Lower Your Guards

A Compositional Pattern-Match Coverage Checker

SEBASTIAN GRAF, Karlsruhe Institute of Technology, Germany
SIMON PEYTON JONES, Microsoft Research, UK
RYAN G. SCOTT, Indiana University, USA

One of a compiler's roles is to warn if a function defined by pattern matching does not cover its inputs—that is, if there are missing or redundant patterns. Generating such warnings accurately is difficult for modern languages due to the myriad of interacting language features when pattern matching. This is especially true in Haskell, a language with a complicated pattern language that is made even more complex by extensions offered by the Glasgow Haskell Compiler (GHC). Although GHC has spent a significant amount of effort towards improving its pattern-match coverage warnings, there are still several cases where it reports inaccurate warnings.

We introduce a coverage checking algorithm called Lower Your Guards, which boils down the complexities of pattern matching into *guard trees*. While the source language may have many exotic forms of patterns, guard trees only have three different constructs, which vastly simplifies the coverage checking process. Our algorithm is modular, allowing for new forms of source-language patterns to be handled with little changes to the overall structure of the algorithm. We have implemented the algorithm in GHC and demonstrate places where it performs better than GHC's current coverage checker, both in accuracy and performance.

Draft March 2020. We would appreciate any comments or feedback to the email addresses below.

1 INTRODUCTION

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

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

The function f has two serious flaws. One obvious problem 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 (or errors) if a function is missing clauses (i.e., if it is *non-exhaustive*), if one of its right-hand sides will never be entered (i.e., if it is *inaccessible*), or if one of its equations can be deleted altogether (i.e., if it is *redundant*). We refer to the combination of checking for exhaustivity, redundancy, and accessibility as *pattern-match coverage checking*. Coverage checking is the first line of defence in catching programmer mistakes when defining code that uses pattern matching.

Coverage checking for a set of equations matching on algebraic data types is a well studied (although still surprisingly tricky) problem—see Section 7 for this related work. But the coverage-checking problem becomes *much* harder when one includes the raft of innovations that have become part of a modern programming language like Haskell, including: view patterns, pattern guards, pattern synonyms, overloaded literals, bang patterns, lazy patterns, as-patterns, strict data constructors, empty case expressions, and long-distance effects (Section 4). Particularly tricky are

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.

GADTs [Xi et al. 2003], where the *type* of a match can determine what *values* can possibly appear; and local type-equality constraints brought into scope by pattern matching [Vytiniotis et al. 2011].

The current state of the art for coverage checking in a richer language of this sort is *GADTs Meet Their Match* [Karachalias et al. 2015], or GMTM for short. It presents an algorithm that handles the intricacies of checking GADTs, lazy patterns, and pattern guards. However GMTM is monolithic and does not account for a number of important language features; it gives incorrect results in certain cases; its formulation in terms of structural pattern matching makes it hard to avoid some serious performance problems; and its implementation in GHC, while a big step forward over its predecessors, has proved complex and hard to maintain.

In this paper we propose a new, compositional coverage-checking algorithm, called Lower Your Guards (LYG), that is simpler, more modular, *and* more powerful than GMTM (see section 7.1). Moreover, it avoids GMTM's performance pitfalls. We make the following contributions:

- We characterise some nuances of coverage checking that not even GMTM handles (Section 2). We also identify issues in GHC's implementation of GMTM.
- We describe a new, compositional coverage checking algorithm, LYG, in Section 3. The key insight is to abandon the notion of structural pattern matching altogether, and instead desugar all the complexities of pattern matching into a very simple language of *guard trees*, with just three constructs (Section 3.1). Coverage checking on these guard trees becomes remarkably simple, returning an *annotated tree* (Section 3.2) decorated with *refinement types*. Finally, provided we have access to a suitable way to find inhabitants of a refinement type, we can report accurate coverage errors (Section 3.3).
- We demonstrate the compositionality of LYG by augmenting it with several language extensions (Section 4). Although these extensions can change the source language in significant ways, the effort needed to incorporate them into the algorithm is comparatively small.
- We discuss how to optimize the performance of LYG (Section 5) and implement a proof of concept in GHC (Section 6).

We discuss the wealth of related work in Section 7.

2 THE PROBLEM WE WANT TO SOLVE

What makes coverage checking so difficult in a language like Haskell? At first glance, implementing a coverage checking 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 (discussed in Section 7) accounts for some of these nuances, but not all of them. In this section we identify some key language features that make coverage checking difficult. While these features may seem disparate at first, we will later show in Section 3 that these ideas can all fit into a unified framework.

2.1 Guards

Guards are a flexible form of control flow in Haskell. Here is a function that demonstrates various capabilities of guards:

```
guardDemo :: Char → Char → Int
guardDemo c1 c2
  | c1 == 'a'                            = 0
  | 'b' ← c1                             = 1
  | let c1′ = c1, 'c' ← c1′, c2 == 'd'   = 2
  | otherwise                            = 3
```

This function has four *guarded right-hand sides* or GRHSs for short. The first GRHS has a *boolean guard*, (*c1* == 'a'), that succeeds if the expression in the guard returns *True*. The second GRHS has a *pattern guard*, ('b' ← *c1*), that succeeds if the pattern in the guard successfully matches. The next line illustrates that a GRHS may have multiple guards, and that guards include **let** bindings, such as **let** *c1′* = *c2*. The fourth GRHS uses *otherwise*, which is simply defined as *True*.

Guards can be thought of as a generalization of patterns, and we would like to include them as part of coverage checking. Checking guards is significantly more complicated than checking ordinary structural pattern matches, however, since guards can contain arbitrary expressions. Consider this implementation of the *signum* function:

```
signum :: Int → Int
signum x | x > 0  = 1
         | x == 0 = 0
         | x < 0  = −1
```

Intuitively, *signum* is exhaustive since the combination of (>), (==), and (<) covers all possible *Int*s. This is much harder for a machine to check, however, since that would require knowledge about the properties of *Int* inequalities. Clearly, coverage checking for guards is undecidable in general. However, while we cannot accurately check *all* uses of guards, we can at least give decent warnings for some common use-cases. For instance, take the following functions:

```
not :: Bool → Bool             not2 :: Bool → Bool      not3 :: Bool → Bool
not b | False ← b = True       not2 False = True        not3 x | x ← False = True
      | True ← b  = False      not2 True  = False       not3 True          = False
```

Clearly all are equivalent. Our coverage checking algorithm should find that all three are exhaustive, and indeed, LYG does so.

2.2 Programmable patterns

Expressions in guards are not the only source of undecidability that the coverage checker must cope with. GHC extends the pattern language in other ways that are also impossible to check in the general case. We consider two such extensions here: view patterns and pattern synonyms.

2.2.1 View patterns. View patterns allow arbitrary computation to be performed while pattern matching. When a value v is matched against a view pattern ($f \rightarrow p$), the match is successful when $f\ v$ successfully matches against the pattern p. For example, one can use view patterns to succinctly define a function that computes the length of Haskell's opaque *Text* data type:

```
Text.null :: Text → Bool   -- Checks if a Text is empty
Text.uncons :: Text → Maybe (Char, Text)   -- If a Text is non-empty, return Just (x, xs),
```

```
                                   -- where x is the first character and xs is the rest
length :: Text → Int
length (Text.null → True)           = 0
length (Text.uncons → Just (_, xs)) = 1 + length xs
```

Again, it would be unreasonable to expect a coverage checking algorithm to prove that *length* is exhaustive, but one might hope for a coverage checking algorithm that handles some common usage patterns. For example, LYG indeed *is* able to prove that *safeLast* function is exhaustive:

```
safeLast :: [a] → Maybe a
safeLast (reverse → [ ])     = Nothing
safeLast (reverse → (x : _)) = Just x
```

2.2.2 *Pattern synonyms.*

Pattern synonyms [Pickering et al. 2016] allow abstraction over patterns themselves. Pattern synonyms and view patterns can be useful in tandem, as the pattern synonym can present an abstract interface to a view pattern that does complicated things under the hood. For example, one can define *length* with pattern synonyms like so:

```
pattern Nil :: Text
pattern Nil ← (Text.null → True)
pattern Cons :: Char → Text → Text
pattern Cons x xs ← (Text.uncons → Just (x, xs))

length :: Text → Int
length Nil = 0
length (Cons x xs) = 1 + length xs
```

How should a coverage checker handle pattern synonyms? One idea is to simply "look through" the definitions of each pattern synonym and verify whether the underlying patterns are exhaustive. This would be undesirable, however, because (1) we would like to avoid leaking the implementation details of abstract pattern synonyms, and (2) even if we *did* look at the underlying implementation, it would be challenging to automatically check that the combination of *Text.null* and *Text.uncons* is exhaustive.

Nevertheless, *Text.null* and *Text.uncons* together are in fact exhaustive, and GHC allows programmers to communicate this fact to the coverage checker using a `COMPLETE` pragma [GHC team 2020]. A `COMPLETE` set is a combination of data constructors and pattern synonyms that should be regarded as exhaustive when a function matches on all of them. For example, declaring `{-# COMPLETE Nil, Cons #-}` is sufficient to make the definition of *length* above compile without any exhaustivity warnings. Since GHC does not (and cannot, in general) check that all of the members of a `COMPLETE` set actually comprise a complete set of patterns, the burden is on the programmer to ensure that this invariant is upheld.

2.3 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:

```
data Void    -- No data constructors; only inhabitant is bottom
data SMaybe a = SJust !a | SNothing

v :: SMaybe Void → Int
v SNothing = 0
v (SJust _) = 1    -- Redundant!
```

The "!" in the definition of *SJust* makes the constructor strict, so $(SJust\ \bot) = \bot$. Curiously, this makes the second equation of v redundant! Since $\bot$ is the only inhabitant of type *Void*, the only inhabitants of *SMaybe Void* are *SNothing* and $\bot$. The former will match on the first equation; the latter will make the first equation diverge. In neither case will execution flow to the second equation, so it is redundant and can be deleted.

2.3.1 Redundancy versus inaccessibility. When reporting unreachable cases, we must distinguish between *redundant* and *inaccessible* cases. Redundant cases can be removed from a function without changing its semantics, whereas inaccessible cases have semantic importance. The examples below illustrate this:

$$\begin{array}{ll}
u :: () \to Int & u' :: () \to Int \\
u\ ()\ \mid False = 1 & u'\ ()\ \mid False = 1 \\
\qquad\ \mid True\ \ = 2 & \qquad\ \ \mid False = 2 \\
u\ _\qquad\quad\ = 3 & u'\ _\qquad\quad\ = 3
\end{array}$$

Within u, the equations that return 1 and 3 could be deleted without changing the semantics of u, so they are classified as redundant. Within u', one can never reach that right-hand sides of the equations that return 1 and 2, but they cannot be removed so easily. Using the definition above, $u'\ \bot\ =\ \bot$, but if the first two equations were removed, then $u'\ \bot\ = 3$. As a result, LYG warns that the first two equations in u' are inaccessible, which suggests to the programmer that u' might benefit from a refactor to avoid this (e.g., $g'\ () = 3$).

Observe that u and u' have completely different warnings, but the only difference between the two functions is whether the second equation uses *True* or *False* in its guard. Moreover, this second equation affects the warnings for *other* equations. This demonstrates that determining whether code is redundant or inaccessible is a non-local problem. Inaccessibility may seem like a tricky corner case, but GHC's users have reported many bugs of this sort (Section 6.2).

