# 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
RYAN G. SCOTT, Indiana University, USA

## 1 INTRODUCTION

Pattern matching is a tremendously useful feature in Haskell and many other programming languages, but it must be wielded with care. Consider the following example of pattern matching gone wrong:

$f :: Int \longrightarrow Bool$
$f\ 0 = True$
$f\ 0 = False$

The $f$ function exhibits two serious flaws. One obvious issue is that there are two clauses that match on 0, and due to the top-to-bottom semantics of pattern matching, this makes the $f\ 0 = False$ clause completely unreachable. Even worse is that $f$ never matches on any patterns besides 0, making it not fully defined. Attempting to invoke $f\ 1$, for instance, will fail.

To avoid these mishaps, compilers for languages with pattern matching often emit warnings whenever a programmer misuses patterns. Such warnings indicate if a function is missing clauses (i.e., if it is *non-exhaustive*) or if a function has overlapping clauses (i.e., if it is *redundant*). We refer to the combination of checking for exhaustivity and redundancy as *pattern-match coverage checking*. Coverage checking is the first line of defence in catching programmer mistakes when defining code that uses pattern matching.

If coverage checking catches mistakes in pattern matches, then who checks for mistakes in the coverage checker itself? It is a surprisingly frequent occurrence for coverage checkers to contain bugs that impact correctness. This is especially true in Haskell, which has an especially rich pattern language, and the Glasgow Haskell Compiler (GHC) complicates the story further by adding pattern-related language extensions. Designing a coverage checker that can cope with all of these features is no small task.

The current state of the art for coverage checking GHC is Karachalias et al. [2015], which presents an algorithm that handles the intricacies of checking GADTs, lazy patterns, and pattern guards. We argue that this algorithm is insufficient in a number of key ways. It does not account for a number of important language features and even gives incorrect results in certain cases. Moreover, the implementation of this algorithm in GHC is inefficient and has proved to be difficult to maintain due to its complexity.

In this paper we propose a new algorithm, called **SYSNAME**, that addresses the deficiencies of Karachalias et al. [2015]. The key insight of **SYSNAME** is to condense all of the complexities of pattern matching into just three constructs: **let** bindings, pattern guards, and bang guards. We make the following contributions: **Ryan:** Cite section numbers in the list below.

- We characterise the nuances of coverage checking that not even the algorithm in [Karachalias et al. 2015] handles. We also identify issues in GHC's implementation of this algorithm.

Authors' addresses: Sebastian Graf, Karlsruhe Institute of Technology, Karlsruhe, Germany, sebastian.graf@kit.edu; Simon Peyton Jones, Microsoft Research, Cambridge, UK, simonpj@microsoft.com; Ryan G. Scott, Indiana University, Bloomington, Indiana, USA, rgscott@indiana.edu.

**Fig. 1.** Definitions used in the text

- **Ryan:** Describe the "Overview Over Our Solution" section and "Formalism" sections.
- We have implemented **SYSNAME** in GHC. **Ryan:** More details.

We discuss the wealth of related work in **TODO:** .

## 2 THE PROBLEM WE WANT TO SOLVE

What makes coverage checking so difficult in a language like Haskell? At first glance, implementing a coverage checker algorithm might appear simple: just check that every function matches on every possible combination of data constructors exactly once. A function must match on every possible combination of constructors in order to be exhaustive, and it must must on them exactly once to avoid redundant matches.

This algorithm, while concise, leaves out many nuances. What constitutes a "match"? Haskell has multiple matching constructs, including function definitions, **case** expressions, and guards. How does one count the number of possible combinations of data constructors? This is not a simple exercise since term and type constraints can make some combinations of constructors unreachable if matched on. Moreover, what constitutes a "data constructor"? In addition to traditional data constructors, GHC features *pattern synonyms* [Pickering et al. 2016], which provide an abstract way to embed arbitrary computation into patterns. Matching on a pattern synonym is syntactically identical to matching on a data constructor, which makes coverage checking in the presence of pattern synonyms challenging.

Prior work on coverage checking (which we will expound upon further in **Ryan:** Cite related work section) accounts for some of these nuances, but not all of them. In this section we identify all of the language features that complicate coverage checking. While these features may seem disparate at first, we will later show in **Ryan:** Cite the relevant section that these ideas can all fit into a unified framework.

### 2.1 Guards

**TODO:**

### 2.2 Strictness

The evaluation order of pattern matching can impact whether a pattern is reachable or not. While Haskell is a lazy language, programmers can opt into extra strict evaluation by giving the fields of a data type strict fields, such as in this example: **Ryan:** Consider moving some of this code to the figure **Ryan:** There is an erroneous space between the ! and the *a*

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

v :: SMaybe Void ⟶ Int
v SNothing = 0
```

The *SJust* constructor is strict in its field, and as a consequence, evaluating *SJust* $\perp$ to weak-head normal form will diverge. This has consequences when coverage checking functions that match on *SMaybe* values, such as *v*. The definition of *v* is curious, since it appears to omit a case for *SJust*. We could imagine adding one:

