The Curious case of Pattern-Match Coverage Checking

Ryan Scott
Indiana University

MuniHac 2018
What is pattern-match coverage checking?
\[
\begin{align*}
\textbf{f} & \::= \text{Maybe Int} \rightarrow \text{Int} \\
\text{f} \ (\text{Just } x) & = x \\
\text{f} \ \text{Nothing} & = 0
\end{align*}
\]
\[ f :: \text{Maybe \ Int} \rightarrow \text{Int} \]
\[ f (\text{Just } x) = x \]
\[ f \text{ Nothing} = 0 \]

\[ g :: \text{Maybe \ Int} \rightarrow \text{Int} \]
\[ g (\text{Just } x) = x \]
\[ f :: \text{Maybe Int} \to \text{Int} \]
\[ g :: \text{Maybe Int} \to \text{Int} \]
\[ f \ (\text{Just } x) = x \]
\[ f \ \text{Nothing} = 0 \]
\[ g \ (\text{Just } x) = x \]

\( \lambda > f \ \text{Nothing} \)
f :: Maybe Int -> Int
f (Just x) = x
f Nothing = 0

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

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

λ> f Nothing
0

λ> g Nothing

f :: Maybe Int -> Int  
g :: Maybe Int -> Int

f (Just x) = x  
g (Just x) = x

λ> f Nothing
0

λ> g Nothing
*** Exception: MuniHac.hs:9:1-14: Non-exhaustive patterns in function g
\[ f :: \text{Maybe Int} \rightarrow \text{Int} \]
\[ f (\text{Just } x) = x \]
\[ f \text{ Nothing} = 0 \]

\[ g :: \text{Maybe Int} \rightarrow \text{Int} \]
\[ g (\text{Just } x) = x \]
\[
\begin{align*}
f & : \text{Maybe Int} \to \text{Int} \\
f(\text{Just } x) & = x \\
f\text{Nothing} & = 0 \\
g & : \text{Maybe Int} \to \text{Int} \\
g(\text{Just } x) & = x \\
g\text{Nothing} & = 0 \\
g\text{Nothing} & = 1
\end{align*}
\]
\[
\begin{align*}
f & : \text{Maybe Int } \to \text{ Int} \\
f(\text{Just } x) &= x \\
f\text{Nothing} &= 0 \\
g & : \text{Maybe Int } \to \text{ Int} \\
g(\text{Just } x) &= x \\
g\text{Nothing} &= 0 \\
g\text{Nothing} &= 1 \\
\end{align*}
\]

\[
\lambda > g \text{ Nothing}
\]
\[ f :: \text{Maybe} \text{ Int} \rightarrow \text{Int} \]
\[ f(\text{Just } x) = x \]
\[ f \text{ Nothing} = 0 \]

\[ g :: \text{Maybe} \text{ Int} \rightarrow \text{Int} \]
\[ g(\text{Just } x) = x \]
\[ g \text{ Nothing} = 0 \]
\[ g \text{ Nothing} = 1 \]

\[ \lambda > g \text{ Nothing} \]
\[ 0 \]
\[ f :: \text{Maybe Int} \rightarrow \text{Int} \]
\[ f (\text{Just } x) = x \]
\[ f \text{ Nothing} = 0 \]

\[ g :: \text{Maybe Int} \rightarrow \text{Int} \]
\[ g (\text{Just } x) = x \]
\[ g \text{ Nothing} = 0 \]
\[ g \text{ Nothing} = 1 \]

\[ \lambda > g \text{ Nothing} \]
\[ 0 \]
Pattern-match coverage checking

Checks that a function’s patterns satisfy two properties:

**Exhaustivity**
(it has no *incomplete* patterns)

\[
g_1 :: \text{Maybe\ Int} \rightarrow \text{Int} \\
g_1 (\text{Just } x) = x
\]

**Non-redundancy**
(it has no *overlapping* patterns)

\[
g_2 :: \text{Maybe\ Int} \rightarrow \text{Int} \\
g_2 (\text{Just } x) = x \\
g_2 \text{ Nothing} = 0 \\
g_2 \text{ Nothing} = 1
\]
Enable -Wall!

