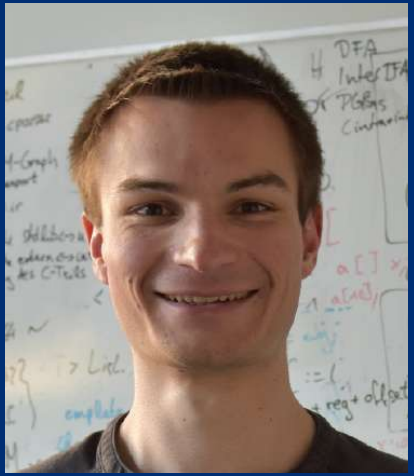
PATTERN MATCH WARNINGS
How hard can it be?



Simon Peyton Jones

Microsoft Research

With lots of help from

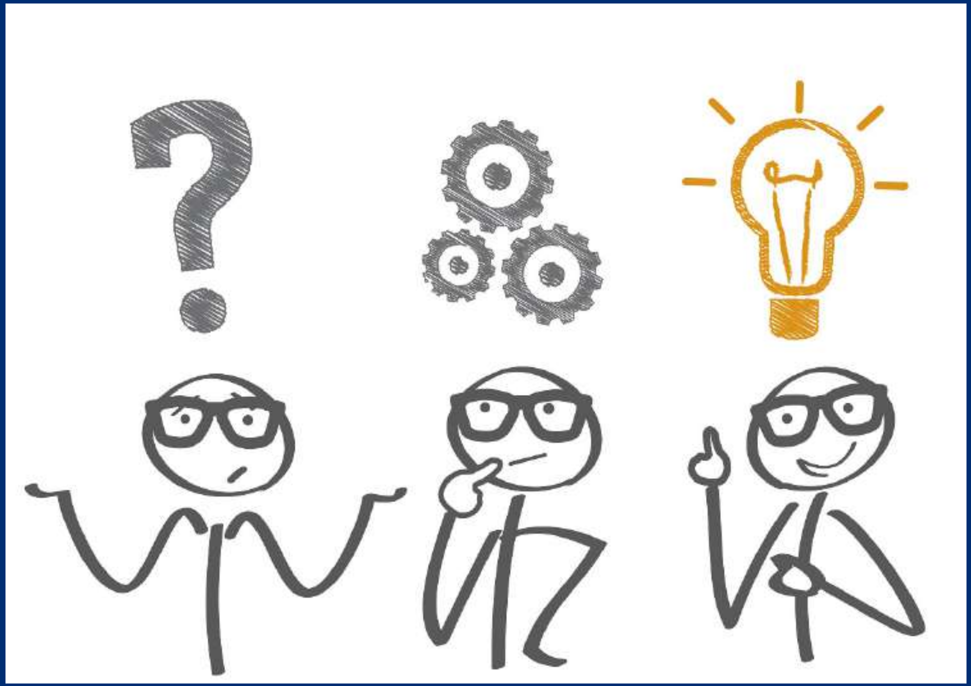
Ryan Scott (Indiana) and Sebastian Graf (Karlsruhe)

October 2019

Programming language research

Excellent research plan:

- Looks hard
- Think think think
- Is easy



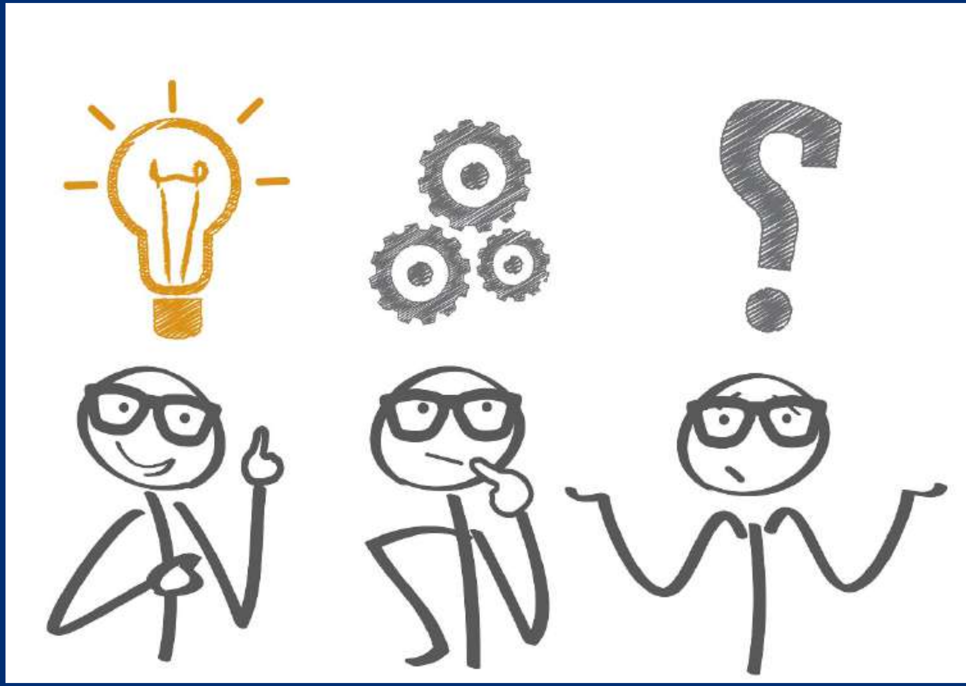
Programming language research

Excellent research plan:

- Looks hard
- Think think think
- Is easy

Less excellent plan

- Looks easy
- Think think think
- Is hard



Pattern match warnings

```
data Maybe a = Nothing
             | Just a

isJust :: Maybe a -> Bool
isJust (Just _) = True
isJust Nothing  = False
```

OK!

