

# GADTs Meet Their Match:

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

SEBASTIAN GRAF, Karlsruhe Institute of Technology, Germany

SIMON PEYTON JONES, Microsoft Research, UK

## 1 OUR SOLUTION

It is customary to define Haskell functions using pattern-matching, possibly with one or more *guarded right-hand sides* (GRHS) per *clause* (see fig. 1). Consider for example this 3 AM attempt at lifting equality over *Maybe*:

```
f Nothing Nothing = True
f (Just x) (Just y)
  | x == y      = True
  | otherwise   = False
```

This function will crash for the call site  $f \text{ (Just 1) Nothing}$ . To see that, we can follow Haskell's top-to-bottom, left-to-right pattern match semantics. The first clause already fails to match *Nothing* against *Just 1*, while the second clause successfully matches  $x$  with 1, but then fails trying to match *Just y* against *Nothing*. There is no third clause, and an *uncovered* value vector that falls out at the bottom of this process will lead to a crash.

Compare that to matching on  $(\text{Just } 1) (\text{Just } 2)$ : While matching against the first clause fails, the second matches  $x$  to 1 and  $y$  to 2. Since there are multiple guarded right-hand sides, every one of them in turn has to be tried in a top-to-bottom fashion. The first GRHS consists of a single boolean guard (in general we have to consider each of them in a left-to-right fashion!) **SG: Maybe an example with more guards would be helpful** that will fail because  $1 \neq 2$ . So the second GRHS is tried successfully, because *otherwise* is a boolean guard that never fails.

Note how both the pattern matching per clause and the guard checking within a syntactic *match* share top-to-bottom and left-to-right semantics. Having to make sense of both pattern and guard semantics seems like a waste of energy. Why can't we just express all pattern matching simply by pattern guards on an auxiliary variable match? See for yourself:

```
f mx my
  | Nothing ← mx, Nothing ← my      = True
  | Just x ← mx, Just y ← my | x == y = True
  | otherwise = False
```

Transforming the first clause with its single GRHS was quite successful. But the second clause already had two GRHSs before, and the resulting tree-like nesting of guards definitely is not valid Haskell! Although intuitively, this is just what we want: After the successful match on the first two guards left-to-right, we try to match each of the GRHSs in turn, top-to-bottom (and their individual guards left-to-right). In fact, it seems rather arbitrary to only allow one level of nested guards! Hence our algorithm desugars the source syntax to the following *guard tree* (see fig. 2 for the full syntax):

**SG: Find shorter aliases for the syntax, maybe make top-to-bottom sequence prefix. Or a more graphic representation, even. Will sketch it out when we have some prose to work on. For now**

Authors' addresses: Sebastian Graf, Karlsruhe Institute of Technology, Karlsruhe, Germany, sebastian.graf@kit.edu; Simon Peyton Jones, Microsoft Research, Cambridge, UK, simonpj@microsoft.com.

Meta variables		Pattern Syntax	
$x, y, z, f, g, h$	Term variables	$defn ::= \overline{clause}$	
$a, b, c$	Type variables	$clause ::= f \overline{pat} \overline{match}$	
$K$	Data constructors	$pat ::= x \mid K \overline{pat}$	
$P$	Pattern synonyms	$match ::= = \overline{expr} \mid \overline{grhss}$	
$T$	Type constructors	$grhss ::= \mid \overline{guard} = \overline{expr}$	
		$guard ::= pat \leftarrow \overline{expr} \mid \overline{expr} \mid \text{let } x = \overline{expr}$	

**Fig. 1.** Source syntax

assume that Guard binds stronger than sequence (;) **SG:** The bangs are distracting. Also the otherwise. Also binding the temporary.

```
Guard (!mx) Guard (Nothing ← mx) Guard (!my) Guard (Nothing ← my) Rhs 1;
Guard (!mx) Guard (Just x ← mx) Guard (!my) Guard (Just y ← my)
(Guard (let t = x == y) Guard (!t) Guard (True ← t) Rhs 2;
Guard (!otherwise) Guard (True ← otherwise) Rhs 3)
```

This representation is quite a bit more explicit than the original program. For one thing, every source-level pattern guard is strict in its scrutinee, whereas the pattern guards in our tree language are not, so we had to insert bang patterns. **SG:** This makes me question again if making pattern guards "lazy" was the right choice. But I like to keep the logic of bang patterns orthogonal to pattern guards in our checking function. For another thing, the pattern guards in Grd only scrutinise variables (and only one level deep), so the comparison in the boolean guard's scrutinee had to be bound to an auxiliary variable in a let binding.

