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

1.1 Desugaring to clause trees

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*:

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

This function will crash for the call site *liftEq* (*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 *Just* 1 against *Nothing*, while the second clause successfully matches 1 with *x*, but then fails trying to match *Nothing* against *Just y*. 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 (*Just* 1) (*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 /= 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:

```
liftEq 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 and fig. 3 the corresponding graphical notation):

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

x, y, z, f, g, h	Term variables
a, b, c	Type variables
K	Data constructors
P	Pattern synonyms
T	Type constructors

Pattern Syntax

$$\begin{array}{rcl}
\mathit{defn} & ::= & \overline{\mathit{clause}} \\
\mathit{clause} & ::= & f\ \overline{\mathit{pat}}\ \overline{\mathit{match}} \\
\mathit{pat} & ::= & x \mid K\ \overline{\mathit{pat}} \\
\mathit{match} & ::= & =\mathit{expr} \mid \overline{\mathit{grhss}} \\
\mathit{grhss} & ::= & \mid \overline{\mathit{guard}} = \mathit{expr} \\
\mathit{guard} & ::= & \mathit{pat} \leftarrow \mathit{expr} \mid \mathit{expr} \mid \texttt{let}\ x = \mathit{expr}
\end{array}$$

Fig. 1. Source syntax

Guard Syntax

$$\begin{array}{rlcl}
K \in & \textsf{Con} & & \\
x, y, a, b \in & \textsf{Var} & & \\
\tau, \sigma \in & \textsf{Type} & & \\
e \in & \textsf{Expr} & ::= & x \\
& & \mid & K\ \overline{\tau}\ \overline{\sigma}\ \overline{\gamma}\ \overline{e} \\
& & \mid & \ldots
\end{array}
\qquad
\begin{array}{rlcl}
n \in & \mathbb{N} & & \\
\gamma \in & \textsf{TyCt} & ::= & \tau_1 \sim \tau_2 \mid \ldots \\
p \in & \textsf{Pat} & ::= & \_ \\
& & \mid & K\ \overline{p} \\
& & \mid & \ldots \\
g \in & \textsf{Grd} & ::= & \textsf{let}\ x : \tau = e \\
& & \mid & K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \\
& & \mid & !x
\end{array}$$

Constraint Formula Syntax

$$\begin{array}{lcll}
\Gamma & ::= & \varnothing \mid \Gamma, x : \tau \mid \Gamma, a & \text{Context} \\
\delta & ::= & \checkmark \mid \times \mid K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mid x \not\approx K \mid x \approx \bot \mid x \not\approx \bot \mid x \approx e & \text{Constraint Literals} \\
\Delta & ::= & \delta \mid \Delta \wedge \Delta \mid \Delta \vee \Delta & \text{Formula} \\
\varphi & ::= & \gamma \mid x \approx K\ \overline{a}\ \overline{y} \mid x \not\approx K \mid x \approx \bot \mid x \not\approx \bot \mid x \approx y & \text{Simple constraints without scoping} \\
\Phi & ::= & \varnothing \mid \Phi, \varphi & \text{Set of simple constraints} \\
\nabla & ::= & \Gamma \triangleright \Phi \mid \times & \text{Inert Set}
\end{array}$$

Clause Tree Syntax

$$\begin{array}{rcl}
t_G, u_G \in \textsf{Gdt} & ::= & \texttt{Rhs}\ n \mid t_G; u_G \mid \texttt{Guard}\ g\ t_G \\
t_A, u_A \in \textsf{Ant} & ::= & \texttt{AccessibleRhs}\ n \mid \texttt{InaccessibleRhs}\ n \mid t_A; u_A \mid \texttt{MayDiverge}\ t_A
\end{array}$$

Fig. 2. IR Syntax

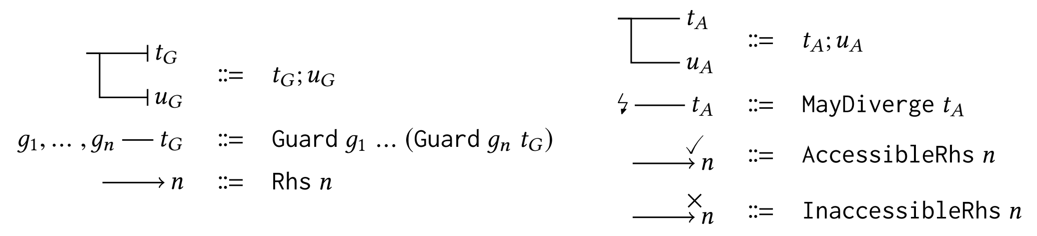
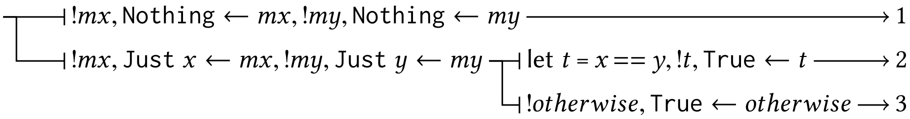


Fig. 3. Graphical notation

SG: TODO: Make the connection between textual syntax and graphic representation. **SG:** The bangs are distracting. Also the otherwise. Also binding the temporary.