Pattern match warnings

```
isJust :: Maybe a -> Bool
isJust Nothing = False
```

Not OK!

```
ghci> isJust (Just True)
*** Exception: <interactive>:16:5-16:
   Non-exhaustive patterns in function isJust
```

Runtime error (bad)

Pattern match warnings

```
isJust :: Maybe a -> Bool
isJust Nothing = False
```

Compile time error (good)

```
ghci> :load Foo.hs
Foo.hs:16:5: warning: [-Wincomplete-patterns]
    Pattern match(es) are non-exhaustive
    In an equation for ‘isJust’: Patterns not matched: Just _
```

Pattern match warnings

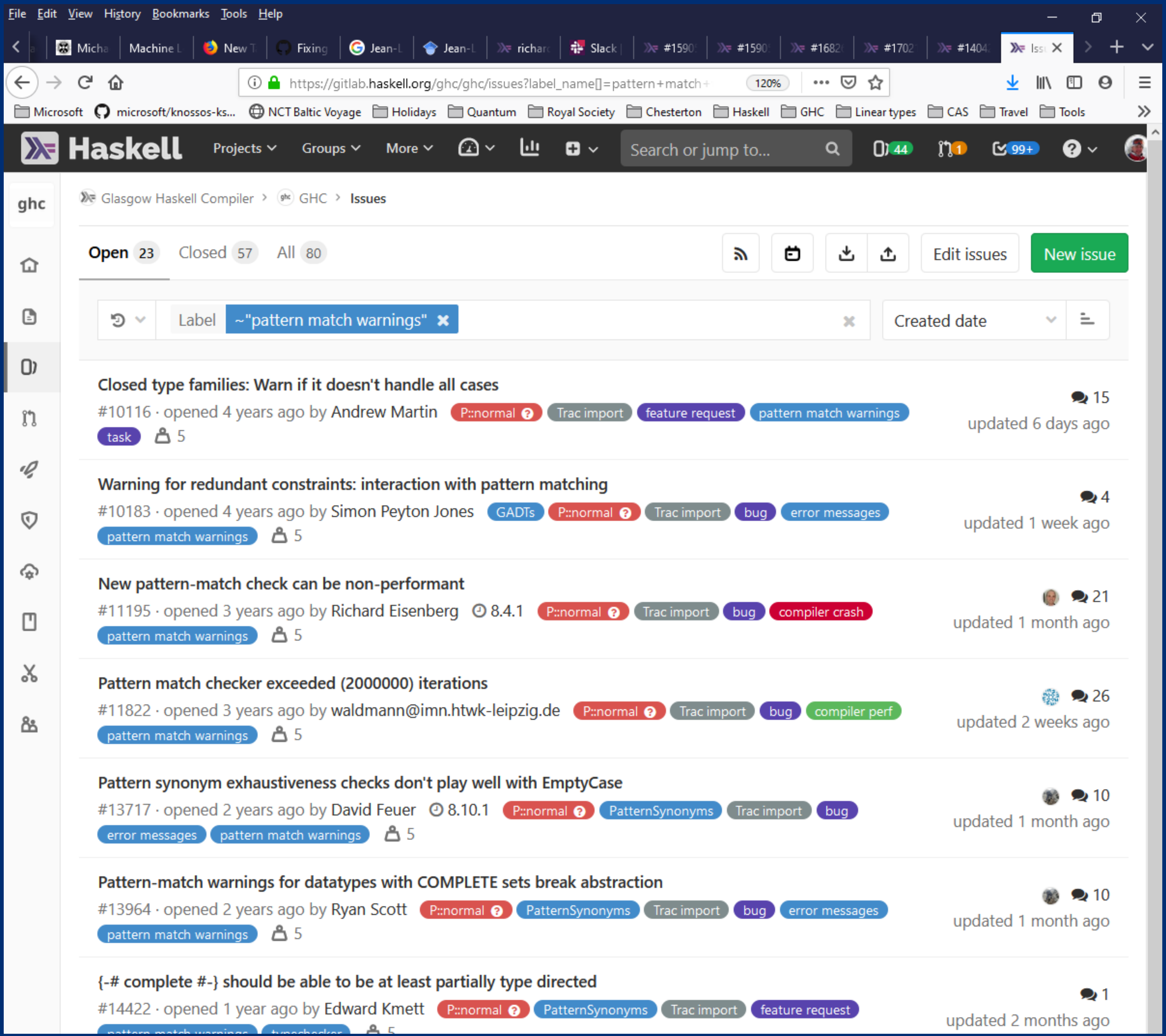
- Task: produce good compile time warnings for
  - Missing equations

```
isJust :: Maybe a -> Bool
isJust Nothing = False
```

  - Redundant equations

```
isJust :: Maybe a -> Bool
isJust Nothing   = False
isJust (Just _)  = True
isJust Nothing   = False
```

- First reaction: **easy peasy**



Easy peasy?

