Lower Your Guards

A Compositional Pattern-Match Coverage Checker

ANONYMOUS AUTHOR(S)

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 interaction of language features when pattern matching. This is especially true in Haskell, a language with a complicated pattern language that is made even more complicated 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.

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:

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

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 if a function is missing clauses (i.e., if it is *non-exhaustive*), if a function has completely overlapping clauses (i.e., if it is *redundant*), or if a function has a right-hand side that cannot be reached (i.e., if it is *inaccessible*). 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 8 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 contructors, empty case expressions, and long-distance effects (Section 5.1). Particularly tricky are 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

2020. 2475-1421/2020/1-ART1 $15.00
https://doi.org/

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. 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 underpin the intuitions of Section 3 with a detailed formal treatment in Section 4.
- We demonstrate the compositionality of LYG by augmenting it with several language extensions (Section 5). 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 6) and implement a proof of concept in GHC (Section 7).

We discuss the wealth of related work in Section 8.

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 8) 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 4 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' $\leftarrow c1$), 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. We explore the subset of guards that LYG can check in more detail in **Ryan:** Cite relevant section**SG:** I think that's mostly in Related Work? Not sure we give a detailed account anywhere.

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 succintly 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 char 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[1] . 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
```

[1] https://downloads.haskell.org/~ghc/8.8.3/docs/html/users_guide/glasgow_exts.html#pragma-COMPLETE

```
v SNothing = 0
v (SJust _) = 1    -- Redundant!
```

The "!" in the definition of *SJust* makes the constructor strict, so (*SJust* ⊥) = ⊥. Curiously, this makes the second equation of *v* redundant! Since ⊥ is the only inhabitant of type *Void*, the only inhabitants of *SMaybe Void* are *SNothing* and ⊥. 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 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:

```
v' :: Maybe Void → Int
v' Nothing = 0
v' (Just !_) = 1    -- Not redundant, but RHS is inaccessible
```

The inhabitants of the type *Mabye Void* are ⊥, *Nothing*, and (*Just* ⊥). The input ⊥ makes the first equation diverge; *Nothing* matches on the first equation; and (*Just* ⊥) makes the second equation diverge because of the bang pattern. None of the three cases reaches the right-hand side of the second equation, but neither is the second equation redundant, because removing it would mean that the input (*Just* ⊥) was matched by no equation. We say that the second equation is not redundant, but its right hand side is *inaccessible*. This may seem like a tricky corner case, but GHC's users reported many bugs of this sort (Section 7.2).

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:

```
data T a b where          g1 :: T Int b → b → Int      g2 :: T a b → T a b → Int
  T1 :: T Int Bool        g1 T1 False = 0              g2 T1 T1 = 0
  T2 :: T Char Bool       g1 T1 True = 1               g2 T2 T2 = 1
```

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

Meta variables

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

Pattern Syntax

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

Fig. 1. Source syntax

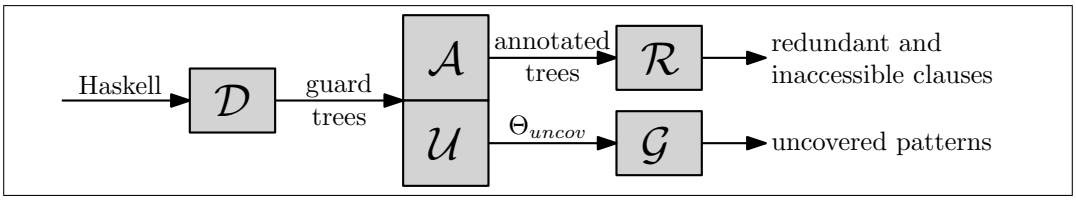


Fig. 2. Bird's eye view of pattern match checking

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. 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]. LYG continues this tradition—see **Ryan:** What section?**SG:** It's a little implicit at the moment, because it just works. Not sure what to reference here. for LYG's take on GADTs.

SLPJ: I deleted the soundness section: it didn't seem to pay its way

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.
- Next, the resulting guard tree is then processed by two different functions (Section 3.2). The function $\mathcal{A}(t)$ produces an *annotated tree* t_A : Ant, which has the same general branching structure as t but desribes 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 $\mathcal{A}$

Guard Syntax

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