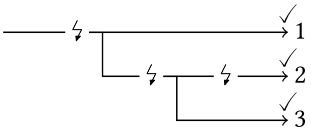

$!mx$, Nothing $\leftarrow mx$, $!my$, Nothing $\leftarrow my$ $\longrightarrow$ 1
$!mx$, Just $x \leftarrow mx$, $!my$, Just $y \leftarrow my$
let $t = x == y$, $!t$, True $\leftarrow t$ $\longrightarrow$ 2
$!otherwise$, True $\leftarrow otherwise$ $\longrightarrow$ 3

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 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.

1.2 Checking guard trees

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 and divergence information. Pattern match checking our guard tree from above should yield an empty uncovered set and an annotated guard tree skeleton like



A GRHS is deemed accessible (✓) whenever there's a non-empty set of values reaching it. For the first GRHS, the set that reaches it looks like $\{(mx, my) \mid mx \not\approx \bot, \texttt{Nothing} \leftarrow mx, my \not\approx \bot, \texttt{Nothing} \leftarrow my\}$, which is inhabited by (Nothing, Nothing). Similarly, we can find inhabitants for the other two clauses.

A ϟ denotes possible divergence in one of the bang patterns and involves testing the set of reaching values for compatibility with i.e. $mx \approx \bot$. We don't know for mx, my and t (hence insert a ϟ), but can certainly rule out $otherwise \approx \bot$ simply by knowing that it is defined as *True*. But since all GRHSs are accessible, there's nothing to report in terms of redundancy and the ϟ decorators are irrelevant.

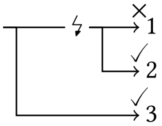
Perhaps surprisingly and most importantly, Grd with its three primitive guards, combined with left-to-right or top-to-bottom semantics in Gdt, is expressive enough to express all pattern matching in Haskell (cf. fig. TODO)! We have yet to find a language extension that doesn't fit into this framework.

1.2.1 Why do we not report redundant GRHSs directly? Why not compute the redundant GRHSs directly instead of building up a whole new tree? Because determining inaccessibility vs. redundancy is a non-local problem. Consider this example: **SG:** I think this kind of detail should be motivated in a prior section and then referenced here for its solution.

$$
\begin{array}{llll}
g :: () \rightarrow \mathit{Int} & & & \\
g\ () & \mid \mathit{False} & = 1 & \\
 & \mid \mathit{True} & = 2 & \\
g\ \_ & & = 3 &
\end{array}
$$

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:



In general, at least one GRHS under a ϟ may not be flagged as redundant. Thus the checking algorithm can't decide which GRHSs are redundant (vs. just inaccessible) when it reaches a particular GRHS.

1.3 Testing for emptiness

The informal style of pattern match checking above represents the set of values reaching a particular node of the guard tree as a *refinement type*. Each guard encountered in the tree traversal refines this set with its own constraints.

Apart from generating inhabitants of the final uncovered set for missing equation warnings, there are two points at which we have to check whether such a refinement type has become empty: To determine whether a right-hand side is inaccessible and whether a particular bang pattern may lead to divergence and requires us to wrap a ϟ.

Take the constraints of the final uncovered set after checking *liftEq* above as an example: **SG:** This doesn't even pick up the trivially empty clauses ending in ×, but is already too complex. Also this is just a Δ, not a full refinement type.