Around
80 tickets

Of which
24 are open

Interactions between arguments

Interactions: not so easy

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

- Which cases (if any) are not matched?

Interactions: not so easy

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

- Which cases (if any) are not matched?

```
berry True  True  True  = …
berry False False False = …
```

Laziness

Laziness: maybe not "easy" at all

```
f :: Bool -> Bool -> Int
f _    False = 1
f True False = 2    -- Is this equation redundant?
f _    _     = 3
```

Laziness: maybe not "easy" at all

```
f :: Bool -> Bool -> Int
f _    False = 1
f True False = 2    -- Is this equation redundant?
f _    _     = 3
```

```
ghci> f (error "urk") True
```

- With equation 2: get "exception: Urk"
- Without equation 2: get 3

So equation 2 is not redundant (cannot be omitted)

Laziness: maybe not "easy" at all

```
f :: Bool -> Bool -> Int
f _    False = 1
f True False = 2      -- Is this equation redundant?
f _    _     = 3
```

```
ghci> f (error "urk") True
```

So equation 2 is not redundant (cannot be omitted)

- With equation 2: get "exception: Urk"
- Without equation 2: get 3
- But can we ever return 2? No!

And yet its RHS is inaccessible

Laziness: maybe not "easy" at all

```
f :: Bool -> Bool -> Int
f _    False = 1
f True False = 2     -- Is this equation redundant?
f _    _     = 3
```

```
<interactive>:1:22: warning: [-Woverlapping-patterns]
    Pattern match has inaccessible right hand side
    In an equation for 'f': f True False = ...
```

- But can we ever return 2? No!

And yet its RHS is inaccessible

Bang patterns
and strict data constructors

Inhabitation

```
data Void            -- No data constructors
```

- The only inhabitant of Void is bottom

```
h :: Int -> Void
h x = h x
```

```
f :: Void -> Bool
f _ = True

g1 = f (error "urk")       -- This call is well typed
g2 = f (h 3)               -- This is well typed too
```

Inhabitation and strict constructors

```
data Void                -- No data constructors
data SMaybe a = SNothing | SJust !a   -- Strict Maybe
```

```
f :: SMaybe Void -> Int
f SNothing   = 1
f (SJust _) = 2   -- Is this redundant?
```

Inhabitation and strict constructors

```
data Void                -- No data constructors
data SMaybe a = SNothing | SJust !a   -- Strict Maybe
```

```
f :: SMaybe Void -> Int
f SNothing   = 1
f (SJust _) = 2      -- Redundant!
```

- The only inhabitants of (SMaybe Void) are
  1. SNothing
  2. bottom
- The first equation matches (1) and diverges on (2)
- So the second equation is redundant

Inhabitation and bang patterns

```
data Void            -- No data constructors
data Maybe a = Nothing | Just a
```

```
f :: Maybe Void -> Int
f Nothing    = 1
f (Just !_) = 2   -- Is this redundant?
```

- The only inhabitants of (Maybe Void) are
  1. Nothing
  2. Just bottom
- The second equation diverges on (2)
- So the second equation is is not redundant, but has inaccessible RHS

Guards and view patterns

Guards

```
sign :: Int -> Ordering
sign x | x < 0       = LT
       | x == 0      = EQ   -- Is this redundant?
       | otherwise = GT   -- Is this redundant?
       -- Are there any missing equations?
```

- Clearly undecidable in general
- But we want to do a good job in special cases
- E.g. otherwise/True always succeeds

Pattern guards

```
last :: [a] -> Maybe a
last xs | (y:_) <- reverse xs = Just y
        | otherwise           = Nothing
```

- Very like

```
last :: [a] -> Maybe a
last xs = case reverse xs of
            (y:_) -> Just y
            _     -> Nothing
```

Pattern guards

```
last :: [a] -> Maybe a
last xs | (y:_) <- reverse xs = Just y
        | []    <- reverse xs = Nothing
```

- Here we might reasonably hope that GHC will see that these equations are exhaustive

Mixing pattern matching and pattern guards

```
get :: Maybe Int -> Int
get Nothing = 0
get x | Just y <- x = y
```

Ordinary pattern match

Pattern guard

- Again, exhaustive...

View patterns (expr -> pat)

```
last :: [a] -> Maybe a
last (reverse -> y:_) = Just y
last (reverse -> [])  = Nothing
```

- Again, we might reasonably hope that GHC will see that these equations are exhaustive

Long distance information

Long distance information

```
data Grade = A | B | C

f :: Grade -> blah
f A = …
f g = … (case g of
            B -> True
            C -> False) …
```

This case is exhaustive

Are we having fun yet?

Multiple arguments

Laziness

Inhabitation, strict data constructors

Bang patterns

Guards and view patterns

Long distance interactions

GADTs: double the fun

GADTs

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

```haskell
getInt :: T Int -> Int
getInt (TInt i) = i
-- Are any equations missing?
```

- What about: getInt (TBool b)?