2.3.2 Bang patterns. Strict fields are one mechanism for adding extra strictness in ordinary Haskell, but GHC adds another in the form of *bang patterns*. A bang pattern such as *!pat* indicates that matching a value v against *pat* always evaluates v to weak-head normal form (WHNF). Here is a variant of v, this time using the standard, lazy *Maybe* data type:

$$\begin{array}{l}
v' :: Maybe\ Void \to Int \\
v'\ Nothing = 0 \\
v'\ (Just\ !_) = 1 \quad \text{-- Not redundant, but RHS is inaccessible}
\end{array}$$

The inhabitants of the type *Maybe Void* are $\bot$, *Nothing*, and $(Just\ \bot)$. The input $\bot$ makes the first equation diverge; *Nothing* matches on the first equation; and $(Just\ \bot)$ makes the second equation diverge because of the bang pattern. Therefore, none of the three inhabitants will result in the right-hand side of the second equation being reached. Note that the second equation is inaccessible, but not redundant (section 2.3.1).

2.4 Type-equality constraints

Besides strictness, another way for pattern matches to be rendered unreachable is by way of *equality constraints*. A popular method for introducing equalities between types is matching on GADTs [Xi et al. 2003]. The following examples demonstrate the interaction between GADTs and coverage checking:

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	$::=$	$x \mid \_ \mid K\ \overline{pat} \mid x@pat \mid !pat \mid expr \rightarrow pat$
P	Pattern synonyms	$match$	$::=$	$= expr \mid \overline{grhs}$
T	Type constructors	$grhs$	$::=$	$\mid\ \overline{guard} = expr$
l	Literal	$guard$	$::=$	$pat \leftarrow expr \mid expr \mid \texttt{let}\ x = expr$
$expr$	Expressions			

Fig. 1. Source syntax

$$
\begin{array}{lll}
\textbf{data}\ T\ a\ b\ \textbf{where} & g1 :: T\ Int\ b \rightarrow b \rightarrow Int & g2 :: T\ a\ b \rightarrow T\ a\ b \rightarrow Int \\
\quad T1 :: T\ Int\ Bool & g1\ T1\ False = 0 & g2\ T1\ T1 = 0 \\
\quad T2 :: T\ Char\ Bool & g1\ T1\ True = 1 & g2\ T2\ T2 = 1
\end{array}
$$

When *g1* matches against *T1*, the *b* in the type *T Int b* is known to be a *Bool*, which is why matching the second argument against *False* or *True* will typecheck. Phrased differently, matching against *T1* brings into scope an *equality constraint* between the types *b* and *Bool*. GHC has a powerful type inference engine that is equipped to reason about type equalities of this sort [Vytiniotis et al. 2011].

Just as important as the code used in the *g1* function is the code that is *not* used in *g1*. One might wonder if *g1* not matching its first argument against *T2* is an oversight. In fact, the exact opposite is true: matching on *T2* would be rejected by the typechecker. This is because *T2* is of type *T Char Bool*, but the first argument to *g1* must be of type *T Int b*. Matching against *T2* would be tantamount to saying that *Int* and *Char* are the same type, which is not the case. As a result, *g1* is exhaustive even though it does not match on all of *T*'s data constructors.

The presence of type equalities is not always as clear-cut as it is in *g1*. Consider the more complex *g2* function, which matches on two arguments of the type *T a b*. While matching the arguments against *T1 T1* or *T2 T2* is possible, it is not possible to match against *T1 T2* or *T2 T1*. To see why, suppose the first argument is matched against *T1*, giving rise to an equality between *a* and *Int*. If the second argument were then matched against *T2*, we would have that *a* equals *Char*. By the transitivity of type equality, we would have that *Int* equals *Char*. This cannot be true, so matching against *T1 T2* is impossible (and similarly for *T2 T1*).

Concluding that *g2* is exhaustive requires some non-trivial reasoning about equality constraints. In GHC, the same engine that typechecks GADT pattern matches is also used to rule out cases made unreachable by type equalities, and LYG adopts a similar approach. Besides GHC's current coverage checker [Karachalias et al. 2015], there are a variety of other coverage checking algorithms that account for GADTs, including those for OCaml [Garrigue and Normand 2011], Dependent ML [Xi 1998a,b, 2003], and Stardust [Dunfield 2007].

3 LOWER YOUR GUARDS: A NEW COVERAGE CHECKER

In this section, we describe our new coverage checking algorithm, LYG. Figure 2 depicts a high-level overview, which divides into three steps:

- First, we desugar the complex source Haskell syntax into a *guard tree* t : Gdt (Section 3.1). The language of guard trees is tiny but expressive, and allows the subsequent passes to be entirely independent of the source syntax. LYG can readily be adapted to other languages simply by changing the desugaring algorithm.

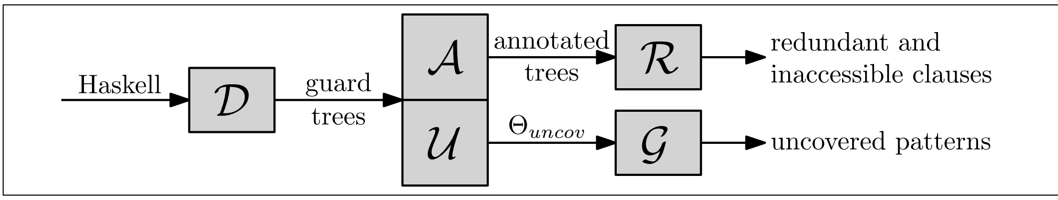


Fig. 2. Bird's eye view of pattern match checking

Guard syntax

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

Refinement type syntax

$$
\begin{array}{lcll}
\Gamma & ::= & \varnothing \mid \Gamma, x : \tau \mid \Gamma, a & \text{Context} \\
\varphi & ::= & \checkmark \mid \times \mid K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mid x \not\approx K \mid x \approx \bot \mid x \not\approx \bot \mid \textsf{let}\ x = e & \text{Literals} \\
\Phi & ::= & \varphi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi & \text{Formula} \\
\Theta & ::= & \langle \Gamma \mid \Phi \rangle & \text{Refinement type}
\end{array}
$$

Clause tree syntax

$$
\begin{array}{rcl}
t \in \textsf{Gdt} & ::= & \textsf{GRhs}\ n \mid t_1; t_2 \mid \textsf{Guard}\ g\ t \\
u \in \textsf{Ant} & ::= & \textsf{ARhs}\ \Theta\ n \mid u_1; u_2 \mid \textsf{Bang}\ \Theta\ u
\end{array}
$$

Graphical notation

┌─ t_1 └─ t_2	::=	$t_1; t_2$	┌─ u_1 └─ u_2	::=	$u_1; u_2$
$g_1, \ldots, g_n$ — t	::=	$\textsf{Guard}\ g_1 \ldots (\textsf{Guard}\ g_n\ t)$	Θ ↯ — u	::=	$\textsf{Bang}\ \Theta\ u$
⟶ n	::=	$\textsf{GRhs}\ n$	⟶ $\Theta\ n$	::=	$\textsf{ARhs}\ \Theta\ n$

Fig. 3. IR syntax

- Next, the resulting guard tree is then processed by two different functions (Section 3.2). The function $\mathcal{A}(t)$ produces an *annotated tree* u : Ant, which has the same general branching structure as t but describes which clauses are accessible, inaccessible, or redundant. The function $\mathcal{U}(t)$, on the other hand, returns a *refinement type* Θ [Rushby et al. 1998; Xi and Pfenning 1998] that describes the set of *uncovered values*, which are not matched by any of the clauses.
- Finally, an error-reporting pass generates comprehensible error messages (Section 3.3). Again there are two things to do. The function $\mathcal{R}$ processes the annotated tree produced by $\mathcal{A}$ to explicitly identify the accessible, inaccessible, or redundant clauses. The function $\mathcal{G}(\Theta)$

produces representative *inhabitants* of the refinement type Θ (produced by $\mathcal{U}$) that describes the uncovered values.

LYG's main contribution when compared to other coverage checkers, such as GMTM, is its incorporation of many small improvements and insights, rather than a single defining breakthrough. In particular, LYG's advantages are:

- Correctly accounting for strictness in identifying redundant and inaccessible code (section 7.4).
- Using detailed term-level reasoning (figs. 6 to 8), which GMTM does not.
- Using *negative information* to sidestep serious performance issues in GMTM without changing the worst-case complexity (section 7.3). This also enables graceful degradation (section 5.2) and the ability to handle `COMPLETE` sets properly (section 5.3).
- Achieving modularity by clearly separating the source syntax (fig. 1) from the intermediate language (fig. 3).
- Fixing various bugs present in GMTM, both in the paper [Karachalias et al. 2015] and in GHC's implementation thereof (section 6.2).

3.1 Desugaring to guard trees

The first step is to desugar the source language into the language of guard trees. The syntax of the source language is given in Figure 1. Definitions *defn* consist of a list of *clauses*, each of which has a list of *patterns*, and a list of *guarded right-hand sides* (GRHSs). Patterns include variables and constructor patterns, of course, but also a representative selection of extensions: wildcards, as-patterns, bang patterns, and view patterns. We explore several other extensions in Section 4.

The language of guard trees Gdt is much smaller; its syntax is given in Figure 3. All of the syntactic redundancy of the source language is translated into a minimal form very similar to pattern guards. We start with an example:

$$
\begin{array}{lll}
f\ (Just\ (!xs, \_))\ & ys@Nothing & = 1 \\
f\ Nothing & (g \rightarrow True) & = 2
\end{array}
$$

This desugars to the following guard tree:

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

Here we use a graphical syntax for guard trees, also defined in Figure 3. The first line says "evaluate x_1; then match x_1 against $Just\ t_1$; then match t_1 against (t_2, t_3); and so on". If any of those matches fail, we fall through into the second line.

More formally, matching a guard tree may *succeed* (with some bindings for the variables bound in the tree), *fail*, or *diverge*. Matching is defined as follows:

- Matching a guard tree (`GRhs` n) succeeds.
- Matching a guard tree $(t_1; t_2)$ means matching against t; if that succeeds, the overall match succeeds; if not, match against t_2.
- Matching a guard tree (`Guard` $!x\ t$) evaluates x; if that diverges the match diverges; if not match t.
- Matching a guard tree (`Guard` $(K\ y_1 \ldots y_n \leftarrow x)\ t$) matches x against constructor K. If the match succeeds, bind $y_1 \ldots y_n$ to the components, and match t; if the constructor match fails, then the entire match fails.
- Matching a guard tree (`Guard` $(\mathsf{let}\ x = e)\ t$) binds x (lazily) to e, and matches t.

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

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

$$\mathcal{D}(clause_1 \ldots clause_n) = \begin{cases} \mathcal{D}(clause_1) \\ \ldots \\ \mathcal{D}(clause_n) \end{cases}$$

$$\mathcal{D}(f\ pat_1 \ldots pat_n = expr) = \;\longrightarrow\!\!| \; \mathcal{D}(x_1, pat_1) \ldots \mathcal{D}(x_n, pat_n) \longrightarrow k_{rhs}$$

$$\mathcal{D}(f\ pat_1 \ldots pat_n\ grhs_1 \ldots grhs_m) = \;\longrightarrow\!\!| \; \mathcal{D}(x_1, pat_1) \ldots \mathcal{D}(x_n, pat_n) \begin{cases} \mathcal{D}(grhs_1) \\ \ldots \\ \mathcal{D}(grhs_m) \end{cases}$$

$$\mathcal{D}(|\ guard_1 \ldots guard_n = expr) = \;\longrightarrow\!\!| \; \mathcal{D}(guard_1) \ldots \mathcal{D}(guard_n) \longrightarrow k$$

$$\begin{array}{lcl}
\mathcal{D}(pat \leftarrow expr) & = & \text{let } x = expr, \mathcal{D}(x, pat) \\
\mathcal{D}(expr) & = & \text{let } b = expr, \mathcal{D}(b, True) \\
\mathcal{D}(\texttt{let } x = expr) & = & \text{let } x = expr \\
\\
\mathcal{D}(x, y) & = & \text{let } y = x \\
\mathcal{D}(x, _) & = & \epsilon \\
\mathcal{D}(x, K\ pat_1 \ldots pat_n) & = & !x, K\ y_1 \ldots y_n \leftarrow x, \mathcal{D}(y_1, pat_1), \ldots, \mathcal{D}(y_n, pat_n) \\
\mathcal{D}(x, y@pat) & = & \text{let } y = x, \mathcal{D}(y, pat) \\
\mathcal{D}(x, !pat) & = & !x, \mathcal{D}(x, pat) \\
\mathcal{D}(x, expr \rightarrow pat) & = & \text{let } y = expr\ x, \mathcal{D}(y, pat)
\end{array}$$

Fig. 4. Desugaring from source language to Gdt

The desugaring algorithm, $\mathcal{D}$, is given in Figure 4. It is a straightforward recursive descent over the source syntax, with a little bit of administrative bureaucracy to account for renaming. It also generates an abundance of fresh temporary variables; in practice, the implementation of $\mathcal{D}$ can be smarter than this by looking at the pattern (which might be a variable match or as-pattern) when choosing a name for a temporary variable.

