Lower Your Guards

A Compositional Pattern-Match Coverage Checker

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

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

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

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

1 INTRODUCTION

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

\[
f :: \text{Int} \to \text{Bool}
f 0 = \text{True}
f 0 = \text{False}
\]

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 \(f 0 = \text{False}\) clause completely unreachable. Even worse is that \(f\) never matches on any patterns besides 0, making it not fully defined. Attempting to invoke \(f 1\), for instance, will fail.

To avoid these mishaps, compilers for languages with pattern matching often emit warnings (or errors) if a function is missing clauses (i.e., if it is non-exhaustive), if one of its right-hand sides will never be entered (i.e., if it is inaccessible), or if one of its equations can be deleted altogether (i.e., if it is redundant). We refer to the combination of checking for exhaustivity, redundancy, and accessibility as pattern-match coverage checking. Coverage checking is the first line of defence in catching programmer mistakes when defining code that uses pattern matching.

Coverage checking for a set of equations matching on algebraic data types is a well studied (although still surprisingly tricky) problem—see Section 7 for this related work. But the coverage-checking problem becomes much harder when one includes the raft of innovations that have become part of a modern programming language like Haskell, including: view patterns, pattern guards, pattern synonyms, overloaded literals, bang patterns, lazy patterns, as-patterns, strict data constructors, empty case expressions, and long-distance effects (Section 4). Particularly tricky are
GADTs [Xi et al. 2003], where the type of a match can determine what values can possibly appear; and local type-equality constraints brought into scope by pattern matching [Vytiniotis et al. 2011].

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

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

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

We discuss the wealth of related work in Section 7.

2 THE PROBLEM WE WANT TO SOLVE

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

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

Prior work on coverage checking (discussed in Section 7) accounts for some of these nuances, but not all of them. In this section we identify some key language features that make coverage checking difficult. While these features may seem disparate at first, we will later show in Section 3 that these ideas can all fit into a unified framework.
2.1 Guards

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

```
guardDemo :: Char → Char → Int

guardDemo c1 c2
| c1 == 'a' = 0
| 'b' ← c1 = 1
| let c1' = c1, 'c' ← c1', c2 == 'd' = 2
| otherwise = 3
```

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

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

```
signon :: Int → Int
signon x
| x > 0 = 1
| x == 0 = 0
| x < 0 = -1
```

Intuitively, signum is exhaustive since the combination of (>), (==), and (<) covers all possible Ints. 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
not b | False ← b = True
| True ← b = False

not2 :: Bool → Bool
not2 False = True
not2 True = False

not3 :: Bool → Bool
not3 x | x ← False = True
| True = False
```

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

2.2 Programmable patterns

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

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

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

length :: Text -> Int
length (Text.null -> True) = 0
length (Text.uncons -> Just (_, xs)) = 1 + length xs

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

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

2.2.2 Pattern synonyms. Pattern synonyms [Pickering et al. 2016] allow abstraction over patterns themselves. Pattern synonyms and view patterns can be useful in tandem, as the pattern synonym can present an abstract interface to a view pattern that does complicated things under the hood.

For example, one can define length with pattern synonyms like so:

```
pattern Nil :: Text
pattern Nil <- (Text.null -> True)
pattern Cons :: Char -> Text -> Text
pattern Cons x xs <- (Text.uncons -> Just (x, xs))

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

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

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

2.3 Strictness

The evaluation order of pattern matching can impact whether a pattern is reachable or not. While Haskell is a lazy language, programmers can opt into extra strict evaluation by giving the fields of a data type strict fields, such as in this example:

```
data Void -- No data constructors; only inhabitant is bottom
data SMaybe a = SJust !a | SNothing
v :: SMaybe Void -> Int
v SNothing = 0
v (SJust _) = 1   -- Redundant!
```
The “!” in the definition of $SJust$ makes the constructor strict, so $(SJust \bot) = \bot$. Curiously, this makes the second equation of $v$ redundant! Since $\bot$ is the only inhabitant of type Void, the only inhabitants of $SMaybe$ $Void$ are $SNothing$ and $\bot$. The former will match on the first equation; the latter will make the first equation diverge. In neither case will execution flow to the second equation, so it is redundant and can be deleted.

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

\[
\begin{align*}
  u :: () \rightarrow \text{Int} \\
  u() | \text{False} = 1 \\
  | \text{True} = 2 \\
  u_\bot &= 3
\end{align*}
\]

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

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

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

\[
\begin{align*}
  v' :: \text{Maybe Void} \rightarrow \text{Int} \\
  v' \text{Nothing} &= 0 \\
  v' (\text{Just } \bot) &= 1 \quad \text{-- Not redundant, but RHS is inaccessible}
\end{align*}
\]