$$
\begin{array}{ll}
 & (mx \not\approx \bot \wedge (mx \not\approx \texttt{Nothing} \vee (\texttt{Nothing} \leftarrow mx \wedge my \not\approx \bot \wedge my \not\approx \texttt{Nothing}))) \\
\wedge & (mx \not\approx \bot \wedge (mx \not\approx \texttt{Just} \vee (\texttt{Just}\ x \leftarrow mx \wedge my \not\approx \bot \wedge (my \not\approx \texttt{Just}))))
\end{array}
$$

A bit of eyeballing *liftEq*'s definition finds *Nothing* (*Just* _) as an uncovered pattern, but eyeballing the constraint formula above seems impossible in comparison. A more systematic approach is to adopt a generate-and-test scheme: Enumerate possible values of the data types for each variable involved (the pattern variables *mx* and *my*, but also possibly the guard-bound *x*, *y* and *t*) and test them for compatibility with the recorded constraints.

Starting from *mx my*, we enumerate all possibilities for the shape of *mx*, and similarly for *my*. The obvious first candidate in a lazy language is ⊥! But that is a contradicting assignment for both *mx* and *my* indepedently. Refining to *Nothing Nothing* contradicts with the left part of the top-level ∧. Trying *Just y* (*y* fresh) instead as the shape for *my* yields our first inhabitant! Note that *y* is unconstrained, so ⊥ is a trivial inhabitant. Similarly for (*Just* _) *Nothing* and (*Just* _) (*Just* _).

Why do we have to test guard-bound variables in addition to the pattern variables? It's because of empty data types and strict fields: **SG:** This definition will probably move to an earlier section

Checking Guard Trees

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

$$\begin{array}{lcl}
\mathcal{U}(\texttt{Rhs}\ n) & = & \times \\
\mathcal{U}(t; u) & = & \mathcal{U}(t) \wedge \mathcal{U}(u) \\
\mathcal{U}(\texttt{Guard}\ (!x)\ t) & = & (x \not\approx \bot) \wedge \mathcal{U}(t) \\
\mathcal{U}(\texttt{Guard}\ (\mathsf{let}\ x = e)\ t) & = & (x \approx e) \wedge \mathcal{U}(t) \\
\mathcal{U}(\texttt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x)\ t) & = & (x \not\approx K) \vee ((K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x) \wedge \mathcal{U}(t))
\end{array}$$

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

$$\begin{array}{lcl}
\mathcal{A}_\Gamma(\Delta, \texttt{Rhs}\ n) & = & \begin{cases} \texttt{InaccessibleRhs}\ n, & \mathcal{G}(\Gamma, \Delta) = \emptyset \\ \texttt{AccessibleRhs}\ n, & \text{otherwise} \end{cases} \\
\mathcal{A}_\Gamma(\Delta, (t; u)) & = & \mathcal{A}_\Gamma(\Delta, t); \mathcal{A}_\Gamma(\Delta \wedge \mathcal{U}(t), u) \\
\mathcal{A}_\Gamma(\Delta, \texttt{Guard}\ (!x)\ t) & = & \begin{cases} \mathcal{A}_\Gamma(\Delta \wedge (x \not\approx \bot), t), & \mathcal{G}(\Gamma, \Delta \wedge (x \approx \bot)) = \emptyset \\ \texttt{MayDiverge}\ \mathcal{A}_\Gamma(\Delta \wedge (x \not\approx \bot), t) & \text{otherwise} \end{cases} \\
\mathcal{A}_\Gamma(\Delta, \texttt{Guard}\ (\mathsf{let}\ x = e)\ t) & = & \mathcal{A}_\Gamma(\Delta \wedge (x \approx e), t) \\
\mathcal{A}_\Gamma(\Delta, \texttt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x)\ t) & = & \mathcal{A}_\Gamma(\Delta \wedge (K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x), t)
\end{array}$$

Putting it all together

(0) Input: Context with match vars Γ 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. 4. Pattern-match checking

```
data Void   -- No data constructors
data SMaybe a = SJust ! a | SNothing
f :: SMaybe Void → Int
f x@SNothing = 0
```

f does not have any uncovered patterns. And our approach better should see that by looking at the constraints of its uncovered set:

$$x \not\approx \bot \wedge x \not\approx \texttt{Nothing}$$

Specifically, the candidate *SJust* y (for fresh y) for x should be rejected, because there is no inhabitant for y! $\bot$ is ruled out by the strict field and *Void* means there is no data constructor to instantiate. Hence it is important to test guard-bound variables for inhabitants, too.