-Wincomplete-patterns
-Woverlapping-patterns
\[ g_1 :: \text{Maybe Int} \rightarrow \text{Int} \]
\[ g_1 (\text{Just } x) = x \]
g1 :: Maybe Int -> Int

\[
g1 \ (\text{Just } x) = x
\]
g2 :: Maybe Int -> Int

\[ g2 \text{ (Just } x) = x \]

\[ g2 \text{ Nothing } = 0 \]

\[ g2 \text{ Nothing } = 1 \]
warning: [-Woverlapping-patterns]
Pattern match is redundant
In an equation for ‘g2’:
g2 Nothing = ...

| g2 Nothing = 1
| ^^^^^^^^^^^^^^^
g2 :: Maybe Int -> Int

g2 (Just x) = x

g2 Nothing = 0

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

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

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

(Generalized Abstract Data Types)

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
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
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
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

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

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

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

getInt :: T Int -> Int
getInt (TInt i) = i
data T a where
    TInt :: Int  -> T Int
    TBool :: Bool -> T Bool

getInt :: T Int  -> Int
getInt (TInt i) = i
getInt (TBool _) = ???
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`
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’
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

getInt :: T Int -> Int
getInt (TInt i) = i
data T a where
    TInt :: Int -> T Int
    TBool :: Bool -> T Bool

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

addAnd :: T a -> T a -> a
addAnd (TInt i1) (TInt i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
data T a where
    TInt :: Int -> T Int
    TBool :: Bool -> T Bool

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 _) = ???
data T a where
   TInt :: Int -> T Int
   TBool :: Bool -> T Bool

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

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’
...
data T a where
    TInt :: Int -> T Int
    TBool :: Bool -> T Bool

addAnd :: T a -> T a -> a
addAnd (TInt i1) (TInt i2) = i1 + i2
addAnd (TBool b1) (TBool b2) = b1 && b2
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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 _)
data T a where
    TInt ::: Int -> T Int
    TBool ::: Bool -> T Bool

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

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

tu :: T a -> U a -> Int
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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

tu :: T a -> U a -> Int
tu (TInt i1) (UInt i2) = i1 + i2
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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

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

λ> tu (TInt 0) (UChar 'a')
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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

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

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

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

tu (TInt i1) (UInt i2) = i1 + i2

λ> tu (TBool True) (UInt 0)
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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

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

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

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

λ> tu (TInt i1) (UInt i2) = i1 + i2

λ> tu (TBool True) (undefined :: U Bool)
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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

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

⊥ :: a
Laziness

\( \bot :: a \)

let x = x
    in x
Laziness

\( \bot :: a \)

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

undefined

error “boom”
data T a where
    TInt :: Int -> T Int
    TBool :: Bool -> T Bool

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

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

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

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

λ> tu (TBool True) (⊥ :: U Bool)
42
data T a where
    TInt ::: Int -> T Int
    TBool ::: Bool -> T Bool

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

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)
⊥
data T a where
  TInt :: Int -> T Int
  TBool :: Bool -> T Bool

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

{--# LANGUAGE EmptyCase --}
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)
⊥
weird :: Bool -> Bool -> Int
weird _ False = 1
weird True False = 2
weird _ _ _ = 3
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 = ...
weird :: Bool -> Bool -> Int
weird _ False = 1
weird True False = 2
weird _ _ _ = 3
weird :: Bool -> Bool -> Int
weird _ False = 1
weird True False = 2
weird _ _ _ = 3

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

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

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

λ> weird ⊥ True
⊥
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 = ...
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

\[
\text{abs} :: \text{Int} \rightarrow \text{Int} \\
\text{abs } x \mid x < 0 \quad = -x \\
\mid \text{otherwise } = x
\]
Guards

\[
\text{abs} :: \text{Int} \rightarrow \text{Int} \\
\text{abs} \ x \mid x < 0 \quad = -x \\
\mid x \geq 0 \quad = x
\]
Guards

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

\[
\tau ::= a \mid \tau_1 \to \tau_2 \mid T \bar{\tau} \mid \ldots
\]

- **Monotypes**
- **Type variables**
- **Type constructors**
- **Typing environment**

### Terms and clauses

\[
f, g, x, y, \ldots
\]