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

2.4 Type-equality constraints

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

- `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

<table>
<thead>
<tr>
<th>Token</th>
<th>Syntax</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>defn</code></td>
<td><code>clause ::= f pat match</code></td>
</tr>
<tr>
<td><code>clause</code></td>
<td><code>pat ::= x \mid \_ \mid K pat \mid x@pat \mid !pat \mid expr \rightarrow pat</code></td>
</tr>
<tr>
<td><code>match</code></td>
<td><code>grhs ::= \mid guard = expr</code></td>
</tr>
<tr>
<td><code>guard</code></td>
<td><code>guard ::= pat \leftarrow expr \mid expr \mid 1 \text{let } x = expr</code></td>
</tr>
</tbody>
</table>

#### Fig. 1. Source syntax

```
data T a b where
g1 :: T Int b \rightarrow b \rightarrow Int
T1 :: T Char Bool
T2 :: T Char Bool

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

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

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

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

### 3 LOWER YOUR GUARDS: A NEW COVERAGE CHECKER

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

- First, we desugar the complex source Haskell syntax into a `guard tree t : Gdt` (Section 3.1). The language of guard trees is tiny but expressive, and allows the subsequent passes to be entirely independent of the source syntax. LYG can readily be adapted to other languages simply by changing the desugaring algorithm.
Fig. 2. Bird’s eye view of pattern match checking

Guard syntax

\[
\begin{align*}
k, n, m & \in \mathbb{N} & \gamma & \in \text{TyCt} & \rightrightarrows & t_1 \sim t_2 \mid \ldots \\
K & \in \text{Con} & p & \in \text{Pat} & \rightrightarrows & - \\
x, y, a, b & \in \text{Var} & \tau, \sigma & \in \text{Type} & \rightrightarrows & K \tilde{p} \\
e & \in \text{Expr} & g & \in \text{Grd} & \rightrightarrows & \text{let } x : \tau = e \\
& & & & \mid K \tilde{r} \tilde{y} \tilde{c} \\
& & & & \mid K \tilde{a} \tilde{y} \tilde{c} \leftarrow x \\
& & & \mid & !x
\end{align*}
\]

Refinement type syntax

\[
\begin{align*}
\Gamma & \rightrightarrows \emptyset \mid \Gamma, x : \tau \mid \Gamma, a \\
\varphi & \rightrightarrows \vee \mid \times \mid K \tilde{a} \tilde{y} \tilde{c} \leftarrow x \mid x \neq K \mid x \approx \bot \mid x \neq \bot \mid \text{let } x = e \\
\Phi & \rightrightarrows \varphi \mid \Phi \land \Phi \mid \Phi \lor \Phi \\
\Theta & \rightrightarrows \langle \Gamma \mid \Phi \rangle
\end{align*}
\]

Clause tree syntax

\[
\begin{align*}
t & \in \text{Gdt} & \rightrightarrows & \text{GRhs } n \mid t_1; t_2 \mid \text{Guard } g \ t \\
u & \in \text{Ant} & \rightrightarrows & \text{ARhs } \Theta \ n \mid u_1; u_2 \mid \text{Bang } \Theta \ u
\end{align*}
\]

Graphical notation

\[
\begin{align*}
t_1 & \rightrightarrows t_1; t_2 \\
t_2 & \rightrightarrows u_1; u_2 \\
g_1, \ldots, g_n & \rightrightarrows \text{Guard } g_1 \ldots (\text{Guard } g_n \ t) \\
t & \rightrightarrows \Theta \ n & \rightrightarrows & \text{Bang } \Theta \ u \\
n & \rightrightarrows \text{GRhs } n & \rightrightarrows & \text{ARhs } \Theta \ n
\end{align*}
\]

Fig. 3. IR syntax

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

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

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

### 3.1 Desugaring to guard trees

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

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

```plaintext
f (Just ([xs, _])) ys@Nothing = 1
f Nothing        (g → True)  = 2
```

This desugars to the following guard tree:

```
!x1, Just t1 ← x1, !t1, (t2, t3) ← t1, !t2, let xs = t2, let ys = x2, !ys, Nothing ← ys ← 1

!x1, Nothing ← x1, let t3 = g x2, !y, True ← t3 ← 2
```

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 $\text{Just } t_1$; then match $t_1$ against $(t_2, t_3)$; and so on”. If any of those matches
fail, we fall through into the second line.

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

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

The desugaring algorithm, $D$, 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 $D$ 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. $g \to True$) 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:

```haskell
liftEq Nothing Nothing = True
liftEq mx (Just y) | Just x <- mx, x == y = True
            | otherwise      = False
```

It desugars thus:
Operations on $\Theta$

$\langle \Gamma \mid \Phi \rangle \wedge \varphi = \langle \Gamma \mid \Phi \wedge \varphi \rangle$
$\langle \Gamma \mid \Phi_1 \rangle \cup \langle \Gamma \mid \Phi_2 \rangle = \langle \Gamma \mid \Phi_1 \lor \Phi_2 \rangle$

Checking guard trees

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

$\mathcal{U}(\langle \Gamma \mid \Phi \rangle, \text{GRhs } n) = \langle \Gamma \mid \times \rangle$
$\mathcal{U}(\Theta, t_1; t_2) = \mathcal{U}(\mathcal{U}(\Theta, t_1), t_2)$
$\mathcal{U}(\Theta, \text{Guard } (x) t) = \mathcal{U}(\Theta \land (x \neq \bot), t)$
$\mathcal{U}(\Theta, \text{Guard } (\text{let } x = e) t) = \mathcal{U}(\Theta \land (\text{let } x = e), t)$
$\mathcal{U}(\Theta, \text{Guard } (K \overline{a} \overline{y} \overline{y}: \overline{\tau} \leftarrow x) t) = (\Theta \land (x \neq K)) \cup \mathcal{U}(\Theta \land (K \overline{a} \overline{y} \overline{y}: \overline{\tau} \leftarrow x), t)$

$\mathcal{A}(\Theta, \text{GRhs } n) = \mathcal{A}\text{Rhs } \Theta n$
$\mathcal{A}(\Theta, (t_1; t_2)) = \mathcal{A}(\Theta, t_1); \mathcal{A}(\mathcal{U}(\Theta, t_1), t_2)$
$\mathcal{A}(\Theta, \text{Guard } (x) t) = \text{Bang } (\Theta \land (x \approx \bot)) \cdot \mathcal{A}(\Theta \land (x \neq \bot), t)$
$\mathcal{A}(\Theta, \text{Guard } (\text{let } x = e) t) = \mathcal{A}(\Theta \land (\text{let } x = e), t)$
$\mathcal{A}(\Theta, \text{Guard } (K \overline{a} \overline{y} \overline{y}: \overline{\tau} \leftarrow x) t) = \mathcal{A}(\Theta \land (K \overline{a} \overline{y} \overline{y}: \overline{\tau} \leftarrow x), t)$

Fig. 5. Coverage checking

![Coverage Checking Diagram]

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

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

3.2 Checking guard trees

In the next step, we transform the guard tree into an annotated tree, Ant, and an uncovered set, $\Theta$.

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 $\Phi$. For example:

$\langle x : \text{Bool} \mid \checkmark \rangle$ denotes $\{\bot, \text{True}, \text{False}\}$
$\langle x : \text{Boo} \mid \times \neq \bot \rangle$ denotes $\{\text{True}, \text{False}\}$
$\langle x : \text{Bool} \mid \text{True} \leftarrow x \rangle$ denotes $\{\text{True}\}$
$\langle mx : \text{Maybe Bool} \mid \text{Just } x \leftarrow mx, x \neq \bot \rangle$ denotes $\{\text{Just True, Just False}\}$

The syntax of $\Phi$ is given in Figure 3. It consists of a collection of literals $\varphi$, 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 $\Phi$, but we omit that here. The literal $\checkmark$ means “true”, as illustrated above; while $\times$ means “false”, so that $\langle \text{Gamma} \mid \times \rangle$ denotes $\emptyset$. 

