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

ACM Reference Format:
Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. Lower Your Guards: A Compositional Pattern-Match Coverage Checker. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2020), 28 pages.

1 INTRODUCTION

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

$f :: \mathit{Int} \rightarrow \mathit{Bool}$
$f\ 0 = \mathit{True}$
$f\ 0 = \mathit{False}$

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

To avoid these mishaps, compilers for languages with pattern matching often emit warnings whenever a programmer misuses patterns. Such warnings indicate if a function is missing clauses (i.e., if it is *non-exhaustive*), 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 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 contructors, empty case expressions, and long-distance effects (**??**). 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. We argue that this algorithm is insufficient in a number of key ways. It does not account for a number of important language

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

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

features and even gives incorrect results in certain cases. Moreover, the implementation of this algorithm in GHC is inefficient and has proved to be difficult to maintain due to its complexity.

In this paper we propose a new, compositional coverage-checking algorithm, called Lower Your Guards (LYG), that is much simpler, more modular, *and* more powerful than GMTM. We make the following contributions:

- We characterise the nuances of coverage checking that not even the algorithm in Karachalias et al. [2015] handles (Section 2). We also identify issues in GHC's implementation of this algorithm.
- We given an overview of our new 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 shore up the intuitions of Section 3 with a formal treatment in Section 4.
- We have implemented LYG in GHC (Section 6). **Ryan:** More details.

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 (which we will expound upon further in Section 7) accounts for some of these nuances, but not all of them. In this section we identify all of the language features that complicate coverage checking. While these features may seem disparate at first, we will later show in 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
```

The first guard is a *boolean guard* that succeeds (i.e., evaluates its right-hand side) if the expression in the guard returns *True*. The second guard is a *pattern guard* that succeeds if the pattern in the guard successfully matches. Moreover, a guard can have **let** bindings or even multiple checks, as the third guard demonstrates. The fourth guard 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 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. In fact, coverage checking for guards in the general case is an undecidable problem. While we cannot accurately check *all* uses of guards, we can at least give decent warnings for some common use cases for guards. 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
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.

2.2 Programmable patterns

Expressions in guards are far from the only source of undecidability that the coverage checker must cope with. GHC extends the pattern language in 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 character and xs is the rest of the Text
length :: Text -> Int
length (Text.null -> True)           = 0
length (Text.uncons -> Just (_, xs)) = 1 + length xs
```

When compiled, a view pattern desugars into a pattern guard. The desugared version of *length*, for instance, would look like this: **Ryan:** Consider putting these versions of *length* side-by-side to save space

$$
\begin{array}{l}
\mathit{length}' :: \mathit{Text} \rightarrow \mathit{Int} \\
\mathit{length}'\ t \mid \mathit{True} \leftarrow \mathit{Text.null}\ t \qquad\quad = \mathit{True} \\
\qquad\quad \mid \mathit{Just}\ (\_, \mathit{xs}) \leftarrow \mathit{Text.uncons}\ t = \mathit{False}
\end{array}
$$

As a result, any coverage-checking algorithm that can handle guards can also handle view patterns, provided that the view patterns desugar to guards that are not too complex. For instance, LYG would not be able to conclude that *length* is exhaustive, but it would be able to conclude that the *safeLast* function below is exhaustive:

$$
\begin{array}{l}
\mathit{safeLast} :: [\,a\,] \rightarrow \mathit{Maybe}\ a \\
\mathit{safeLast}\ (\mathit{reverse} \rightarrow [\,])\qquad\ = \mathit{Nothing} \\
\mathit{safeLast}\ (\mathit{reverse} \rightarrow (x : \_)) = \mathit{Just}\ x
\end{array}
$$

2.2.2 Pattern synonyms.

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:

$$
\begin{array}{l}
\mathit{pattern\ Nil} :: \mathit{Text} \\
\mathit{pattern\ Nil} \leftarrow (\mathit{Text.null} \rightarrow \mathit{True}) \\
\mathit{pattern\ Cons} :: \mathit{Char} \rightarrow \mathit{Text} \rightarrow \mathit{Text} \\
\mathit{pattern\ Cons}\ x\ \mathit{xs} \leftarrow (\mathit{Text.uncons} \rightarrow \mathit{Just}\ (x, \mathit{xs}))
\end{array}
$$

$$
\begin{array}{l}
\mathit{length} :: \mathit{Text} \rightarrow \mathit{Int} \\
\mathit{length}\ \mathit{Nil} = 0 \\
\mathit{length}\ (\mathit{Cons}\ x\ \mathit{xs}) = 1 + \mathit{length}\ \mathit{xs}
\end{array}
$$

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.