Refinement Type Syntax

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

Clause Tree Syntax

$$
\begin{array}{rcl}
t_G, u_G \in \mathrm{Gdt} & ::= & \mathsf{GRhs}\ n \mid t_G; u_G \mid \mathsf{Guard}\ g\ t_G \\
t_A, u_A \in \mathrm{Ant} & ::= & \mathsf{ARhs}\ \Theta\ n \mid t_A; u_A \mid \mathsf{Bang}\ \Theta\ t_A
\end{array}
$$

Graphical Notation

$$
\begin{array}{rcl@{\qquad}rcl}
\begin{array}{l} t_G \\ \quad\big\vert \\ u_G \end{array} & ::= & t_G; u_G & \begin{array}{l} t_A \\ \quad\big\vert \\ u_A \end{array} & ::= & t_A; u_A \\
g_1, \ldots, g_n \text{ —— } t_G & ::= & \mathsf{Guard}\ g_1\ \ldots\ (\mathsf{Guard}\ g_n\ t_G) & \lightning \text{ —— } t_A & ::= & \mathsf{Bang}\ \_\ t_A \\
\longrightarrow n & ::= & \mathsf{GRhs}\ n & \longrightarrow n & ::= & \mathsf{ARhs}\ \_\ n
\end{array}
$$

Fig. 3. IR Syntax

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.

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 5.

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:

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

This desugars to the following guard tree:

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

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

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

$$\mathcal{D}(f\ pat_1 \ldots pat_n = expr) = \dashv \mathcal{D}(x_1, pat_1) \ldots \mathcal{D}(x_n, pat_n) \longrightarrow k$$

$$\mathcal{D}(f\ pat_1 \ldots pat_n\ grhs_1 \ldots grhs_m) = \dashv \mathcal{D}(x_1, pat_1) \ldots \mathcal{D}(x_n, pat_n) \begin{array}{l} \mathcal{D}(grhs_1) \\ \ldots \\ \mathcal{D}(grhs_m) \end{array}$$

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

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

Fig. 4. Desugaring from source language to Gdt

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

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_G; u_G) means matching against t_G; if that succeeds, the overall match succeeds; if not, match against u_G.
- Matching a guard tree (`Guard` $!x\ t_G$) evaluates x; if that diverges the match diverges; if not match t_G.
- Matching a guard tree (`Guard` $(K\ y_1 \ldots y_n \leftarrow x)\ t_G$) matches x against constructor K. If the match succeeds, bind $y_1 \ldots y_n$ to the components, and match t_G.
- Matching a guard tree (`Guard` $(\text{let } x = e)\ t_G$) binds x (lazily) to e, and matches t_G.

Operations on Θ

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

Checking Guard Trees

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

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

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

$$\begin{array}{lcl} \mathcal{A}(\Theta, \texttt{GRhs}\ n) & = & \texttt{ARhs}\ \Theta\ n \\ \mathcal{A}(\Theta, (t; u)) & = & \mathcal{A}(\Theta, t); \mathcal{A}(\mathcal{U}(\Theta, t), u) \\ \mathcal{A}(\Theta, \texttt{Guard}\ (!x)\ t) & = & \texttt{Bang}\ (\Theta \mathbin{\dot\wedge} (x \approx \bot))\ \mathcal{A}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), t) \\ \mathcal{A}(\Theta, \texttt{Guard}\ (\text{let } x = e)\ t) & = & \mathcal{A}(\Theta \mathbin{\dot\wedge} (\text{let } x = e), t) \\ \mathcal{A}(\Theta, \texttt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t) & = & \mathcal{A}(\Theta \mathbin{\dot\wedge} (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t) \end{array}$$

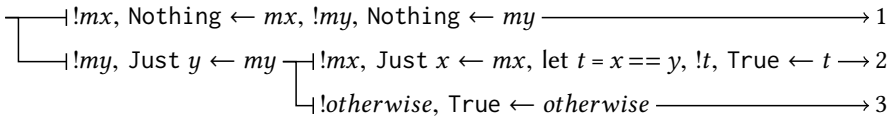
Fig. 5. Coverage checking

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. $g \to \mathit{True}$) 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:

$$\begin{array}{lll} \mathit{liftEq}\ \mathit{Nothing}\ \mathit{Nothing} & = & \mathit{True} \\ \mathit{liftEq}\ \mathit{mx}\quad (\mathit{Just}\ y) & \mid\ \mathit{Just}\ x \leftarrow \mathit{mx}, x == y & = \mathit{True} \\ & \mid\ \mathit{otherwise} & = \mathit{False} \end{array}$$

It desugars thus:



Notice that the pattern guard ($\mathit{Just}\ x \leftarrow \mathit{mx}$) 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:

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

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. **SLPJ:** It would be nice to add it. The literal $\checkmark$ means "true", as illustrated above; while $\times$ means "false", so that $\langle\, \mathit{Gamma} \mid \times \,\rangle = \emptyset$.

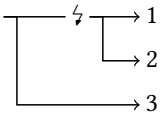
SLPJ: I'm unhappy with having to subscribe our trees with G all the time, thus t_G. Can we just us a different letter for guard trees and annotated trees? The uncovered-set function $\mathcal{U}(\Theta, t_G)$, defined in Figure 5, computes a refinement type describing the values in Θ that are not covered by the guard tree t_G. 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 sequential 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 `MayDiverge` 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[2].

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