```
v (SJust _) = 1
```

**Meta variables**

| | |
|---|---|
| $x, y, z, f, g, h$ | Term variables |
| $a, b, c$ | Type variables |
| $K$ | Data constructors |
| $P$ | Pattern synonyms |
| $T$ | Type constructors |
| $l$ | Literal |
| $expr$ | Expressions |

**Pattern Syntax**

$$\begin{array}{rcl}
defn & ::= & \overline{clause} \\
clause & ::= & f\ \overline{pat}\ \overline{match} \\
pat & ::= & x \mid \_ \mid K\ \overline{pat} \mid x@pat \mid !pat \mid {\sim}pat \mid x + l \\
match & ::= & = expr \mid \overline{grhs} \\
grhs & ::= & \mid \overline{guard} = expr \\
guard & ::= & pat \leftarrow expr \mid expr \mid \texttt{let}\ x = expr
\end{array}$$

**Fig. 2.** Source syntax

It turns out, however, that the RHS of this case can never be reached. The only way to use *SJust* to construct a value of type *SMaybe Void* is *SJust* $\bot$, since *Void* has no data constructors. Because *SJust* is strict in its field, matching on *SJust* will cause *SJust* $\bot$ to diverge, since matching on a data constructor evaluates it to WHNF. As a result, there is no argument one could pass to $v$ to make it return 1, which makes the *SJust* case unreachable.

Although Karachalias et al. [2015] incorporates strictness constraints into their algorithm, it does not consider constraints that arise from strict fields. **Ryan:** Say more here?

*2.2.1 Bang patterns.* Strict fields are the primary mechanism for adding extra strictness in Haskell, but GHC adds another mechanism in the form of *bang patterns*. A bang pattern such as $!pat$ indicates that matching against *pat always* evaluates it to WHNF. While data constructor matches are normally the only patterns that match strictly, bang patterns extend this treatment to other patterns. For example, one can rewrite the earlier $v$ example to use the ordinary, lazy *Maybe* data type: **Ryan:** I actually wanted to write Just !_, but LaTeX won't parse that :(

$$\begin{array}{l}
v' :: Maybe\ Void \longrightarrow Int \\
v'\ Nothing = 0 \\
v'\ (Just\ !\ x) = 1
\end{array}$$

**Ryan:** Finish me

#### *2.2.2 Newtypes.* TODO:

### 2.3 Programmable patterns

TODO:

#### *2.3.1 Overload literals.* TODO:

#### *2.3.2 Pattern synonyms.* TODO:

### 2.4 Term and type constraints

TODO:

#### *2.4.1 Long-distance information.* TODO:

## 3 OVERVIEW OVER OUR SOLUTION

In this section, we aim to provide an intuitive understanding of our pattern match checking algorithm, by way of deriving the intermediate representations of the pipeline step by step from motivating examples.

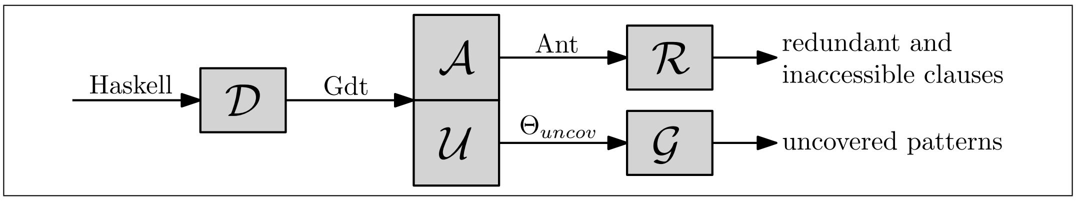


**Fig. 3.** Bird's eye view of pattern match checking

**Guard Syntax**

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

**Constraint Formula Syntax**

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

**Clause Tree Syntax**

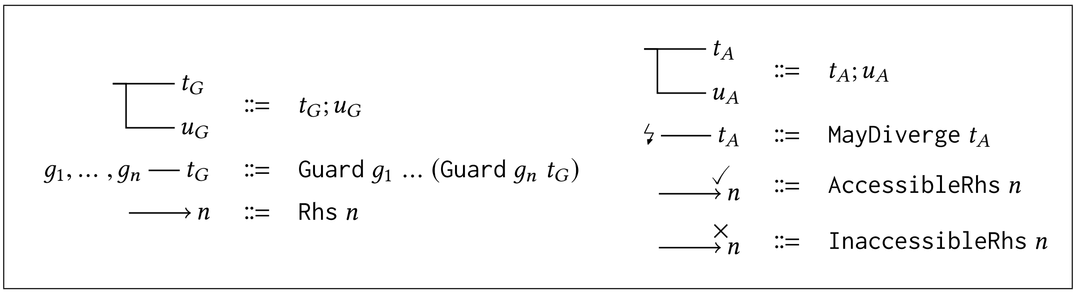
$$
\begin{array}{rcl}
t_G, u_G \in \text{Gdt} & ::= & \texttt{Rhs}\ n \mid t_G; u_G \mid \texttt{Guard}\ g\ t_G \\
t_A, u_A \in \text{Ant} & ::= & \texttt{AccessibleRhs}\ n \mid \texttt{InaccessibleRhs}\ n \mid t_A; u_A \mid \texttt{MayDiverge}\ t_A
\end{array}
$$

**Fig. 4.** IR Syntax

Figure 3 depicts a high-level overview over this pipeline. Desugaring the complex source Haskell syntax to the very elementary language of guard trees Gdt via $\mathcal{D}$ is an incredible simplification for the checking process. At the same time, $\mathcal{D}$ is the only transformation that is specific to Haskell, implying easy applicability to other languages. The resulting guard tree is then processed by two different functions, $\mathcal{A}$ and $\mathcal{U}$, which compute redundancy information and uncovered patterns, respectively. $\mathcal{A}$ boils down this information into an annotated tree Ant, for which the set of redundant and inaccessible right-hand sides can be computed in a final pass of $\mathcal{R}$ **SG:** TODO: Write $\mathcal{R}$. $\mathcal{U}$ on the other hand returns a *refinment type* representing the set of *uncovered values*, for which $\mathcal{G}$ can generate the inhabiting patterns to show to the user.

## 3.1 Desugaring to Guard Trees

It is customary to define Haskell functions using pattern-matching, possibly with one or more *guarded right-hand sides* (GRHS) per syntactic *clause* (see fig. 2). Consider for example this 3am attempt at lifting equality over *Maybe*:



$t_G$
$u_G$
::= $t_G; u_G$
$g_1, \ldots, g_n$ — $t_G$ ::= Guard $g_1$ … (Guard $g_n$ $t_G$)
⟶ $n$ ::= Rhs $n$
$t_A$
$u_A$
::= $t_A; u_A$
↯ — $t_A$ ::= MayDiverge $t_A$
$\xrightarrow{\checkmark} n$ ::= AccessibleRhs $n$
$\xrightarrow{\times} n$ ::= InaccessibleRhs $n$

**Fig. 5.** Graphical notation