Intuitively, *Text.null* and *Text.uncons* together are exhaustive. GHC allows programmers to communicate this sort of intuition to the coverage checker in the form of *COMPLETE* sets. **Ryan:** Cite the *COMPLETE* section of the users guide. **SG:** I'm using `COMPLETE` for marking up COMPLETE pragmas. But I'm not sold on either way. 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* 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:

$$
\begin{array}{l}
\textbf{data}\ \mathit{Void}\quad \text{-- No data constructors} \\
\textbf{data}\ \mathit{SMaybe}\ a = \mathit{SJust}\ !a \mid \mathit{SNothing}
\end{array}
$$

$v :: \mathit{SMaybe}\ \mathit{Void} \longrightarrow \mathit{Int}$
$v\ \mathit{SNothing} = 0$

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

$v\ (\mathit{SJust}\ \_) = 1$

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

2.3.1 Bang patterns. Strict fields are the primary mechanism for adding extra strictness in ordinary Haskell, but GHC adds another mechanism in the form of *bang patterns*. A bang pattern such as $!\mathit{pat}$ indicates that matching against *pat* always evaluates it to WHNF. While data constructor matches are normally the only patterns that match strictly, bang patterns extend this treatment to other patterns. For example, one can rewrite the earlier v example to use the standard, lazy *Maybe* data type:

$v' :: \mathit{Maybe}\ \mathit{Void} \longrightarrow \mathit{Int}$
$v'\ \mathit{Nothing} = 0$
$v'\ (\mathit{Just}\ !\_) = 1$

The *Just* case in v' is unreachable for the same reasons that the *SJust* case in v is unreachable. Due to the presence of bang patterns, a strictness-aware coverage-checking algorithm must be consider the effects of strictness on any possible pattern, not just those arising from matching on data constructors with strict fields.

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]. Here is one example that demonstrates the interaction between GADTs and coverage checking: **Ryan:** Lay these out side by side with an array or something

data $T\ a\ b$ **where**	$s :: T\ \mathit{Bool}\ b \longrightarrow b$
$\quad T1 :: T\ \mathit{Bool}\ \mathit{Int}$	$s\ T1 = 42$
$\quad T2 :: T\ \mathit{Char}\ \mathit{Int}$	

When s matches against *T1*, the b in the type $T\ \mathit{Bool}\ b$ is known to be an *Int* on the right-hand side of the clause, which is why the use of 42 typechecks. Phrased differently, matching against *T1* brings into scope an *equality constraint* between the types b and *Int*. 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 s function is the code that is *not* used in s. One might wonder if s not matching on the *T2* constructor is an oversight. In fact, the exact opposite is true: matching on s in *T2* would be rejected by the typechecker. This is because *T2* is of type $T\ \mathit{Char}\ \mathit{Int}$, but the argument to s must be of type $T\ \mathit{Bool}\ b$. Matching against *T2* would be tantamount to saying that *Bool* and *Char* are the same type, which is not the case. As a result, s is exhaustive even though it does not match on all of T's data constructors.

Meta variables

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

Pattern Syntax

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

Fig. 1. Source syntax

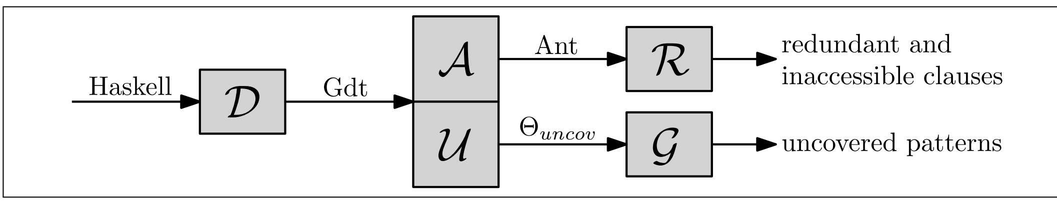


Fig. 2. Bird's eye view of pattern match checking

The same engine that typechecks GADT pattern matches is also used to rule out cases made unreachable by type equalities. There are a variety of coverage checking algorithms that account for GADTs, including GHC's current coverage checker [Karachalias et al. 2015], as well as the checkers for OCaml [Garrigue and Normand 2011], Dependent ML [Xi 1998a,b, 2003], and Stardust [Dunfield 2007]. LYG continues this tradition—see **Ryan:** What section? for LYG's take on GADTs.