```
g :: () → Int
g () | False = 1
     | True  = 2
g _          = 3
```

$\rightsquigarrow$



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

[2] Our implementaiton avoids this duplicated work – see Section 6.1 – but the fomulation in Figure 5 emphasises clarity over effiiency.

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

3.3 Reporting errors

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

$f :: \mathit{Maybe\ Int} \to \mathit{Bool}$
$f\ (\mathit{Just}\ 0) = \mathit{True}$

If t is the guard tree obtained from f, fhe function $\mathcal{U}(t)$ will produce a refinement type describing values that are not matched, something like this:

$$\mathcal{U}(t) = \Theta_f = \langle\, x{:}\mathit{Maybe\ Int} \mid x \not\approx \bot,\ \mathit{Just}\ y \leftarrow x,\ \text{let } b = (y == 0),\ b \not\approx \bot,\ \mathit{True} \leftarrow b \,\rangle$$

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

```
Missing equations for function 'f':
f Nothing = ...
f (Just y) = ..., where y /= 0
```

SLPJ: Working here

The predicate literals φ of refinement types look quite similar to the original Grd language, so how is checking them for emptiness an improvement over reasoning about about guard trees directly? To appreciate the transition, it is important to realise that semantics of Grds are *highly non-local*! Left-to-right and top-to-bottom match semantics means that it is hard to view Grds in isolation; we always have to reason about whole Gdts. By contrast, refinement types are self-contained, which means the process of generating inhabitants can be treated separately from the process of coverage checking.

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

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

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

Why do we have to test guard-bound variables in addition to the pattern variables? It is because of empty data types and strict fields. For example, *v* from section 2.3 does not have any uncovered patterns. And our approach should see that by looking at its uncovered set $\langle\, x : \mathit{Maybe\ Void} \mid x \not\approx \bot \wedge x \not\approx \texttt{Nothing} \,\rangle$. Specifically, the candidate *SJust y* (for fresh y) for x should be rejected, because there is no inhabitant for y! ⊥ is ruled out by the strict field and *Void* has no data constructors with which to instantiate y. Hence it is important to test guard-bound variables for inhabitants, too.

Collect accessible, inaccessible and redundant GRHSs

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

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

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

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

Inert Set Syntax

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
Δ	$::=$	$\varnothing \mid \Delta, \delta$	Set of constraints
∇	$::=$	$\times \mid \Gamma \triangleright \Delta$	Inert Set

Generate inhabitants of Θ

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

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

Construct inhabited ∇s from Φ

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

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

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

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

Expand variables to Pat with ∇

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

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

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

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 ∇

4 FORMALISM

SLPJ: This entire section is scheduled for deletion, once we have moved out everything we need

The previous section gave insights into how we represent coverage checking problems as guard trees and provided an intuition for how to check them for exhaustiveness and redundancy. This section formalises these intuitions in terms of the syntax (cf. fig. 3) we introduced earlier.

As in the previous section, we divide this section into three main parts: desugaring, coverage checking, and finding inhabitants of the resulting refinement types. The latter subtask proves challenging enough to warrant two additional subsections.

SG: We should talk about why "constructor applications" in Expr, Grd, φ and δ have different number of arguments. Not sure where.

4.1 Desugaring to guard trees

SLPJ: I think this section has nothing useful left

Figure 4 outlines the desugaring step from source Haskell to our guard tree language Gdt. It is assumed that the top-level match variables x_1 through x_n in the *clause* cases have special, fixed names. All other variables that aren't bound in arguments to $\mathcal{D}$ have fresh names.

Consider this example function:

$$f\ (\mathit{Just}\ (!xs, \_))\ ys@\mathit{Nothing} = 1$$
$$f\ \mathit{Nothing}\qquad (g \to \mathit{True}) = 2$$

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

$$\dashv !x_1, \mathit{Just}\ t_1 \leftarrow x_1, !t_1, (t_2, t_3) \leftarrow t_1, !t_2, \mathsf{let}\ xs = t_2, \mathsf{let}\ ys = x_2, !ys, \mathit{Nothing} \leftarrow ys \longrightarrow 1$$
$$\dashv !x_1, \mathit{Nothing} \leftarrow x_1, \mathsf{let}\ t_3 = g\ x_2, !y, \mathit{True} \leftarrow t_3 \longrightarrow 2$$

The definition of $\mathcal{D}$ is straight-forward, but a little expansive because of the realistic source language. Its most intricate job is keeping track of all the renaming going on to resolve name mismatches. Other than that, the desugaring follows from the restrictions on the Grd language, such as the fact that source-level pattern guards also need to emit a bang guard on the variable representing the scrutinee.

Note how our naïve desugaring function 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 variable.

4.2 Checking guard trees

SLPJ: I think this section has nothing useful left

Figure 5 shows the two main functions for checking guard trees. $\mathcal{U}$ carries out exhaustiveness checking by computing the set of uncovered values for a particular guard tree, whereas $\mathcal{A}$ computes the corresponding annotated tree, capturing redundancy information. $\mathcal{R}$ extracts a triple of accessible, inaccessible and redundant GRHS from such an annotated tree.

Both $\mathcal{U}$ and $\mathcal{A}$ take as their second parameter the set of values *reaching* the particular guard tree node. If no value reaches a particular tree node, that node is inaccessible. The definition of $\mathcal{U}$ follows the intuition we built up earlier: It refines the set of reaching values as a subset of it falls through from one clause to the next. This is most visible in the ; case (top-to-bottom composition), where the set of values reaching the right (or bottom) child is exactly the set of values that were uncovered by the left (or top) child on the set of values reaching the whole node. A GRHS covers every reaching value. The left-to-right semantics of `Guard` are respected by refining the set of values reaching the wrapped subtree, depending on the particular guard. Bang guards and let bindings don't do anything beyond that refinement, whereas pattern guards additionally account for the

possibility of a failed pattern match. Note that a failing pattern guard is the *only* way in which the uncovered set can become non-empty!

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

Pattern guard semantics are important for $\mathcal{U}$ and bang guard semantics are important for $\mathcal{A}$. But what about let bindings? They are in fact completely uninteresting to the checking process, but making sense of them is important for the precision of the emptiness check involving $\mathcal{G}$. Of course, "making sense" of an expression is an open-ended endeavour, but we'll see a few reasonable ways to improve precision considerably at almost no cost, both in section 4.4 and section 5.3.

4.3 Generating inhabitants of a refinement type

The key function for the emptiness test is $\mathcal{G}$ in fig. 6, which generates a set of patterns which inhabit a given refinement type Θ. There might be multiple inhabitants, and C will construct multiple ∇s, each representing at least one inhabiting assignment of the refinement predicate Φ. Each such assignment corresponds to a pattern vector, so $\mathcal{E}$ expands the assignments in a ∇ into multiple pattern vectors.