```plaintext
\text{Just } x \leftarrow mx, \text{let } t = x == y, \text{lt, True } \leftarrow t \rightarrow 2
\text{Just } y \leftarrow my \rightarrow 1
\text{Nothing } \leftarrow mx \rightarrow 1
\text{Nothing } \leftarrow my \rightarrow 1
\text{otherwise, True } \leftarrow \text{otherwise } \rightarrow 3
```
The uncovered set function \( U(\Theta, t) \), defined in Figure 5, computes a refinement type describing the values in \( \Theta \) that are not covered by the guard tree \( t \). It is defined by a simple recursive descent over the guard tree, using the operation \( \Theta \hat{\land} \varphi \) (also defined in Figure 5) to extend \( \Theta \) with an extra literal \( \varphi \).

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

3.3 Reporting errors

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

\[
\text{data } T = A \mid B \mid C
\]

\[
f (\text{Just } A) = \text{True}
\]

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

\[
\Theta_f = \langle x: \text{Maybe } T \mid x \neq \bot \land (x \neq \text{Just } \vee (\text{Just } y \leftarrow x \land y \neq \bot \land (y \neq A \lor (A \leftarrow y \land x)))) \rangle
\]

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

Missing equations for function ‘f’:

\[
\begin{align*}
f \text{ Nothing} &= \ldots \\
f (\text{Just } B) &= \ldots \\
f (\text{Just } C) &= \ldots
\end{align*}
\]

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

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

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

\[
\begin{align*}
u' () | \text{False} = 1 &\quad \Theta_1 \downarrow \Theta_2 1 \\
| \text{False} = 2 &\sim \Theta_3 2 \\
\underline{u'} = 3 &\quad \Theta_4 3
\end{align*}
\]

\(^1\) Our implementation avoids this duplicated work – see Section 5.1 – but the formulation in Figure 5 emphasises clarity over efficiency.
Collect accessible \((\bar{k})\), inaccessible \((\bar{n})\) and redundant \((\bar{m})\) GRHSs

\[
\mathcal{R}(u) = (\bar{k}, \bar{n}, \bar{m})
\]

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

\[
\mathcal{R}(t; u) = (k k', \bar{n} n', \bar{m} m') \quad \text{where} \quad (k k', \bar{n} n', \bar{m} m') = \mathcal{R}(t)
\]

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

Normalised refinement type syntax

\[
\nabla := \times \mid (\Gamma \parallel \Delta) \quad \text{Normalised refinement type}
\]

\[
\Delta := \emptyset \mid \Delta, \delta \quad \text{Set of constraints}
\]

\[
\delta := \gamma \mid x \approx K \bar{a} \bar{y} \mid x \neq K \mid x \approx \perp \mid x \neq \perp \mid x \approx y \quad \text{Constraints}
\]

Generate inhabitants of \(\Theta\)

\[
\mathcal{G}(\Theta) = \mathcal{P}(\bar{p})
\]

\[
\mathcal{G}(\langle \Gamma \mid \Phi \rangle) = \{ \mathcal{E}(\nabla, \text{dom}(\Gamma)) \mid \nabla \in C(\langle \Gamma \parallel \emptyset \rangle, \Phi) \}
\]

Construct inhabited \(\nabla\)s from \(\Phi\)

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

\[
C(\nabla, \Phi_1 \land \Phi_2) = \bigcup \{ C(\nabla, \Phi_2) \mid \nabla' \in C(\nabla, \Phi_1) \}
\]

\[
C(\nabla, \Phi_1 \lor \Phi_2) = C(\nabla, \Phi_1) \cup C(\nabla, \Phi_2)
\]

Expand variables to \(\text{Pat}\) with \(\nabla\)

\[
\mathcal{E}(\nabla, \bar{x}) = \bar{p}
\]

\[
\mathcal{E}(\nabla, \epsilon) = \epsilon
\]

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

Finding the representative of a variable in \(\Delta\)

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

Fig. 6. Generating inhabitants of \(\Theta\) via \(\nabla\)
\( \Theta_2 \) and \( \Theta_3 \) are uninhabited (because of the False guards). But we cannot delete both GRHSs as redundant, because that would make the call \( u \downarrow \bot \) return 3 rather than diverging. Rather, we want to report the first GRHSs as inaccessible, leaving all the others as redundant.

### 3.4 Generating inhabitants of a refinement type

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

- Flatten \( \Theta \) into a set of normalised refinement types \( \nabla \), by the call \( C(\langle \Gamma \parallel \emptyset \rangle, \Phi) \); see Section 3.6.
- For each such \( \nabla \), expand \( \Gamma \) into a list of patterns, by the call \( E(\nabla, \text{dom}(\Gamma)) \); see Section 3.5.

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

- \( \Delta \) is simply a conjunction of literals \( \delta \); there are no disjunctions. Instead, disjunction reflects in the fact that \( C \) returns a set of normalised refinement types.

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

1. **Mutual compatibility**: No two constraints in \( \Delta \) should conflict with each other, where \( x \approx \bot \) conflicts with \( x \neq \bot \) and \( x \approx K \_ \_ \) conflicts with \( x \neq K \) for all \( x \).
2. **Triangular form**: A \( x \approx y \) constraint implies absence of any other constraints mentioning \( x \) in its left-hand side.
3. **Single solution**: There is at most one positive constructor constraint \( x \approx K \_ \_ \_ \_ \) for a given \( x \).