Pattern guards in Grd are the only guards that can possibly fail to match, in which case the value of the scrutinee was not of the shape of the constructor application it was matched against. The Gdt tree language determines how to cope with a failed guard. Left-to-right matching semantics is captured by Guard, whereas top-to-bottom backtracking is expressed by sequence (;). The leaves in this tree, Rhs, each correspond to a GRHS. **SG:** The preceding and following paragraph would benefit from illustrations. It's hard to come up with something concrete that doesn't go into too much detail. GMTM just shows a top-to-bottom pipeline. But why should we leave out left-to-right composition? Also we produce an annotated syntax tree Ant instead of a covered set.

Pattern match checking works by gradually refining the set of uncovered values as they flow through the tree and produces two values: The uncovered set that wasn't covered by any clause and an annotated guard tree skeleton Ant with the same shape as the guard tree to check, capturing redundancy (AccessibleRhs vs. InaccessibleRhs when the uncovered set that reaches the Rhs was empty) and divergence (expressed through a MayDiverge wrapper when a bang pattern forces a value that still can diverge) information.

We can generate missing clauses from the final uncovered set falling out at the bottom, just by generating inhabitants. The annotated tree on the other hand can be used to compute inaccessible and redundant GRHSs. **SG:** I think this kind of detail should be motivated in a prior section and then referenced here for its solution. Why not compute the redundant GRHSs directly? Because determining inaccessibility vs. redundancy is a non-local problem. Consider this example:

```
g :: () → Int
g () | False = 1
```

Guard Syntax		
$K \in$	Con	$n \in \mathbb{N}$
$x, y, a, b \in$	Var	$\gamma \in \text{TyCt} ::= \tau_1 \sim \tau_2 \mid \dots$
$\tau, \sigma \in$	Type	$p \in \text{Pat} ::= \_$
$e \in$	Expr	$ ::= x$
		$ ::= K \bar{\tau} \bar{\sigma} \bar{\gamma} \bar{e}$
		$ ::= \dots$
		$g \in \text{Grd} ::= \text{let } x : \tau = e$
		$ ::= K \bar{a} \bar{\gamma} \bar{y} : \bar{\tau} \leftarrow x$
		$ ::= !x$
Constraint Formula Syntax		
$\Gamma ::=$	$\emptyset \mid \Gamma, x : \tau \mid \Gamma, a$	Context
$\delta ::=$	$\checkmark \mid \times \mid K \bar{a} \bar{\gamma} \bar{y} : \bar{\tau} \leftarrow x \mid x \not\approx K \mid x \approx \perp \mid x \not\approx \perp \mid x \approx e$	Constraint Literals
$\Delta ::=$	$\delta \mid \Delta \wedge \Delta \mid \Delta \vee \Delta$	Formula
$\varphi ::=$	$\gamma \mid x \approx K \bar{a} \bar{\gamma} \mid x \not\approx K \mid x \approx \perp \mid x \not\approx \perp \mid x \approx y$	Simple constraints without scoping
$\Phi ::=$	$\emptyset \mid \Phi, \varphi$	Set of simple constraints
$\nabla ::=$	$\Gamma \triangleright \Phi \mid \times$	Inert Set
Clause Tree Syntax		
$t_G, u_G \in$	Gdt	$ ::= \text{Rhs } n \mid t_G; u_G \mid \text{Guard } g \ t_G$
$t_A, u_A \in$	Ant	$ ::= \text{AccessibleRhs } n \mid \text{InaccessibleRhs } n \mid t_A; u_A \mid \text{MayDiverge } t_A$

Fig. 2. IR Syntax

$\mid \text{True} = 2$   
 $g - = 3$

Is the first clause inaccessible or even redundant? Although the match on  $()$  forces the argument, we can delete the first clause without changing program semantics, so clearly it's redundant. But that wouldn't be true if the second clause wasn't there to "keep alive" the  $()$  pattern!

Here is the corresponding annotated tree after checking:

MayDiverge (InaccessibleRhs 1; AccessibleRhs 2); AccessibleRhs 3

In general, at least one GRHS under a MayDiverge may not be flagged as redundant. The decision which GRHSs are redundant (vs. just inaccessible) can thus only happen in an additional pass over the annotated tree, rather than when the checking algorithm reaches a particular Rhs.