Notice that both "structural" pattern-matching in the source language (e.g. the match on *Nothing* in the second equation), and view patterns (e.g. $g \longrightarrow True$) can readily be compiled to a single form of matching in guard trees. The same holds for pattern guards. For example, consider this (stylistically contrived) definition of *liftEq*, which is inexhaustive:

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

It desugars thus:

Operations on Θ

$$
\begin{aligned}
\langle\, \Gamma \mid \Phi \,\rangle \mathrel{\dot\wedge} \varphi &= \langle\, \Gamma \mid \Phi \wedge \varphi \,\rangle \\
\langle\, \Gamma \mid \Phi_1 \,\rangle \cup \langle\, \Gamma \mid \Phi_2 \,\rangle &= \langle\, \Gamma \mid \Phi_1 \vee \Phi_2 \,\rangle
\end{aligned}
$$

Checking guard trees

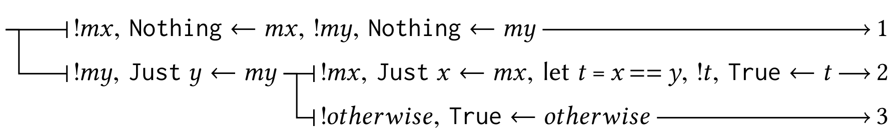
$$\boxed{\mathcal{U}(\Theta, t) = \Theta}$$

$$
\begin{aligned}
\mathcal{U}(\langle\, \Gamma \mid \Phi \,\rangle, \mathtt{GRhs}\ n) &= \langle\, \Gamma \mid \times \,\rangle \\
\mathcal{U}(\Theta, t_1; t_2) &= \mathcal{U}(\mathcal{U}(\Theta, t_1), t_2) \\
\mathcal{U}(\Theta, \mathtt{Guard}\ (!x)\ t) &= \mathcal{U}(\Theta \mathrel{\dot\wedge} (x \not\approx \bot), t) \\
\mathcal{U}(\Theta, \mathtt{Guard}\ (\mathtt{let}\ x = e)\ t) &= \mathcal{U}(\Theta \mathrel{\dot\wedge} (\mathtt{let}\ x = e), t) \\
\mathcal{U}(\Theta, \mathtt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t) &= (\Theta \mathrel{\dot\wedge} (x \not\approx K)) \cup \mathcal{U}(\Theta \mathrel{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t)
\end{aligned}
$$

$$\boxed{\mathcal{A}(\Theta, t) = u}$$

$$
\begin{aligned}
\mathcal{A}(\Theta, \mathtt{GRhs}\ n) &= \mathtt{ARhs}\ \Theta\ n \\
\mathcal{A}(\Theta, (t_1; t_2)) &= \mathcal{A}(\Theta, t_1); \mathcal{A}(\mathcal{U}(\Theta, t_1), t_2) \\
\mathcal{A}(\Theta, \mathtt{Guard}\ (!x)\ t) &= \mathtt{Bang}\ (\Theta \mathrel{\dot\wedge} (x \approx \bot))\ \mathcal{A}(\Theta \mathrel{\dot\wedge} (x \not\approx \bot), t) \\
\mathcal{A}(\Theta, \mathtt{Guard}\ (\mathtt{let}\ x = e)\ t) &= \mathcal{A}(\Theta \mathrel{\dot\wedge} (\mathtt{let}\ x = e), t) \\
\mathcal{A}(\Theta, \mathtt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t) &= \mathcal{A}(\Theta \mathrel{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t)
\end{aligned}
$$

Fig. 5. Coverage checking



Notice that the pattern guard (*Just* $x \leftarrow mx$) and the boolean guard ($x == y$) have both turned into the same constructor-matching construct in the guard tree.

In a way there is nothing very deep here, but it took us a surprisingly long time to come up with the language of guard trees. We recommend it!

3.2 Checking guard trees

In the next step, we transform the guard tree into an *annotated tree*, Ant, and an *uncovered set*, Θ.

Taking the latter first, the uncovered set describes all the input values of the match that are not covered by the match. We use the language of *refinement types* to describe this set (see Figure 3). The refinement type $\Theta = \langle\, x_1{:}\tau_1, \ldots, x_n{:}\tau_n \mid \Phi \,\rangle$ denotes the vector of values $x_1 \ldots x_n$ that satisfy the predicate Φ. For example:

$$
\begin{aligned}
\langle\, x{:}\mathit{Bool} \mid \checkmark \,\rangle \quad &\text{denotes} \quad \{\bot, \mathit{True}, \mathit{False}\} \\
\langle\, x{:}\mathit{Bool} \mid x \not\approx \bot \,\rangle \quad &\text{denotes} \quad \{\mathit{True}, \mathit{False}\} \\
\langle\, x{:}\mathit{Bool} \mid \mathit{True} \leftarrow x \,\rangle \quad &\text{denotes} \quad \{\mathit{True}\} \\
\langle\, mx{:}\mathit{Maybe\ Bool} \mid \mathit{Just}\ x \leftarrow mx, x \not\approx \bot \,\rangle \quad &\text{denotes} \quad \{\mathit{Just\ True}, \mathit{Just\ False}\}
\end{aligned}
$$

The syntax of Φ is given in Figure 3. It consists of a collection of literals φ, combined with conjunction and disjunction. Unconventionally, however, a literal may bind one or more variables, and those bindings are in scope in conjunctions to the right. This can readily be formalised by giving a type system for Φ, but we omit that here. The literal ✓ means "true", as illustrated above; while × means "false", so that $\langle\, \mathit{Gamma} \mid \times \,\rangle$ denotes ∅.

The uncovered set function $\mathcal{U}(\Theta, t)$, defined in Figure 5, computes a refinement type describing the values in Θ that are not covered by the guard tree t. It is defined by a simple recursive descent over the guard tree, using the operation $\Theta \mathbin{\dot\wedge} \varphi$ (also defined in Figure 5) to extend Θ with an extra literal φ.

While $\mathcal{U}$ finds a refinement type describing values that are *not* matched by a guard tree, the function $\mathcal{A}$ finds refinements describing values that *are* matched by a guard tree, or that cause matching to diverge. It does so by producing an *annotated tree*, whose syntax is given in Figure 3. An annotated tree has the same general structure as the guard tree from whence it came: in particular the top-to-bottom compositions ";" are in the same places. But in an annotated tree, each `Rhs` leaf is annotated with a refinement type describing the input values that will lead to that right-hand side; and each `Bang` node is annotated with a refinement type that describes the input values on which matching will diverge. Once again, $\mathcal{A}$ can be defined by a simple recursive descent over the guard tree (Figure 5), but note that the second equation uses $\mathcal{U}$ as an auxiliary function[1].

3.3 Reporting errors

The final step is to report errors. First, let us focus on reporting missing equations. Consider the following definition

data $T = A \mid B \mid C$
$f\ (\mathit{Just}\ A) = \mathit{True}$

If t is the guard tree obtained from f, the expression $\mathcal{U}(\langle\, x : \mathit{Maybe}\ T \mid \checkmark\,\rangle, t)$ will produce this refinement type describing values that are not matched:

$$\Theta_f = \langle\, x{:}\mathit{Maybe}\ T \mid x \not\approx \bot \wedge (x \not\approx \mathit{Just} \vee (\mathit{Just}\ y \leftarrow x \wedge y \not\approx \bot \wedge (y \not\approx A \vee (A \leftarrow y \wedge \times))))\,\rangle$$

But this is not very helpful to report to the user. It would be far preferable to produce one or more concrete *inhabitants* of Θ_f to report, something like this:

```
Missing equations for function 'f':
  f Nothing  = ...
  f (Just B) = ...
  f (Just C) = ...
```

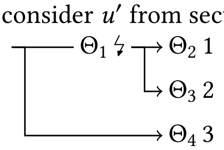
Producing these inhabitants is done by $\mathcal{G}(\Theta)$ in Figure 6, which we discuss next in Section 3.4. But before doing so, notice that the very same function $\mathcal{G}$ allows us to report accessible, inaccessible, and redundant GRHSs. The function $\mathcal{R}$, also defined in Figure 6 does exactly this, returning a triple of (accessible, inaccessible, redundant) GRHSs:

- Having reached a leaf `ARhs` $\Theta\ n$, if the refinement type Θ is uninhabited ($\mathcal{G}(\Theta) = \emptyset$), then no input values can cause execution to reach this right-hand side, and it is redundant.
- Having reached a node `Bang` $\Theta\ t$, if Θ is inhabited there is a possibility of divergence. Now suppose that all the GRHSs in t are redundant. Then we should pick the first of them and mark it as inaccessible.
- The case for $\mathcal{R}(t; u)$ is trivial: just combine the classifications of t and u.

To illustrate the second case consider u' from section 2.3.1 and its annotated tree:

$u'\ ()\mid \mathit{False} = 1$
$\quad\quad\ \mid \mathit{False} = 2$
$u'\ \_ \quad\quad\ = 3$

$\rightsquigarrow$



[1] Our implementation avoids this duplicated work – see Section 5.1 – but the formulation in Figure 5 emphasises clarity over efficiency.