```
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 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 the *uncovered* tuple of values (*Just* 1) *Nothing* 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 (GRHSs), each 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!) that will fail because $1 /= 2$. So the second GRHS is tried successfully, because *otherwise* is a boolean guard that never fails.

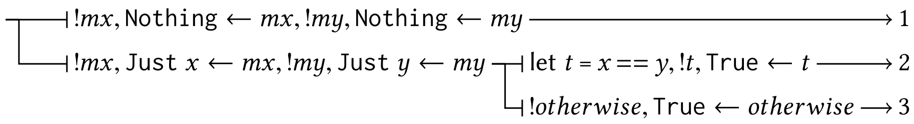
Note how both the pattern matching per clause and the guard checking within a syntactic *match* share top-to-bottom and left-to-right semantics. Having to make sense of both pattern and guard semantics seems like a waste of energy. Perhpas we can express all pattern matching by (nested) pattern guards, thus:

```
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 is easy. But the second clause already had two GRHSs, so we need to use *nested* pattern guards. This is not a feature that Haskell offers (yet), but it allows a very convenient uniformity for our purposes: 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).

Hence our algorithm desugars the source syntax to the following *guard tree* (see fig. 4 for the full syntax and fig. 5 the corresponding 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.

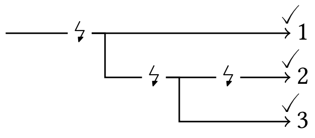

$!mx, \texttt{Nothing} \leftarrow mx, !my, \texttt{Nothing} \leftarrow my$
1
$!mx, \texttt{Just}\ x \leftarrow mx, !my, \texttt{Just}\ y \leftarrow my$
$\texttt{let}\ t = x == y, !t, \texttt{True} \leftarrow t$
2
$!otherwise, \texttt{True} \leftarrow otherwise$
3

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 guards*. In analogy to bang patterns, $!x$ evaluates $x$ to WHNF, which, as a guard, always succeeds or diverges. 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.

## 3.2 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 $(\texttt{Nothing}, \texttt{Nothing})$. Similarly, we can find inhabitants for the other two clauses.

A ↯ denotes possible divergence in one of the bang guards 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 ↯), 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 ↯ decorators are irrelevant.

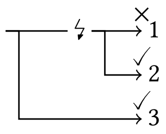
Perhaps surprisingly and most importantly, Grd with its three primitive guards, combined with left-to-right or top-to-bottom semantics in Gdt, is expressive enough to express all pattern matching in Haskell (cf. fig. 6)! We have yet to find a language extension that doesn't fit into this framework.

*3.2.1 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.

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



In general, at least one GRHS under a $\lightning$ 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.

## 3.3 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* (which is the $\Theta$ from fig. 4). 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 guard may lead to divergence and requires us to wrap a $\lightning$.

Take the the final uncovered set $\{\, (mx : Maybe\ a, my : Maybe\ a) \mid \Phi \,\}$ after checking *liftEq* above as an example, where the predicate $\Phi$ is: **SG:** This doesn't even pick up the trivially empty clauses ending in $\times$, but is already qutie complex.

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

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 $\bot$! 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 $\wedge$. Trying *Just* $y$ ($y$ fresh) instead as the shape for *my* yields our first inhabitant! Note that $y$ is unconstrained, so $\bot$ 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 example will probably move to an earlier section

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

$v$ does not have any uncovered patterns. And our approach better should see that by looking at its uncovered set $\{\, x : \mathit{Maybe\ Void} \mid x \not\approx \bot \wedge x \not\approx \texttt{Nothing} \,\}$.

Specifically, the candidate $\mathit{SJust}\ \ y$ (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.

## 4 FORMALISM

The previous section gave insights into how we represent pattern match checking problems as clause trees and provided an intuition for how to check them for exhaustiveness and redundancy. This section formalises these intuitions in terms of the syntax (cf. fig. 4) we introduced earlier.

As in the previous section, this comes in two main parts: Pattern match checking and finding inhabitants of the arising refinement types.