But what *is* ∇? It's a pair of a type context Γ and a Δ, a set of mutually compatible constraints δ, or a proven incomatibility $\times$ between such a set of constraints. C will arrange it that every constructed ∇ satisfies a number of well-formedness constraints:

- I1 *Mutual compatibility*: No two constraints in ∇ should conflict with each other.
- 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.

We refer to such a ∇ as an *inert set*, in the sense that its constraints are of canonical form and already checked for mutual compatibility (I1), in analogy to a typechecker's implementation.

It is helpful at times 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 later in section 4.4, this view is supported by immense overlap 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 (in the sense of understanding Δ as a partial function) twice: The first lookup will find x's representative y, the second lookup on y will then find the solution *Nothing*.

In denoting looking up the representative by $\Delta(x)$ (cf. fig. 6), we can assert that x has *Nothing* as a solution simply by writing $\Delta(x) \approx \mathit{Nothing} \in \Delta$.

Each Δ is one of possibly many valid variable assignments of the particular Φ it is constructed for. In contrast to Φ, there is no disjunction in Δ, which makes it easy to check if a new constraint

is compatible with the existing ones without any backtracking. Another fundamental difference is that δ has no binding constructs (so every variable has to be bound in the Γ part of ∇), whereas pattern bindings in φ bind constructor arguments.

$\mathcal{C}$ is the function that breaks down a Φ into multiple ∇s, maintaining the invariant that no such ∇ is $\times$. At the heart of $\mathcal{C}$ is adding a φ literal to the ∇ under construction via $\oplus_\varphi$ and filtering out any unsuccessful attempts (via intercepting the $\times$ failure mode of $\oplus_\varphi$) to do so. Conjunction is handled by the equivalent of a *concatMap*, whereas a disjunction corresponds to a plain union.

Expanding a ∇ to a pattern vector in $\mathcal{E}$ 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*. I3 guarantees that there is at most one such solution and $\mathcal{E}$ is well-defined.

4.4 Extending the inert set

After tearing down abstraction after abstraction in the previous sections we are nearly at the heart of LYG: Figure 7 depicts how to add a φ constraint to an inert set ∇.

It does so by expressing a φ in terms of once again simpler constraints δ and calling out to $\oplus_\delta$. Specifically, for a lack of binding constructs in δ, pattern bindings extend the context and disperse into separate type constraints and a positive constructor constraint arising from the binding. The fourth case of $\oplus_\delta$ finally performs some limited, but important reasoning about let bindings: In case the right-hand side was a constructor application (which is not to be confused with a pattern binding, if only for the difference in binding semantics!), we add appropriate positive constructor and type constraints, as well as recurse into the field expressions, which might in turn contain nested constructor applications. All other let bindings are simply discarded. We'll see an extension in section 5.3 which will expand here. The last case of $\oplus_\varphi$ turns the syntactically and semantically identical subset of φ into δ and adds that constraint via $\oplus_\delta$.

Which brings us to the prime unification procedure, $\oplus_\delta$. Consider adding a positive constructor constraint like $x \approx \mathit{Just}\ y$: The unification procedure will first look for any positive constructor constraint involving the representative of x with *that same constructor*. Let's say there is $\Delta(x) = z$ and $z \approx \mathit{Just}\ u \in \Delta$. Then $\oplus_\delta$ decomposes the new constraint just like a classic unification algorithm operating 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 (thus maintaining wellformed-ness condition I1), is not added to the inert set because of I2.

If there was no positive constructor constraint with the same constructor, it will look for such a constraint involving a different constructor, like $x \approx \mathit{Nothing}$, in which case the new constraint is incompatible with the existing solution. There are two other ways in which the constraint can be incompatible: If there was a negative constructor constraint $x \not\approx \mathit{Just}$ or if any of the fields were not inhabited, which is checked by the $\nabla \vdash x$ judgment in fig. 8. Otherwise, the constraint is compatible and is added to Δ.

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

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

Add a formula literal to the inert set

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

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

Add a constraint to the inert set

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

$$\times \oplus_\delta \delta = \times$$

$$\Gamma \triangleright \Delta \oplus_\delta \gamma = \begin{cases} \Gamma \triangleright (\Delta, \gamma) & \text{if type checker deems } \gamma \text{ compatible with } \Delta \\ & \text{and } \forall x \in \mathrm{dom}(\Gamma) : \Gamma \triangleright (\Delta, \gamma) \vdash \Delta(x) \\ \times & \text{otherwise} \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \approx K\ \overline{a}\ \overline{y} = \begin{cases} \Gamma \triangleright \Delta \oplus_\delta \overline{a \sim b} \oplus_\delta \overline{y \approx z} & \text{if } \Delta(x) \approx K\ \overline{b}\ \overline{z} \in \Delta \\ \times & \text{if } \Delta(x) \approx K'\ \overline{b}\ \overline{z} \in \Delta \\ \Gamma \triangleright (\Delta, \Delta(x) \approx K\ \overline{a}\ \overline{y}) & \text{if } \Delta(x) \not\approx K \notin \Delta \text{ and } \overline{\Gamma \triangleright \Delta \vdash \Delta(y)} \\ \times & \text{otherwise} \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \not\approx K = \begin{cases} \times & \text{if } \Delta(x) \approx K\ \overline{a}\ \overline{y} \in \Delta \\ \times & \text{if not } \Gamma \triangleright (\Delta, \Delta(x) \not\approx K) \vdash \Delta(x) \\ \Gamma \triangleright (\Delta, \Delta(x) \not\approx K) & \text{otherwise} \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \approx \bot = \begin{cases} \times & \text{if } \Delta(x) \not\approx \bot \in \Delta \\ \Gamma \triangleright (\Delta, \Delta(x) \approx \bot) & \text{otherwise} \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \not\approx \bot = \begin{cases} \times & \text{if } \Delta(x) \approx \bot \in \Delta \\ \times & \text{if not } \Gamma \triangleright (\Delta, \Delta(x) \not\approx \bot) \vdash \Delta(x) \\ \Gamma \triangleright (\Delta, \Delta(x) \not\approx \bot) & \text{otherwise} \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \approx y = \begin{cases} \Gamma \triangleright \Delta & \text{if } \Delta(x) = \Delta(y) \\ \Gamma \triangleright ((\Delta \setminus \Delta(x)), \Delta(x) \approx \Delta(y)) \oplus_\delta ((\Delta \cap \Delta(x))[\Delta(y)/\Delta(x)]) & \text{otherwise} \end{cases}$$

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

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

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

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

Fig. 7. Adding a constraint to the inert set ∇

Test if x is inhabited considering ∇

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

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

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

Find data constructors of τ

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

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

Instantiate x to data constructor K

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

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

Fig. 8. Inhabitance test

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.