Collect accessible ($\overline{k}$), inaccessible ($\overline{n}$) and redundant ($\overline{m}$) GRHSs

$$\boxed{\mathcal{R}(u) = (\overline{k}, \overline{n}, \overline{m})}$$

$$\mathcal{R}(\textsf{ARhs}\ \Theta\ n) = \begin{cases} (\epsilon, \epsilon, n), & \text{if } \mathcal{G}(\Theta) = \emptyset \\ (n, \epsilon, \epsilon), & \text{otherwise} \end{cases}$$

$$\mathcal{R}(t; u) = (\overline{k}\,\overline{k'}, \overline{n}\,\overline{n'}, \overline{m}\,\overline{m'}) \text{ where } \begin{array}{l} (\overline{k}, \overline{n}, \overline{m}) = \mathcal{R}(t) \\ (\overline{k'}, \overline{n'}, \overline{m'}) = \mathcal{R}(u) \end{array}$$

$$\mathcal{R}(\textsf{Bang}\ \Theta\ t) = \begin{cases} (\epsilon, m, \overline{m'}), & \text{if } \mathcal{G}(\Theta) \neq \emptyset \text{ and } \mathcal{R}(t) = (\epsilon, \epsilon, m\,\overline{m'}) \\ \mathcal{R}(t), & \text{otherwise} \end{cases}$$

Normalised refinement type syntax

∇	$::=$	$\times \mid \langle \Gamma \parallel \Delta \rangle$	Normalised refinement type
Δ	$::=$	$\varnothing \mid \Delta, \delta$	Set of constraints
δ	$::=$	$\gamma \mid x \approx K\ \overline{a}\ \overline{y} \mid x \not\approx K \mid x \approx \bot \mid x \not\approx \bot \mid x \approx y$	Constraints

Generate inhabitants of Θ

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

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

Construct inhabited ∇s from Φ

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

$$C(\nabla, \varphi) = \begin{cases} \{\langle \Gamma' \parallel \Delta' \rangle\} & \text{where } \langle \Gamma' \parallel \Delta' \rangle = \nabla \oplus_\varphi \varphi \\ \emptyset & \text{otherwise} \end{cases}$$

$$C(\nabla, \Phi_1 \wedge \Phi_2) = \bigcup \{ C(\nabla', \Phi_2) \mid \nabla' \in C(\nabla, \Phi_1) \}$$

$$C(\nabla, \Phi_1 \vee \Phi_2) = C(\nabla, \Phi_1) \cup C(\nabla, \Phi_2)$$

Expand variables to Pat with ∇

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

$$\mathcal{E}(\nabla, \epsilon) = \epsilon$$

$$\mathcal{E}(\langle \Gamma \parallel \Delta \rangle, x_1 ... x_n) = \begin{cases} (K\ q_1 ... q_m)\, p_2 ... p_n & \text{if } \Delta(x_1) \approx K\ \overline{a}\ \overline{y} \in \Delta \\ & \text{and } (q_1 ... q_m\, p_2 ... p_n) = \mathcal{E}(\langle \Gamma \parallel \Delta \rangle, y_1 ... y_m x_2 ... x_n) \\ \_\ p_2 ... p_n & \text{where } (p_2 ... p_n) = \mathcal{E}(\langle \Gamma \parallel \Delta \rangle, x_2 ... x_n) \end{cases}$$

Finding the representative of a variable in Δ

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

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

Fig. 6. Generating inhabitants of Θ via ∇

Θ_2 and Θ_3 are uninhabited (because of the *False* guards). But we cannot delete both GRHSs as redundant, because that would make the call $u' \perp$ return 3 rather than diverging. Rather, we want to report the first GRHSs as inaccessible, leaving all the others as redundant.

3.4 Generating inhabitants of a refinement type

Thus far, all our functions have been very simple, syntax-directed transformations, but they all ultimately depend on the single function $\mathcal{G}$, which does the real work. That is our new focus. As Figure 6 shows, $\mathcal{G}(\Theta)$ takes a refinement type $\Theta = \langle \Gamma \mid \Phi \rangle$ and returns a (possibly-empty) set of patterns $\overline{p}$ (syntax in Figure 3) that give the shape of values that inhabit Θ. We do this in two steps:

- Flatten Θ into a set of *normalised refinement types* ∇, by the call $\mathcal{C}(\langle \Gamma \parallel \varnothing \rangle, \Phi)$; see Section 3.6.
- For each such ∇, expand Γ into a list of patterns, by the call $\mathcal{E}(\nabla, \mathsf{dom}(\Gamma))$; see Section 3.5.

A normalised refinement type ∇ is either empty ($\times$) or of the form $\langle \Gamma \parallel \Delta \rangle$. It is similar to a refinement type $\Theta = \langle \Gamma \mid \Phi \rangle$, but is in a much more restricted form:

- Δ is simply a conjunction of literals δ; there are no disjunctions. Instead, disjunction reflects in the fact that $\mathcal{C}$ returns a *set* of normalised refinement types.

Beyond these syntactic differences, we enforce the following semantic invariants on a $\nabla = \langle \Gamma \parallel \Delta \rangle$:

- I1 *Mutual compatibility*: No two constraints in Δ should *conflict* with each other, where $x \approx \perp$ conflicts with $x \not\approx \perp$ and $x \approx K\ \_\ \_$ conflicts with $x \not\approx K$ for all x.
- I2 *Triangular form*: A $x \approx y$ constraint implies absence of any other constraints mentioning x in its left-hand side.
- I3 *Single solution*: There is at most one positive constructor constraint $x \approx K\ \overline{a}\ \overline{y}$ for a given x.
- I4 *Incompletely matched*: If $x{:}\tau \in \Gamma$ and τ reduces to a data type under type constraints in Δ, there must be at least one constructor K (or $\perp$) which x can be instantiated to without contradicting I1; see Section 3.7.

It is often helpful to think of a Δ as a partial function from x to its *solution*, informed by the single positive constraint $x \approx K\ \overline{a}\ \overline{y} \in \Delta$, if it exists. For example, $x \approx \mathit{Nothing}$ can be understood as a function mapping x to *Nothing*. This reasoning is justified by I3. Under this view, Δ looks like a substitution. As we'll see in section 3.6, this view is supported by a close correspondence with unification algorithms.

I2 is actually a condition on the represented substitution. Whenever we find out that $x \approx y$, for example when matching a variable pattern y against a match variable x, we have to merge all the other constraints on x into y, and say that y is the representative of x's equivalence class. This is so that every new constraint we record on y also affects x and vice versa. The process of finding the solution of x in $x \approx y, y \approx \mathit{Nothing}$ then entails *walking* the substitution, because we have to look up constraints twice: The first lookup will find x's representative y, the second lookup on y will then find the solution *Nothing*.

We use $\Delta(x)$ to look up the representative of x in Δ (see Figure 6). Therefore, we can assert that x has *Nothing* as a solution simply by writing $\Delta(x) \approx \mathit{Nothing} \in \Delta$.

3.5 Expanding a normalised refinement type to a pattern

Expanding a ∇ to a pattern vector, by calling $\mathcal{E}(\nabla)$ in Figure 6, is syntactically heavy, but straightforward. When there is a solution like $\Delta(x) \approx \mathit{Just}\ y$ in Δ for the head x of the variable vector of interest, expand y in addition to the rest of the vector and wrap it in a *Just*. Invariant I3 guarantees that there is at most one such solution and $\mathcal{E}$ is well-defined.

Add a formula literal to ∇ $\boxed{\nabla \oplus_\varphi \varphi = \nabla}$

$$
\begin{array}{rcll}
\nabla \oplus_\varphi \times & = & \times & (1)\\
\nabla \oplus_\varphi \checkmark & = & \nabla & (2)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\varphi K\ \overline{a}\ \overline{\gamma}\ \overline{y{:}\tau} \leftarrow x & = & \langle \Gamma, \overline{a}, \overline{y{:}\tau} \,\|\, \Delta \rangle \oplus_\delta \overline{\gamma} \oplus_\delta \overline{y' \not\approx \bot} \oplus_\delta x \approx K\ \overline{a}\ \overline{y} & (3)\\
 & & \quad \text{where } \overline{y'} \text{ bind strict fields} & \\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\varphi \text{let } x{:}\tau = K\ \overline{\sigma}\ \overline{\gamma}\ \overline{e} & = & \langle \Gamma, x{:}\tau, \overline{a} \,\|\, \Delta \rangle \oplus_\delta \overline{a \sim \sigma} \oplus_\delta x \approx K\ \overline{a}\ \overline{y} \oplus_\varphi \overline{\text{let } y{:}\tau' = e} & (4)\\
 & & \quad \text{where } \overline{a}\,\overline{y} \# \Gamma, \overline{e{:}\tau'} & \\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\varphi \text{let } x{:}\tau = y & = & \langle \Gamma, x{:}\tau \,\|\, \Delta \rangle \oplus_\delta x \approx y & (5)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\varphi \text{let } x{:}\tau = e & = & \langle \Gamma, x{:}\tau \,\|\, \Delta \rangle & (6)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\varphi \varphi & = & \langle \Gamma \,\|\, \Delta \rangle \oplus_\delta \varphi & (7)
\end{array}
$$

Add a constraint to ∇ $\boxed{\nabla \oplus_\delta \delta = \nabla}$

$$
\begin{array}{rcll}
\times \oplus_\delta \delta & = & \times & (8)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta \gamma & = & \begin{cases} \langle \Gamma \,\|\, (\Delta, \gamma) \rangle & \text{if type checker deems } \gamma \text{ compatible with } \Delta \\ & \text{and } \forall x \in \text{dom}(\Gamma) : \langle \Gamma \,\|\, (\Delta, \gamma) \rangle \vdash \Delta(x) \\ \times & \text{otherwise} \end{cases} & (9)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \approx K\ \overline{a}\ \overline{y} & = & \begin{cases} \langle \Gamma \,\|\, \Delta \rangle \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 \\ \langle \Gamma \,\|\, (\Delta, \Delta(x) \approx K\ \overline{a}\ \overline{y}) \rangle & \text{if } \Delta(x) \not\approx K \notin \Delta \\ \times & \text{otherwise} \end{cases} & (10)\\
\langle \Gamma \,\|\, \Delta \rangle \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 } \langle \Gamma \,\|\, (\Delta, \Delta(x) \not\approx K) \rangle \vdash \Delta(x) \\ \langle \Gamma \,\|\, (\Delta, \Delta(x) \not\approx K) \rangle & \text{otherwise} \end{cases} & (11)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \approx \bot & = & \begin{cases} \times & \text{if } \Delta(x) \not\approx \bot \in \Delta \\ \langle \Gamma \,\|\, (\Delta, \Delta(x) \approx \bot) \rangle & \text{otherwise} \end{cases} & (12)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \not\approx \bot & = & \begin{cases} \times & \text{if } \Delta(x) \approx \bot \in \Delta \\ \times & \text{if not } \langle \Gamma \,\|\, (\Delta, \Delta(x) \not\approx \bot) \rangle \vdash \Delta(x) \\ \langle \Gamma \,\|\, (\Delta, \Delta(x) \not\approx \bot) \rangle & \text{otherwise} \end{cases} & (13)\\
\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \approx y & = & \begin{cases} \langle \Gamma \,\|\, \Delta \rangle & \text{if } x' = y' \\ \langle \Gamma \,\|\, ((\Delta \setminus x'), x' \approx y') \rangle \oplus_\delta (\Delta|_{x'}\ [y'/x']) & \text{otherwise} \end{cases} & (14)\\
 & & \text{where } x' = \Delta(x) \text{ and } y' = \Delta(y) &
\end{array}
$$

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

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

$\boxed{\Delta|_x = \Delta}$

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

Fig. 7. Adding a constraint to the normalised refinement type ∇

3.6 Normalising a refinement type

Normalisation, carried out by $\mathcal{C}$ in Figure 6, is largely a matter of repeatedly adding a literal φ to a normalised type, thus $\nabla \oplus_\varphi \varphi$. This function is where all the work is done, in Figure 7. It does so by expressing a φ in terms of once again simpler constraints δ and calling out to $\oplus_\delta$. Specifically, in Equation (3) a pattern guard extends the context and adds suitable type constraints and a positive constructor constraint arising from the binding. Equation (4) of $\oplus_\varphi$ performs some limited, but important reasoning about let bindings: it flattens possibly nested constructor applications, such as let $x = \mathit{Just}\ \mathit{True}$. Note that equation (6) simply discards let bindings that cannot be expressed in ∇; we'll see an extension in section 4.3 that avoids this information loss.

That brings us to the prime unification procedure, $\oplus_\delta$. When adding $x \approx \mathit{Just}\ y$, equation (10), the unification procedure will first look for a solution for x with *that same constructor*. Let's say there is $\Delta(x) \approx \mathit{Just}\ u \in \Delta$. Then $\oplus_\delta$ operates on the transitively implied equality $\mathit{Just}\ y \approx \mathit{Just}\ u$ by equating type and term variables with new constraints, i.e. $y \approx u$. The original constraint, although not conflicting, is not added to the normalised refinement type because of I2.