- **Term variables**
- **Expression**
- **Clause**

\[
e
\]

\[
c ::= \bar{p} \to e
\]

### Patterns

\[
K
\]

- **Data constructors**

\[
p, q ::= x \mid K \bar{p} \mid G
\]

- **Pattern**
- **Guard**

\[
G ::= p \leftarrow e
\]

### Value abstractions

\[
S, C, U, D ::= \bar{v}
\]

- **Value set abstraction**
- **Value vector abstraction**
- **Value abstraction**

\[
v ::= \Gamma \vdash \bar{v} \triangleright \Delta
\]

\[
u, w ::= x \mid K \bar{u}
\]

### Constraints

\[
\Delta ::= \epsilon \mid \Delta \cup \Delta
\]

\[
\lfloor Q \quad x \approx e \quad x \approx \bot \quad \Delta \rfloor
\]

\[
Q ::= \tau \sim \tau
\]

- **Type constraint**
- **Term-equality constraint**
- **Strictness constraint**
- **Type-equality constraint**
- **other constraint**

---

**Figure 2: Syntax**
Types
\[ \tau ::= a \mid \tau_1 \to \tau_2 \mid T\overline{\tau} \mid \ldots \]  
Monotypes
Type variables
Type constructors
Typing environment

Terms and clauses

Constraints
\[ \Delta ::= \epsilon \mid \Delta \cup \Delta \]
\[ Q \]
\[ x \approx e \]
\[ x \approx \bot \]

\[ Q ::= \tau \sim \tau \]
\[ \ldots \]

Type constraint
Term-equality constraint
Strictness constraint
Type-equality constraint
other constraint

Figure 2: Syntax
The End?
GADTs Meet Their Match:
Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

Georgios Karachalios
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
#15305 closed bug (fixed)

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

Reported by: jkoppel
Priority: normal
Component: Compiler (Type checker)
Keywords: PatternMatchWarnings
Operating System: Unknown/Multiple
Type of failure: Incorrect error/warning at compile-time

Owned by:
Milestone: 8.8.1
Version: 8.4.3
Cc: alanz, sh.najd@gmail.com
Architecture: Unknown/Multiple
Test Case: pmcheck/should_compile/T15305

Blocked By: 
Differential Rev(s): Phab:D5087
data ABool a where
  ABool :: Bool -> ABool Bool
data AnInt a where
  AnInt :: Int -> AnInt Int
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)
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

getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
data ABool a where
  ABool :: Bool -> ABool Bool

data AnInt a where
  AnInt :: Int -> AnInt Int

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 _)

data MustBe a
  MustBe1 !(ABool a)
  MustBe2 !(AnInt a)
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)

getBool :: MustBe Bool -> Bool
getBool (MustBe1 (ABool b)) = b
getBool (MustBe2 _) = False -- ??
data ABool a where
  ABool :: Bool -> ABool Bool

data AnInt a where
  AnInt :: Int -> AnInt Int

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

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

λ> getBool (MustBe2 (AnInt 42))
data ABool a where
  ABool :: Bool -> ABool Bool

data AnInt a where
  AnInt :: Int -> AnInt Int

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

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
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)