GADTs

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

```
getInt :: T Int -> Int
getInt (TInt i) = i
-- Are any equations missing?  No!!
```

- No: this single equation is exhaustive

GADTs and long distance information

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

This case is exhaustive

```
foo :: T a -> T a -> T a
foo (TInt i1)  y = …(case y of TInt i2 -> …) …
foo (TBool b1) y = …(case y of TBool b2 -> …) …
```

GADTs and multiple arguments

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

```
data U a where
  UChar :: Char -> U Char
  UBool :: Bool -> U Bool
```

```
foo :: T a -> U a -> Bool
foo (TBool b1) (UBool b2) = b1 || b2
-- Are any equations missing?
```

GADTs and multiple arguments

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

```
data U a where
  UChar :: Char -> U Char
  UBool :: Bool -> U Bool
```

```
foo :: T a -> U a -> Bool
foo (TBool b1) (UBool b2) = b1 || b2
-- Are any equations missing?  Yes!
```

- What about: foo (TInt i1) (error "urk")?
- Yikes! This is well typed; and fails to match the first eqn

GADTs and multiple arguments

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

```
data U a where
  UChar :: Char -> U Char
  UBool :: Bool -> U Bool
```

```
foo :: T a -> U a -> Bool
foo (TBool b1) (UBool b2) = b1 || b2
foo (TInt _)   _          = True
```

- or…

GADTs and multiple arguments

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

```
data U a where
  UChar :: Char -> U Char
  UBool :: Bool -> U Bool
```

```
foo :: T a -> U a -> Bool
foo (TBool b1) (UBool b2) = b1 || b2
foo (TInt _)   y          = case y of { }
```

- {-# LANGUAGE EmptyCase #-}
- The empty case is strict, so will force y.
- But we should check that the case y of {} is exhaustive.. long distance.

Pattern synonyms

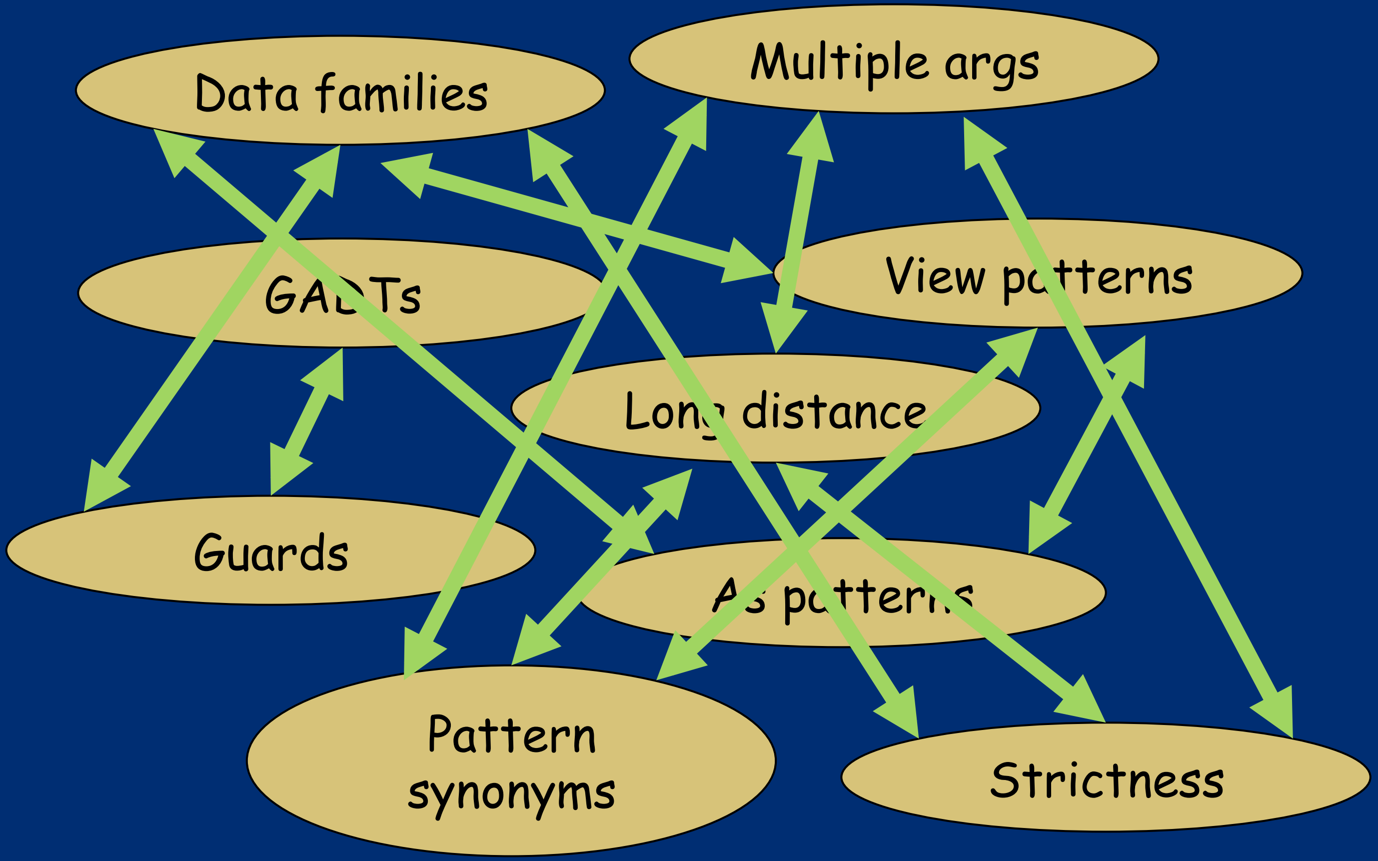
Pattern synonyms

```
pattern Snoc xs x <- (reverse -> (x:xs))
{-# COMPLETE Snoc, [] #-}

last :: [a] -> Maybe a
last []           = Nothing
last (Snoc xs x)  = Just x
```