3 OVERVIEW OF OUR SOLUTION

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

Figure 2 depicts a high-level overview over this pipeline. Desugaring the complex source Haskell syntax to the very elementary language of guard trees Gdt via $\mathcal{D}$ is an incredible simplification for the checking process. At the same time, $\mathcal{D}$ is the only transformation that is specific to Haskell, implying easy applicability to other languages. The resulting guard tree is then processed by two different functions, $\mathcal{A}$ and $\mathcal{U}$, which compute redundancy information and uncovered patterns, respectively. $\mathcal{A}$ boils down this information into an annotated tree Ant, for which the set of redundant and inaccessible right-hand sides can be computed in a final pass of $\mathcal{R}$. $\mathcal{U}$, on the other hand, returns a *refinement type* representing the set of *uncovered values*, for which $\mathcal{G}$ can generate the inhabiting patterns to show to the user.

3.1 Desugaring to Guard Trees

To understand what language we should desugar to, consider the following 3am attempt at lifting equality over *Maybe*:

$$
\begin{array}{l}
liftEq\ Nothing\ Nothing = True \\
liftEq\ (Just\ x)\ (Just\ y) \\
\quad \mid x == y \qquad = True \\
\quad \mid otherwise = False
\end{array}
$$

Guard Syntax

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

Constraint Formula Syntax

$$
\begin{array}{lll l}
\Gamma & ::= & \varnothing \mid \Gamma, x : \tau \mid \Gamma, a & \text{Context} \\
\varphi & ::= & \checkmark \mid \times \mid K\ \overline{a}\ \overline{\gamma}\ \overline{y : \tau} \leftarrow x \mid x \not\approx K \mid x \approx \bot \mid x \not\approx \bot \mid \mathsf{let}\ x = e & \text{Literals} \\
\Phi & ::= & \varphi \mid \Phi \wedge \Phi \mid \Phi \vee \Phi & \text{Formula} \\
\Theta & ::= & \{\, \Gamma \mid \Phi \,\} & \text{Refinement Type} \\
\delta & ::= & \gamma \mid x \approx K\ \overline{a}\ \overline{y} \mid x \not\approx K \mid x \approx \bot \mid x \not\approx \bot \mid x \approx y & \text{Constraints} \\
\Delta & ::= & \varnothing \mid \Delta, \delta & \text{Set of constraints} \\
\nabla & ::= & \times \mid \Gamma \triangleright \Delta & \text{Inert Set}
\end{array}
$$

Clause Tree Syntax

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

Fig. 3. IR Syntax

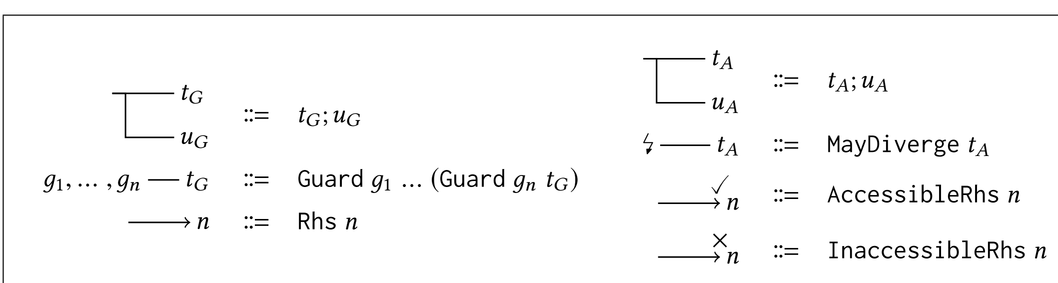


Fig. 4. Graphical notation

Function definitions in Haskell allow one or more *guarded right-hand sides* (GRHS) per syntactic *clause* (see fig. 1). For example, *liftEq* has two clauses, the second of which defines two GRHSs. Semantically, neither of them will match the call site *liftEq* (*Just* 1) *Nothing*, leading to a crash.