### 4.1 Desugaring to Guard Trees

**SG:** I find this section quite boring. There's nothing to see in fig. 6 that wasn't already clear after reading 3.1.

Figure 6 outlines the desugaring step from source Haskell to our guard tree language Gdt. It is assumed that the top-level match variables $x_1$ through $x_n$ in the *clause* cases have special, fixed names. **SG:** If we had a different font for meta variables than for object variables, we could make that visible in syntax. But we don't... All other variables that aren't bound in arguments to $\mathcal{D}$ have fresh names.

Consider this example function **SG:** Maybe use the same function as in 3.1? But we already desugar it there...:

$$
\begin{array}{llll}
f\ (\mathit{Just}\ (!xs, \_)) & ys@\mathit{Nothing} & = & 1 \\
f\ \mathit{Nothing} & zs & = & 2
\end{array}
$$

Under $\mathcal{D}$, this desugars to

$$
\begin{array}{l}
\dashv\, !x_1, \mathit{Just}\ t_1 \leftarrow x_1, !t_1, (t_2, t_3) \leftarrow t_1, !t_2, \mathsf{let}\ xs = t_2, \mathsf{let}\ ys = x_2, !ys, \mathit{Nothing} \leftarrow ys \longrightarrow 1 \\
\dashv\, !x_1, \mathit{Nothing} \leftarrow x_1, \mathsf{let}\ zs = x_2 \longrightarrow 2
\end{array}
$$

The definition of $\mathcal{D}$ is mostly straight-forward, but a little expansive because of the realistic source language. Its most intricate job is keeping track of all the renaming going on to resolve name mismatches. Other than that, the desugaring follows from the restrictions on the Grd language.

Note how our naive desugaring function generates an abundance of fresh temporary variables. In pratice, the implementation of $\mathcal{D}$ can be smarter about it, by looking at the pattern when choosing a name for the variable.

### 4.2 Checking Guard Trees

Figure 7 shows the two main functions for checking guard trees. $\mathcal{U}$ carries out exhaustiveness checking by computing the set of uncovered values for a particular guard tree, whereas $\mathcal{A}$ computes the corresponding annotated tree, capturing redundancy information.

Both functions take as input the set of values *reaching* the particular guard tree node passed in as second parameter. The definition of $\mathcal{U}$ follows the intuition we built up earlier: It refines the set of reaching values as a subset of it falls through from one clause to the next. This is most visible in the ; case (top-to-bottom composition), where the set of values reaching the right (or bottom) child is exactly the set of values that were uncovered by the left (or top) child on the set of values reaching

$$\boxed{\mathcal{D}(defn) = \text{Gdt}, \mathcal{D}(clause) = \text{Gdt}, \mathcal{D}(grhs) = \text{Gdt}}$$

$$\boxed{\mathcal{D}(guard) = \overline{\text{Grd}}, \mathcal{D}(x, pat) = \overline{\text{Grd}}}$$

$$
\begin{array}{lcl}
\mathcal{D}(clause_1 \ldots clause_n) & = & \left[\begin{array}{l} \mathcal{D}(clause_1) \\ \ldots \\ \mathcal{D}(clause_n) \end{array}\right. \\[2ex]
\mathcal{D}(f\ pat_1 \ldots pat_n = expr) & = & \longrightarrow\!\!\!\!\dashv \mathcal{D}(x_1, pat_1) \ldots \mathcal{D}(x_n, pat_n) \longrightarrow k \\[1ex]
\mathcal{D}(f\ pat_1 \ldots pat_n\ grhs_1 \ldots grhs_m) & = & \longrightarrow\!\!\!\!\dashv \mathcal{D}(x_1, pat_1) \ldots \mathcal{D}(x_n, pat_n) \left[\begin{array}{l} \mathcal{D}(grhs_1) \\ \ldots \\ \mathcal{D}(grhs_m) \end{array}\right. \\[2ex]
\mathcal{D}(\mid guard_1 \ldots guard_n = expr) & = & \longrightarrow\!\!\!\!\dashv \mathcal{D}(guard_1) \ldots \mathcal{D}(guard_n) \longrightarrow k \\[2ex]
\mathcal{D}(pat \leftarrow expr) & = & \text{let } x = expr, \mathcal{D}(x, pat) \\
\mathcal{D}(expr) & = & \text{let } b = expr, \mathcal{D}(b, True) \\
\mathcal{D}(\texttt{let}\ x = expr) & = & \text{let } x = expr \\[2ex]
\mathcal{D}(x, y) & = & \text{let } y = x \\
\mathcal{D}(x, \_) & = & \epsilon \\
\mathcal{D}(x, K\ pat_1 \ldots pat_n) & = & !x, K\ y_1 \ldots y_n \leftarrow x, \mathcal{D}(y_1, pat_1), \ldots, \mathcal{D}(y_n, pat_n) \\
\mathcal{D}(x, y@pat) & = & \text{let } y = x, \mathcal{D}(y, pat) \\
\mathcal{D}(x, !pat) & = & !x, \mathcal{D}(x, pat) \\
\mathcal{D}(x, \sim pat) & = & \epsilon \\
\mathcal{D}(x, y + l) & = & \mathcal{D}(x \geqslant l), \text{let } y = x - l
\end{array}
$$

**Fig. 6.** Desugaring Haskell to Gdt

the whole node. A GRHS covers every reaching value. The left-to-right semantics of `Guard` are respected by refining the set of values reaching the wrapped subtree, depending on the particular guard. Bang guards and let bindings don't do anything beyond that refinement, whereas pattern guards additionally account for the possibility of a failed pattern match. Note that ultimately, a failing pattern guard is the only way in which the uncovered set can become non-empty!