If there is a solution involving a different constructor like $\Delta(x) \approx \mathit{Nothing}$ or if there was a negative constructor constraint $\Delta(x) \not\approx \mathit{Just}$, the new constraint is incompatible with the existing solution. Otherwise, the constraint is compatible and is added to Δ.

Adding a negative constructor constraint $x \not\approx \mathit{Just}$ is quite similar (equation (11)), except that we have to make sure that x still satisfies I4, which is checked by the $\nabla \vdash \Delta(x)$ judgment (cf. section 3.7) in fig. 8. Handling positive and negative constraints involving $\bot$ is analogous.

Adding a type constraint γ (equation (9)) entails calling out to the type checker to assert that the constraint is consistent with existing type constraints. Afterwards, we have to ensure I4 is upheld for *all* variables in the domain of Γ, because the new type constraint could have rendered a type empty. To demonstrate why this is necessary, imagine we have $\langle x : a \,\|\, x \not\approx \bot \rangle$ and try to add $a \sim \mathit{Void}$. Although the type constraint is consistent, x in $\langle x : a \,\|\, x \not\approx \bot, a \sim \mathit{Void} \rangle$ is no longer inhabited. There is room for being smart about which variables we have to re-check: For example, we can exclude variables whose type is a non-GADT data type.

Equation (14) of $\oplus_\delta$ equates two variables ($x \approx y$) by merging their equivalence classes. Consider the case where x and y aren't in the same equivalence class. Then $\Delta(y)$ is arbitrarily chosen to be the new representative of the merged equivalence class. To uphold I2, all constraints mentioning $\Delta(x)$ have to be removed and renamed in terms of $\Delta(y)$ and then re-added to Δ, one of which in turn might uncover a contradiction.

3.7 Testing for inhabitation

The process for adding a constraint to a normalised type above (which turned out to be a unification procedure in disguise) makes use of an *inhabitation test* $\nabla \vdash x$, depicted in fig. 8. This tests whether there are any values of x that satisfy ∇. If not, ∇ does not uphold I4. For example, the conjunction $x \not\approx \mathit{Just}, x \not\approx \mathit{Nothing}, x \not\approx \bot$ does not satisfy I4, because no value of x satisfies all those constraints.

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 ∇. Rule $\vdash$NoCpl will accept unconditionally when its type is not a data type, i.e. for $x : \mathit{Int} \to \mathit{Int}$.

Note that the outlined approach is complete in the sense that $\nabla \vdash x$ is derivable (if and) only if x is actually inhabited in ∇, because that means we don't have any ∇s floating around in the checking process that actually aren't inhabited and trigger false positive warnings. But that also means that the $\vdash$ relation is undecidable! Consider the following example:

Test if x is inhabited considering ∇ $\boxed{\nabla \vdash x}$

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

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

Find data constructors of τ $\boxed{\mathsf{Cons}(\langle\Gamma \,\|\, \Delta\rangle, \tau) = \overline{K}}$

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

Instantiate x to data constructor K $\boxed{\mathsf{Inst}(\nabla, x, K) = \nabla}$

$$\mathsf{Inst}(\langle\Gamma \,\|\, \Delta\rangle, x, K) = \langle\Gamma, \overline{a}, \overline{y : \sigma} \,\|\, \Delta\rangle \oplus_\delta \tau_x \sim \tau \oplus_\delta \overline{\gamma} \oplus_\delta x \approx K\ \overline{a}\ \overline{y} \oplus_\delta \overline{y' \not\approx \bot}$$
$$\text{where } K : \forall \overline{a}.\overline{\gamma} \Rightarrow \overline{\sigma} \rightarrow \tau,\ \overline{a}\ \overline{y} \,\#\, \Gamma,\ x : \tau_x \in \Gamma,\ \overline{y'} \text{ bind strict fields}$$

Fig. 8. Testing for inhabitation

data $T = \mathit{MkT}\ !T$
$f :: \mathit{SMaybe}\ T \rightarrow ()$
$f\ \mathit{SNothing} = ()$

This is exhaustive, because T is an uninhabited type. Upon adding the constraint $x \not\approx \mathit{SNothing}$ on the match variable x via $\oplus_\delta$, we perform an inhabitation test, which tries to instantiate the SJust constructor via $\vdash$Inst. That implies adding (via $\oplus_\delta$) the constraints $x \approx \mathit{SJust}\ y, y \not\approx \bot$, the latter of which leads to an inhabitation test on y. That leads to instantiation of the MkT constructor, which leads to constraints $y \approx \mathit{MkT}\ z, z \not\approx \bot$, and so on for z etc.. An infinite chain of fruitless instantiation attempts!

In practice, we implement a fuel-based approach that conservatively assumes that a variable is inhabited after n such iterations and consider supplementing that with a simple termination analysis in the future.

4 POSSIBLE EXTENSIONS

LYG is well equipped to handle the fragment of Haskell it was designed to handle. But GHC (and other languages, for that matter) extends Haskell in non-trivial ways. This section exemplifies easy accommodation of new language features and measures to increase precision of the checking process, demonstrating the modularity and extensibility of our approach.

4.1 Long-distance information

Coverage checking should also work for **case** expressions and nested function definitions, like

$f\ \mathit{True} = 1$
$f\ x \quad = \ldots(\textbf{case}\ x\ \textbf{of}\ \{\mathit{False} \rightarrow 2; \mathit{True} \rightarrow 3\})\ldots$

LYG as is will not produce any warnings for this definition. But the reader can easily make the "long distance connection" that the last GRHS of the **case** expression is redundant! That simply follows by context-sensitive reasoning, knowing that x was already matched against *True*.

In terms of LYG, the input values of the second GRHS Θ_2 (which determine whether the GRHS is accessible) encode the information we are after. We just have to start checking the **case** expression starting from Θ_2 as the initial set of reaching values instead of $\langle\, x : \mathit{Bool} \mid \checkmark \,\rangle$.

4.2 Empty case

As can be seen in fig. 1, Haskell function definitions need to have at least one clause. That leads to an awkward situation when pattern matching on empty data types, like *Void*:

$$
\begin{array}{ll}
\mathit{absurd1}\ \_\ = \bot & \mathit{absurd1}, \mathit{absurd2}, \mathit{absurd3} :: \mathit{Void} \longrightarrow a \\
\mathit{absurd2}\ !\_\ = \bot & \mathit{absurd3}\ x = \mathbf{case}\ x\ \mathbf{of}\ \{\,\}
\end{array}
$$

absurd1 returns $\bot$ when called with $\bot$, thus masking the original $\bot$ with the error thrown by $\bot$. *absurd2* would diverge alright, but LYG will report its RHS as inaccessible! Hence GHC provides an extension, called `EmptyCase`, that allows the definition of *absurd3* above. Such a **case** expression without any alternatives evaluates its argument to WHNF and crashes when evaluation returns.

It is quite easy to see that Gdt lacks expressive power to desugar `EmptyCase` into, since all leaves in a guard tree need to have corresponding RHSs. Therefore, we need to introduce `GEmpty` to Gdt and `AEmpty` to Ant. This is how they affect the checking process:

$$
\mathcal{U}(\Theta, \texttt{GEmpty}) = \Theta \qquad \mathcal{A}(\Theta, \texttt{GEmpty}) = \texttt{AEmpty}
$$

Since `EmptyCase`, unlike regular **case**, evaluates its scrutinee to WHNF *before* matching any of the patterns, the set of reaching values is refined with a $x \not\approx \bot$ constraint *before* traversing the guard tree, thus $\mathcal{U}(\langle\, \Gamma \mid x \not\approx \bot \,\rangle, \texttt{GEmpty})$.

4.3 View patterns

Our source syntax had support for view patterns to start with (cf. fig. 1). And even the desugaring we gave as part of the definition of $\mathcal{D}$ in fig. 4 is accurate. But this desugaring alone is insufficient for the checker to conclude that *safeLast* from section 2.2.1 is an exhaustive definition! To see why, let's look at its guard tree:

$$
\begin{array}{l}
\dashv \mathsf{let}\ y_1 = \mathit{reverse}\ x_1, !y_1, \mathit{Nothing} \leftarrow y_1 \longrightarrow 1 \\
\dashv \mathsf{let}\ y_2 = \mathit{reverse}\ x_1, !y_2, \mathit{Just}\ t_1 \leftarrow y_2, !t_1, (t_2, t_3) \leftarrow t_1 \longrightarrow 2
\end{array}
$$

As far as LYG is concerned, the matches on both y_1 and y_2 are non-exhaustive. But that's actually too conservative: Both bind the same value! By making the connection between y_1 and y_2, the checker could infer that the match was exhaustive.

This can be fixed by maintaining equivalence classes of semantically equivalent expressions in Δ, similar to what we already do for variables. We simply extend the syntax of δ and change the last **let** case of $\oplus_\varphi$. Then we can handle the new constraint in $\oplus_\delta$, as follows:

$$
\delta = \ldots \mid e \approx x \qquad \langle \Gamma \,\|\, \Delta \rangle \oplus_\varphi \mathsf{let}\ x : \tau = e = \langle \Gamma, x : \tau \,\|\, \Delta \rangle \oplus_\delta e \approx x
$$

$$
\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta e \approx x = \begin{cases} \langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \approx y, & \text{if } e' \approx y \in \Delta \text{ and } e \equiv_\Delta e' \\ \langle \Gamma \,\|\, \Delta, e \approx \Delta(x) \rangle, & \text{otherwise} \end{cases}
$$

Where $\equiv_\Delta$ is (an approximation to) semantic equivalence modulo substitution under Δ. A clever data structure is needed to answer queries of the form $e \approx \_ \in \Delta$, efficiently. In our implementation, we use a trie to index expressions rapidly and sacrifice reasoning modulo Δ in doing so. Plugging in an SMT solver to decide $\equiv_\Delta$ would be more precise, but certainly less efficient.

4.4 Pattern synonyms

To accommodate checking of pattern synonyms P, we first have to extend the source syntax and IR syntax by adding the syntactic concept of a *ConLike*:

$$\begin{array}{rcl} cl & ::= & K \mid P \\ pat & ::= & x \mid \_ \mid cl\ \overline{pat} \mid x@pat \mid \ldots \end{array} \qquad \begin{array}{rlcl} P \in & \text{PS} & & \\ C \in & \text{CL} & ::= & K \mid P \\ p \in & \text{Pat} & ::= & \_ \mid C\ \overline{p} \mid \ldots \end{array}$$

Assuming every definition encountered so far is changed to handle ConLikes C now instead of data constructors K, everything should work almost fine. Why then introduce the new syntactic variant in the first place? Consider

$pattern\ P = ()$
$pattern\ Q = ()$
$n = \mathbf{case}\ P\ \mathbf{of}\ Q \longrightarrow 1; P \longrightarrow 2$

Knowing that the definitions of P and Q completely overlap, we can see that the match on Q will cover all values that could reach P, so clearly P is redundant. A sound approximation to that would be not to warn at all. And that's reasonable, after all we established in section 2.2.2 that reasoning about pattern synonym definitions is undesirable.

But equipped with long-distance information from the scrutinee expression, the checker would mark the *first case alternative* as redundant, which clearly is unsound! Deleting the first alternative would change its semantics from returning 1 to returning 2. In general, we cannot assume that arbitrary pattern synonym definitions are disjoint, in stark contrast to data constructors.