To see that, we can follow Haskell's top-to-bottom, left-to-right pattern match semantics. The first clause fails to match *Just* 1 against *Nothing*, while the second clause successfully matches 1 with x but then fails trying to match *Nothing* against *Just* y. There is no third clause, and the *uncovered* tuple of values (*Just* 1) *Nothing* that falls out at the bottom of this process will lead to a crash.

Compare that to matching on (*Just* 1) (*Just* 2): While matching against the first clause fails, the second matches x to 1 and y to 2. Since there are multiple GRHSs, each of them in turn has to be tried in a top-to-bottom fashion. The first GRHS consists of a single boolean guard (in general we have to consider each of them in a left-to-right fashion!) that will fail because $1 \mathrel{/}= 2$. The second GRHS is tried next, and because *otherwise* is a boolean guard that never fails, this successfully matches.

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

$$
\begin{array}{l}
\mathit{liftEq}\ \mathit{mx}\ \mathit{my} \\
\quad \mid \mathit{Nothing} \leftarrow \mathit{mx}, \mathit{Nothing} \leftarrow \mathit{my} \qquad\qquad\quad = \mathit{True} \\
\quad \mid \mathit{Just}\ x \leftarrow \mathit{mx}, \mathit{Just}\ y \leftarrow \mathit{my} \mid x == y \quad\ \ = \mathit{True} \\
\qquad\qquad\qquad\qquad\qquad\qquad\qquad\quad \mid \mathit{otherwise} = \mathit{False}
\end{array}
$$

Transforming the first clause with its single GRHS is easy. But the second clause already has two GRHSs, so we need to use *nested* pattern guards. This is not a feature that Haskell offers (yet), but it allows a very convenient uniformity for our purposes: after the successful match on the first two guards left-to-right, we try to match each of the GRHSs in turn, top-to-bottom (and their individual guards left-to-right).

Hence LYG desugars the source syntax to the following *guard tree* (see fig. 3 for the full syntax and fig. 4 the corresponding graphical notation):

$$
\begin{array}{l}
\dashv\ !\mathit{mx}, \mathtt{Nothing} \leftarrow \mathit{mx}, !\mathit{my}, \mathtt{Nothing} \leftarrow \mathit{my} \longrightarrow 1 \\
\dashv\ !\mathit{mx}, \mathtt{Just}\ x \leftarrow \mathit{mx}, !\mathit{my}, \mathtt{Just}\ y \leftarrow \mathit{my} \begin{array}{l} \dashv \mathsf{let}\ t = x == y, !t, \mathtt{True} \leftarrow t \longrightarrow 2 \\ \dashv\ !\mathit{otherwise}, \mathtt{True} \leftarrow \mathit{otherwise} \longrightarrow 3 \end{array}
\end{array}
$$

This representation is quite a bit more explicit than the original program. For one thing, every source-level pattern guard is strict in its scrutinee, whereas the pattern guards in our tree language are not, so we had to insert *bang guards*. By analogy with bang patterns, $!x$ evaluates x to WHNF, which will either succeed or diverge. Moreover, the pattern guards in Grd only scrutinise variables (and only one level deep), so the comparison in the boolean guard's scrutinee had to be bound to an auxiliary variable in a let binding.

Pattern guards in Grd are the only guards that can possibly fail to match, in which case the value of the scrutinee was not of the shape of the constructor application it was matched against. The Gdt tree language determines how to cope with a failed guard. Left-to-right matching semantics is captured by `Guard`, whereas top-to-bottom backtracking is expressed by sequence (;). The leaves in a guard tree each correspond to a GRHS.