Perhaps surprisingly and most importantly, Grd with its three primitive guards, combined with left-to-right or top-to-bottom semantics in Gdt, is expressive enough to express all pattern matching in Haskell (cf. fig. TODO)! We have yet to find a language extension that doesn't fit into this framework.

## 2 END TO END EXAMPLE

We'll start from the following source Haskell program and see how each of the steps (translation to guard trees, checking guard trees and ultimately generating inhabitants of the occurring  $\Delta$ s) work.

$f :: \text{Maybe Int} \rightarrow \text{Int}$   
 $f \text{ Nothing} = 0 \quad \text{-- RHS 1}$   
 $f x \mid \text{Just } y \leftarrow x = y \quad \text{-- RHS 2}$

### Checking Guard Trees

$$\boxed{\mathcal{U}(t_G) = \Delta}$$

$$\begin{aligned} \mathcal{U}(\text{Rhs } n) &= \times \\ \mathcal{U}(t; u) &= \mathcal{U}(t) \wedge \mathcal{U}(u) \\ \mathcal{U}(\text{Guard } (!x) \ t) &= (x \neq \perp) \wedge \mathcal{U}(t) \\ \mathcal{U}(\text{Guard } (\text{let } x = e) \ t) &= (x \approx e) \wedge \mathcal{U}(t) \\ \mathcal{U}(\text{Guard } (K \ \bar{a} \ \bar{y} \ \bar{y} : \bar{\tau} \leftarrow x) \ t) &= (x \neq K) \vee ((K \ \bar{a} \ \bar{y} \ \bar{y} : \bar{\tau} \leftarrow x) \wedge \mathcal{U}(t)) \end{aligned}$$

$$\boxed{\mathcal{A}_\Gamma(\Delta, t_G) = t_A}$$

$$\begin{aligned} \mathcal{A}_\Gamma(\Delta, \text{Rhs } n) &= \begin{cases} \text{InaccessibleRhs } n, & \mathcal{G}(\Gamma, \Delta) = \emptyset \\ \text{AccessibleRhs } n, & \text{otherwise} \end{cases} \\ \mathcal{A}_\Gamma(\Delta, (t; u)) &= \mathcal{A}_\Gamma(\Delta, t); \mathcal{A}_\Gamma(\Delta \wedge \mathcal{U}(t), u) \\ \mathcal{A}_\Gamma(\Delta, \text{Guard } (!x) \ t) &= \begin{cases} \mathcal{A}_\Gamma(\Delta \wedge (x \neq \perp), t), & \mathcal{G}(\Gamma, \Delta \wedge (x \approx \perp)) = \emptyset \\ \text{MayDiverge } \mathcal{A}_\Gamma(\Delta \wedge (x \neq \perp), t) & \text{otherwise} \end{cases} \\ \mathcal{A}_\Gamma(\Delta, \text{Guard } (\text{let } x = e) \ t) &= \mathcal{A}_\Gamma(\Delta \wedge (x \approx e), t) \\ \mathcal{A}_\Gamma(\Delta, \text{Guard } (K \ \bar{a} \ \bar{y} \ \bar{y} : \bar{\tau} \leftarrow x) \ t) &= \mathcal{A}_\Gamma(\Delta \wedge (K \ \bar{a} \ \bar{y} \ \bar{y} : \bar{\tau} \leftarrow x), t) \end{aligned}$$

### Putting it all together

- (0) Input: Context with match vars  $\Gamma$  and desugared Gdt  $t$
- (1) Report  $n$  pattern vectors of  $\mathcal{G}(\Gamma, \mathcal{U}(t))$  as uncovered
- (2) Report the collected redundant and not-redundant-but-inaccessible clauses in  $\mathcal{A}_\Gamma(\checkmark, t)$   
(TODO: Write a function that collects the RHSs).

Fig. 3. Pattern-match checking

## 2.1 Translation to guard trees

The program (by a function we probably only give in the appendix?) corresponds to the following guard tree  $t_f$ :

Guard (!x) Guard (Nothing  $\leftarrow$  x) Rhs 1;  
Guard (!x) Guard (Just  $y \leftarrow$  x) Rhs 2

Data constructor matches are strict, so we add a bang for each match.