SG: GMTM goes into detail about type constraints, term constraints and worst-case complexity here. That feels a bit out of place.

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 Δs) work.

Generate inhabitants of Δ

$$\boxed{\mathcal{G}(\Gamma, \Delta) = \mathcal{P}(\overline{p})}$$

$$\mathcal{G}(\Gamma, \Delta) = \bigcup \{\mathcal{E}(\nabla, \mathsf{dom}(\Gamma)) \mid \nabla \in \mathcal{C}(\Gamma \triangleright \varnothing, \Delta)\}$$

Construct inhabited ∇s from Δ

$$\boxed{\mathcal{C}(\nabla, \Delta) = \mathcal{P}(\nabla)}$$

$$\begin{array}{lcl}
\mathcal{C}(\nabla, \delta) & = & \begin{cases} \{\Gamma' \triangleright \Phi'\} & \text{where } \Gamma' \triangleright \Phi' = \nabla \oplus_\delta \delta \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{C}(\nabla, \Delta_1 \wedge \Delta_2) & = & \bigcup \{\mathcal{C}(\nabla', \Delta_2) \mid \nabla' \in \mathcal{C}(\nabla, \Delta_1)\} \\
\mathcal{C}(\nabla, \Delta_1 \vee \Delta_2) & = & \mathcal{C}(\nabla, \Delta_1) \cup \mathcal{C}(\nabla, \Delta_2)
\end{array}$$

Expand variables to Pat with ∇

$$\boxed{\mathcal{E}(\nabla, \overline{x}) = \mathcal{P}(\overline{p})}$$

$$\begin{array}{lcl}
\mathcal{E}(\nabla, \epsilon) & = & \{\epsilon\} \\
\mathcal{E}(\Gamma \triangleright \Phi, x_1 ... x_n) & = & \begin{cases} \{(K\ q_1 ... q_m)\, p_2 ... p_n \mid (q_1 ... q_m\, p_2 ... p_n) \in \mathcal{E}(\Gamma \triangleright \Phi, y_1 ... y_m x_2 ... x_n)\} & \text{if } \Phi(x) \approx K\ \overline{a}\ \overline{y} \in \\ \{\_\ p_2 ... p_n \mid (p_2 ... p_n) \in \mathcal{E}(\Gamma \triangleright \Phi, x_2 ... x_n)\} & \text{otherwise} \end{cases}
\end{array}$$

Finding the representative of a variable in Φ

$$\boxed{\Phi(x) = y}$$

$$\Phi(x) \quad = \quad \begin{cases} \Phi(y) & x \approx y \in \Phi \\ x & \text{otherwise} \end{cases}$$

Fig. 5. Bridging between the facade Δ and ∇

$$\begin{array}{lll}
f :: \mathit{Maybe}\ \mathit{Int} \longrightarrow \mathit{Int} & & \\
f\ \mathit{Nothing} & = 0 & \text{-- RHS 1} \\
f\ x \mid \mathit{Just}\ y \longleftarrow x = y & & \text{-- RHS 2}
\end{array}$$

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:

$$\begin{array}{c}
\mathtt{Guard}\ (!x)\ \mathtt{Guard}\ (\mathtt{Nothing} \longleftarrow x)\ \mathtt{Rhs}\ 1; \\
\mathtt{Guard}\ (!x)\ \mathtt{Guard}\ (\mathtt{Just}\ y \longleftarrow x)\ \mathtt{Rhs}\ 2
\end{array}$$

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 Δs, after the first and the second clause respectively.