The last case of $\oplus_\delta$ equates two variables ($x \approx y$) by merging their equivalence classes. Consider the case where x and y don't already belong to the same equivalence class and thus have different representatives $\Delta(x)$ and $\Delta(y)$. $\Delta(y)$ is arbitrarily chosen to be the new representative of the merged equivalence class. Now, to uphold the well-formedness condition I2, all constraints mentioning $\Delta(x)$ have to be removed and renamed in terms of $\Delta(y)$ and then re-added to Δ. That might fail, because $\Delta(x)$ might have a constraint that conflicts with constraints on $\Delta(y)$, so it is better to use $\oplus_\delta$ rather than to add it blindly to Δ.

4.5 Inhabitation test

SG: We should find better subsection titles that clearly distinguish "Testing (Θ) for Emptiness" from "Inhabitation Test(ing a particular variable in ∇)". The process for adding a constraint to an inert set above (which turned out to be a unification procedure in disguise) frequently made use of an *inhabitation test* $\nabla \vdash x$, depicted in fig. 8. In contrast to the emptiness test in fig. 6, this one focuses on a particular variable and works on a ∇ rather than a much higher-level Θ.

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:

$$\textbf{data}\ T = MkT\ !\ T$$
$$f :: SMaybe\ T \to ()$$
$$f\ SNothing = ()$$

This is exhaustive, because T is an uninhabited type. Upon adding the constraint $x \not\approx SNothing$ 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 $x \approx SJust\ y, y \not\approx \bot$, the latter of which leads to an inhabitation test on y. That leads to instantiation of the MkT constructor, which leads to constraints $y \approx MkT\ z, z \not\approx \bot$, 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.

5 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 how our solution can be easily supplemented to deal with new language features or measures for increasing the precision of the checking process.

5.1 Long-distance information

Coverage checking as described also works for **case** expressions (with the appropriate desugaring function) and nested function definitions, like in the following example:

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

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 fact, LYG does exactly the same kind of reasoning when checking f! Specifically, the set of values reaching the second GRHS (which we test for inhabitants to determine whether the GRHS is accessible) Θ_{rhs2} encodes the information we are after. We just have to start checking the **case** expression starting from Θ_{rhs2} as the initial set of reaching values instead of $\langle\, x : Bool \mid \checkmark \,\rangle$.

5.2 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*:

$$absurd :: Void \to a$$
$$absurd\ \_ = \bot$$
$$absurd\ !\_ = \bot$$

Clearly, neither option is satisfactory to implement *absurd*: The first one would actually return $\bot$ when called with $\bot$, thus masking the original $\bot$ with the error thrown by $\bot$. The second one

would diverge alright, but it is unfortunate that we still have to provide a RHS that we know will never be entered. In fact, LYG will mark the second option as having an inaccessible RHS!

GHC provides an extension, called `EmptyCase`, that introduces the following bit of new syntax:

$$\mathit{absurd}\ x = \textbf{case}\ x\ \textbf{of}\ \{\}$$

Such a **case** expression without any alternatives evaluates its argument to WHNF and crashes when evaluation returns.

Although we did not give the syntax of **case** expressions in fig. 1, 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. The new `GEmpty` case has to be handled by the checking functions and is a neutral element to ; as far as $\mathcal{U}$ is concerned:

$$\begin{array}{lcl} \mathcal{U}(\Theta, \texttt{GEmpty}) & = & \Theta \\ \mathcal{A}(\Theta, \texttt{GEmpty}) & = & \texttt{AEmpty} \end{array}$$

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. So, for checking an empty **case**, the call to $\mathcal{U}$ looks like $\mathcal{U}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), \texttt{GEmpty})$, where Θ is the context-sensitive set of reaching values, possibly enriched with long distance information (cf. section 5.1).

5.3 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:

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

Although y_1 and y_2 bind syntactically equivalent expressions, our simple desugaring function doesn't see that and allocated fresh names for each of them. That in turn means that both the match on y_1 and y_2 by itself are non-exhaustive. But due to referential transparency, the result of $\mathit{reverse}\ x_1$ doesn't change! By making the connection between y_1 and y_2, the checker could infer that the match was exhaustive.

This can be fixed at any level of abstraction (i.e. in $\mathcal{D}$ or $\oplus_\varphi$) by maintaining equivalence classes of semantically equivalent expressions. For the example above, handling let $y_2 = \mathit{reverse}\ x_1$ in the second branch would entail looking up the equivalence class of $\mathit{reverse}\ x_1$ and finding out that it is also bound by y_1, so we can handle let $y_2 = y_1$ instead and make sense of the $y_1 \not\approx \mathit{Nothing}$ constraint that fell through from the first branch to conclude that the match is exhaustive.

In fact, that is just like performing an on-the-fly global value numbering (GVN) of expression [Kildall 1973]! We decided to perform (an approximation to) GVN at the level of $\oplus_\varphi$, because it is more broadly applicable there and a very localised change:

$$\Gamma \triangleright \Delta \oplus_\varphi \text{let } x : \tau = e \quad = \quad \Gamma \triangleright \Delta \oplus_\varphi \text{let } x : \tau = r_i \quad \text{where } i \text{ is global value number of } e$$

Where r_i is the representative of the equivalence class of expressions with global value number i. Thus, our implementation will not emit any warnings for a definition like *safeLast*.

5.4 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*:

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

SG: For coverage checking purposes, we assume that pattern synonym matches are strict, just like data constructor matches. This is not generally true, but #17357 has a discussion of why being conservative is too disruptive to be worth the trouble. Should we talk about that? It concerns the definition of $\mathcal{D}$, namely whether to add a $!x$ on the match var or not. Maybe a footnote?

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

$$\begin{array}{l} \mathit{pattern}\ P = () \\ \mathit{pattern}\ Q = () \\ n = \mathbf{case}\ P\ \mathbf{of}\ Q \to 1; P \to 2 \end{array}$$

Knowing that the definitions of P and Q completely overlap, we can see that 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 disjunct. That is in stark contrast to data constructors, which never overlap.

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

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

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.

5.5 `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 ⊢Inst from fig. 8 currently making sure that this `COMPLETE` set is in fact inhabited. We also have ⊢NoCpl that handles the case when we can't find *any* `COMPLETE` set for the given type

(think $x : Int \to Int$). 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:

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

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

SG: What do you think of the indexing on C_i? It's not entirely accurate, but do we want to cloud the presentation with i.e. $\overline{C_{i,1}, \ldots, C_{i,n_i}}^{\,i}$?

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 6.

5.6 Literals

The source syntax in fig. 1 deliberately left out literal patterns l. Literals are very similar to nullary data constructors, with one caveat: They don't come with a builtin `COMPLETE` set. Before section 5.5, that would have meant quite a bit of hand waving and complication to the ⊢ judgment. Now, literals can be handled like disjunct pattern synonyms (i.e. $l_1 \cap l_2 = \emptyset$ for any two literals l_1, l_2) without a `COMPLETE` set!

We can even handle overloaded literals, but will find ourselves in a similar situation as with pattern synonyms:

```
instance Num () where
  fromInteger _ = ()
n = case (0 :: ()) of 1 -> 1; 0 -> 2
```

Considering overloaded literals to be disjunct would mean marking the first alternative as redundant, which is unsound. Hence we regard overloaded literals as possibly overlapping, so they behave exactly like nullary pattern synonyms without a `COMPLETE` set.

5.7 Newtypes

Newtypes have strange semantics. Here are two key examples that distinguish it from data types:

```
newtype N a = N a
g1 :: N () -> Bool -> Int
g1 !(N _) True = 1
g1 (N !_) True = 2

g2 :: N () -> Bool -> Int
g2 (N !_) True = 2
g2 !(N _) True = 1

f :: N Void -> Bool -> Int
f _     True = 1
f (N _) True = 2
f !_    True = 3
```

The definition of f is subtle. Contrary to the situation with data constructors, the second GRHS is *redundant*: The pattern match on the Newtype constructor is a no-op. Conversely, the bang pattern in the third GRHS forces not only the Newtype constructor, but also its wrapped thing. That could lead to divergence for a call site like $f \perp False$, so the third GRHS is *inaccessible* (because every value it could cover was already covered by the first GRHS), but not redundant. A perhaps

$$cl ::= K \mid P \mid \boxed{N} \qquad \begin{array}{ll} N \in & \mathsf{NT} \\ C \in & K \mid P \mid \boxed{N} \end{array}$$

$$\mathcal{D}(x, N\ pat_1 \ldots pat_n) = N\ y_1 \ldots y_n \leftarrow x, \mathcal{D}(y_1, pat_1), \ldots, \mathcal{D}(y_n, pat_n)$$

$$\frac{(\Gamma \triangleright \Delta \oplus_\delta x \approx \bot) \neq \times \qquad x : \tau \in \Gamma \quad \tau \text{ not a Newtype}}{\Gamma \triangleright \Delta \vdash x}\ \vdash\textsc{Bot} \qquad \frac{x : \tau \in \Gamma \quad \mathsf{Cons}(\Gamma \triangleright \Delta, \tau) = \overline{C_1, \ldots, C_{n_i}}^{\,i} \qquad \overline{\mathsf{Inst}(\Gamma \triangleright \Delta, x, C_j) \neq \times}^{\,i} \quad \tau \text{ not a Newtype}}{\Gamma \triangleright \Delta \vdash x}\ \vdash\textsc{Inst}$$

$$\frac{\tau \text{ Newtype with constructor } N \text{ wrapping } \sigma \qquad x : \tau \in \Gamma \quad y \# \Gamma \quad \Gamma, y : \sigma \triangleright \Delta \oplus_\delta x \approx N\ y \vdash y}{\Gamma \triangleright \Delta \vdash x}\ \vdash\textsc{Inst}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \approx \bot \;=\; \begin{cases} \times & \text{if } \Delta(x) \not\approx \bot \in \Delta \\ \Gamma \triangleright \Delta \oplus_\delta x \approx N\ y \oplus_\delta y \approx \bot & \text{if } x : \tau \in \Gamma, \tau \text{ Newtype with constructor } N \text{ wrappin} \\ \Gamma \triangleright (\Delta, \Delta(x) \approx \bot) & \text{otherwise} \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_\delta x \not\approx \bot \;=\; \begin{cases} \times & \text{if } \Delta(x) \approx \bot \in \Delta \\ \times & \text{if not } \Gamma \triangleright (\Delta, \Delta(x) \not\approx \bot) \vdash \Delta(x) \\ \Gamma \triangleright \Delta \oplus_\delta x \not\approx N\ y \oplus_\delta y \not\approx \bot & \text{if } x : \tau \in \Gamma, \tau \text{ Newtype with constructor } N \text{ wrappin} \\ \Gamma \triangleright (\Delta, \Delta(x) \not\approx \bot) & \text{otherwise} \end{cases}$$

Fig. 9. Extending coverage checking to handle Newtypes

surprising consequence is that the definition of f is exhaustive, because after $N\ \mathit{Void}$ was deprived of its sole inhabitant $\bot \equiv N\ \bot$ by the third GRHS, there is nothing left to match on.

Figure 9 outlines a solution (based on that for pattern synonyms for brevity) that handles f correctly. The idea is to treat Newtype pattern matches lazily (so compared to data constructor matches, $\mathcal{D}$ omits the $!x$). The other significant change is to the $\vdash$ judgment form, where we introduce a new rule $\vdash$NT that is specific to Newtypes, which can no longer be proven inhabited by either $\vdash$Inst or $\vdash$Bot.

SG: I think that just assuming Newtype constructors to have strict fields achieves the same, but is a more surgical change, probably saving some space.

But *g1* crushes this simple hack. We would mark its second GRHS as inaccessible when it is clearly redundant, because the $x \not\approx \bot$ constraint on the match variable x wasn't propagated to the wrapped (). The inner bang pattern has nothing to evaluate. This is arguably a small downside and doesn't even regress in terms of soundness.

SG: We could save about 1/4 of a page by stopping here and omitting the changes to $\oplus_\delta$.

We counter that with another refinement: We just add $x \approx Ny$ and $y \not\approx \bot$ constraints whenever we add $x \not\approx \bot$ constraints when we know that x is a Newtype with constructor N (similarly for $x \approx \bot$). Both *g1* and *g2* will be handled correctly.

SG: Needless to say, we won't propagate $\bot$ constraints when we only find out (by additional type info) that something is a Newtype *after* adding the constraints (think *SMaybe a* and we later find that $a \sim N\ \mathit{Void}$), but let's call it a day.

An alternative, less hacky solution would be treating Newtype wrappers as coercions and at the level of Δ consider equivalence classes modulo coercions. That entails a slew of modifications and has deep ramifications throughout the presentation.

5.8 Strictness

Instead of extending the source language, let's discuss ripping out a language feature, for a change! So far, we have focused on Haskell as the source language of the checking process, which is lazy by default. Although the desugaring function makes sure that the difference in evaluation strategy of the source language quickly becomes irrelevant, it raises the question of how much our approach could be simplified if we targeted a source language that was strict by default, such as OCaml or Idris.

First off, both languages offer language support for laziness and lazy pattern matches, so the question rather becomes whether the gained simplification is actually worth risking unusable or even unsound warning messages when making use of laziness. If the answer is "No", then there isn't anything to simplify, just relatively more $x \approx \bot$ constraints to handle.

Otherwise, in a completely eager language we could simply drop $!x$ from Grd and Bang from Ant. Actually, Ant and $\mathcal{R}$ could go altogether and $\mathcal{A}$ could just collect the redundant GRHS directly!

Since there wouldn't be any bang guards, there is no reason to have $x \approx \bot$ and $x \not\approx \bot$ constraints either. Most importantly, the $\vdash$Bot judgment form has to go, because $\bot$ does not inhabit any types anymore.

6 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 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.

SG: New implementation (pre !2753) has 3850 lines, out of which 1753 is code. Previous impl as of GHC 8.6.5 had 3118 lines, out of which 1438 were code. Not sure how to sell that.

6.1 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 reaching 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!