3.2 Checking Guard Trees

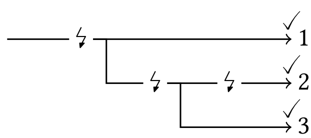
Coverage checking works by gradually refining the set of reaching values **Ryan:** Did you mean to write "reachable values" here? "Reaching values" reads strangely to me. **SG:** I was thinking "reaching values" as in "reaching definitions": The set of values that reach that particular piece of the guard tree. as they flow through the guard tree until it produces two outputs. One output is the set of uncovered values that wasn't covered by any clause, and the other output is an annotated guard tree skeleton Ant with the same shape as the guard tree to check, capturing redundancy and divergence information.

For the example of *liftEq*'s guard tree t_{liftEq}, we represent the set of values reaching the first clause by the *refinement type* $\{ (\mathit{mx} : \mathit{Maybe}\ a, \mathit{my} : \mathit{Maybe}\ a) \mid \checkmark \}$ (which is a Θ from fig. 3) . This set is gradually refined until finally we have $\Theta_{liftEq} := \{ (mx : Maybe\ a, my : Maybe\ a) \mid \Phi \}$ as the uncovered set, where the predicate Φ is semantically equivalent to:

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

Every $\lor$ disjunct corresponds to one way in which a pattern guard in the tree could fail. It is not obvious at all for humans to read off inhbaitants from this representation, but we will give an intuitive treatment of how to do so in the next subsection.

The annotated guard tree skeleton corresponding to t_{liftEq} looks like this:



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

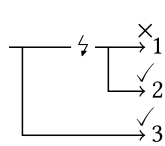
A ϟ denotes possible divergence in one of the bang guards and involves testing the set of reaching values for compatibility with i.e. $mx \approx \bot$. We cannot know in advance whether mx, my or t are $\bot$ (hence the three uses of ϟ), but we can certainly rule out $otherwise \approx \bot$ simply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

Perhaps surprisingly and most importantly, Grd with its three primitive guards, combined with left-to-right or top-to-bottom semantics in Gdt, is expressive enough to express all pattern matching in Haskell (cf. the desugaring function $\mathcal{D}$ in fig. 5)! We have yet to find a language extension that does not fit into this framework.

3.2.1 *Why do we not report redundant GRHSs directly?*

Why not compute the redundant GRHSs directly instead of building up a whole new tree? Because determining inaccessibility vs. redundancy is a non-local problem. Consider this example 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.

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



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!

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

3.3 Generating Inhabitants of a Refinement Type

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. **SG:** Do we need to give its predicate here again? 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: **SG:** This example will probably move to an earlier section

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

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

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

4 FORMALISM

The previous section gave insights into how we represent 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.

4.1 Desugaring to Guard Trees

SG: I find this section quite boring. There's nothing to see in fig. 5 that wasn't already clear after reading 3.1.

Figure 5 outlines the desugaring step from source Haskell to our guard tree language Gdt. It is assumed that the top-level match variables x_1 through x_n in the *clause* cases have special, fixed names. **SG:** If we had a different font for meta variables than for object variables, we could make

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

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

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

Fig. 5. Desugaring Haskell to Gdt

that visible in syntax. But we don't... All other variables that aren't bound in arguments to $\mathcal{D}$ have fresh names.

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

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

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

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

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.

Operations on Θ

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

Checking Guard Trees

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

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

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

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

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

$$
\begin{array}{lcl}
\mathcal{R}(\texttt{AccessibleRhs}\ n) & = & (n, \epsilon, \epsilon) \\
\mathcal{R}(\texttt{InaccessibleRhs}\ n) & = & (\epsilon, n, \epsilon) \\
\mathcal{R}(t; u) & = & (\overline{k}\,\overline{k'}, \overline{n}\,\overline{n'}, \overline{m}\,\overline{m'}) \ \text{ where } \begin{array}{l} (\overline{k}, \overline{n}, \overline{m}) \ \ = \mathcal{R}(t) \\ (\overline{k'}, \overline{n'}, \overline{m'}) = \mathcal{R}(u) \end{array} \\
\mathcal{R}(\texttt{MayDiverge}\ t) & = & \begin{cases} (\epsilon, m, \overline{m'}) & \text{if } \mathcal{R}(t) = (\epsilon, \epsilon, m\,\overline{m'}) \\ \mathcal{R}(t) & \text{otherwise} \end{cases}
\end{array}
$$

Fig. 6. Coverage checking

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 @-pattern) when choosing a name for a variable.