(1)

$$\begin{array}{lcl}
\Delta_1 & := & \mathcal{U}(\mathtt{Guard}\ (!x)\ \mathtt{Guard}\ (\mathtt{Nothing} \longleftarrow x)\ \mathtt{Rhs}\ 1) \\
 & = & x \not\approx \bot \wedge (x \not\approx \mathtt{Nothing} \vee \times)
\end{array}$$

Add a constraint to the inert set

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

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

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 \mathrm{dom}(\Gamma) : \Gamma \triangleright (\Phi, \gamma) \vdash \Phi(x) \\ \times & \text{otherwise} \end{cases} \\
&\Gamma \triangleright \Phi \oplus_\varphi x \approx K\ \overline{a}\ \overline{y} &&= \begin{cases} \Gamma \triangleright \Phi \oplus_\varphi \overline{a \sim b} \oplus_\varphi \overline{y \approx z} & \text{if } \Phi(x) \approx K\ \overline{b}\ \overline{z} \in \Phi \\ \Gamma' \triangleright (\Phi', \Phi(x) \approx K\ \overline{a}\ \overline{y}) & \text{where } \Gamma' \triangleright \Phi' = \Gamma \triangleright \Phi \oplus_\varphi \overline{\gamma} \\ & \text{and } \Phi'(x) \not\approx K \notin \Phi' \text{ and } \overline{\Gamma' \triangleright \Phi' \vdash y} \\ \times & \text{otherwise} \end{cases} \\
&\Gamma \triangleright \Phi \oplus_\varphi x \not\approx K &&= \begin{cases} \times & \text{if } \Phi(x) \approx K\ \overline{a}\ \overline{y} \in \Phi \\ \times & \text{if not } \Gamma \triangleright (\Phi, \Phi(x) \not\approx K) \vdash \Phi(x) \\ \Gamma \triangleright (\Phi, \Phi(x) \not\approx K) & \text{otherwise} \end{cases} \\
&\Gamma \triangleright \Phi \oplus_\varphi x \approx \bot &&= \begin{cases} \bot & \text{if } \Phi(x) \not\approx \bot \in \Phi \\ \Gamma \triangleright (\Phi, \Phi(x) \approx \bot) & \text{otherwise} \end{cases} \\
&\Gamma \triangleright \Phi \oplus_\varphi x \not\approx \bot &&= \begin{cases} \times & \text{if } \Phi(x) \approx \bot \in \Phi \\ \times & \text{if not } \Gamma \triangleright (\Phi, \Phi(x) \not\approx \bot) \vdash \Phi(x) \\ \Gamma \triangleright (\Phi, \Phi(x) \not\approx \bot) & \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}$$

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

$$\begin{aligned}
&\varnothing \cap x &&= \varnothing \\
&(\Phi, x \approx K\ \overline{a}\ \overline{y}) \cap x &&= (\Phi \cap x), x \approx K\ \overline{a}\ \overline{y} \\
&(\Phi, x \not\approx K) \cap x &&= (\Phi \cap x), x \not\approx K \\
&(\Phi, x \approx \bot) \cap x &&= (\Phi \cap x), x \approx \bot \\
&(\Phi, x \not\approx \bot) \cap x &&= (\Phi \cap x), x \not\approx \bot \\
&(\Phi, \varphi) \cap x &&= \Phi \cap x
\end{aligned}$$

Fig. 6. Adding a constraint to the inert set ∇

(2)

$$\Delta_2 \quad := \quad \mathcal{U}(t_{\mathfrak{f}}) = \Delta_1 \wedge x \not\approx \bot \wedge (x \not\approx \texttt{Just} \vee \times)$$

Test if x is inhabited considering ∇

$$\boxed{\nabla \vdash x}$$

$$\frac{(\Gamma \triangleright \Phi \oplus_\varphi x \approx \bot) \neq \times}{\Gamma \triangleright \Phi \vdash x} \qquad \frac{\begin{array}{c} x : \tau \in \Gamma \quad K \in \mathsf{Cons}(\Gamma \triangleright \Phi, \tau) \\ \mathsf{Inst}(\Gamma, x, K) = \overline{\varphi} \\ (\Gamma, \overline{y : \tau'} \triangleright \Phi \oplus_\varphi \overline{\varphi}) \neq \times \end{array}}{\Gamma \triangleright \Phi \vdash x}$$

$$\frac{x : \tau \in \Gamma \quad \mathsf{Cons}(\Gamma \triangleright \Phi, \tau) = \bot}{\Gamma \triangleright \Phi \vdash x} \qquad \frac{\begin{array}{c} x : \tau \in \Gamma \quad K \in \mathsf{Cons}(\Gamma \triangleright \Phi, \tau) \\ \mathsf{Inst}(\Gamma, x, K) = \bot \end{array}}{\Gamma \triangleright \Phi \vdash x}$$

Find data constructors of τ

$$\boxed{\mathsf{Cons}(\Gamma \triangleright \Phi, \tau) = \overline{K}}$$

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