Clearly, the space usage won't actually grow exponentially due to sharing in the implementation, but the problems for runtime performance remain. 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 really build up a refinement type but pass around the result of calling $\mathcal{C}$ on what would have been the set of reaching values.

You can see the resulting definition in fig. 10. The readability of the interleaving of both functions is clouded by unwrapping of pairs. Other than that, all references to Θ were replaced by a vector of ∇s. $\mathcal{UA}$ requires that each ∇ individually is non-empty, i.e. not $\times$. This invariant is maintained by

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

$$\begin{array}{rcl}
\epsilon \dot{\oplus}_\varphi \varphi & = & \epsilon \\
(\nabla_1 \dots \nabla_n) \dot{\oplus}_\varphi \varphi & = & \begin{cases} (\Gamma \triangleright \Delta)\,(\nabla_2 \dots \nabla_n \dot{\oplus}_\varphi \varphi) & \text{if } \Gamma \triangleright \Delta = \nabla \oplus_\varphi \varphi \\ (\nabla_2 \dots \nabla_n) \dot{\oplus}_\varphi \varphi & \text{otherwise} \end{cases}
\end{array}$$

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

$$\begin{array}{rcl}
\mathcal{UA}(\overline{\nabla}, \mathsf{GRhs}\ n) & = & (\epsilon, \mathsf{ARhs}\ \overline{\nabla}\ n) \\
\mathcal{UA}(\overline{\nabla}, t_G; u_G) & = & (\overline{\nabla}_2, t_A; u_A) \text{ where } \begin{array}{l} (\overline{\nabla}_1, t_A) = \mathcal{UA}(\overline{\nabla}, t_G) \\ (\overline{\nabla}_2, u_A) = \mathcal{UA}(\overline{\nabla}_1, u_G) \end{array} \\
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (!x)\ t_G) & = & \begin{cases} (\overline{\nabla}', t_A), & \overline{\nabla} \dot{\oplus}_\varphi (x \approx \bot) = \epsilon \\ (\overline{\nabla}', \mathsf{Bang}\ \overline{\nabla}\ t_A) & \text{otherwise} \end{cases} \\
 & & \quad \text{where } (\overline{\nabla}', t_A) = \mathcal{UA}(\overline{\nabla} \dot{\oplus}_\varphi (x \not\approx \bot), t_G) \\
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (\mathsf{let}\ x = e)\ t) & = & \mathcal{UA}(\overline{\nabla} \dot{\oplus}_\varphi (\mathsf{let}\ x = e), t) \\
\mathcal{UA}(\overline{\nabla}, \mathsf{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x)\ t_G) & = & ((\overline{\nabla} \dot{\oplus}_\varphi (x \not\approx K))\,\overline{\nabla}', t_A) \\
 & & \quad \text{where } (\overline{\nabla}', t_A) = \mathcal{UA}(\overline{\nabla} \dot{\oplus}_\varphi (K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x), t_G)
\end{array}$$

Fig. 10. Fast coverage checking

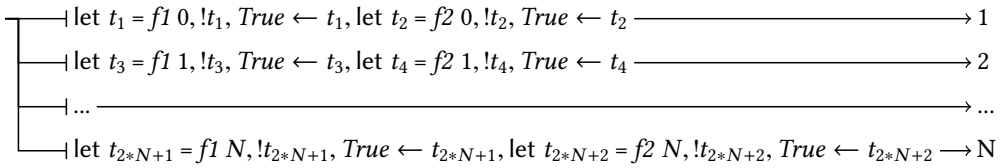
adding φ constraints through $\dot{\oplus}_\varphi$, which filters out any ∇ that would become empty. All mentions of $\mathcal{G}$ are gone, because we never were interested in inhabitants in the first place, only whether there where any inhabitants at all! In this new representation, whether a vector of ∇ is inhabited is easily seen by syntactically comparing it to the empty vector, ϵ.

6.2 Throttling for graceful degradation

Even with the tweaks from section 6.1, checking certain pattern matches remains NP-hard **SG:** Cite something here or earlier, bring an example. Naturally, there will be cases where we have to conservatively approximate in order not to slow down compilation too much. After all, coverage checking is just a static analysis pass without any effect on the produced binary! Consider the following example:

$f1, f2 :: Int \to Bool$
$g\ \_$
$\quad \mid True \leftarrow f1\ 0,\ True \leftarrow f2\ 0\ = ()$
$\quad \mid True \leftarrow f1\ 1,\ True \leftarrow f2\ 1\ = ()$
$\quad \dots$
$\quad \mid True \leftarrow f1\ N, True \leftarrow f2\ N = ()$

Here's the corresponding guard tree:



Each of the N GRHS can fall through in two distinct ways: By failure of either pattern guard involving *f1* or *f2*. Each way corresponds to a way in which the vector of reaching ∇s is split. For example, the single, unconstrained ∇ reaching the first equation will be split in one ∇ that records that either $t_1 \not\approx \mathit{True}$ or that $t_2 \not\approx \mathit{True}$. Now two ∇s fall through and reach the second branch, where they are split into four ∇s. This exponential pattern repeats N times, and leads to horrible performance!

There are a couple of ways to go about this. First off, that it is always OK to overapproximate the set of reaching values! Instead of *refining* ∇ with the pattern guard, leading to a split, we could just continue with the original ∇, thus forgetting about the $t_1 \not\approx \mathit{True}$ or $t_2 \not\approx \mathit{True}$ constraints. In terms of the modeled refinement type, ∇ is still a superset of both refinements.

Another realisation is that each of the temporary variables binding the pattern guard expressions are only scrutinised once, within the particular branch they are bound. That makes one wonder why we record a fact like $t_1 \not\approx \mathit{True}$ in the first place. Some smart "garbage collection" process might get rid of this additional information when falling through to the next equation, where the variable is out of scope and can't be accessed. The same procedure could even find out that in the particular case of the split that the ∇ falling through from the *f1* match models a superset of the ∇ falling through from the *f2* match (which could additionally diverge when calling *f2*). This approach seemed far to complicated for us to pursue.

Instead, we implement *throttling*: We limit the number of reaching ∇s to a constant. Whenever a split would exceed this limit, we continue with the original reaching ∇ (which as we established is a superset, thus 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:

$$\begin{aligned}\mathcal{UA}(\overline{\nabla}, \texttt{Guard}\ (K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x)\ t_G) = (\left\lfloor (\overline{\nabla} \dot{\oplus}_\varphi (x \not\approx K))\ \overline{\nabla}' \right\rfloor_{\overline{\nabla}}, t_A) \\ \text{where } (\overline{\nabla}', t_A) = \mathcal{UA}(\overline{\nabla} \dot{\oplus}_\varphi (K\ \overline{a}\ \overline{\gamma}\ \overline{y:\tau} \leftarrow x), t_G)\end{aligned}$$

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.

For the sake of our above example we'll use 4 as the limit. The initial ∇ will be split by the first equation in two, which in turn results in 4 ∇s reaching the third equation. Here, splitting would result in 8 ∇s, so we throttle, so that the same four ∇s reaching the third equation also reach the fourth equation, and so on. Basically, every equation is checked for overlaps *as if* it was the third equation, because we keep on forgetting what was matched beyond that.

6.3 Maintaining residual `COMPLETE` sets

Our implementation applies a few hacks 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 :: T → Int
pattern P = ...                    f A1     = 1
 {-# COMPLETE A1, P #-}            f A2     = 2
                                    ...
                                   f A1000 = 1000
```