## 2.2 Checking

**2.2.1 Uncovered values.** First compute the uncovered  $\Delta$ s, after the first and the second clause respectively.

(1)

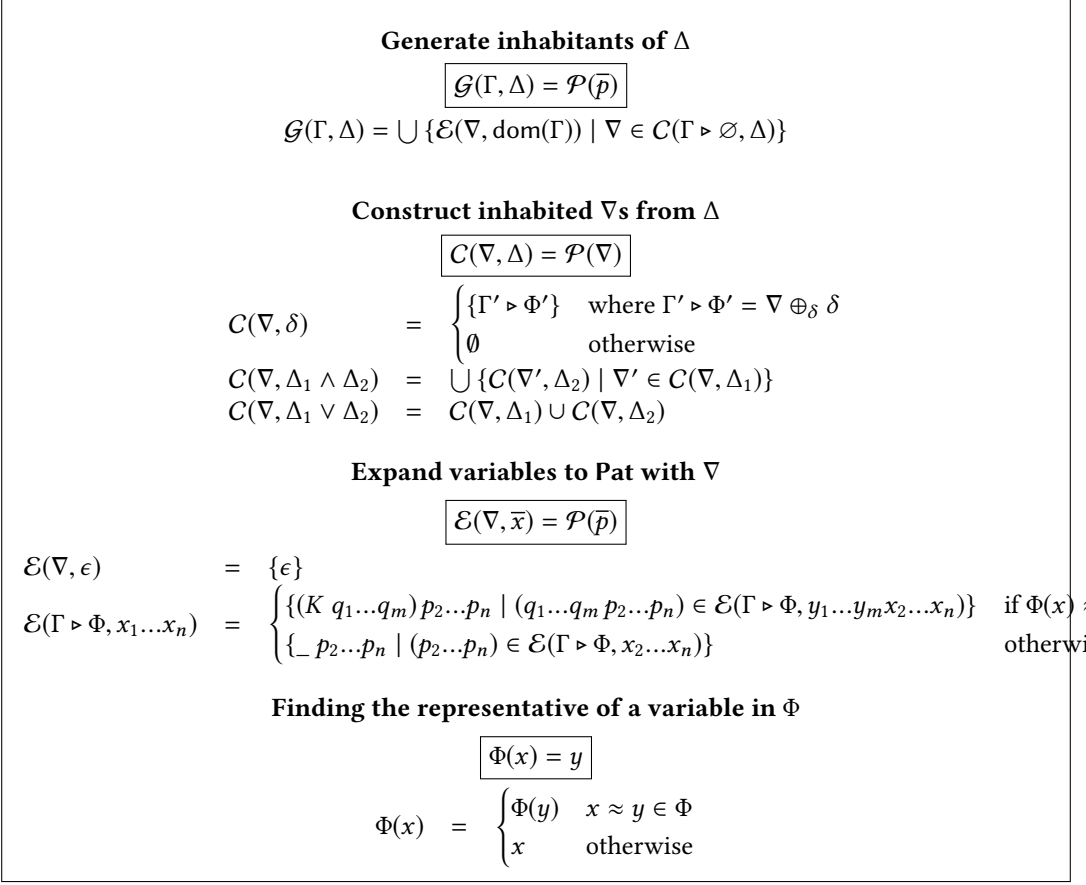
$$\begin{aligned} \Delta_1 &:= \mathcal{U}(\text{Guard } (!x) \ \text{Guard } (\text{Nothing} \leftarrow x) \ \text{Rhs } 1) \\ &= x \neq \perp \wedge (x \neq \text{Nothing} \vee \times) \end{aligned}$$

(2)

$$\Delta_2 := \mathcal{U}(t_f) = \Delta_1 \wedge x \neq \perp \wedge (x \neq \text{Just} \vee \times)$$

The right operands of  $\vee$  are vacuous, but the purely syntactical transformation doesn't see that.

We can see that  $\Delta_2$  is in fact uninhabited, because the three constraints  $x \neq \perp$ ,  $x \neq \text{Nothing}$  and  $x \neq \text{Just}$  cover all possible data constructors of the Maybe data type. And indeed  $\mathcal{G}(x : \text{Maybe Int}, \Delta_2) = \emptyset$ , as we'll see later.