Instantiate x to data constructor K

$$\boxed{\mathsf{Inst}(\Gamma, x, K) = \overline{\varphi}}$$

$$\mathsf{Inst}(\Gamma, x, K) = \begin{cases} \tau_x \sim \tau, \overline{\gamma}, x \approx K\ \overline{a}\ \overline{y}, \overline{y' \not\approx \bot} & K : \forall \overline{a}. \overline{\gamma} \Rightarrow \overline{\sigma} \rightarrow \tau, \overline{y} \# \Gamma, \overline{a} \# \Gamma, x : \tau_x \in \Gamma, \overline{y'} \text{ bind strict fields} \\ \bot & \text{otherwise} \end{cases}$$

Fig. 7. Inhabitance test

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

We can see that Δ_2 is in fact uninhabited, because the three constraints $x \not\approx \bot$, $x \not\approx \texttt{Nothing}$ and $x \not\approx \texttt{Just}$ cover all possible data constructors of the `Maybe` data type. And indeed $\mathcal{G}(x : \texttt{Maybe Int}, \Delta_2) = \emptyset$, as we'll see later.

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

Except for Δ_5, these are all inhabited, i.e. $\mathcal{G}(x : \texttt{Maybe Int}, \Delta_i) \neq \emptyset$ (as we'll see in the next section).

Thus, we will get the following annotated tree:

$$\texttt{MayDiverge}\ \texttt{AccessibleRhs}\ 1; \texttt{AccessibleRhs}\ 2$$

2.3 Generating inhabitants

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

$\mathcal{G}(,)$ proceeds in two steps: First it constructs zero, one or many *inert sets* ∇ with $\mathcal{C}(,)$ (each of them representing a set of mutually compatible constraints) and then expands each of the returned inert sets into one or more pattern vectors $\overline{p}$ with $\mathcal{E}(,)$, which is the preferred representation to show to the user.

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

Let's see how that works for Δ_3 above. Recall that $\Gamma = x : \texttt{Maybe Int}$ and $\Delta_3 = \checkmark \wedge x \approx \bot$:

$$
\begin{array}{ll}
 & \mathcal{C}(\Gamma, \checkmark \wedge x \approx \bot) \\
= & \quad \{ \text{ Conjunction } \} \\
 & \bigcup \{ \mathcal{C}(\Gamma' \triangleright \nabla', x \approx \bot) \mid \Gamma' \triangleright \nabla' \in \mathcal{C}(\Gamma \triangleright \varnothing, \checkmark) \} \\
= & \quad \{ \text{ Single constraint } \} \\
 & \begin{cases} \mathcal{C}(\Gamma' \triangleright \nabla', x \approx \bot) & \text{where } \Gamma' \triangleright \nabla' = \Gamma \triangleright \varnothing \oplus_\varphi \checkmark \\ \emptyset & \text{otherwise} \end{cases} \\
= & \quad \{ \checkmark \text{ case of } \oplus_\varphi \} \\
 & \mathcal{C}(\Gamma \triangleright \varnothing, x \approx \bot) \\
= & \quad \{ \text{ Single constraint } \} \\
 & \begin{cases} \{\Gamma' \triangleright \nabla'\} & \text{where } \Gamma' \triangleright \nabla' = \Gamma \triangleright \varnothing \oplus_\varphi x \approx \bot \\ \emptyset & \text{otherwise} \end{cases} \\
= & \quad \{ x \approx \bot \text{ case of } \oplus_\varphi \} \\
 & \{ \Gamma \triangleright x \approx \bot \}
\end{array}
$$

Let's start with $\mathcal{G}(\Gamma, \Delta_3)$, where $\Gamma = x : \texttt{Maybe Int}$ and recall that $\Delta_3 = \checkmark \wedge x \approx \bot$. The first constraint $\checkmark$ is added very easily to the initial nabla by discarding it, the second one ($x \approx \bot$) is not conflicting with any $x \not\approx \bot$ constraint in the incoming, still empty ($\varnothing$) nabla, so we end up with $\Gamma \triangleright x \approx \bot$ as proof that Δ_3 is in fact inhabited. Indeed, $\mathcal{E}(\Gamma \triangleright x \approx \bot, x)$ 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 $\mathcal{C}(\Gamma \triangleright \varnothing, \Delta_4)$, which in turn will add $x \not\approx \bot$ to an initially empty ∇. That entails an inhabitance check to see if x might take on any values besides $\bot$.