4.2 Checking Guard Trees

Figure 6 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 $x \approx \perp$, 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. 7, which generates a set of patterns which inhabit a given refinement type Θ. There might be multiple inhabitants, and $\mathcal{C}$ 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. **SG:** Currently, $\mathcal{E}$ will only expand positive constraints and not produce multiple pattern vectors for a ∇ with negative info (see the TODO comment attached to $\mathcal{E}$'s definition)

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. $\mathcal{C}$ 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 constraint of the form $x \approx \_$.

SG: We don't maintain I3 in fig. 8 as is, because we might have $x \approx \perp$ and $x \approx \mathit{Nothing}$. Maybe relax it to apply only to data constructor solutions? 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 $x \approx \_ \in \Delta$, 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

Generate inhabitants of Θ

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

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

Construct inhabited ∇s from Φ

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

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

Expand variables to Pat with ∇

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

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

Finding the representative of a variable in Δ

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

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

Fig. 7. Generating inhabitants of Θ via ∇

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. 7), 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.

SG: $\mathcal{E}$ undoubtly needs some love, but that's a TODO for later. 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 8 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. 9. 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.

SG: Maybe move down the type constraint case in the definition? 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 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.

Add a formula literal to the inert set

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

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

Add a constraint to the inert set

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

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

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

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

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

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

Fig. 8. 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

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

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

Fig. 9. Inhabitance test

SG: We need to brag about how this representation is better than GMTMs. Example:

```
data T = A1 | ... | A1000
f :: T → T → ()
f A1 _ = ()
f _ A1 = ()
```

This will split (a term which is introduced in section 6) into a million value vectors in GMTMs model, whereas there will only be ever fall through one ∇ from one equation to the next because of negative constraints.

Also GMTM comitting to a particular COMPLETE set the first time it splits on a constructor pattern means buggy COMPLETE pragma handling. I think this comparison should go into Related Work.

4.5 Inhabitation Test

SG: We need to 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. 9. In contrast to the emptiness test in fig. 7, 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 $x : Int \rightarrow Int$.

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 ∇, 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:

$$\begin{array}{l}\textbf{data}\ T = MkT\ !\ T \\ f :: SMaybe\ T \rightarrow () \\ f\ SNothing = ()\end{array}$$

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

SG: This currently doesn't mention the term "long distance information" even once...

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

$$\begin{array}{l}f\ Nothing = 1 \\ f\ x@(Just\ 15) = \ldots(\textbf{case}\ x\ \textbf{of} \\ \quad Nothing \rightarrow 2 \\ \quad Just\ 15 \rightarrow 3 \\ \quad Just\ \_ \rightarrow 4)\ \ldots\end{array}$$

LYG as is will not produce any warnings for this definition. But for the reader it is as plain as it can be that the **case** expression has two redundant GRHSs! That simply follows by context-sensitive reasoning, knowing that x was successfully matched to $Just\ 15$ in the outer match.

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 $\{\, x : Maybe\ Int \mid \checkmark \,\}$.

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 $Void$:

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

SG: lhs2TeX chokes on wildcards and will format *absurd* ! x as infix. Yuck 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:

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

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 `Empty` to Gdt and `AntEmpty` to Ant. The new `Empty` case has to be handled by the checking functions and is a neutral element to ; as far as $\mathcal{U}$ is concerned:

$$\begin{array}{rcl} \mathcal{U}(\Theta, \texttt{Empty}) & = & \Theta \\ \mathcal{A}(\Theta, \texttt{Empty}) & = & \texttt{AntEmpty} \end{array}$$

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 $\mathcal{U}(\Theta \mathbin{\dot\wedge} (x \not\approx \bot), \texttt{Empty})$, where Θ is the context-sensitive set of reaching values, possibly enriched with long distance information (cf. section 5.1).