**Fig. 4.** Bridging between the facade  $\Delta$  and  $\nabla$

**2.2.2 Redundancy.** In order to compute the annotated clause tree  $\mathcal{A}_\Gamma(\checkmark, t_f)$ , we need to perform the following four inhabitation checks, one for each bang (for knowing whether we need to wrap a `MayDiverge` and one for each RHS (where we have to decide for `InaccessibleRhs` or `AccessibleRhs`):

- (1) The first divergence check:  $\Delta_3 := \checkmark \wedge x \approx \perp$
- (2) Upon reaching the first RHS:  $\Delta_4 := \checkmark \wedge x \not\approx \perp \wedge \text{Nothing} \leftarrow x$
- (3) The second divergence check:  $\Delta_5 := \checkmark \wedge \Delta_1 \wedge x \approx \perp$
- (4) Upon reaching the second RHS:  $\Delta_6 := \checkmark \wedge \Delta_1 \wedge x \not\approx \perp \wedge \text{Just } y \leftarrow x$

Except for  $\Delta_5$ , these are all inhabited, i.e.  $\mathcal{G}(x : \text{Maybe Int}, \Delta_i) \neq \emptyset$  (as we'll see in the next section).

Thus, we will get the following annotated tree:

MayDiverge AccessibleRhs 1; AccessibleRhs 2

### 2.3 Generating inhabitants

The last section left open how  $\mathcal{G}(\cdot)$  works, which was used to establish or refute vacuity of a  $\Delta$ .

$\mathcal{G}(\cdot)$  proceeds in two steps: First it constructs zero, one or many *inert sets*  $\nabla$  with  $C(\cdot)$  (each of them representing a set of mutually compatible constraints) and then expands each of the returned

### Add a constraint to the inert set

$$\boxed{\nabla \oplus_{\delta} \delta = \nabla}$$

$$\begin{aligned}
\nabla \oplus_{\delta} \times &= \times \\
\nabla \oplus_{\delta} \checkmark &= \nabla \\
\Gamma \triangleright \Phi \oplus_{\delta} K \bar{a} \bar{y} \overline{y : \tau} \leftarrow x &= \Gamma, \bar{a}, \bar{y} : \tau \triangleright \Phi \oplus_{\varphi} \bar{y} \oplus_{\varphi} x \approx K \bar{a} \bar{y} \\
\Gamma \triangleright \Phi \oplus_{\delta} x \approx K \bar{\tau}' \bar{\tau} \bar{y} \bar{e} &= \Gamma, \bar{a}, \bar{y} : \sigma \triangleright \Phi \oplus_{\delta} K \bar{a} \bar{y} \bar{y} \leftarrow x \oplus_{\varphi} \bar{a} \sim \bar{\tau} \oplus_{\delta} \bar{y} \approx \bar{e} \text{ where } \bar{a} \# \Gamma, \bar{y} \# \Gamma, \bar{e} : \sigma \\
\nabla \oplus_{\delta} x \approx e &= \nabla \\
\Gamma \triangleright \Phi \oplus_{\delta} \delta &= \Gamma \triangleright \Phi \oplus_{\varphi} \delta
\end{aligned}$$

### Add a simple constraint to the inert set

$$\boxed{\nabla \oplus_{\varphi} \varphi = \nabla}$$

$$\begin{aligned}
\times \oplus_{\varphi} \varphi &= \times \\
\Gamma \triangleright \Phi \oplus_{\varphi} \gamma &= \begin{cases} \Gamma \triangleright (\Phi, \gamma) & \text{if type checker deems } \gamma \text{ compatible with } \Phi \\ & \text{and } \forall x \in \text{dom}(\Gamma) : \Gamma \triangleright (\Phi, \gamma) \vdash \Phi(x) \\ \times & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Phi \oplus_{\varphi} x \approx K \bar{a} \bar{y} &= \begin{cases} \Gamma \triangleright \Phi \oplus_{\varphi} \bar{a} \sim \bar{b} \oplus_{\varphi} \bar{y} \approx \bar{z} & \text{if } \Phi(x) \approx K \bar{b} \bar{z} \in \Phi \\ \Gamma' \triangleright (\Phi', \Phi(x) \approx K \bar{a} \bar{y}) & \text{where } \Gamma' \triangleright \Phi' = \Gamma \triangleright \Phi \oplus_{\varphi} \bar{y} \\ & \text{and } \Phi'(x) \approx K \notin \Phi' \text{ and } \Gamma' \triangleright \Phi' \vdash y \\ \times & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Phi \oplus_{\varphi} x \approx K &= \begin{cases} \times & \text{if } \Phi(x) \approx K \bar{a} \bar{y} \in \Phi \\ \times & \text{if not } \Gamma \triangleright (\Phi, \Phi(x) \approx K) \vdash \Phi(x) \\ \Gamma \triangleright (\Phi, \Phi(x) \approx K) & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Phi \oplus_{\varphi} x \approx \perp &= \begin{cases} \perp & \text{if } \Phi(x) \approx \perp \in \Phi \\ \Gamma \triangleright (\Phi, \Phi(x) \approx \perp) & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Phi \oplus_{\varphi} x \approx \perp &= \begin{cases} \times & \text{if } \Phi(x) \approx \perp \in \Phi \\ \times & \text{if not } \Gamma \triangleright (\Phi, \Phi(x) \approx \perp) \vdash \Phi(x) \\ \Gamma \triangleright (\Phi, \Phi(x) \approx \perp) & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Phi \oplus_{\varphi} x \approx y &= \begin{cases} \Gamma \triangleright \Phi & \text{if } \Phi(x) = \Phi(y) \\ \Gamma \triangleright (\Phi, \Phi(x) \approx \Phi(y)) \oplus_{\varphi} ((\Phi \cap \Phi(x))[\Phi(y)/\Phi(x)]) & \text{otherwise} \end{cases}
\end{aligned}$$