Asserts that {Snoc,[]} covers all values

These equations are complete

Panic!
My head just exploded



The answer: ICFP 2015

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

$$\boxed{patVectProc(\vec{p}, S) = \langle C, U, D\rangle}$$

$$patVectProc\,(\vec{p}, S) = \langle C, U, D\rangle \quad \text{where} \quad \begin{array}{l} C = \{w \mid v \in S, w \in \mathcal{C}\,\vec{p}\,v,\ \vdash_{\text{SAT}} w\} \\ U = \{w \mid v \in S, w \in \mathcal{U}\,\vec{p}\,v,\ \vdash_{\text{SAT}} w\} \\ D = \{w \mid v \in S, w \in \mathcal{D}\,\vec{p}\,v,\ \vdash_{\text{SAT}} w\} \end{array}$$

$$\boxed{\mathcal{C}\,\vec{p}\,v = C \ \ \text{(always empty or singleton set)}}$$

$$\begin{array}{llll}
[\text{CNIL}] & \mathcal{C}\ \epsilon & (\Gamma \vdash \epsilon \rhd \Delta) & = \{\Gamma \vdash \epsilon \rhd \Delta\} \\
[\text{CCONCON}] & \mathcal{C}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta) & = \begin{cases} map\ (kcon\ K_i)\ (\mathcal{C}\ (\vec{p}\,\vec{q})\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta)) & \text{if } K_i = K_j \\ \varnothing & \text{if } K_i \neq K_j \end{cases} \\
[\text{CCONVAR}] & \mathcal{C}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = \mathcal{C}\ ((K_i\,\vec{p})\,\vec{q})\ (\Gamma' \vdash (K_i\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \quad \text{where } \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x{:}\tau_x) \in \Gamma \quad K_i :: \forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau \\
 & & & \qquad \Gamma' = \Gamma, \vec{a}, \vec{y}{:}\vec{\tau} \\
 & & & \qquad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i\,\vec{y} \\
[\text{CVAR}] & \mathcal{C}\ (x\,\vec{p}) & (\Gamma \vdash u\,\vec{u} \rhd \Delta) & = map\ (ucon\ u)\ (\mathcal{C}\ (\vec{p})\ (\Gamma, x{:}\tau \vdash \vec{u} \rhd \Delta \cup x \approx u)) \quad \text{where } x\#\Gamma \quad \Gamma \vdash u : \tau \\
[\text{CGUARD}] & \mathcal{C}\ ((p \leftarrow e)\,\vec{p}) & (\Gamma \vdash \vec{u} \rhd \Delta) & = map\ tail\ (\mathcal{C}\ (p\,\vec{p})\ (\Gamma, y{:}\tau \vdash y\,\vec{u} \rhd \Delta \cup y \approx e)) \quad \text{where } y\#\Gamma \quad \Gamma \vdash e : \tau
\end{array}$$

$$\boxed{\mathcal{U}\,\vec{p}\,v = U}$$

$$\begin{array}{llll}
[\text{UNIL}] & \mathcal{U}\ \epsilon & (\Gamma \vdash \epsilon \rhd \Delta) & = \varnothing \\
[\text{UCONCON}] & \mathcal{U}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta) & = \begin{cases} map\ (kcon\ K_i)\ (\mathcal{U}\ (\vec{p}\,\vec{q})\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta) & \text{if } K_i = K_j \\ \{\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta\} & \text{if } K_i \neq K_j \end{cases} \\
[\text{UCONVAR}] & \mathcal{U}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = \bigcup_{K_j} \mathcal{U}\ ((K_i\,\vec{p})\,\vec{q})\ (\Gamma' \vdash (K_j\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \quad \text{where } \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x{:}\tau_x) \in \Gamma \quad K_j :: \forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau \\
 & & & \qquad \Gamma' = \Gamma, \vec{a}, \vec{y}{:}\vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_j\,\vec{y} \\
[\text{UVAR}] & \mathcal{U}\ (x\,\vec{p}) & (\Gamma \vdash u\,\vec{u} \rhd \Delta) & = \textit{exactly like [CVAR], with } \mathcal{U} \textit{ instead of } \mathcal{C} \\
[\text{UGUARD}] & \mathcal{U}\ ((p \leftarrow e)\,\vec{p}) & (\Gamma \vdash \vec{u} \rhd \Delta) & = \textit{exactly like [CGUARD], with } \mathcal{U} \textit{ instead of } \mathcal{C}
\end{array}$$