5.3 View Patterns

Extending source syntax for view patterns is straight-forward, so is its desugaring in terms of Grd:

$$pat ::= \boxed{expr \rightarrow pat} \mid \ldots \qquad \boxed{\mathcal{D}(x, expr \rightarrow pat) = \mathsf{let}\ y = expr\ x, \mathcal{D}(y, pat)}$$

SG: Should we also generate a $!x$? That wouldn't be true for *const False* $\rightarrow$ *True*. I'm not sure if there's a conservative way to handle that case! Where y is a fresh variable. But this 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:

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

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 inexhaustive. But due to referential transparency, the result of $reverse\ x_1$ 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 $y_2 = reverse\ x_1$ in the second branch would entail looking up the equivalence class of $reverse\ x_1$ 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 $y_1 \not\approx Nothing$ 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:

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

Where r_i is the representative of the equivalence class of expressions with global value number i. Thus, our implementation will not emit any warning 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*:

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

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

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

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 $x \approx C\ \overline{a}\ \overline{y}$:

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

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. 9 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:

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

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

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. $\overline{C_{i,1}, ..., C_{i,n_i}}^{\,i}$?

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

$$pat ::= N\ pat \mid \ldots \qquad \begin{array}{rlll} N \in & \mathrm{NT} & & \\ co \in & \mathrm{Co} & ::= & \mathsf{sym}\ co \mid \langle\tau\rangle \mid co_1; co_2 \mid \mathsf{newt}\ N \\ e \in & \mathrm{Expr} & ::= & x \triangleright co \mid K\ \overline{\tau}\ \overline{\sigma}\ \overline{\gamma}\ \overline{e} \mid \ldots \end{array}$$

$$\mathcal{D}(x, N\ pat) = \mathsf{let}\ x = y \triangleright \mathsf{newt}\ \ N, \mathcal{D}(y, pat)$$

$$\delta ::= \ldots \mid x \approx y \triangleright co \qquad \Gamma \triangleright \Delta \oplus_{\varphi} \mathsf{let}\ x : \tau = y \triangleright co = \Gamma, x : \tau \triangleright \Delta \oplus_{\delta} x \approx y \triangleright co$$

$$\Gamma \triangleright \Delta(x) = \begin{cases} (z, co_1; co_2) & x \approx y \triangleright co_1 \in \Delta, (z, co_2) = \nabla(y) \\ (x, \langle\tau\rangle) & \text{where } x : \tau \in \Gamma \end{cases}$$

$$\Gamma \triangleright \Delta \oplus_{\delta} x \approx y \triangleright co = \begin{cases} \Gamma \triangleright \Delta & \text{if } \Delta(x)_1 = \Delta(y)_1 \\ \Gamma \triangleright ((\Delta \setminus \Delta(x)_1), \Delta(x)_1 \approx \Delta(y)_1 \triangleright co') \oplus_{\delta} ((\Delta \cap \Delta(x)_1)[\Delta(y)_1 / \Delta(x)_1]) & \text{if } x : \tau \in \Gamma \\ \text{and } co' = \mathsf{sym}\ \Delta(x)_2; co; \Delta(y)_2 & \end{cases}$$

Fig. 10. Extending coverage checking to handle Newtypes

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, so the third GRHS is *inaccessible* (because every value it could cover was already covered by the first GRHS), but not redundant. A perhaps surprising consequence is that the definition of f is exhaustive, because after $N\ Void$ was deprived of its sole inhabitant $\perp \equiv N\ \perp$, there is nothing left to match on.

If it was only for f, we could express this semantics simply by desugaring Newtype pattern matches as lazy (so we wouldn't generate a $!x$ on the match var x in $\mathcal{D}$), but treat N as if it had a strict field in Cons.

g crushes this simple hack. We would mark its second GRHS as inaccessible when it is clearly redundant: The inner bang pattern has nothing to evaluate. This is arguably a small downside and doesn't even regress in terms of soundness.

We'll show how to fix this infelicity by treating Newtype wrappers as coercions. That entails a slew of modifications, the gist of which is depicted in fig. 10. We have to extend source syntax in a similar manner as for pattern synonyms (section 5.4) and add coercions to IR expressions. Then we can desugar Newtype matches to coercions on a fresh match variable, which ultimately turns into an extended δ constraint $x \approx y \triangleright co$ via $\oplus_{\varphi}$.