$$\boxed{\Phi \cap x = \Phi}$$

$$\begin{aligned}
\emptyset \cap x &= \emptyset \\
(\Phi, x \approx K \bar{a} \bar{y}) \cap x &= (\Phi \cap x), x \approx K \bar{a} \bar{y} \\
(\Phi, x \approx K) \cap x &= (\Phi \cap x), x \approx K \\
(\Phi, x \approx \perp) \cap x &= (\Phi \cap x), x \approx \perp \\
(\Phi, x \approx \perp) \cap x &= (\Phi \cap x), x \approx \perp \\
(\Phi, \varphi) \cap x &= \Phi \cap x
\end{aligned}$$

**Fig. 5.** Adding a constraint to the inert set  $\nabla$

inert sets into one or more pattern vectors  $\bar{p}$  with  $\mathcal{E}(\cdot, \cdot)$ , which is the preferred representation to show to the user.

**Test if  $x$  is inhabited considering  $\nabla$**

$$\begin{array}{c}
 \boxed{\nabla \vdash x} \\
 x : \tau \in \Gamma \quad K \in \text{Cons}(\Gamma \triangleright \Phi, \tau) \\
 \frac{(\Gamma \triangleright \Phi \oplus_{\varphi} x \approx \perp) \neq \times}{\Gamma \triangleright \Phi \vdash x} \quad \frac{\text{Inst}(\Gamma, x, K) = \bar{\varphi} \quad (\Gamma, y : \tau' \triangleright \Phi \oplus_{\varphi} \bar{\varphi}) \neq \times}{\Gamma \triangleright \Phi \vdash x} \\
 \\
 \frac{x : \tau \in \Gamma \quad \text{Cons}(\Gamma \triangleright \Phi, \tau) = \perp}{\Gamma \triangleright \Phi \vdash x} \quad \frac{x : \tau \in \Gamma \quad K \in \text{Cons}(\Gamma \triangleright \Phi, \tau) \quad \text{Inst}(\Gamma, x, K) = \perp}{\Gamma \triangleright \Phi \vdash x}
 \end{array}$$

**Find data constructors of  $\tau$**

$$\begin{array}{c}
 \boxed{\text{Cons}(\Gamma \triangleright \Phi, \tau) = \bar{K}} \\
 \text{Cons}(\Gamma \triangleright \Phi, \tau) = \begin{cases} \bar{K} & \tau = T \bar{\sigma} \text{ and } T \text{ data type with constructors } \bar{K} \\ & \text{(after normalisation according to the type constraints in } \Phi) \\ \perp & \text{otherwise} \end{cases}
 \end{array}$$

**Instantiate  $x$  to data constructor  $K$**

$$\begin{array}{c}
 \boxed{\text{Inst}(\Gamma, x, K) = \bar{\varphi}} \\
 \text{Inst}(\Gamma, x, K) = \begin{cases} \tau_x \sim \tau, \bar{y}, x \approx K \bar{a} \bar{y}, \bar{y}' \not\approx \perp & K : \forall \bar{a}. \bar{y} \Rightarrow \bar{\sigma} \rightarrow \tau, \bar{y} \# \Gamma, \bar{a} \# \Gamma, x : \tau_x \in \Gamma, \bar{y}' \text{ bind strict fields} \\ \perp & \text{otherwise} \end{cases}
 \end{array}$$

**Fig. 6.** Inhabitation test

The interesting bit happens in  $C(\cdot)$ , where a  $\Delta$  is basically transformed into disjunctive normal form, represented by a set of independently inhabited  $\nabla$ . This ultimately happens in the base case of  $C(\cdot)$ , by gradually adding individual constraints to the incoming inert set with  $\oplus_{\varphi}$ , which starts out empty in  $\mathcal{G}(\cdot)$ . Conjunction is handled by performing the equivalent of a *concatMap*, whereas disjunction simply translates to set union.