4. **Incompletely matched**: If \( x: \tau \in \Gamma \) and \( \tau \) reduces to a data type under type constraints in \( \Delta \), there must be at least one constructor \( K \) (or \( \bot \)) which \( x \) can be instantiated to without contradicting I1; see Section 3.7.

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

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

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

### 3.5 Expanding a normalised refinement type to a pattern

Expanding a \( \nabla \) to a pattern vector, by calling \( E(\nabla) \) in Figure 6, is syntactically heavy, but straightforward. When there is a solution like \( \Delta(x) \approx \text{Just} \ y \in \Delta \) 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 \( \text{Just} \). Invariant I3 guarantees that there is at most one such solution and \( E \) is well-defined.
Add a formula literal to $\nabla$: $\nabla \oplus_{\varphi} \varphi = \nabla$

$$\nabla \oplus_{\varphi} \times = \times$$ (1)

$$\nabla \oplus_{\varphi} \checkmark = \nabla$$ (2)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\varphi} K \overline{a} \overline{y} \overline{y}: \overline{\varphi} \leftarrow x = \langle \Gamma, \overline{a}, \overline{y}: \overline{\varphi} \parallel \Delta \rangle \oplus_{\delta} \overline{y} \oplus_{\delta} \overline{y}' \neq \perp \oplus_{\delta} x \approx K \overline{a} \overline{y}$$ (3)

where $\overline{y}'$ bind strict fields

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\varphi} \text{let } x: \tau = K \overline{a} \overline{y} \overline{\varphi} = \langle \Gamma, x: \tau, \overline{a} \parallel \Delta \rangle \oplus_{\delta} \overline{a} \sim \overline{\sigma} \oplus_{\delta} x \approx K \overline{a} \overline{y} \oplus_{\varphi} \text{ let } y: \tau' = e$$ (4)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\varphi} \Delta \| x \approx y$$ (5)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\varphi} \text{let } x: \tau = e$$ (6)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\varphi} \varphi = \langle \Gamma \parallel \Delta \rangle \oplus_{\delta} \varphi$$ (7)

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

$$\times \oplus_{\delta} \delta = \times$$ (8)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\delta} \varphi = \begin{cases} \langle \Gamma \parallel (\Delta, \varphi) \rangle & \text{if type checker deems } \varphi \text{ compatible with } \Delta \\ \times & \text{otherwise} \end{cases}$$ (9)

$$\times \oplus_{\delta} (a \sim b \oplus_{\delta} \overline{a} \overline{y} \approx \overline{z}) = \langle \Gamma \parallel (\Delta, \Delta(x) \approx K \overline{a} \overline{y}) \rangle \text{ if } \Delta(x) \approx K \overline{a} \overline{y} \in \Delta$$ (10)

$$\times \oplus_{\delta} \times = \langle \Gamma \parallel (\Delta, \Delta(x) \neq K) \rangle \text{ if } \Delta(x) \neq K$$ (11)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\delta} \Delta(x) \approx \perp$$ (12)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\delta} \Delta(x) \neq \perp$$ (13)

$$\langle \Gamma \parallel \Delta \rangle \oplus_{\delta} x \approx y$$ (14)

where $x' = \Delta(x)$ and $y' = \Delta(y)$

$$\nabla \mid x = \Delta$$

$$\nabla \mid x = \Delta$$

$$(\Delta, x \approx K \overline{a} \overline{y}) \mid x = \Delta \mid x$$

$$(\Delta, x \neq \perp) \mid x = \Delta \mid x$$

$$(\Delta, x \neq K) \mid x = \Delta \mid x$$

$$(\Delta, x \approx \perp) \mid x = \Delta \mid x$$

$$(\Delta, x \neq \perp) \mid x = \Delta \mid x$$

$$(\Delta, \delta) \mid x = (\Delta \mid x), \delta$$

Fig. 7. Adding a constraint to the normalised refinement type $\nabla$. 
3.6 Normalising a refinement type

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

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

If there is a solution involving a different constructor like $\Delta(x) \approx Nothing$ or if there was a negative constructor constraint $\Delta(x) \neq Just$, the new constraint is incompatible with the existing solution. Otherwise, the constraint is compatible and is added to $\Delta$.

Adding a negative constructor constraint $x \neq Just$ is quite similar (equation (11)), except that we have to make sure that $x$ still satisfies I4, which is checked by the $\nabla \vdash \Delta(x)$ judgment (cf. section 3.7) in fig. 8. Handling positive and negative constraints involving $\bot$ is analogous.

Adding a type constraint $y$ (equation (9)) entails calling out to the type checker to assert that the constraint is consistent with existing type constraints. Afterwards, we have to ensure I4 is upheld for all variables in the domain of $\Gamma$, because the new type constraint could have rendered a type empty. To demonstrate why this is necessary, imagine we have $\langle x : a \parallel x \neq \bot \rangle$ and try to add $a \sim Void$. Although the type constraint is consistent, $x$ in $\langle x : a \parallel x \neq \bot, a \sim Void \rangle$ is no longer inhabited. There is room for being smart about which variables we have to re-check: For example, we can exclude variables whose type is a non-GADT data type.