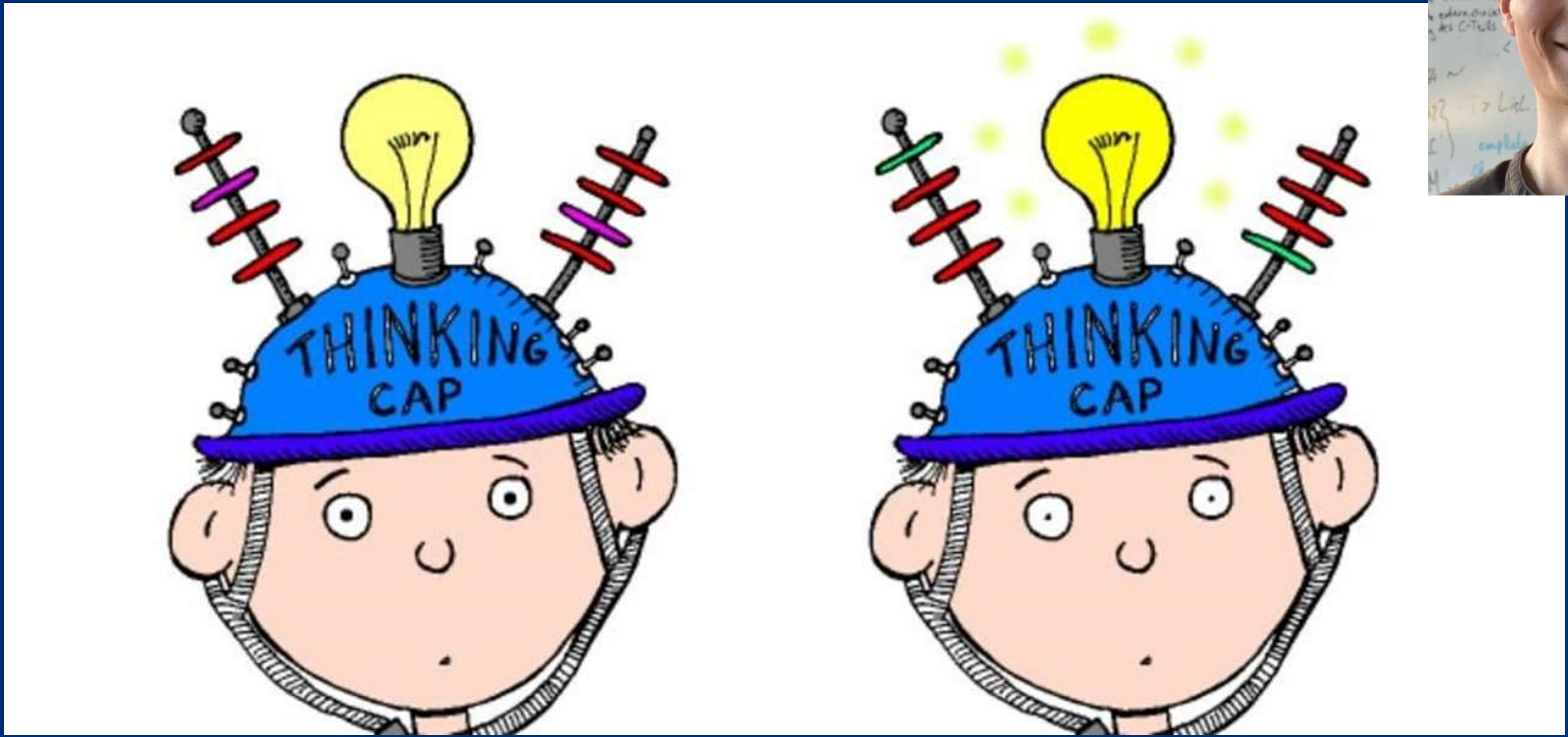
$$\boxed{\mathcal{D}\,\vec{p}\,v = D}$$

$$\begin{array}{llll}
[\text{DNIL}] & \mathcal{D}\ \epsilon & (\Gamma \vdash \epsilon \rhd \Delta) & = \varnothing \\
[\text{DCONCON}] & \mathcal{D}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta) & = \begin{cases} map\ (kcon\ K_i)\ (\mathcal{D}\ (\vec{p}\,\vec{q})\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta) & \text{if } K_i = K_j \\ \varnothing & \text{if } K_i \neq K_j \end{cases} \\
[\text{DCONVAR}] & \mathcal{D}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = \{\Gamma \vdash x\,\vec{u} \rhd \Delta \cup (x \approx \bot)\} \ \cup\ \mathcal{D}\ ((K_i\,\vec{p})\,\vec{q})\ (\Gamma' \vdash (K_i\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \quad \text{where } \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x{:}\tau_x) \in \Gamma \quad K_i :: \forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau \\
 & & & \qquad \Gamma' = \Gamma, \vec{a}, \vec{y}{:}\vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i\,\vec{y} \\
[\text{DVAR}] & \mathcal{D}\ (x\,\vec{p}) & (\Gamma \vdash u\,\vec{u} \rhd \Delta) & = \textit{exactly like [CVAR], with } \mathcal{D} \textit{ instead of } \mathcal{C} \\
[\text{DGUARD}] & \mathcal{D}\ ((p \leftarrow e)\,\vec{p}) & (\Gamma \vdash \vec{u} \rhd \Delta) & = \textit{exactly like [CGUARD], with } \mathcal{D} \textit{ instead of } \mathcal{C}
\end{array}$$