Let's see how that works for  $\Delta_3$  above. Recall that  $\Gamma = x : \text{Maybe Int}$  and  $\Delta_3 = \checkmark \wedge x \approx \perp$ :

$$\begin{aligned}
& C(\Gamma, \checkmark \wedge x \approx \perp) \\
= & \{ \text{Conjunction} \} \\
& \cup \{ C(\Gamma' \triangleright \nabla', x \approx \perp) \mid \Gamma' \triangleright \nabla' \in C(\Gamma \triangleright \emptyset, \checkmark) \} \\
= & \{ \text{Single constraint} \} \\
& \begin{cases} C(\Gamma' \triangleright \nabla', x \approx \perp) & \text{where } \Gamma' \triangleright \nabla' = \Gamma \triangleright \emptyset \oplus_{\varphi} \checkmark \\ \emptyset & \text{otherwise} \end{cases} \\
= & \{ \checkmark \text{ case of } \oplus_{\varphi} \} \\
& C(\Gamma \triangleright \emptyset, x \approx \perp) \\
= & \{ \text{Single constraint} \} \\
& \begin{cases} \{ \Gamma' \triangleright \nabla' \} & \text{where } \Gamma' \triangleright \nabla' = \Gamma \triangleright \emptyset \oplus_{\varphi} x \approx \perp \\ \emptyset & \text{otherwise} \end{cases} \\
= & \{ x \approx \perp \text{ case of } \oplus_{\varphi} \} \\
& \{ \Gamma \triangleright x \approx \perp \}
\end{aligned}$$

Let's start with  $\mathcal{G}(\Gamma, \Delta_3)$ , where  $\Gamma = x : \text{Maybe Int}$  and recall that  $\Delta_3 = \checkmark \wedge x \approx \perp$ . The first constraint  $\checkmark$  is added very easily to the initial nabla by discarding it, the second one ( $x \approx \perp$ ) is not conflicting with any  $x \not\approx \perp$  constraint in the incoming, still empty ( $\emptyset$ ) nabla, so we end up with  $\Gamma \triangleright x \approx \perp$  as proof that  $\Delta_3$  is in fact inhabited. Indeed,  $\mathcal{E}(\Gamma \triangleright x \approx \perp, x)$  generate  $\_$  as the inhabitant (which is rather unhelpful, but correct).

The result of  $\mathcal{G}(\Gamma, \Delta_3)$  is thus  $\{ \_ \}$ , which is not empty. Thus,  $\mathcal{A}_{\Gamma}(\Delta, t)$  will wrap a `MayDiverge` around the first RHS.

Similarly,  $\mathcal{G}(\Gamma, \Delta_4)$  needs  $C(\Gamma \triangleright \emptyset, \Delta_4)$ , which in turn will add  $x \not\approx \perp$  to an initially empty  $\nabla$ . That entails an inhabitation check to see if  $x$  might take on any values besides  $\perp$ .

This is one possible derivation of the  $\Gamma \triangleright x \not\approx \perp \vdash x$  predicate:

$$\begin{array}{c}
x : \text{Maybe Int} \in \Gamma \quad \text{Nothing} \in \text{Cons}(\Gamma \triangleright x \not\approx \perp, \text{Maybe Int}) \\
\text{Inst}(\Gamma, x, \text{Nothing}) = \text{Nothing} \leftarrow x \\
(\Gamma \triangleright x \not\approx \perp \oplus_{\varphi} \text{Nothing} \leftarrow x) \neq \perp \\
\hline
\Gamma \triangleright x \not\approx \perp \vdash x
\end{array}$$

The subgoal  $\Gamma \triangleright x \not\approx \perp \oplus_{\varphi} \text{Nothing} \leftarrow x$  is handled by the second case of the match on constructor pattern constraints, because there are no other constructor pattern constraints yet in the incoming  $\nabla$ . Since there are no type constraints carried by `Nothing`, no fields and no constraints of the form  $x \not\approx K$  in  $\nabla$ , we end up with  $\Gamma \triangleright x \not\approx \perp, \text{Nothing} \leftarrow x$ . Which is not  $\perp$ , thus we conclude our proof of  $\Gamma \triangleright x \not\approx \perp \vdash x$ .