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

3.7 Testing for inhabitation

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

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

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 $\nabla$, because that means we don’t have any $\nablas$ floating around in the checking process that actually aren’t inhabited and trigger false positive warnings. But that also means that the $\vdash$ relation is undecidable! Consider the following example:
#### Test if x is inhabited considering \( \nabla \)
\[
\frac{\left( \Gamma \parallel \Delta \right) \oplus \delta \ x \approx \bot \neq x}{\nabla \vdash x} \quad \vdash \text{BOT} \quad x : \tau \in \Gamma \quad \text{Cons}(\left( \Gamma \parallel \Delta \right), \tau) = \bot \quad \vdash \text{NoCpl}
\]

\[
\frac{x : \tau \in \Gamma \quad K \in \text{Cons}(\left( \Gamma \parallel \Delta \right), \tau)}{\text{Inst}(\left( \Gamma \parallel \Delta \right), x, K) \neq \times} \quad \vdash \text{INST}
\]

#### Find data constructors of \( \tau \)
\[
\text{Cons}(\left( \Gamma \parallel \Delta \right), \tau) = \begin{cases} 
\overline{K} & \tau = T \sigma \text{ and } T \text{ data type with constructors } \overline{K} \\
\bot & \text{otherwise}
\end{cases}
\]

#### Instantiate \( x \) to data constructor \( K \)
\[
\text{Inst}(\nabla, x, K) = \nabla
\]

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

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

#### 4.1 Long-distance information
Coverage checking should also work for \textit{case} expressions and nested function definitions, like

\[
f \ True = 1
\]

\[
f \ x = \ldots (\text{case } x \text{ of } \{ \text{False} \rightarrow 2; \ True \rightarrow 3 \}) \ldots
\]
LYG as is will not produce any warnings for this definition. But the reader can easily make the “long distance connection” that the last GRHS of the case expression is redundant! That simply follows by context-sensitive reasoning, knowing that \( x \) was already matched against \( \text{True} \).

In terms of LYG, the input values of the second GRHS \( \Theta_2 \) (which determine whether the GRHS is accessible) encode the information we are after. We just have to start checking the case expression starting from \( \Theta_2 \) as the initial set of reaching values instead of \( \langle x : \text{Bool} \mid √ \rangle \).

### 4.2 Empty case

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

\[
\begin{align*}
\text{absurd1 } _\_ & = ⊥ \\
\text{absurd2 } ! _\_ & = ⊥ \\
\text{absurd3 } x & = \text{case } x \text{ of } \{ \}
\end{align*}
\]

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

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

\[
\begin{align*}
\mathcal{U}(\Theta, \Theta(\text{GEmpty})) & = \Theta \\
\mathcal{A}(\Theta, \Theta(\text{GEmpty})) & = \text{AEmpty}
\end{align*}
\]

Since \( \text{EmptyCase} \), unlike regular case, evaluates its scrutinee to WHNF before matching any of the patterns, the set of reaching values is refined with a \( x ≠ ⊥ \) constraint before traversing the guard tree, thus \( \mathcal{U}(\langle \Gamma \mid x ≠ ⊥ \rangle, \Theta(\text{GEmpty})) \).

### 4.3 View patterns

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

\[
\begin{align*}
\text{let } y_1 & = \text{reverse } x_1, ! y_1, \text{Nothing} ← y_1 \quad 1 \\
\text{let } y_2 & = \text{reverse } x_2, ! y_2, \text{Just } t_1 ← y_2, ! t_1, (t_2, t_3) ← t_1 \quad 2
\end{align*}
\]

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

This can be fixed by maintaining equivalence classes of semantically equivalent expressions in \( \Delta \), similar to what we already do for variables. We simply extend the syntax of \( δ \) and change the last let case of \( θ_φ \). Then we can handle the new constraint in \( θ_δ \), as follows:

\[
\begin{align*}
δ & = ... \mid e ≃ x \\
\langle \Gamma \parallel Δ \rangle \oplus_δ \text{let } x : τ = e & = \langle \Gamma, x : τ \parallel Δ \rangle \oplus_δ e ≃ x \\
\langle \Gamma \parallel Δ \rangle \oplus_δ e ≃ x & = \begin{cases} 
\langle \Gamma \parallel Δ \rangle \oplus_δ x ≃ y, & \text{if } e' ≃ y \in Δ \text{ and } e \equiv_Δ e'
\langle \Gamma \parallel Δ, e ≃ Δ(x) \rangle, & \text{otherwise}
\end{cases}
\end{align*}
\]

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

### 4.4 Pattern synonyms

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

\[
\begin{align*}
  \text{cl} & ::= K \mid P \\
  \text{pat} & ::= x \mid - \mid \text{cl pat} \mid \text{x@pat} \mid \ldots
\end{align*}
\]

\[
\begin{align*}
  P & \in \text{PS} \\
  C & \in \text{CL} \\
  P & \in \text{Pat} \quad ::= \_ \mid C \overrightarrow{p} \mid \ldots
\end{align*}
\]

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

\[
\begin{align*}
\text{pattern } P &= () \\
\text{pattern } Q &= ()
\end{align*}
\]