The solution is to tweak the clause of $\oplus_\delta$ dealing with positive ConLike constraints $x \approx C\ \overline{a}\ \overline{y}$:

$$\langle \Gamma \,\|\, \Delta \rangle \oplus_\delta x \approx C\ \overline{a}\ \overline{y} \quad = \quad \begin{cases} \langle \Gamma \,\|\, \Delta \rangle \oplus_\delta \overline{a \sim b} \oplus_\delta \overline{y \approx z} & \text{if } \Delta(x) \approx C\ \overline{b}\ \overline{z} \in \Delta \\ \times & \text{if } \Delta(x) \approx C'\ \overline{b}\ \overline{z} \in \Delta \text{ and } C \cap C' = \emptyset \\ \langle \Gamma \,\|\, (\Delta, \Delta(x) \approx C\ \overline{a}\ \overline{y}) \rangle & \text{if } \Delta(x) \not\approx C \notin \Delta \text{ and } \overline{\langle \Gamma \,\|\, \Delta \rangle \vdash \Delta(y)} \\ \times & \text{otherwise} \end{cases}$$

Where the suggestive notation $C \cap C' = \emptyset$ is only true if C and C' don't overlap, if both are data constructors, for example.

Note that the slight relaxation means that the constructed ∇ might violate $I3$, specifically when $C \cap C' \neq \emptyset$. In practice that condition only matters for the well-definedness of $\mathcal{E}$, which in case of multiple solutions (i.e. $x \approx P, x \approx Q$) has to commit to one them for the purposes of reporting warnings. Fixing that requires a bit of boring engineering.

4.5 `COMPLETE` pragmas

In a sense, every algebraic data type defines its own builtin `COMPLETE` set, consisting of all its data constructors, so the coverage checker already manages a single `COMPLETE` set.

We have $\vdash$Inst from fig. 8 currently making sure that this `COMPLETE` set is in fact inhabited. We also have $\vdash$NoCpl that handles the case when we can't find *any* `COMPLETE` set for the given type (think $x : Int \longrightarrow Int$). The obvious way to generalise this is by looking up all `COMPLETE` sets attached to a type and check that none of them is completely covered:

$$\frac{(\langle \Gamma \parallel \Delta \rangle \oplus_\delta x \approx \bot) \neq \times}{\langle \Gamma \parallel \Delta \rangle \vdash x} \vdash\textsc{Bot} \qquad \frac{x : \tau \in \Gamma \quad \mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \overline{C_1, ..., C_{n_i}}^i \qquad \overline{\mathsf{Inst}(\langle \Gamma \parallel \Delta \rangle, x, C_j) \neq \times}^i}{\langle \Gamma \parallel \Delta \rangle \vdash x} \vdash\textsc{Inst}$$

$$\mathsf{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \begin{cases} \overline{C_1, ..., C_{n_i}}^i & \tau = T\ \overline{\sigma} \text{ and } T \text{ type constructor with COMPLETE sets } \overline{C_1, ..., C_{n_i}}^i \\ & \text{(after normalisation according to the type constraints in } \Delta) \\ \epsilon & \text{otherwise} \end{cases}$$

Cons was changed to return a list of all available `COMPLETE` sets, and ⊢Inst tries to find an inhabiting ConLike in each one of them in turn. Note that ⊢NoCpl is gone, because it coincides with ⊢Inst for the case where the list returned by Cons was empty. The judgment has become simpler and and more general at the same time! Note that checking against multiple `COMPLETE` sets so frequently is computationally intractable. We will worry about that in section 5.

4.6 Other extensions

We consider further extensions, including overloaded literals, newtypes, and a strict-by-default source syntax, in Appendix A.

5 IMPLEMENTATION

The implementation of LYG in GHC accumulates quite a few tricks that go beyond the pure formalism. This section is dedicated to describing these.

Warning messages need to reference source syntax in order to be comprehensible by the user. At the same time, coverage checks involving GADTs need a type checked program, so the only reasonable design is to run the coverage checker between type checking and desugaring to GHC Core, a typed intermediate representation lacking the connection to source syntax. We perform coverage checking in the same tree traversal as desugaring.

5.1 Interleaving $\mathcal{U}$ and $\mathcal{A}$

The set of reaching values is an argument to both $\mathcal{U}$ and $\mathcal{A}$. Given a particular set of input values and a guard tree, one can see by a simple inductive argument that both $\mathcal{U}$ and $\mathcal{A}$ are always called at the same arguments! Hence for an implementation it makes sense to compute both results together, if only for not having to recompute the results of $\mathcal{U}$ again in $\mathcal{A}$.

But there's more: Looking at the last clause of $\mathcal{U}$ in fig. 5, we can see that we syntactically duplicate Θ every time we have a pattern guard. That can amount to exponential growth of the refinement predicate in the worst case and for the time to prove it empty!

What we really want is to summarise a Θ into a more compact canonical form before doing these kinds of *splits*. But that's exactly what ∇ is! Therefore, in our implementation we don't pass around and annotate refinement types, but the result of calling $\mathcal{C}$ on them directly.

You can see the resulting definition in fig. 9. The readability is clouded by unwrapping of pairs. $\mathcal{UA}$ requires that each ∇ individually is non-empty, i.e. not $\times$. This invariant is maintained by adding φ constraints through $\dot{\oplus}_\varphi$, which filters out any ∇ that would become empty.

5.2 Throttling for graceful degradation

Even with the tweaks from section 5.1, checking certain pattern matches remains NP-hard Sekar et al. [1995]. Naturally, there will be cases where we have to conservatively approximate in order

$$\boxed{\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} \varphi = \overline{\nabla}}$$

$$\begin{array}{lcl}
\epsilon \mathbin{\dot{\oplus}_\varphi} \varphi & = & \epsilon \\
(\nabla_1 \ldots \nabla_n) \mathbin{\dot{\oplus}_\varphi} \varphi & = & \begin{cases} (\langle \Gamma \parallel \Delta \rangle)\,(\nabla_2 \ldots \nabla_n \mathbin{\dot{\oplus}_\varphi} \varphi) & \text{if } \langle \Gamma \parallel \Delta \rangle = \nabla \oplus_\varphi \varphi \\ (\nabla_2 \ldots \nabla_n) \mathbin{\dot{\oplus}_\varphi} \varphi & \text{otherwise} \end{cases}
\end{array}$$

$$\boxed{\mathcal{UA}(\overline{\nabla}, t) = (\overline{\nabla}, \mathsf{Ant})}$$

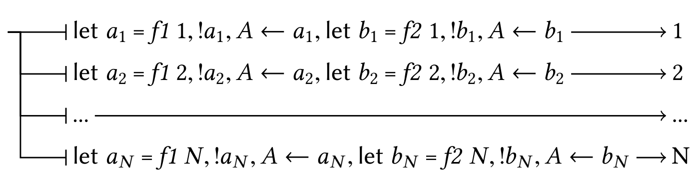
$$\begin{array}{lcl}
\mathcal{UA}(\overline{\nabla}, \mathsf{GRhs}\ n) & = & (\epsilon, \mathsf{ARhs}\ \overline{\nabla}\ n) \\
\mathcal{UA}(\overline{\nabla}, t_1; t_2) & = & (\overline{\nabla}_2, u_1; u_2) \ \text{where} \ \begin{array}{l} (\overline{\nabla}_1, u_1) = \mathcal{UA}(\overline{\nabla}, t_1) \\ (\overline{\nabla}_2, u_2) = \mathcal{UA}(\overline{\nabla}_1, t_2) \end{array} \\
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (!x)\ t) & = & \mathsf{Bang}\ (\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \approx \bot))\ u \\
 & & \quad \text{where } (\overline{\nabla}', u) = \mathcal{UA}(\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \not\approx \bot), t) \\
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (\mathsf{let}\ x = e)\ t) & = & \mathcal{UA}(\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (\mathsf{let}\ x = e), t) \\
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t) & = & ((\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \not\approx K))\ \overline{\nabla}', u) \\
 & & \quad \text{where } (\overline{\nabla}', u) = \mathcal{UA}(\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t)
\end{array}$$

Fig. 9. Fast coverage checking

not to slow down compilation too much. Consider the following example and its corresponding guard tree:

data $T = A \mid B; f1, f2 :: Int \longrightarrow T$

$g\ \_$
$\quad \mid A \leftarrow f1\ 1,\ \ A \leftarrow f2\ 1\ \ = ()$
$\quad \mid A \leftarrow f1\ 2,\ \ A \leftarrow f2\ 2\ \ = ()$
$\quad \ldots$
$\quad \mid A \leftarrow f1\ N, A \leftarrow f2\ N = ()$



Each of the N GRHS can fall through in two distinct ways: By failure of either pattern guard involving *f1* or *f2*. Initially, we start out with a single input ∇. After the first equation it will split into two sub-∇s, after the second into four, and so on. This exponential pattern repeats N times, and leads to horrible performance!

Instead of *refining* ∇ with the pattern guard, leading to a split, we could just continue with the original ∇, thus forgetting about the $a_1 \not\approx A$ or $b_1 \not\approx A$ constraints. In terms of the modeled refinement type, ∇ is still a superset of both refinements, and thus a sound overapproximation.

In our implementation, we call this *throttling*: We limit the number of reaching ∇s to a constant. Whenever a split would exceed this limit, we continue with the original input ∇s, a conservative estimate, instead. Intuitively, throttling corresponds to *forgetting* what we matched on in that particular subtree. Throttling is refreshingly easy to implement! Only the last clause of $\mathcal{UA}$, where splitting is performed, needs to change:

$$\begin{array}{l}
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t) = (\left\lfloor (\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (x \not\approx K))\ \overline{\nabla}' \right\rfloor_{\overline{\nabla}}, u) \\
\qquad\qquad\qquad\qquad \text{where } (\overline{\nabla}', u) = \mathcal{UA}(\overline{\nabla} \mathbin{\dot{\oplus}_\varphi} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t)
\end{array}$$

where the new throttling operator $\lfloor \_ \rfloor_{\_}$ is defined simply as

$$\left\lfloor \overline{\nabla} \right\rfloor_{\overline{\nabla}'} = \begin{cases} \overline{\nabla} & \text{if } |\{\overline{\nabla}\}| \leqslant K \\ \overline{\nabla}' & \text{otherwise} \end{cases}$$

with K being an arbitrary constant. We use 30 as an arbitrary limit in our implementation (dynamically configurable via a command-line flag) without noticing any false positives in terms of exhaustiveness warnings outside of the test suite.

5.3 Maintaining residual `COMPLETE` sets

Our implementation tries hard to make the inhabitation test as efficient as possible. For example, we represent Δs by a mapping from variables to their positive and negative constraints for easier indexing. But there are also asymptotical improvements. Consider the following function:

```
data T = A1 | ... | A1000          f A1    = 1
pattern P = ...                    f A2    = 2
 {-# COMPLETE A1, P #-}              ...
                                   f A1000 = 1000
```

f is exhaustively defined. To see that we need to perform an inhabitation test for the match variable x after the last clause. The test will conclude that the builtin `COMPLETE` set was completely overlapped. But in order to conclude that, our algorithm tries to instantiate x (via $\vdash$Inst) to each of its 1000 constructors and try to add a positive constructor constraint! What a waste of time, given that we could just look at the negative constraints on x *before* trying to instantiate x. But asymptotically it shouldn't matter much, since we're doing this only once at the end.

Except that is not true, because we also perform redundancy checking! At any point in f's definition there might be a match on P, after which all remaining clauses would be redundant by the user-supplied `COMPLETE` set. Therefore, we have to perform the expensive inhabitation test *after every clause*, involving $O(n)$ instantiations each.

Clearly, we can be smarter about that! Indeed, we cache *residual* `COMPLETE` *sets* in our implementation: Starting from the full `COMPLETE` sets, we delete ConLikes from them whenever we add a new negative constructor constraint, maintaining the invariant that each of the sets is inhabited by at least one constructor. Note how we never need to check the same constructor twice (except after adding new type constraints), thus we have an amortised $O(n)$ instantiations for the whole checking process.

