns not matched: _
   |
   | abs x | x < 0      = -x
   | ^^^^^^^^^^^^^^^^^^^^^^...
```

GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalias
Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

Types

τ	$::= a \mid \tau_1 \rightarrow \tau_2 \mid T\,\overline{\tau} \mid \ldots$	Monotypes
$a, b, a', b', \ldots$		Type variables
T		Type constructors
Γ	$::= \epsilon \mid \Gamma, a \mid \Gamma, x : \tau$	Typing environment

Terms and clauses

$f, g, x, y, \ldots$		Term variables
e		Expression
c	$::= \vec{p} \rightarrow e$	Clause

Patterns

K		Data constructors
p, q	$::= x \mid K\,\vec{p} \mid G$	Pattern
G	$::= p \leftarrow e$	Guard

Value abstractions

S, C, U, D	$::= \overline{v}$	Value set abstraction
v	$::= \Gamma \vdash \vec{u} \triangleright \Delta$	Value vector abstraction
u, w	$::= x \mid K\,\vec{u}$	Value abstraction

Constraints

Δ	$::= \epsilon \mid \Delta \cup \Delta$	
	$\mid Q$	Type constraint
	$\mid x \approx e$	Term-equality constraint
	$\mid x \approx \bot$	Strictness constraint
Q	$::= \tau \sim \tau$	Type-equality constraint
	$\mid \ldots$	other constraint

Figure 2: Syntax

Types

τ	$::= a \mid \tau_1 \to \tau_2 \mid T\,\overline{\tau} \mid \ldots$	Monotypes
$a, b, a', b', \ldots$		Type variables
T		Type constructors
Γ	$::= \epsilon \mid \Gamma, a \mid \Gamma, x : \tau$	Typing environment

Terms and clauses

Constraints

Δ	$::= \epsilon \mid \Delta \cup \Delta$	
	$\mid Q$	Type constraint
	$\mid x \approx e$	Term-equality constraint
	$\mid x \approx \bot$	Strictness constraint
Q	$::= \tau \sim \tau$	Type-equality constraint
	$\mid \ldots$	other constraint

Constraints

Δ	$::= \epsilon \mid \Delta \cup \Delta$	
	$\mid Q$	Type constraint
	$\mid x \approx e$	Term-equality constraint
	$\mid x \approx \bot$	Strictness constraint
Q	$::= \tau \sim \tau$	Type-equality constraint
	$\mid \ldots$	other constraint

Figure 2: Syntax

The End?

GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalias
Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalias
Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalias
Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalias
Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

The awkward bits

Haskell has a number of features that complicate coverage checking:

- **GADTs**
- **Guards**
- **Laziness**
- **Strictness annotations** (new?)

The awkward bits

Haskell has a number of features that complicate coverage checking:

- **GADTs**
- **Guards**
- **Laziness**
- **Strictness annotations** (new?)

Strictness annotations

https://ghc.haskell.org/trac/ghc/ticket/15305

GHC Trac Home
GHC Home

Joining In
Report a bug
Newcomers info
Mailing Lists & IRC
The GHC Team

Documentation
GHC Status Info
Building Guide
Working conventions
Commentary
Debugging
Infrastructure

View Tickets
My Tickets
Tickets I Created
By Milestone
By OS
By Architecture
Patches for review

#15305 closed bug (fixed)

Opened 4 months ago
Closed 2 months ago
Last modified 2 months ago

Erroneous "non-exhaustive pattern match" using nested GADT with strictness annotation

Reported by:	jkoppel	Owned by:	
Priority:	normal	Milestone:	8.8.1
Component:	Compiler (Type checker)	Version:	8.4.3
Keywords:	PatternMatchWarnings	Cc:	alanz, sh.najd@gmail.com
Operating System:	Unknown/Multiple	Architecture:	Unknown/Multiple
Type of failure:	Incorrect error/warning at compile-time	Test Case:	pmcheck/should_compile/T15305
Blocked By:		Blocking:	
Related Tickets:		Differential Rev(s):	Phab:D5087
Wiki Page:			

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int

data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
```

```
GHCi, version 8.6.1:
    Pattern match(es) are non-exhaustive
    In an equation for ‘getBool’:
    Patterns not matched: (MustBe2 _)
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

[redacted]

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 (AnInt 42))
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 (AnInt 42))
  Couldn't match type ‘Int’ with ‘Bool’
  Expected type: MustBe Bool
    Actual type: MustBe Int
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 ⊥)
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 ⊥)
⊥
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 ⊥)
⊥
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 ⊥)
⊥
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
λ> getBool (MustBe2 ⊥)
⊥
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
GHCi, version 8.6.1:
    Pattern match(es) are non-exhaustive
    In an equation for ‘getBool’:
    Patterns not matched: (MustBe2 _)
```

A revised checking algorithm

When coverage-checking a clause

```
f (MkD d1 ... dn) = ...
```

A revised checking algorithm

When coverage-checking a clause

`f (MkD d1 ... dn) = ...`

- Collect all the strict fields of `MkD`.

A revised checking algorithm

When coverage-checking a clause

```
f (MkD d1 ... dn) = ...
```

- Collect all the strict fields of `MkD`.
- For each strict field's type, find the possible inhabitants of that type.