Knowing that the definitions of $P$ and $Q$ completely overlap, we can see that the match on $Q$ will cover all values that could reach $P$, so clearly $P$ is redundant. A sound approximation to that would be not to warn at all. Why not introduce the new syntactic variant in the first place? Consider

\[
\begin{align*}
\text{pattern } P &= () \\
\text{pattern } Q &= ()
\end{align*}
\]

\[
\begin{align*}
 n = \text{case } P \text{ of } Q & \rightarrow 1; P \rightarrow 2
\end{align*}
\]

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

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

\[
\langle \Gamma \parallel \Delta \rangle \otimes_\delta x \approx C \overrightarrow{a} \overrightarrow{y} = \begin{cases} 
\langle \Gamma \parallel (\Delta(x) \approx C \overrightarrow{b} \overrightarrow{z} \in \Delta \otimes_\delta \overrightarrow{a} \overrightarrow{y} ) \times \Delta \rangle & \text{if } \Delta(x) \approx C \overrightarrow{b} \overrightarrow{z} \in \Delta \\
\langle \Gamma \parallel (\Delta(x) \approx C \overrightarrow{a} \overrightarrow{y} ) \rangle & \text{if } \Delta(x) \not\approx C \notin \Delta \text{ and } \langle \Gamma \parallel \Delta \rangle \vdash \Delta(y) \\
\langle \Gamma \parallel (\Delta(x) \approx C \overrightarrow{a} \overrightarrow{y} ) \rangle & \text{otherwise}
\end{cases}
\]

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

#### 4.5 COMPLETE pragmas

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

We have $\text{I-INST}$ from fig. 8 currently making sure that this COMPLETE set is in fact inhabited. We also have $\text{I-NoCpl}$ that handles the case when we can’t find any COMPLETE set for the given type
(think \(x : \text{Int} \rightarrow \text{Int}\)). 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:

\[
((\Gamma \parallel \Delta) \, \Theta, x \approx \bot) \neq \times \quad \frac{x : \tau \in \Gamma}{\langle \Gamma \parallel \Delta \rangle \vdash x} \quad \frac{\text{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = C_1, \ldots, C_n}{\langle \Gamma \parallel \Delta \rangle \vdash x} \quad \frac{\text{Inst}(\langle \Gamma \parallel \Delta \rangle, x, C_j) \neq \times}{(\langle \Gamma \parallel \Delta \rangle \vdash x)}
\]

\[
\text{Cons}(\langle \Gamma \parallel \Delta \rangle, \tau) = \begin{cases} C_1, \ldots, C_n & \tau = T \overline{s} \text{ and } T \text{ type constructor with COMPLETE sets } C_1, \ldots, C_n \\ \emptyset & \text{otherwise} \end{cases}
\]

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

4.6 Other extensions

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

5 IMPLEMENTATION

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

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

5.1 Interleaving \(\mathcal{U} \) and \(\mathcal{A} \)

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

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

What we really want is to summarise a \(\Theta \) into a more compact canonical form before doing these kinds of splits. But that’s exactly what \(\nabla \) is! Therefore, in our implementation we don’t pass around and annotate refinement types, but the result of calling \(C \) on them directly.

You can see the resulting definition in fig. 9. The readability is clouded by unwrapping of pairs. \(\mathcal{UA} \) requires that each \(\nabla \) individually is non-empty, i.e. not \(\times \). This invariant is maintained by adding \(\phi \) constraints through \(\ominus \phi \), which filters out any \(\nabla \) that would become empty.

5.2 Throttling for graceful degradation

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

\[(\vphi_1 \ldots \vphi_n) \oplus_\varphi \vphi = \begin{cases} 
(\Gamma \parallel \Delta) (\vphi_2 \ldots \vphi_n) & \text{if } (\Gamma \parallel \Delta) = \vphi \oplus_\varphi \vphi \\
(\vphi_2 \ldots \vphi_n) & \text{otherwise}
\end{cases}\]

\[\text{fig. 9. Fast coverage checking}\]

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

\textbf{data} \(T = A \mid B; f1, f2 :: \text{Int} \rightarrow T\)

\[g -
| A \leftarrow f1 1, A \leftarrow f2 1 = ()
| A \leftarrow f1 2, A \leftarrow f2 2 = ()
| A \leftarrow f1 N, A \leftarrow f2 N = ()
\]

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

Instead of refining \(\vphi\) with the pattern guard, leading to a split, we could just continue with the original \(\vphi\), thus forgetting about the \(a_1 \neq A\) or \(b_1 \neq A\) constraints. In terms of the modeled refinement type, \(\vphi\) is still a superset of both refinements, and thus a sound overapproximation.

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

\[\text{UA}(\vphi, \text{Guard}(K a \bar{y} y : x) t) = \left[\left(\vphi_\varphi (x \neq K) \vphi'\right), u\right], \text{UA}(\vphi_\varphi (K a \bar{y} y : x) t)\]
where the new throttling operator $\lfloor \_ \rfloor$ is defined simply as

$$\lfloor \nabla \rfloor_{\nabla'} = \begin{cases} \nabla & \text{if } |\{\nabla}\}| \leq K \\ \nabla' & \text{otherwise} \end{cases}$$

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

### 5.3 Maintaining residual COMPLETE sets

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