5.4 Reporting uncovered patterns

The expansion function $\mathcal{E}$ in fig. 6 exists purely for presenting uncovered patterns to the user. It is very simple and doesn't account for negative information, leading to surprising warnings. Consider a definition like $f\ \mathit{True} = ()$. The computed uncovered set of f is the refinement type $\langle x : \mathit{Bool} \mid x \not\approx \bot, x \not\approx \mathit{True} \rangle$, which crucially contains no positive information! As a result, expanding the resulting ∇ (which looks quite similar) with $\mathcal{E}$ just unhelpfully reports _ as an uncovered pattern. Our implementation thus splits the ∇ into (possibly multiple) sub-∇s with positive information on variables we have negative information on before handing off to $\mathcal{E}$.

6 EVALUATION

We have implemented LYG in a to-be-released version of GHC. To put the new coverage checker to the test, we performed a survey of real-world Haskell code using the `head.hackage` repository [2].

[2] https://gitlab.haskell.org/ghc/head.hackage/commit/30a310fd8033629e1cbb5a9696250b22db5f7045

	Time (milliseconds)			Megabytes allocated		
	8.8.3	HEAD	% change	8.8.3	HEAD	% change
T11276	1.16	1.69	45.7%	1.86	2.39	28.6%
T11303	28.1	18.0	-36.0%	60.2	39.9	-33.8%
T11303b	1.15	0.39	-65.8%	1.65	0.47	-71.8%
T11374	4.62	3.00	-35.0%	6.16	3.20	-48.1%
T11822	1,060	16.0	-98.5%	2,010	27.9	-98.6%
T11195	2,680	22.3	-99.2%	3,080	39.5	-98.7%
T17096	7,470	16.6	-99.8%	17,300	35.4	-99.8%
PmSeriesS	44.5	2.58	-94.2%	52.9	6.19	-88.3%
PmSeriesT	48.3	6.86	-85.8%	61.4	17.6	-71.4%
PmSeriesV	131	4.54	-96.5%	139	9.53	-93.2%

Fig. 10. The relative compile-time performance of GHC 8.8.3 (which implements GMTM) and HEAD (which implements LYG) on test cases designed to stress-test coverage checking.

`head.hackage` contains a sizable collection of libraries and minimal patches necessary to make them build with a development version of GHC. We identified those libraries which compiled without coverage warnings using GHC 8.8.3 (which uses GMTM as its checking algorithm) but emitted warnings when compiled using our LYG version of GHC.

Of the 361 libraries in `head.hackage`, seven of them revealed coverage issues that only LYG warned about. Two of the libraries, `pandoc` and `pandoc-types`, have cases that were flagged as redundant due to LYG's improved treatment of guards and term equalities. One library, `geniplate-mirror`, has a case that was redundant by way of long-distance information. Another library, `generic-data`, has a case that is redundant due to bang patterns.

The last three libraries—`Cabal`, `HsYAML`, and `network`—were the most interesting. `HsYAML` in particular defines this function:

$go' \_ \_ \_ \; xs \mid False = error \; (show \; xs)$
$go' \_ \_ \_ \; xs = err \; xs$

The first clause is clearly unreachable, and LYG now flags it as such. However, the authors of `HsYAML` likely left in this clause because it is useful for debugging purposes. One can uncomment the second clause and remove the *False* guard to quickly try out a code path that prints a more detailed error message. Moreover, leaving the first clause in the code ensures that it is typechecked and less susceptible to bitrotting over time.

We may consider adding a primitive function *keepAlive* such that *keepAlive False* does not get marked as redundant in order to support use cases like `HsYAML`'s. The unreachable code in `Cabal` and `network` is of a similar caliber and would also benefit from *keepAlive*.

6.1 Performance tests

To compare the efficiency of GMTM and LYG quantitatively, we collected a series of test cases from GHC's test suite that are designed to test the compile-time performance of coverage checking. Figure 10 lists each of these 11 test cases. Test cases with a `T` prefix are taken from user-submitted bug reports about the poor performance of GMTM. Test cases with a `PmSeries` prefix are adapted from Maranget [2007], which presents several test cases that caused GHC to exhibit exponential running times during coverage checking.

We compiled each test case with GHC 8.8.3, which uses GMTM as its checking algorithm, and GHC HEAD, which uses LYG. We measured (1) the time spent in the desugarer, the phase of compilation in which coverage checking occurs, and (2) how many megabytes were allocated during desugaring. Figure 10 shows these figures as well as the percent change going from 8.8.3 to HEAD. Most cases exhibit a noticeable improvement under LYG, with the exception of T11276. Investigating T11276 suggests that the performance of GHC's equality constraint solver has become more expensive in HEAD [GHC issue 2020c], and these extra costs outweigh the performance benefits of using LYG.

6.2 GHC issues

Implementing LYG in GHC has fixed over 30 bug reports related to coverage checking. These include:

- Better compile-time performance [GHC issue 2015a, 2016e, 2019a,b]
- More accurate warnings for empty **case** expressions [GHC issue 2015b, 2017f, 2018e,g, 2019c]
- More accurate warnings due to LYG's desugaring [GHC issue 2016c,d, 2017d, 2018a, 2020d]
- More accurate warnings due to improved term-level reasoning [GHC issue 2016a, 2017a, 2018b,c,d,h, 2019d,e,h]
- More accurate warnings due to tracking long-distance information [GHC issue 2019k, 2020a,b]
- Improved treatment of COMPLETE sets [GHC issue 2016b, 2017b,c,e,g, 2018j, 2019f,g,i]
- Better treatment of strictness, bang patterns, and newtypes [GHC issue 2018f,i, 2019j,l]

7 RELATED WORK

7.1 Comparison with GADTs Meet Their Match

Karachalias et al. [2015] present GADTs Meet Their Match (GMTM), an algorithm which handles many of the subtleties of GADTs, guards, and laziness mentioned in section 2. Despite this, the GMTM algorithm still gives incorrect warnings in many cases.

7.1.1 GMTM does not consider laziness in its full glory. The formalism in Karachalias et al. [2015] incorporates strictness constraints, but these constraints can only arise from matching against data constructors. GMTM does not consider strict matches that arise from strict fields of data constructors or bang patterns. A consequence of this is that GMTM would incorrectly warn that v (section 2.3) is missing a case for *SJust*, even though such a case is unreachable. LYG, on the other hand, more thoroughly tracks strictness when desugaring Haskell programs.

7.1.2 GMTM's treatment of guards is shallow. GMTM can only reason about guards through an abstract term oracle. Although the algorithm is parametric over the choice of oracle, in practice the implementation of GMTM in GHC uses an extremely simple oracle that can only reason about guards in a limited fashion. More sophisticated uses of guards, such as in this variation of the *safeLast* function from section 2.2.1, will cause GMTM to emit erroneous warnings:

$$\begin{aligned}
&\mathit{safeLast2}\ \mathit{xs}\\
&\quad \mid (x : _) \leftarrow \mathit{reverse}\ \mathit{xs} = \mathit{Just}\ x\\
&\quad \mid [\,] \qquad\ \leftarrow \mathit{reverse}\ \mathit{xs} = \mathit{Nothing}
\end{aligned}$$

While GMTM's term oracle is customisable, it is not as simple to customize as one might hope. The formalism in Karachalias et al. [2015] represents all guards as $p \leftarrow e$, where p is a pattern and e is an expression. This is a straightforward, syntactic representation, but it also makes it more difficult to analyse when e is a complicated expression. This is one of the reasons why it is difficult for GMTM to accurately give warnings for the *safeLast* function, since it would require recognizing that both clauses scrutinise the same expression in their view patterns.

LYG makes analysing term equalities simpler by first desugaring guards from the surface syntax to guard trees. The $\oplus_\varphi$ function, which is roughly a counterpart to GMTM's term oracle, can then reason about terms arising from patterns. While $\oplus_\varphi$ is already more powerful than a trivial term oracle, its real strength lies in the fact that it can easily be extended, as LYG's treatment of view patterns (section 4.3) demonstrates. While GMTM's term oracle could be improved to accomplish the same thing, it is unlikely to be as straightforward of a process as extending $\oplus_\varphi$.

7.2 Comparison with similar coverage checkers

7.2.1 Structural and semantic pattern matching analysis in Haskell. Kalvoda and Kerckhove [2019] implement a variation of GMTM that leverages an SMT solver to give more accurate coverage warnings for programs that use guards. For instance, their implementation can conclude that the *signum* function from section 2.1 is exhaustive. This is something that LYG cannot do out of the box, although it would be possible to extend $\oplus_\varphi$ with SMT-like reasoning about booleans and linear integer arithmetic.

7.2.2 Warnings for pattern matching. Maranget [2007] presents a coverage checking algorithm for OCaml. While OCaml is a strict language, the algorithm claims to be general enough to handle languages with non-strict semantics such as Haskell. That claim however builds on a broken understanding of laziness. Given the following definition:

$$
\begin{aligned}
&f\ \mathit{True} = 1\\
&f\ \_ \quad\ \ = 2
\end{aligned}
$$

Maranget implies that $f\ \bot$ evaluates to 2, which is of course incorrect. Also, replacing the wild card by a match on *False* would no longer be a complete match according to their formalism.

7.2.3 Elaborating dependent (co)pattern matching. Cockx and Abel [2018] design a coverage checking algorithm for a dependently typed language with both pattern matching and *copattern* matching, which is a feature that GHC lacks. While the source language for their algorithm is much more sophisticated than GHC's, their algorithm is similar to LYG in that it first desugars definitions by clauses to *case trees*. Case trees present a simplified form of pattern matching that is easier to check for coverage, much like guard trees in LYG. Guard trees could take inspiration from case trees should a future version of GHC add dependent types or copatterns.

7.3 Positive and negative information

LYG's use of positive and negative constructor constraints is inspired by Sestoft [1996], which uses positive and negative information to implement a pattern-match compiler for ML. Sestoft utilises positive and negative information to generate decision trees that avoid scrutinizing the same terms repeatedly. This insight is equally applicable to coverage checking and is one of the primary reasons for LYG's efficiency.

Besides efficiency, the accuracy of redundancy warnings involving `COMPLETE` sets hinge on negative constraints. To see why this isn't possible in other checkers that only track positive information, such as those of Karachalias et al. [2015] (section 7.1) and Maranget [2007] (section 7.2.2), consider the following example:

$$\mathit{pattern}\ \mathit{True}' = \mathit{True}$$
{-# COMPLETE True', False #-}

$$
\begin{aligned}
&f\ \mathit{False}\ = 1\\
&f\ \mathit{True}' = 2\\
&f\ \mathit{True}\ \ = 3
\end{aligned}
$$

GMTM would have to commit to a particular `COMPLETE` set when encountering the match on *False*, without any semantic considerations. Choosing $\{\mathit{True}', \mathit{False}\}$ here will mark the third GRHS as redundant, while choosing $\{True, False\}$ won't. GHC's implementation used to try each `COMPLETE` set in turn and would disambiguate using a complicated metric based on the number and kinds of warnings the choice of each oset would generate [GHC team 2020], which was broken still [GHC issue 2017g].

Negative constraints make LYG efficient in other places too, such as in this example:

$$\textbf{data}\ T = A1 \mid \ldots \mid A1000 \qquad\qquad \begin{aligned} h\ A1\ \_\ &= 1 \\ h\ \_\ \ A1 &= 2 \end{aligned}$$

In h, GMTM would split the value vector (which is like LYG's Δs without negative constructor constraints) into 1000 alternatives over the first match variable, and then *each* of the 999 value vectors reaching the second GRHS into another 1000 alternatives over the second match variable. Negative constraints allow LYG to compress the 999 value vectors falling through into a single one indicating that the match variable can no longer be $A1$. Maranget detects wildcard matches to prevent blowup, but only can find a subset of all uncovered patterns in doing so (section 7.2.2).