A revised checking algorithm

When coverage-checking a clause

`f (MkD d1 ... dn) = ...`

- Collect all the strict fields of `MkD`.
- For each strict field's type, find the possible inhabitants of that type.
- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

A revised checking algorithm

When coverage-checking a clause

`f (MkD d1 ... dn) = ...`

- Collect all the strict fields of MkD.
- For each strict field's type, find the possible inhabitants of that type.
- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

A revised checking algorithm

When coverage-checking a clause

`f (MkD d1 ... dn) = ...`

- Collect all the strict fields of MkD.
- For each strict field's type, find the possible inhabitants of that type.

$\perp :: a$

- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

A revised checking algorithm

When coverage-checking a clause
`f (MkD d1 ... dn) = ...`

- Collect all the strict fields of MkD.

- For each strict field's type, find the possible *terminating* inhabitants of that type.

- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

```haskell
data Abyss = MkAbyss !Abyss
```

- For each strict field's type, find the possible *terminating* inhabitants of that type.

- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

```
data Abyss = MkAbyss !Abyss

gazeIntoTheAbyss :: Abyss -> a
gazeIntoTheAbyss x = case x of {}
```

- For each strict field's type, find the possible *terminating* inhabitants of that type.

- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

```haskell
data Abyss = MkAbyss !Abyss

gazeIntoTheAbyss :: Abyss -> a
gazeIntoTheAbyss x = case x of {}
```

- For each strict field's type, find the possible *terminating* inhabitants of that type. If recursion is detected, bail out and conservatively assume there is an inhabitant.

- If any of these types has no possible inhabitants, that clause is unreachable (i.e., redundant).

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
GHCi, version 8.6.1:
    Pattern match(es) are non-exhaustive
    In an equation for ‘getBool’:
    Patterns not matched: (MustBe2 _)
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _)         = False -- ??
```

```
GHCi, version 8.7 (HEAD):
```

```haskell
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
```

```haskell
data MustBe a
  = MustBe1 !(ABool a)
  | MustBe2 !(AnInt a)
```

```haskell
getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
```

~~getBool (MustBe2 _)   = False -- ??~~

```
GHCi, version 8.7 (HEAD):
```

The End?!?!??

Keywords contains PatternMatchWarnings

Custom Query (27 matches)

▼ Filters

Keywords contains PatternMatchWarnings

Status closed infoneeded merge new patch upstream

And Or

▶ Columns

Group results by descending Show under each result: Description Max items per page 100 Update

Ticket	Summary	Status	Keywords	Owner	Type	Priority
#14899	Significant compilation time regression between 8.4 and HEAD due to coverage checking	new	PatternMatchWarnings, newcomer		bug	highest
#14253	Pattern match checker mistakenly concludes pattern match on pattern synonym is unreachable	new	PatternSynonyms, PatternMatchWarnings		bug	high
#10116	Closed type families: Warn if it doesn't handle all cases	new	TypeFamilies, PatternMatchWarnings		feature request	normal
#11195	New pattern-match check can be non-performant	new	PatternMatchWarnings		bug	normal
#11253	Duplicate warnings for pattern guards and relevant features (e.g. View Patterns)	new	pattern matching, exhaustiveness, pattern checker, PatternMatchWarnings	gkaracha	bug	normal
#11503	TypeError woes (incl. pattern match checker)	new	PatternMatchWarnings, CustomTypeErrors		bug	normal
#11822	Pattern match checker exceeded (2000000) iterations	new	PatternMatchWarnings	gkaracha	bug	normal
#12694	GHC HEAD no longer reports inaccessible code	new	PatternMatchWarnings		bug	normal
#12949	Pattern coverage checker ignores dictionary arguments	new	PatternMatchWarnings	gkaracha	bug	normal
#13021	Inaccessible RHS warning is confusing for users	new	PatternMatchWarnings		bug	normal
#13363	Wildcard patterns and COMPLETE sets can lead to misleading redundant pattern-match warnings	new	PatternSynonyms, PatternMatchWarnings		bug	normal
#13717	Pattern synonym exhaustiveness checks don't play well with EmptyCase	new	PatternSynonyms, PatternMatchWarnings		bug	normal
#13766	Confusing "redundant pattern match" in 8.0, no warning at all in 8.2	new	PatternMatchWarnings		bug	normal
#13964	Pattern-match warnings for datatypes with COMPLETE sets break abstraction	new	PatternSynonyms, PatternMatchWarnings		bug	normal
#13965	COMPLETE sets nerf redundant pattern-match warnings	new	PatternSynonyms, PatternMatchWarnings		bug	normal
#14059	COMPLETE sets don't work at all with data family instances	new	PatternSynonyms, PatternMatchWarnings		bug	normal
#14133	COMPLETE pragmas seem to be ignored when using view patterns	new	PatternSynonyms, PatternMatchWarnings		bug	normal
#14838	missing "incomplete-patterns" warning for TH-generated functions	new	PatternMatchWarnings		bug	normal
#14851	"Pattern match has inaccessible right hand side" with TypeRep	new	PatternMatchWarnings, PatternSynonyms		bug	normal
#14987	Memory usage exploding for complex pattern matching	new	PatternMatchWarnings		bug	normal

The End
(for real this time!)

Pattern-match coverage checking

- **Immensely useful, but surprisingly tricky to get right**
- **Haskell/GHC features make this analysis more interesting**
- **We need your help in fixing the remaining bugs!**

Thank you for listening!