Next, we have to add  $\text{Nothing} \leftarrow x$  to our  $\nabla = x \not\approx \perp$ , which amounts to computing  $\Gamma \triangleright x \not\approx \perp \oplus_{\varphi} \text{Nothing} \leftarrow x$ . Conveniently, we just did that! So the result of  $C(\Gamma \triangleright \emptyset, \Delta_4)$  is  $\Gamma \triangleright x \not\approx \perp, \text{Nothing} \leftarrow x$ .

Now, we see that  $\mathcal{E}(\Gamma \triangleright (x \not\approx \perp, \text{Nothing} \leftarrow x), x) = \{ \text{Nothing} \}$ , which is also the result of  $\mathcal{G}(\Gamma, \Delta_4)$ .

The checks for  $\Delta_5$  and  $\Delta_6$  are quite similar, only that we start from  $C(\Gamma \triangleright \emptyset, \Delta_1)$  (which occur syntactically in  $\Delta_5$  and  $\Delta_6$ ) as the initial  $\nabla$ . So, we first compute that.

Fast forward to computing  $\Gamma \triangleright x \not\approx \perp \oplus_{\varphi} x \not\approx \text{Nothing}$ . Ultimately, this entails a proof of  $\Gamma \triangleright x \not\approx \perp, x \not\approx \text{Nothing} \vdash x$ , for which we need to instantiate the `Just` constructor:

$$\begin{array}{c}
x : \text{Maybe Int} \in \Gamma \quad \text{Just} \in \text{Cons}(\Gamma \triangleright (x \not\approx \perp, x \not\approx \text{Nothing}), \text{Maybe Int}) \\
\text{Inst}(\Gamma, x, \text{Just}) = \text{Just } y \leftarrow x \\
(\Gamma, y : \text{Int} \triangleright (x \not\approx \perp, x \not\approx \text{Nothing}) \oplus_{\varphi} \text{Just } y \leftarrow x) \neq \perp \\
\hline
\Gamma \triangleright x \not\approx \perp, x \not\approx \text{Nothing} \vdash x
\end{array}$$



$\Gamma, y : \text{Int} \triangleright (x \neq \perp, x \neq \text{Nothing}) \oplus_{\varphi} \text{Just } y \leftarrow x$  is in fact not  $\perp$ , which is enough to conclude  $\Gamma \triangleright x \neq \perp, x \neq \text{Nothing} \vdash x$ .

The second operand of  $\vee$  in  $\Delta_1$  is similar, but ultimately ends in  $\times$ , so will never produce a  $\nabla$ , so  $C(\Gamma \triangleright \emptyset, \Delta_1) = \Gamma \triangleright x \neq \perp, x \neq \text{Nothing}$ .

$C(\Gamma \triangleright \emptyset, \Delta_5)$  will then just add  $x \approx \perp$  to that  $\nabla$ , which immediately refutes with  $x \neq \perp$ . So no `MayDiverge` around the second RHS.

$C(\Gamma \triangleright \emptyset, \Delta_6)$  is very similar to the situation with  $\Delta_4$ , just with more (non-conflicting) constraints in the incoming  $\nabla$  and with  $\text{Just } y \leftarrow x$  instead of  $\text{Nothing} \leftarrow x$ . Thus,  $\mathcal{G}(\Gamma, \Delta_6) = \{\text{Just } \_ \}$ .

The last bit concerns  $\mathcal{G}(\Gamma, \Delta_2)$ , which is empty because we ultimately would add  $x \neq \text{Just}$  to the inert set  $x \neq \perp, x \neq \text{Nothing}$ , which refutes by the second case of  $\_ \oplus_{\varphi} \_$ . (The  $\vee$  operand with  $\times$  in it is empty, as usual).

So we have  $\mathcal{G}(\Gamma, \Delta_2) = \emptyset$  and the pattern-match is exhaustive.

The result of  $\mathcal{A}_r(\Gamma, t)$  is thus `MayDiverge AccessibleRhs 1; AccessibleRhs 2`.