7.4 Strict fields in inhabitation testing

To our knowledge, the Inst function in fig. 8 is the first inhabitation test in a coverage checking algorithm to take strict fields into account. This is essential in order to conclude that the v function from section 2.3 is exhaustive, which is something that even coverage checkers for call-by-value languages get wrong. For example, we ported v to OCaml and Idris [3]:

$$\begin{aligned} &\textbf{type}\ void;; \\ &\textbf{let}\ v\ (None : void\ option) : int = 0;; \end{aligned} \qquad\qquad \begin{aligned} &v : Maybe\ Void \rightarrow Int \\ &v\ Nothing = 0 \end{aligned}$$

OCaml 4.07.1 incorrectly warns that v is missing a case on $Some\ \_$. Idris 1.3.2 does not warn, but if one adds an extra $v\ (Just\ \_) = 1$ clause, it will not warn that the extra clause is redundant.

7.5 Refinement types in coverage checking

In addition to LYG, Liquid Haskell uses refinement types to perform a limited form of exhaustivity checking [Vazou et al. 2014, 2017]. While exhaustiveness checks are optional in ordinary Haskell, they are mandatory for Liquid Haskell, as proofs written in Liquid Haskell require user-defined functions to be total (and therefore exhaustive) in order to be sound. For example, consider this non-exhaustive function:

$$\begin{aligned} &fibPartial :: Integer \rightarrow Integer \\ &fibPartial\ 0 = 0 \\ &fibPartial\ 1 = 1 \end{aligned}$$

When compiled, GHC fills out this definition by adding an extra $fibPartial\ \_ = error$ `"undefined"` clause. Liquid Haskell leverages this by giving *error* the refinement type:

$$error :: \{\, v : String \mid false \,\} \rightarrow a$$

As a result, attempting to use *fibPartial* in a proof will yield an inconsistent environment (and therefore fail to verify) unless the user can prove that *fibPartial* is only ever invoked with the arguments 0 or 1.

8 CONCLUSION

In this paper, we describe Lower Your Guards, a coverage checking algorithm that distills rich pattern matching into simple guard trees. Guard trees are amenable to analyses that are not easily

[3] Idris has separate compile-time and runtime semantics, the latter of which is call by value.

expressible in coverage checkers that work over structural pattern matches. This allows LYG to report more accurate warnings while also avoiding performance issues when checking complex programs. Moreover, LYG is extensible, and we anticipate that this will streamline the process of checking new forms of patterns in the future.

REFERENCES

Jesper Cockx and Andreas Abel. 2018. Elaborating Dependent (Co)Pattern Matching. *Proc. ACM Program. Lang.* 2, ICFP, Article Article 75 (July 2018), 30 pages. https://doi.org/10.1145/3236770

Joshua Dunfield. 2007. *A Unified System of Type Refinements*. Ph.D. Dissertation. Carnegie Mellon University. CMU-CS-07-129.

Jacques Garrigue and Jacques Le Normand. 2011. Adding GADTs to OCaml: the direct approach. In *Workshop on ML*.

GHC issue. 2015a. New pattern-match check can be non-performant. https://gitlab.haskell.org/ghc/ghc/issues/11195

GHC issue. 2015b. No non-exhaustive pattern match warning given for empty case analysis. https://gitlab.haskell.org/ghc/ghc/issues/10746

GHC issue. 2016a. In a record-update construct:ghc-stage2: panic! (the 'impossible' happened). https://gitlab.haskell.org/ghc/ghc/issues/12957

GHC issue. 2016b. Inaccessible RHS warning is confusing for users. https://gitlab.haskell.org/ghc/ghc/issues/13021

GHC issue. 2016c. Pattern coverage checker ignores dictionary arguments. https://gitlab.haskell.org/ghc/ghc/issues/12949

GHC issue. 2016d. Pattern match incompleteness / inaccessibility discrepancy. https://gitlab.haskell.org/ghc/ghc/issues/11984

GHC issue. 2016e. Representation of value set abstractions as trees causes performance issues. https://gitlab.haskell.org/ghc/ghc/issues/11528

GHC issue. 2017a. -Woverlapping-patterns warns on wrong patterns for Int. https://gitlab.haskell.org/ghc/ghc/issues/14546

GHC issue. 2017b. COMPLETE sets don't work at all with data family instances. https://gitlab.haskell.org/ghc/ghc/issues/14059

GHC issue. 2017c. COMPLETE sets nerf redundant pattern-match warnings. https://gitlab.haskell.org/ghc/ghc/issues/13965

GHC issue. 2017d. Incorrect pattern match warning on nested GADTs. https://gitlab.haskell.org/ghc/ghc/issues/14098

GHC issue. 2017e. Pattern match checker mistakenly concludes pattern match on pattern synonym is unreachable. https://gitlab.haskell.org/ghc/ghc/issues/14253

GHC issue. 2017f. Pattern synonym exhaustiveness checks don't play well with EmptyCase. https://gitlab.haskell.org/ghc/ghc/issues/13717

GHC issue. 2017g. Wildcard patterns and COMPLETE sets can lead to misleading redundant pattern-match warnings. https://gitlab.haskell.org/ghc/ghc/issues/13363

GHC issue. 2018a. -Wincomplete-patterns gets confused when combining GADTs and pattern guards. https://gitlab.haskell.org/ghc/ghc/issues/15385

GHC issue. 2018b. Bogus -Woverlapping-patterns warning with OverloadedStrings. https://gitlab.haskell.org/ghc/ghc/issues/15713

GHC issue. 2018c. Compiling a function with a lot of alternatives bottlenecks on insertIntHeap. https://gitlab.haskell.org/ghc/ghc/issues/14667

GHC issue. 2018d. Completeness of View Patterns With a Complete Set of Output Patterns. https://gitlab.haskell.org/ghc/ghc/issues/15884

GHC issue. 2018e. EmptyCase thinks pattern match involving type family is not exhaustive, when it actually is. https://gitlab.haskell.org/ghc/ghc/issues/14813

GHC issue. 2018f. Erroneous "non-exhaustive pattern match" using nested GADT with strictness annotation. https://gitlab.haskell.org/ghc/ghc/issues/15305

GHC issue. 2018g. Inconsistency w.r.t. coverage checking warnings for EmptyCase under unsatisfiable constraints. https://gitlab.haskell.org/ghc/ghc/issues/15450

GHC issue. 2018h. Inconsistent pattern-match warnings when using guards versus case expressions. https://gitlab.haskell.org/ghc/ghc/issues/15753

GHC issue. 2018i. nonVoid is too conservative w.r.t. strict argument types. https://gitlab.haskell.org/ghc/ghc/issues/15584

GHC issue. 2018j. "Pattern match has inaccessible right hand side" with TypeRep. https://gitlab.haskell.org/ghc/ghc/issues/14851

GHC issue. 2019a. 67-pattern COMPLETE pragma overwhelms the pattern match checker. https://gitlab.haskell.org/ghc/ghc/issues/17096

GHC issue. 2019b. Add Luke Maranget's series in "Warnings for Pattern Matching". https://gitlab.haskell.org/ghc/ghc/issues/17264

GHC issue. 2019c. `case (x :: Void) of _ -> ()` should be flagged as redundant. https://gitlab.haskell.org/ghc/ghc/issues/17376

GHC issue. 2019d. GHC thinks pattern match is exhaustive. https://gitlab.haskell.org/ghc/ghc/issues/16289

GHC issue. 2019e. Incorrect non-exhaustive pattern warning with PatternSynonyms. https://gitlab.haskell.org/ghc/ghc/issues/16129

GHC issue. 2019f. Minimality of missing pattern set depends on constructor declaration order. https://gitlab.haskell.org/ghc/ghc/issues/17386

GHC issue. 2019g. Panic during tyConAppArgs. https://gitlab.haskell.org/ghc/ghc/issues/17112

GHC issue. 2019h. Pattern-match checker: True /= False. https://gitlab.haskell.org/ghc/ghc/issues/17251

GHC issue. 2019i. Pattern match checking open unions. https://gitlab.haskell.org/ghc/ghc/issues/17149

GHC issue. 2019j. Pattern match overlap checking doesn't consider -XBangPatterns. https://gitlab.haskell.org/ghc/ghc/issues/17234

GHC issue. 2019k. Pattern match warnings are per Match, not per GRHS. https://gitlab.haskell.org/ghc/ghc/issues/17465

GHC issue. 2019l. PmCheck treats Newtype patterns the same as constructors. https://gitlab.haskell.org/ghc/ghc/issues/17248

GHC issue. 2020a. -Wincomplete-record-updates ignores context. https://gitlab.haskell.org/ghc/ghc/issues/17783

GHC issue. 2020b. Pattern match checker stumbles over reasonably tricky pattern-match. https://gitlab.haskell.org/ghc/ghc/issues/17703

GHC issue. 2020c. Pattern match coverage checker allocates twice as much for trivial program with instance constraint vs. without. https://gitlab.haskell.org/ghc/ghc/issues/17891

GHC issue. 2020d. Pattern match warning emitted twice. https://gitlab.haskell.org/ghc/ghc/issues/17646

GHC team. 2020. COMPLETE pragmas. https://downloads.haskell.org/~ghc/8.8.3/docs/html/users_guide/glasgow_exts.html#pragma-COMPLETE

Pavel Kalvoda and Tom Sydney Kerckhove. 2019. Structural and semantic pattern matching analysis in Haskell. arXiv:cs.PL/1909.04160

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

Luc Maranget. 2007. Warnings for pattern matching. *Journal of Functional Programming* 17 (2007), 387–421. Issue 3.

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

John Rushby, Sam Owre, and Natarajan Shankar. 1998. Subtypes for specifications: Predicate subtyping in PVS. *IEEE Transactions on Software Engineering* 24, 9 (1998), 709–720.

R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. 1995. Adaptive Pattern Matching. *SIAM J. Comput.* 24, 6 (Dec. 1995), 1207âĂŞ1234. https://doi.org/10.1137/S0097539793246252

Peter Sestoft. 1996. ML pattern match compilation and partial evaluation. In *Partial Evaluation*. Springer, 446–464.

Niki Vazou, Eric L. Seidel, Ranjit Jhala, Dimitrios Vytiniotis, and Simon Peyton-Jones. 2014. Refinement Types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming (ICFP '14)*. ACM, New York, NY, USA, 269–282. https://doi.org/10.1145/2628136.2628161

Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. 2017. Refinement Reflection: Complete Verification with SMT. *Proc. ACM Program. Lang.* 2, POPL, Article Article 53 (Dec. 2017), 31 pages. https://doi.org/10.1145/3158141

Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. Outsidein(x) Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

Hongwei Xi. 1998a. Dead Code Elimination Through Dependent Types. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages (PADL '99)*. Springer-Verlag, London, UK, 228–242.

Hongwei Xi. 1998b. *Dependent Types in Practical Programming*. Ph.D. Dissertation. Carnegie Mellon University.

Hongwei Xi. 2003. Dependently typed pattern matching. *Journal of Universal Computer Science* 9 (2003), 851–872.

Hongwei Xi, Chiyan Chen, and Gang Chen. 2003. Guarded Recursive Datatype Constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '03)*. ACM, New York, NY, USA, 224–235. https://doi.org/10.1145/604131.604150

Hongwei Xi and Frank Pfenning. 1998. Eliminating Array Bound Checking through Dependent Types. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation (PLDI '98)*. Association for Computing Machinery, New York, NY, USA, 249–257. https://doi.org/10.1145/277650.277732