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

$$
\frac{\begin{array}{c} x : \texttt{Maybe Int} \in \Gamma \quad \texttt{Nothing} \in \mathsf{Cons}(\Gamma \triangleright x \not\approx \bot, \texttt{Maybe Int}) \\ \mathsf{Inst}(\Gamma, x, \texttt{Nothing}) = \texttt{Nothing} \longleftarrow x \\ (\Gamma \triangleright x \not\approx \bot \oplus_\varphi \texttt{Nothing} \longleftarrow x) \neq \bot \end{array}}{\Gamma \triangleright x \not\approx \bot \vdash x}
$$

The subgoal $\Gamma \triangleright x \not\approx \bot \oplus_\varphi \texttt{Nothing} \longleftarrow x$ 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 ∇. Since there are no type constraints carried by `Nothing`, no fields and no constraints of the form $x \not\approx K$ in ∇, we end up with $\Gamma \triangleright x \not\approx \bot, \texttt{Nothing} \longleftarrow x$. Which is not $\bot$, thus we conclude our proof of $\Gamma \triangleright x \not\approx \bot \vdash x$.

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

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

The checks for Δ_5 and Δ_6 are quite similar, only that we start from $\mathcal{C}(\Gamma \triangleright \varnothing, \Delta_1)$ (which occur syntactically in Δ_5 and Δ_6) as the initial ∇. So, we first compute that.

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

$$\frac{\begin{array}{c} x : \texttt{Maybe Int} \in \Gamma \quad \texttt{Just} \in \mathsf{Cons}(\Gamma \triangleright (x \not\approx \bot, x \not\approx \mathsf{Nothing}), \texttt{Maybe Int}) \\ \mathsf{Inst}(\Gamma, x, \mathsf{Just}) = \texttt{Just}\ y \longleftarrow x \\ (\Gamma, y : \texttt{Int} \triangleright (x \not\approx \bot, x \not\approx \mathsf{Nothing}) \oplus_\varphi \mathsf{Just}\ y \longleftarrow x) \neq \bot \end{array}}{\Gamma \triangleright x \not\approx \bot, x \not\approx \mathsf{Nothing} \vdash x}$$

$\Gamma, y : \texttt{Int} \triangleright (x \not\approx \bot, x \not\approx \texttt{Nothing}) \oplus_\varphi \texttt{Just}\ y \longleftarrow x)$ is in fact not $\bot$, which is enough to conclude $\Gamma \triangleright x \not\approx \bot, x \not\approx \texttt{Nothing} \vdash x$.

The second operand of $\vee$ in Δ_1 is similar, but ultimately ends in $\times$, so will never produce a ∇, so $\mathcal{C}(\Gamma \triangleright \varnothing, \Delta_1) = \Gamma \triangleright x \not\approx \bot, x \not\approx \texttt{Nothing}$.

$\mathcal{C}(\Gamma \triangleright \varnothing, \Delta_5)$ will then just add $x \approx \bot$ to that ∇, which immediately refutes with $x \not\approx \bot$. So no $\texttt{MayDiverge}$ around the second RHS.

$\mathcal{C}(\Gamma \triangleright \varnothing, \Delta_6)$ is very similar to the situation with Δ_4, just with more (non-conflicting) constraints in the incoming ∇ and with $\texttt{Just}\ y \longleftarrow x$ instead of $\texttt{Nothing} \longleftarrow x$. Thus, $\mathcal{G}(\Gamma, \Delta_6) = \{\texttt{Just \_}\}$.

The last bit concerns $\mathcal{G}(\Gamma, \Delta_2)$, which is empty because we ultimately would add $x \not\approx \texttt{Just}$ to the inert set $x \not\approx \bot, x \not\approx \texttt{Nothing}$, 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 $\mathcal{A}_\Gamma(\Gamma, t)$ is thus $\texttt{MayDiverge AccessibleRhs 1}; \texttt{AccessibleRhs 2}$.