f is exhaustively defined. For seeing 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 ⊢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.

6.4 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 $\mathcal{E}$.

7 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 [3]. `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`, has 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.

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

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 $keepAlive :: a \to a$ primitive function 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*.

7.1 Performance tests

Ryan: I ran out of time today, but we should measure the difference between GHC 8.8.3 and HEAD on the following perf tests from GHC's test suite: T11303, T11276, T11303b, T11374, T11822, T11195, T17096, PmSeriesS, PmSeriesT, PmSeriesV, and PmSeriesG. These all live under the `testsuite/tests/pmcheck/should_compile` directory.

7.2 GHC issues

Implementing LYG in GHC has fixed a litany of 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, 2020c]
- 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]

8 RELATED WORK

8.1 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.

8.1.1 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.

8.1.2 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 the *safeLast* function from section 2.2.1. But, will cause GMTM to emit erroneous warnings.

SG: *safeLast* isn't about guards, strictly speaking. But we could have

```
safeLast2 xs
  | (x : _) ← reverse xs = Just x
  | [ ]     ← reverse xs = Nothing
```

which is handled by LYG, but not by GMTM.

While GMTM's term oracle is customizable, 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 5.3) 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 $\oplus_\varphi$.

Ryan: Should we mention something about the performance of GMTM here?

8.2 Comparison with similar coverage checkers

8.2.1 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 integer arithmetic. **Ryan:** Sebastian: is this the thing that would need to be extended? **SG:** Yes, I imagine that $\oplus_\varphi$ would match on arithmetic expressions and then add some kind of new δ constraint to Δ. $\oplus_\delta$ would then have to do the actual linear arithmetic reasoning, e.g., conclude from $x \not< e, x \not\approx e, x \not> e$ (and $x \not\approx \bot$) that x is not inhabited, quite similar to a `COMPLETE` set.

8.2.2 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

```
f True = 1
f _    = 2
```

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.

SG: We can shorten this as needed. It's just a rant at this point, I'm afraid...

Apart from that, his algorithm would report *Nothing* as a missing clause of the definition $g\ (Just\ True) = 1$, but would fail to report the nested *Just False*, which is clearly unsound according to our definition in ??.

Maranget comes up with surprisingly tricky test cases which exponentially blew up compilation time of GHC at the time. We added them into our testsuite[4] as regression tests and made sure that throttling (section 6.2) maintains linear performance characteristics. **Ryan:** The use of "We" in this sentence risks de-anonymizing us...

8.2.3 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.

8.3 Positive and negative information

LYG's use of positive and negative constructor constraints in inert sets 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.

But also accurate redundancy warnings involving `COMPLETE` sets hinge on negative constraints. For why this isn't possible in other checkers that only track positive information, such as those of Karachalias et al. [2015] (section 8.1) and Maranget [2007] (section 8.2.2), consider the following example:

```
pattern True' = True
  {-# COMPLETE True', False #-}
f False = 1
f True' = 2
f True  = 3
```

GMTM would have to commit to a particular `COMPLETE` set when encountering the match on *False*, without any semantic considerations. Choosing $\{True', False\}$ 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 had a complicated metric based on the number and kinds of warnings the choice of each one would generate to disambiguate[5] , which was broken still[6].

On the front of efficiency, consider

```
data T = A1 | ... | A1000
h A1 _  = 1
h _  A1 = 2
```

GMTM first splits the value vector (roughly corresponding to one of our 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 us 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 8.2.2).

[4] #17264
[5] "Disambiguating between multiple `COMPLETE` pragmas" in the user's guide of GHC 8.6.5
[6] #13363

8.4 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 [7]:

type *void*; ;	$v : \mathit{Maybe\ Void} \to \mathit{Int}$
let v (*None* : *void option*) : *int* = 0; ;	$v\ \mathit{Nothing} = 0$

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

8.5 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 (cf. **??**). For example, consider this non-exhaustive function:

$\mathit{fibPartial} :: \mathit{Integer} \to \mathit{Integer}$
$\mathit{fibPartial}\ 0 = 0$
$\mathit{fibPartial}\ 1 = 1$

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

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

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.

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

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

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 warning emitted twice. https://gitlab.haskell.org/ghc/ghc/issues/17646
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
Gary A. Kildall. 1973. A Unified Approach to Global Program Optimization. In *Proceedings of the 1st Annual ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages (POPL '73)*. Association for Computing Machinery, New York, NY, USA, 194–206. https://doi.org/10.1145/512927.512945
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.

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