When $\mathcal{A}$ hits a GRHS, it asks $\mathcal{G}$ for inhabitants of $\Theta$ to decide whether the GRHS is accessible or not. Since $\mathcal{A}$ needs to compute and maintain the set of reaching values just the same as $\mathcal{U}$, it has to call out to $\mathcal{U}$ for the ; case. Out of the three guard cases, the one handling bang guards is the only one doing more than just refining the set of reaching values for the subtree (thus respecting left-to-right semantics). A bang guard $!x$ is handled by testing whether the set of reaching values $\Theta$ is compatible with the assignment $x \approx \bot$, which again is done by asking $\mathcal{G}$ for concrete inhabitants of the resulting refinement type. If it *is* inhabited, then the bang guard might diverge and we need to wrap the annotated subtree in a $\lightning$.

Pattern guard semantics are important for $\mathcal{U}$ and bang guard semantics are important for $\mathcal{A}$. But what about let bindings? They are in fact completely uninteresting to the checking process,

**Operations on $\Theta$**

$$\begin{array}{lcl}\{\,\Gamma \mid \Phi\,\} \mathbin{\dot\wedge} \varphi & = & \{\,\Gamma \mid \Phi \wedge \varphi\,\} \\ \{\,\Gamma \mid \Phi_1\,\} \cup \{\,\Gamma \mid \Phi_2\,\} & = & \{\,\Gamma \mid \Phi_1 \vee \Phi_2\,\}\end{array}$$

**Checking Guard Trees**

$$\boxed{\mathcal{U}(\Theta, t_G) = \Theta}$$

$$\begin{array}{lcl}\mathcal{U}(\{\,\Gamma \mid \Phi\,\}, \texttt{Rhs}\ n) & = & \{\,\Gamma \mid \times\,\} \\ \mathcal{U}(\Theta, t; u) & = & \mathcal{U}(\mathcal{U}(\Theta, t), u) \\ \mathcal{U}(\Theta, \texttt{Guard}\ (!x)\ t) & = & \mathcal{U}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), t) \\ \mathcal{U}(\Theta, \texttt{Guard}\ (\textsf{let}\ x = e)\ t) & = & \mathcal{U}(\Theta \mathbin{\dot\wedge} (x \approx e), t) \\ \mathcal{U}(\Theta, \texttt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t) & = & (\Theta \mathbin{\dot\wedge} (x \not\approx K)) \cup \mathcal{U}(\Theta \mathbin{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t)\end{array}$$

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

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

**Putting it all together**

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

**Fig. 7.** Pattern-match checking

but making sense of them is important for the precision of the emptiness check involving $\mathcal{G}$, as we'll see in section 4.3.

## 4.3 Testing for Emptiness

**SG:** Maybe the this paragraph should somewhere else, possibly earlier. The predicate literals $\varphi$ of refinement types looks quite similar to the original Grd language, so how is checking them for emptiness an improvement over reasoning about about guard trees directly? To appreciate the translation step we just described, it is important to realise that semantics of Grds are *highly non-local*! Left-to-right and top-to-bottom match semantics means that it's hard to view Grds in isolation, we always have to reason about whole Gdts. By contrast, refinement types are self-contained, which means the emptiness test can be treated in separation from the whole pattern match checking problem.

The key function for the emptiness test is $\mathcal{G}$ in fig. 8, which generates a set of patterns which inhabit a given refinement type $\Theta$. There might be multiple inhabitants, and $\mathcal{C}$ will construct multiple $\nabla$s, each representing at least one inhabiting assignment of the refinement predicate $\Phi$. Each such assignment corresponds to a pattern vector, so $\mathcal{E}$ expands the assignments in a

**Generate inhabitants of $\Theta$**

$$\boxed{\mathcal{G}(\Theta) = \mathcal{P}(\overline{p})}$$

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

**Construct inhabited $\nabla$s from $\Phi$**

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

$$\begin{array}{lcl}
\mathcal{C}(\nabla, \varphi) & = & \begin{cases} \{\Gamma' \triangleright \Phi'\} & \text{where } \Gamma' \triangleright \Phi' = \nabla \oplus_\varphi \varphi \\ \emptyset & \text{otherwise} \end{cases} \\
\mathcal{C}(\nabla, \Phi_1 \wedge \Phi_2) & = & \bigcup \{\mathcal{C}(\nabla', \Phi_2) \mid \nabla' \in \mathcal{C}(\nabla, \Phi_1)\} \\
\mathcal{C}(\nabla, \Phi_1 \vee \Phi_2) & = & \mathcal{C}(\nabla, \Phi_1) \cup \mathcal{C}(\nabla, \Phi_2)
\end{array}$$

**Expand variables to Pat with $\nabla$**

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

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

**Finding the representative of a variable in $\Delta$**

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

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