A big step forward

But
- tricky
- buggy
- slow

Sebastian Graf



A new, simple, modular approach

Two simple ideas

1. Desugar all pattern matching to guards
2. Collect the available facts into a fact-base Δ

Desugar pattern matching to guards

```
f (Just (!xs,_)) ys@(y:_) | y > 3 = rhs1
f Nothing        zs             = rhs2
```

desugars thus:

```
f as ys
  | Just t <- as
  , (xs,v) <- t
  , !xs
  , (y:w) <- ys
  , let b = (y > 3)
  , True <- b
  = rhs1
  | Nothing <- as
  , let zs = ys
  = rhs2
```

Desugar pattern matching to guards

```
f (Just (!xs,_)) ys@(y:_) | y > 3 = rhs1
f Nothing        zs             = rhs2
```

```
f as ys
  | Just t <- as
  , (xs,v) <- t
  , !xs
  , (y:w) <- ys
  , let b = (y > 3)
  , True <- b
  = rhs1
  | Nothing <- as
  , let zs = ys
  = rhs2
```

One-level matching

Matching only on a variable

Simply evaluates xs

Ordinary let-binding

Fix name differences

Desugar pattern matching to guards

$$\begin{array}{rcl} grd & ::= & !x \\ & \mid & K\ x_1 \ldots x_n \leftarrow y \\ & \mid & let\ x = e \end{array}$$

This is enough to express

- As-patterns
- View patterns
- Record patterns
- Pattern guards
- Wildcard patterns
- Overloaded literal patterns
- List and tuple patterns
- n+k patterns
- Bang patterns
- Lazy patterns
- Pattern synonyms

After desugaring

```
f x y
  | g11, g12, g13       = rhs1
  | g21, g22            = rhs2
  | g31, g32, g33, g34  = rhs3
```

Clause 1

Clause 2

$$
\begin{aligned}
grd ::=\ & !x \\
|\ & K\ x_1 \ldots x_n \leftarrow y \\
|\ & let\ x = e
\end{aligned}
$$

NB: this desugaring is for pattern-match overlap checking only,
not execution

Missing equations

All values

Clause 1

Values not covered by clause 1

Clause 2

Values not covered by clauses 1 or 2

Clause 3

Values not covered by clauses 1, 2, or 3

We want to report these – could be a runtime error

Missing equations

All values

Clause 1

Values not covered by clause 1

Clause 2

Values not covered by clauses 1 or 2

Clause 3

Values not covered by clauses 1, 2, or 3

Question
How do we represent a (possibly infinite) set of values?

We want to report these – could be a runtime error

Idea: represent set of values by a factbase Δ

- $\Delta = \{x{:}Maybe\ Bool \mid \epsilon\}$ represents
  $\{\bot, Nothing, Just\ \bot, Just\ True, Just\ False\}$
- $\Delta = \{x{:}Maybe\ Bool \mid Nothing \leftarrow x\}$ represents
  $\{Nothing\}$
- $\Delta = \{x{:}Maybe\ Bool \mid Just\ (y{:}Bool) \leftarrow x\}$ represents
  $\{Just\ \bot, Just\ True, Just\ False\}$
- $\Delta = \{x{:}Maybe\ Bool \mid Just\ (y{:}Bool) \leftarrow x, True \leftarrow y\}$ represents
  $\{Just\ True\}$

Things that are true about every value in the set

Idea: represent set of values by a factbase Δ

- A type describes a set of values
- So does Δ.   So Δ is a sort of type.
- Indeed a well-known sort of type: a **refinement type**

$\{x{:}Maybe\ Int \mid Just\ (y{:}Int) \leftarrow x,\ y > 3\}$

Example

```
f (Just True) = rhs
```

Desugars to

```
f x | Just y <- x, True <- y = rhs
```

Example

$\Delta = \{x{:}Maybe\ Bool \mid \epsilon\}$

```
f x | Just y <- x, True <- y = rhs
```

Values not covered by the equation

$\Delta = \{x{:}Maybe\ Bool \mid x \neq Just,\ x \neq \bot\} \cup$

$\{x{:}Maybe\ Bool \mid Just\ (y{:}Bool) \leftarrow x,\ y \neq True,\ y \neq \bot\}$

How do we do that in general?

Computing the uncovered set

$U(\Delta, gs)$ = the subset of Δ whose values do not match the guards gs

$U(\Delta, [\,]) = \emptyset$

$U(\Delta, (K\ ys \leftarrow x) : gs) = (\Delta + x \neq K, x \neq \bot) \cup U(\Delta + (K\ ys \leftarrow x), gs)$

$U(\Delta, (!x) : gs) = U(\Delta + x \neq \bot, gs)$

$U(\Delta, (let\ x = e) : gs) = U(\Delta + (let\ x = e), gs)$

...and that is all!