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

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

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

λ> getBool (MustBe2 ⊥)
⊥
```
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)

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

λ> getBool (MustBe2 ⊥)
⊥
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)

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

λ> getBool (MustBe2 ⊥)
⊥
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

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

λ> getBool (MustBe2 ⊥)
⊥
data ABool a where
  ABool :: Bool -> ABool Bool

data AnInt a where
  AnInt :: Int -> AnInt Int

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

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
\[ f (MkD \ d1 \ldots\ dn) = \ldots \]
A revised checking algorithm

When coverage-checking a clause
\[ f (MkD \ d1 \ldots \ dn) = \ldots \]

- Collect all the strict fields of MkD.
A revised checking algorithm

When coverage-checking a clause
\[ f \left( \text{MkD } d_1 \ldots d_n \right) = \ldots \]

- 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
\[ f (MkD \ d1 \ldots \ dn) = \ldots \]

• 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
\[ f (MkD \ d_1 \ldots \ d_n) = \ldots \]

- 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
\( f(\text{MkD } d_1 \ldots d_n) = \ldots \)

- Collect all the strict fields of \( \text{MkD} \).

- 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

\[ f (MkD \ d1 \ldots \ dn) = \ldots \]

• Collect all the strict fields of MkD.

• For each strict field’s type, find the possible \textit{terminating} 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 $f(MkD \ d_1 \ ... \ d_n) = ...$
  - 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).
A revised checking algorithm

When coverage-checking a clause
\[ f(MkD \, d_1 \ldots \, d_n) = \ldots \]

- 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

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).
A revised checking algorithm

When coverage-checking a clause \( f(MkD \ d_1 \ldots \ d_n) = \ldots \)

- Collect all the strict fields of \( MkD \).
- 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 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).
data ABool a where
  ABool :: Bool -> ABool Bool

data AnInt a where
  AnInt :: Int -> AnInt Int

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 _)
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)

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

GHCi, version 8.7 (HEAD):
data ABool a where
  ABool :: Bool -> ABool Bool

data AnInt a where
  AnInt :: Int -> AnInt Int

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

GHCi, version 8.7 (HEAD):
The End?!?!??
<table>
<thead>
<tr>
<th>Ticket</th>
<th>Summary</th>
<th>Status</th>
<th>Keywords</th>
<th>Owner</th>
<th>Type</th>
<th>Priority</th>
</tr>
</thead>
<tbody>
<tr>
<td>#14899</td>
<td>Significant compilation time regression between 8.4 and HEAD due to coverage checking</td>
<td>new</td>
<td>PatternMatchWarnings, newcomer</td>
<td></td>
<td>bug</td>
<td>highest</td>
</tr>
<tr>
<td>#14253</td>
<td>Pattern match checker mistakenly concludes pattern match on pattern synonym is unreachable</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td></td>
</tr>
<tr>
<td>#10116</td>
<td>Closed type families: Warn if it doesn’t handle all cases</td>
<td>new</td>
<td>TypeFamilies, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>high</td>
</tr>
<tr>
<td>#11195</td>
<td>New pattern-match check can be non-performant</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td></td>
</tr>
<tr>
<td>#11253</td>
<td>Duplicate warnings for pattern guards and relevant features (e.g. View Patterns)</td>
<td>new</td>
<td>pattern matching, exhaustiveness, pattern checker, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td></td>
</tr>
<tr>
<td>#11503</td>
<td><code>TypeError</code> woes (incl. pattern match checker)</td>
<td>new</td>
<td>PatternMatchWarnings, CustomTypeError</td>
<td>gkaracha</td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#11822</td>
<td>Pattern match checker exceeded (2000000) iterations</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#12531</td>
<td>GHC HEAD no longer reports inaccessible code</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td>gkaracha</td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#12549</td>
<td>Pattern coverage checker ignores dictionary arguments</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td>gkaracha</td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#13021</td>
<td>Inaccessible RHS warning is confusing for users</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td>gkaracha</td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#13363</td>
<td>Wildcard patterns and COMPLETE sets can lead to misleading redundant pattern-match warnings</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#13717</td>
<td>Pattern synonym exhaustiveness checks don’t play well with EmptyCase</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#13766</td>
<td>Confusing “redundant pattern match” in 8.0, no warning at all in 8.2</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#13964</td>
<td>Pattern-match warnings for datatypes with COMPLETE sets break abstraction</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#13965</td>
<td>COMPLETE sets nrf redundant pattern-match warnings</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#14095</td>
<td>COMPLETE sets don’t work at all with data family instances</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#14133</td>
<td>COMPLETE pragmas seem to be ignored when using view patterns</td>
<td>new</td>
<td>PatternSynonyms, PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#14838</td>
<td>missing “incomplete-patterns” warning for TH-generated functions</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#14951</td>
<td>“Pattern match has inaccessible right hand side” with TypeRep</td>
<td>new</td>
<td>PatternMatchWarnings, PatternSynonyms</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
<tr>
<td>#14987</td>
<td>Memory usage exploding for complex pattern matching</td>
<td>new</td>
<td>PatternMatchWarnings</td>
<td></td>
<td>bug</td>
<td>normal</td>
</tr>
</tbody>
</table>
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!