**Fig. 8.** Generating inhabitants of $\Theta$ via $\nabla$

$\nabla$ into multiple pattern vectors. **SG:** Currently, $\mathcal{E}$ will only expand positive constraints and not produce multiple pattern vectors for a $\nabla$ with negative info (see the TODO comment attached to $\mathcal{E}$'s definition)

But what *is* $\nabla$? To a first approximation, it is a set of mutually compatible constraints $\delta$ (or a proven incomatibility $\times$ between them). It is also a unifier to the particular $\Phi$ it is constructed for, in that the recorded constraints are valid assignments for the variables occuring in the orginal predicate **SG:** This is not true of $\times$. Each $\nabla$ is the trace of commitments to a left or right disjunct in a $\Phi$ **SG:** Not sure how to say this more accurately, which are checked in isolation. So in contrast to $\Phi$, there is no disjunction in $\Delta$. Which makes it easy to check if a new constraint is compatible with the existing ones without any backtracking. Another fundamental difference is that $\delta$ has no binding construts (so every variable has to be bound in the $\Gamma$ part of $\nabla$), whereas pattern bindings in $\varphi$ bind constructor arguments.

$\mathcal{C}$ is the function that breaks down a $\Phi$ into multiple $\nabla$s. At the heart of $\mathcal{C}$ is adding a $\varphi$ literal to the $\nabla$ under construction via $\oplus_\varphi$ and filtering out any unsuccessful attempts (via intercepting the $\times$ failure mode) to do so. Conjunction is handled by the equivalent of a *concatMap*, whereas a disjunction corresponds to a plain union.

Expanding a $\nabla$ to a pattern vector in $\mathcal{E}$ is syntactically heavy, but straightforward: When there is a positive constraint like $x \approx \mathit{Just}\ y$ in $\Delta$ for the head $x$ of the variable vector of interest, expand $y$ in addition to the other variables and wrap it in a $\mathit{Just}$. Only that it's not plain $x \approx \mathit{Just}\ y$, but $\Delta(x) \approx \mathit{Just}\ y$. That's because $\Delta$ is in *triangular form* (alluding to *triangular substitutions*, as opposed to an idempotent substitution): We have to follow $x \approx y$ constraints in $\Delta$ until we find the representative of its equality class, to which all constraints apply. Note that a $x \approx y$ constraint implies absence of any other constraints mentioning $x$ in its left-hand side ($x \approx y \in \Delta \Longrightarrow (\Delta \cap x = x \approx y)$, foreshadowing notation from fig. 9). For $\mathcal{E}$ to be well-defined, there needs to be at most one positive constraint in $\Delta$.

Thus, constraints within $\nabla$s constructed by $\oplus_\varphi$ satisfy a number of well-formedness constraints, like mutual compatibility, triangular form and the fact that there is at most one positive constraint $x \approx \_$ per variable $x$. We refer to such $\nabla$s as an *inert set*, in the sense that constraints inside it satisfy it are of canonical form and already checked for mutual compatibility, in analogy to a typechecker's implementation **SG:** Feel free to flesh out or correct this analogy.

## 4.4 Extending the inert set

After tearing down abstraction after abstraction in the previous sections we nearly hit rock bottom: Figure 9 depicts how to add a $\varphi$ constraint to an inert set $\nabla$.

It does so by expressing a $\varphi$ in terms of once again simpler constraints $\delta$ and calling out to $\oplus_\delta$. Specifically, for a lack of binding constructs in $\delta$, pattern bindings extend the context and disperse into separate type constraints and a positive constructor constraint arising from the binding. The fourth case of $\oplus_\delta$ makes sense of let bindings: In case the right-hand side was a constructor application (which is not to be confused with a pattern binding, if only for the difference in binding semantics!), we add appropriate positive constructor and type constraints, as well as recurse into the field expressions, which might in turn contain nested constructor applications. The last case of $\oplus_\varphi$ turns the syntactically and semantically identical subset of $\varphi$ into $\delta$ and adds that constraint via $\oplus_\delta$.