Reporting uncovered sets

$\Delta = [x{:}Maybe\ Bool \mid \epsilon]$

```
f x | Just y <- x, True <- y = rhs
```

$\Delta = \{\, x{:}Maybe\ Bool \mid x \neq Just,\ x \neq \perp \} \cup$

$\{\, x{:}Maybe\ Bool \mid Just\ (y{:}Bool) \leftarrow x,\ y \neq True,\ y \neq \perp \}$

- Next question: what values satisfy the Δ that falls out of the bottom – these are the cases that are not covered
- Empty => equations are exhaustive.

Reporting uncovered sets

```
f x | Just y <- x, True <- y = rhs
```

$$\Delta = \{\, x{:}Maybe\ Bool \mid x \neq Just,\ x \neq \bot \,\} \cup$$

$$\{\, x{:}Maybe\ Bool \mid Just\ (y{:}Bool) \leftarrow x,\ y \neq True,\ y \neq \bot \,\}$$

Easy!

- Pick each disjunct in turn $[\, x : Maybe\ Bool \mid \theta \,]$
- Start from $x{:}Maybe\ Bool$
- Pick a value of x that works for θ
- Repeat

Example

$\Delta = \{\, x{:}Maybe\ Bool \mid x \neq Just,\ x \neq \perp \} \cup$ …

- Start from $x{:}Maybe\ Bool$
- Pick a value of x that works for $x \neq Just,\ x \neq \perp$
- $x = Nothing$ looks good. (NB in general there may be many.)
- Done

Example

$\Delta = \ldots \cup \{x{:}Maybe\ Bool \mid Just\ (y{:}Bool) \leftarrow x,\ y \neq True,\ y \neq \bot\}$

- Start from $x{:}Maybe\ Bool$
- Pick a value of x that works for $Just\ (y{:}Bool) \leftarrow x$
- $x = Just\ (y{:}Bool)$ looks good.
- Pick a value of y that works for $y \neq True,\ y \neq \bot$
- $y = False$ looks good

Result: $x = Just\ False$

Example

```
f (Just True) = rhs
```

desugars to

```
f x | Just y <- x, True <- y = rhs
```

reports uncovered possibilities

x=Nothing

x=Just False

Empty Δ

```
g (Just y)  = y
g Nothing   = False
```

desugars to

```
g x | Just y  <- x = y
    | Nothing <- x = 0
```

$\Delta = \{x{:}Maybe\ Bool \mid x \neq Just,\ x \neq \bot,\ x \neq Nothing\}$

- **What values does this Δ represent?**

Empty Δ

```
g (Just y) = y
g Nothing  = False
```

desugars to

```
g x | Ju
    | N                        0
```

Δ represents the empty set - no values satisfy it

So g is exhaustive.

$\Delta = \{x{:}Maybe\ Bool \mid x \neq Just, \ \ \neq \perp, x \neq Nothing\}$

- **What values does this Δ represent?**

Scaling up to all of Haskell

Redundant/inaccessible equations

- Modifying $U(\Delta, gs)$ a little bit deals with **redundant/ inaccessible equations**
- **Pattern synonyms**: some footwork when coming up with uncovered sets. E.g. what values are expressed by

$$\Delta=\{x{:}[Int] \mid x\neq[\,], x\neq Snoc, x\neq\bot\}$$

Answer: none, because {[], Snoc} is COMPLETE

Pattern synonyms

- Just needs some footwork when coming up with uncovered sets.
- E.g. what values are expressed by

$$\Delta = \{x : [Int] \mid x \neq [\,], x \neq Snoc, x \neq \bot\}$$

- Answer: none, because {[], Snoc} is COMPLETE

GADTs

- Δ contains **type equalities** as well as term equalities

```
data T a where
  TBool  :: T Bool
  ...

f :: a -> T a -> a
f x y | TBool <- y, True <- x = …
```

- $\Delta = \{x{:}a\,,\, y{:}T\ a \mid TBool \leftarrow y,\ \boldsymbol{a \sim Bool}, True \leftarrow x\}$
- Re-uses GHC's type-constraint solver

Long distance information: easy!

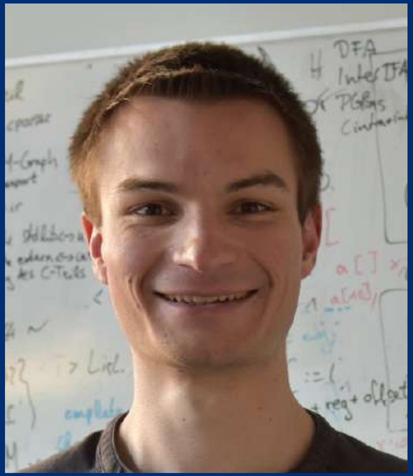
```
data Grade = A | B | C

f :: Bool -> Grade -> blah
f _    A = …
f True g = … (case g of
                B -> True
                C -> False) …
```

Simply start this case from the enclosing Δ!

We get to this RHS with $\Delta = \{b{:}Bool,\ g{:}Grade \mid g \neq A,\ True \leftarrow b\}$

Conclusion





- A long, long road
- A satisfying conclusion
  - Theory is a lot simpler
  - Code is a lot simpler
  - And a lot shorter
  - And runs faster
  - And nails many bugs