```haskell
data T = A1 | ... | A1000

pattern P = ...
{-# COMPLETE A1, P #-} ...

f A1 = 1
f A2 = 2
f A1000 = 1000
```

$f$ is exhaustively defined. To see that we need to perform an inhabitation test for the match variable $x$ after the last clause. The test will conclude that the built-in COMPLETE set was completely overlapped. But in order to conclude that, our algorithm tries to instantiate $x$ (via $\text{\textit{\textit{\textbackslash \Gamma}}}\text{\textit{-INST}}$) 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 \textit{after} every clause, involving $O(n)$ instantiations each.

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

### 5.4 Reporting uncovered patterns

The expansion function $E$ 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 $f \text{ True } = ()$. The computed uncovered set of $f$ is the refinement type $\langle x : \text{Bool} | x \neq \bot, x \neq \text{True} \rangle$, which crucially contains no positive information! As a result, expanding the resulting $\nabla$ (which looks quite similar) with $E$ just unhelpfully reports $\_\_\_$ as an uncovered pattern. Our implementation thus splits the $\nabla$ into (possibly multiple) sub-$\nabla$s with positive information on variables we have negative information on before handing off to $E$.

### 6 EVALUATION

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

---

2https://gitlab.haskell.org/ghc/head.hackage/commit/30a310fd8033629e1cbb5a9696250b22db5f7045
<table>
<thead>
<tr>
<th>Time (milliseconds)</th>
<th>Megabytes allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.8.3</td>
<td>HEAD</td>
</tr>
<tr>
<td>T11276</td>
<td>1.16</td>
</tr>
<tr>
<td>T11303</td>
<td>28.1</td>
</tr>
<tr>
<td>T11303b</td>
<td>1.15</td>
</tr>
<tr>
<td>T11374</td>
<td>4.62</td>
</tr>
<tr>
<td>T11822</td>
<td>1.060</td>
</tr>
<tr>
<td>T11195</td>
<td>2.680</td>
</tr>
<tr>
<td>T17096</td>
<td>7,470</td>
</tr>
<tr>
<td>PmSeriesS</td>
<td>44.5</td>
</tr>
<tr>
<td>PmSeriesT</td>
<td>48.3</td>
</tr>
<tr>
<td>PmSeriesV</td>
<td>131</td>
</tr>
</tbody>
</table>

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

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

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

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

go' _ _ xs | False = error (show xs)
go' _ _ xs = err xs

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

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

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

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

6.2 GHC issues
Implementing LYG in GHC has fixed over 30 bug reports related to coverage checking. These
include:
  - Better compile-time performance [GHC issue 2015a, 2016e, 2019a,b]
  - More accurate warnings for empty case expressions [GHC issue 2015b, 2017f, 2018e,g, 2019c]
  - More accurate warnings due to LYG’s desugaring [GHC issue 2016c,d, 2017d, 2018a, 2020d]
  - More accurate warnings due to improved term-level reasoning [GHC issue 2016a, 2017a,
    2018b,c,d,h, 2019d,e,h]
  - More accurate warnings due to tracking long-distance information [GHC issue 2019k, 2020a,b]
  - Improved treatment of COMPLETE sets [GHC issue 2016b, 2017b,c,e,g, 2018j, 2019f,g,i]
  - Better treatment of strictness, bang patterns, and newtypes [GHC issue 2018f,i, 2019j,l]

7 RELATED WORK

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

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

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

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

While GMTM’s term oracle is customisable, it is not as simple to customize as one might hope.
The formalism in Karachalias et al. [2015] represents all guards as p ← e, where p is a pattern and
e is an expression. This is a straightforward, syntactic representation, but it also makes it more
difficult to analyse when e is a complicated expression. This is one of the reasons why it is difficult
for GMTM to accurately give warnings for the safeLast function, since it would require recognizing
that both clauses scrutinise the same expression in their view patterns.
LYG makes analysing term equalities simpler by first desugaring guards from the surface syntax to guard trees. The $\oplus_\varphi$ function, which is roughly a counterpart to GMTM’s term oracle, can then reason about terms arising from patterns. While $\oplus_\varphi$ is already more powerful than a trivial term oracle, its real strength lies in the fact that it can easily be extended, as LYG’s treatment of view patterns (section 4.3) demonstrates. While GMTM’s term oracle could be improved to accomplish the same thing, it is unlikely to be as straightforward of a process as extending $\oplus_\varphi$.

7.2 Comparison with similar coverage checkers

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

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

\begin{align*}
f \text{ True} &= 1 \\
f \bot &= 2
\end{align*}

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

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

7.3 Positive and negative information

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

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

\begin{align*}
\text{pattern True’} &= \text{ True} \\
\{-# COMPLETE True’, False #-\} &\quad f \text{ False} = 1 \\
\text{pattern True’} &= \text{ True} \\
\{-# COMPLETE False, True’ #-\} &\quad f \text{ True’} = 2 \\
&\quad f \text{ True} = 3
\end{align*}

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

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

\begin{verbatim}
data T = A1 | ... | A1000
\end{verbatim}

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

7.4 Strict fields in inhabitation testing

To our knowledge, the \texttt{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 \texttt{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 \texttt{v} to OCaml and Idris 3:

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

7.5 Refinement types in coverage checking

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

\begin{verbatim}
fibPartial :: Integer \rightarrow Integer
fibPartial 0 = 0
fibPartial 1 = 1
\end{verbatim}

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

\begin{verbatim}
error :: \{v : String | false\} \rightarrow a
\end{verbatim}

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

8 CONCLUSION

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

\footnote{Idris has separate compile-time and runtime semantics, the latter of which is call by value.}
expressible in coverage checkers that work over structural pattern matches. This allows LYG to report more accurate warnings while also avoiding performance issues when checking complex programs. Moreover, LYG is extensible, and we anticipate that this will streamline the process of checking new forms of patterns in the future.

REFERENCES


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


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


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


GHC issue. 2018i. nonVoid is too conservative w.r.t. strict argument types. https://gitlab.haskell.org/ghc/ghc/issues/15573


Journal of Functional Programming


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/17714