Which brings us to the prime unification procedure, $\oplus_\delta$. Consider adding a positive constructor constraint like $x \approx \mathit{Just}\ y$: The unification procedure will first look for any positive constructor constraint involving the representative of $x$ with *that same constructor*. Let's say there is $\Delta(x) = z$ and $z \approx \mathit{Just}\ u \in \Delta$. Then $\oplus_\delta$ decomposes the new constraint just like a classic unification algorithm, by equating type and term variables with new constraints, i.e. $y \approx u$. If there was no positive constructor constraint with the same constructor, it will look for such a constraint involving a different constructor, like $x \approx \mathit{Nothing}$. In this case the new constraint is incompatible by *generativity* of data constructors [Eisenberg 2016]. There are two other ways in which the constraint can be incompatible: If there was a negative constructor constraint $x \not\approx \mathit{Just}$ or if any of the fields were not inhabited, which is checked by the $\nabla \vdash x$ judgment in fig. 10. Otherwise, the constraint is compatible and is added to $\Delta$.

Adding a negative constructor constraint $x \not\approx \mathit{Just}$ is quite similar, so is handling of positive and negative constraints involving $\bot$. The scheme is that whenever we add a negative constraint that doesn't contradict with positive constraints, we still have to test if there are any inhabitants left.

**SG:** Maybe move down the type constraint case in the definition? Adding a type constraint drives this paranoia to a maximum: After calling out to the type-checker (the logic of which we do not and would not replicate in this paper or our implementation) to assert that the constraint is consistent with the inert set, we have to test *all* variables in the domain of $\Gamma$ for inhabitants, because the new type constraint could have rendered a type empty. To demonstrate why this is necessary, imagine we have $x : a \triangleright x \not\approx \bot$ and try to add $a\ \mathit{Void}$. Although the type constraint is consistent, $x$ in $x : a \triangleright x \not\approx \bot, a\ \mathit{Void}$ is no longer inhabited. There is room for being smart about which variables

**Add a formula literal to the inert set**

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

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

**Add a constraint to the inert set**

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

$$\begin{array}{lcl}
\times \oplus_\delta \delta & = & \times \\
\Gamma \triangleright \Delta \oplus_\delta \gamma & = & \begin{cases} \Gamma \triangleright (\Delta, \gamma) & \text{if type checker deems } \gamma \text{ compatible with } \Delta \\ & \text{and } \forall x \in \mathrm{dom}(\Gamma) : \Gamma \triangleright (\Delta, \gamma) \vdash \Delta(x) \\ \times & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Delta \oplus_\delta x \approx K\ \overline{a}\ \overline{y} & = & \begin{cases} \Gamma \triangleright \Delta \oplus_\delta \overline{a \sim b} \oplus_\delta \overline{y \approx z} & \text{if } \Delta(x) \approx K\ \overline{b}\ \overline{z} \in \Delta \\ \times & \text{if } \Delta(x) \approx K'\ \overline{b}\ \overline{z} \in \Delta \\ \Gamma \triangleright (\Delta, \Delta(x) \approx K\ \overline{a}\ \overline{y}) & \text{if } \Delta(x) \not\approx K \notin \Delta \text{ and } \overline{\Gamma \triangleright \Delta \vdash \Delta(y)} \\ \times & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Delta \oplus_\delta x \not\approx K & = & \begin{cases} \times & \text{if } \Delta(x) \approx K\ \overline{a}\ \overline{y} \in \Delta \\ \times & \text{if not } \Gamma \triangleright (\Delta, \Delta(x) \not\approx K) \vdash \Delta(x) \\ \Gamma \triangleright (\Delta, \Delta(x) \not\approx K) & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Delta \oplus_\delta x \approx \bot & = & \begin{cases} \times & \text{if } \Delta(x) \not\approx \bot \in \Delta \\ \Gamma \triangleright (\Delta, \Delta(x) \approx \bot) & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Delta \oplus_\delta x \not\approx \bot & = & \begin{cases} \times & \text{if } \Delta(x) \approx \bot \in \Delta \\ \times & \text{if not } \Gamma \triangleright (\Delta, \Delta(x) \not\approx \bot) \vdash \Delta(x) \\ \Gamma \triangleright (\Delta, \Delta(x) \not\approx \bot) & \text{otherwise} \end{cases} \\
\Gamma \triangleright \Delta \oplus_\delta x \approx y & = & \begin{cases} \Gamma \triangleright \Delta & \text{if } \Delta(x) = \Delta(y) \\ \Gamma \triangleright ((\Delta \setminus \Delta(x)), \Delta(x) \approx \Delta(y)) \oplus_\delta ((\Delta \cap \Delta(x))[\Delta(y)/\Delta(x)]) & \text{otherwise} \end{cases}
\end{array}$$

$$\boxed{\Delta \setminus x = \Delta}$$

$$\begin{array}{rcl}
\varnothing \setminus x & = & \varnothing \\
(\Delta, x \approx K\ \overline{a}\ \overline{y}) \setminus x & = & \Delta \setminus x \\
(\Delta, x \not\approx K) \setminus x & = & \Delta \setminus x \\
(\Delta, x \approx \bot) \setminus x & = & \Delta \setminus x \\
(\Delta, x \not\approx \bot) \setminus x & = & \Delta \setminus x \\
(\Delta, \delta) \setminus x & = & (\Delta \setminus x), \delta
\end{array}$$

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

$$\begin{array}{rcl}
\varnothing \cap x & = & \varnothing \\
(\Delta, x \approx K\ \overline{a}\ \overline{y}) \cap x & = & (\Delta \cap x), x \approx K\ \overline{a}\ \overline{y} \\
(\Delta, x \not\approx K) \cap x & = & (\Delta \cap x), x \not\approx K \\
(\Delta, x \approx \bot) \cap x & = & (\Delta \cap x), x \approx \bot \\
(\Delta, x \not\approx \bot) \cap x & = & (\Delta \cap x), x \not\approx \bot \\
(\Delta, \delta) \cap x & = & \Delta \cap x
\end{array}$$

**Fig. 9.** Adding a constraint to the inert set $\nabla$

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

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

$$\frac{(\Gamma \triangleright \Delta \oplus_\delta x \approx \bot) \neq \times}{\Gamma \triangleright \Delta \vdash x}\vdash\textsc{Bot} \qquad \frac{x : \tau \in \Gamma \quad \mathsf{Cons}(\Gamma \triangleright \Delta, \tau) = \bot}{\Gamma \triangleright \Delta \vdash x}\vdash\textsc{NoCpl}$$

$$\frac{\begin{array}{c} x : \tau \in \Gamma \quad K \in \mathsf{Cons}(\Gamma \triangleright \Delta, \tau) \\ \mathsf{Inst}(\Gamma \triangleright \Delta, x, K) \neq \times \end{array}}{\Gamma \triangleright \Delta \vdash x}\vdash\textsc{Inst}$$

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

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

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

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

$$\boxed{\mathsf{Inst}(\nabla, x, K) = \nabla}$$

$$\begin{array}{l} \mathsf{Inst}(\Gamma \triangleright \Delta, x, K) = \Gamma, \overline{a}, \overline{y : \sigma} \triangleright \Delta \oplus_\delta \tau_x \sim \tau \oplus_\delta \overline{\gamma} \oplus_\delta x \approx K\ \overline{a}\ \overline{y} \oplus_\delta \overline{y' \not\approx \bot} \\ \qquad \text{where } K : \forall \overline{a}. \overline{\gamma} \Rightarrow \overline{\sigma} \to \tau,\ \overline{y} \# \Gamma,\ \overline{a} \# \Gamma,\ x : \tau_x \in \Gamma,\ \overline{y'} \text{ bind strict fields} \end{array}$$

**Fig. 10.** Inhabitance test

we have to re-check: For example, we can exclude variables whose type is a non-GADT data type. **SG:** That trick probably belongs in the implementation section.

The last case of $\oplus_\delta$ equates two variables ($x \approx y$) by merging their equivalence classes. Consider the case where $x$ and $y$ don't already belong to the same equivalence class, so have different representatives $\Delta(x)$ and $\Delta(y)$. $\Delta(y)$ is arbitrarily chosen to be the new representative of the merged equivalence class. Now to uphold one of the well-formedness conditions **SG:** Which one? Better have a list of them and reference them here., all constraints mentioning $\Delta(x)$ have to be removed and renamed in terms of $\Delta(y)$ and then re-added to $\Delta$. That might fail, because $\Delta(x)$ might have a constraint that conflicts with constraints on $\Delta(y)$, so better use $\oplus_\delta$ rather than add it blindly to $\Delta$.

## 4.5 Inhabitance Test

**SG:** We need to find better subsection titles that clearly distinguish "Testing ($\Theta$) for Emptiness" from "Inhabitance Test(ing a particular variable in $\nabla$)". The process for adding a constraint to an inert set above (which turned out to be a unification procedure in disguise) frequently made use of an *inhabitation test* $\nabla \vdash x$, depicted in fig. 10. In contrast to the emptiness test in fig. 8, this one focuses on a particular variable and works on a $\nabla$ rather than a much higher-level $\Theta$.

The $\vdash$Bot judgment of $\nabla \vdash x$ tries to instantiate $x$ to $\bot$ to conclude that $x$ is inhabited. $\vdash$Inst instantiates $x$ to one of its data constructors. That will only work if its type ultimately reduces to a data type under the type constraints in $\nabla$. Rule $\vdash$NoCpl will accept unconditionally when its type is not a data type, i.e. for $x : Int \to Int$.

## REFERENCES

Richard A. Eisenberg. 2016. *Dependent Types in Haskell: Theory and Practice.* Ph.D. Dissertation. arXiv:cs.PL/1610.07978

Georgios Karachalias, Tom Schrijvers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. *GADTs meet their match (extended version).* Technical Report. KU Leuven. https://people.cs.kuleuven.be/~tom.schrijvers/Research/papers/icfp2015.pdf

Matthew Pickering, Gergő Érdi, Simon Peyton Jones, and Richard A. Eisenberg. 2016. Pattern Synonyms. In *Proceedings of the 9th International Symposium on Haskell (Haskell 2016).* Association for Computing Machinery, New York, NY, USA, 80–91. https://doi.org/10.1145/2976002.2976013