Before we finally talk about $\oplus_{\delta}$, we have to change the definition of $\Delta(x)$ to also return the coercion along the transitive chain of $x \approx y \triangleright co$ constraints it had to follow to find the representative. Since that will now return a pair, many definitions change in syntactically drastic, but semantically non-meaningful way. The only exception is the last clause of $\oplus_{\delta}$, where we have to build the proper coercion co' when adding the new $\Delta(x)_1 \approx \Delta(y)_1 \triangleright co'$ constraint. **SG:** TODO: the defn still uses $\Delta(x)$ instead of $\nabla(x)$ all over the place. yuck

Surprisingly, no more coercion handling is needed! We can see that $\Delta(x) \approx y \triangleright co \in \Delta$ is impossible for all x and y. So all representatives either don't have a solution (in which case $\Delta \cap \Delta(x)_1$ is empty) or are representatives of a data constructor solution or $\perp$ themselves, so can be trivially renamed with the $[\Delta(y)_1 / \Delta(x)_1]$ suffix.

Other than that, $\mathcal{E}$ (which for the purposes of this paper is just concerned with presenting uncovered patterns to the user) will have to turn the sequence of coercions back into source-level Newtype applications.

SG: I'm no longer convinced that we want to have this in the paper.

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 **SG:** Cite something?, 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 `MayDiverge` 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.

SG: Treat type information as an extension?

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. 6, 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. 11. 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 these ∇s are non-empty, i.e. not $\times$. This invariant is maintained by adding φ constraints through $\dot{\oplus}_\varphi$, which filters out any ∇ that would become empty. All mentions of $\mathcal{G}$ are

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

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

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

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

Fig. 11. Fast coverage checking

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:

$$\begin{array}{l}
\mathit{f1}, \mathit{f2} :: \mathit{Int} \longrightarrow \mathit{Bool} \\
g\; _ \\
\quad \mid \mathit{True} \leftarrow \mathit{f1}\; 0,\;\; \mathit{True} \leftarrow \mathit{f2}\; 0 \;\; = () \\
\quad \mid \mathit{True} \leftarrow \mathit{f1}\; 1,\;\; \mathit{True} \leftarrow \mathit{f2}\; 1 \;\; = () \\
\quad \ldots \\
\quad \mid \mathit{True} \leftarrow \mathit{f1}\; N, \mathit{True} \leftarrow \mathit{f2}\; N = ()
\end{array}$$

Here's the corresponding guard tree:

$$\begin{array}{l}
\dashv \mathsf{let}\; t_1 = \mathit{f1}\; 0, !t_1, \mathit{True} \leftarrow t_1, \mathsf{let}\; t_2 = \mathit{f2}\; 0, !t_2, \mathit{True} \leftarrow t_2 \longrightarrow 1 \\
\dashv \mathsf{let}\; t_3 = \mathit{f1}\; 1, !t_3, \mathit{True} \leftarrow t_3, \mathsf{let}\; t_4 = \mathit{f2}\; 1, !t_4, \mathit{True} \leftarrow t_4 \longrightarrow 2 \\
\dashv \ldots \longrightarrow \ldots \\
\dashv \mathsf{let}\; t_{2*N+1} = \mathit{f1}\; N, !t_{2*N+1}, \mathit{True} \leftarrow t_{2*N+1}, \mathsf{let}\; t_{2*N+2} = \mathit{f2}\; N, !t_{2*N+2}, \mathit{True} \leftarrow t_{2*N+2} \longrightarrow \mathrm{N}
\end{array}$$

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:

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

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

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

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

$f :: T \longrightarrow Int$
$f\ A1 = 1$
$f\ A2 = 2$
$\ldots$
$f\ A1000 = 1000$

f is exhaustively defined. For 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 ($\vdash$Inst) to each of its 1000 constructors and try to add a positive constructor constraint! What a waste of time, given that we could just look at the negative constraints on x *before* trying to instantiate x! But asymptotically it shouldn't matter much, since we're doing this only once at the end.

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

Cleary, we can be smarter about that! Indeed, we cache *residual* `COMPLETE` *sets* in our implementation: Starting from the full `COMPLETE` sets, we delete ConLike from them whenever we add a new negative constructor constraint, making sure that each of the sets is inhabited by at least one constructor. Note how we never need to check the same constructor twice, thus we have an amortised $\mathcal{O}(n)$ instantiations for the whole checking process.

SG: I'm not sure what other hacks we should mention beyond this. I don't think we want to write about ad-hoc details like 6.2 in GMTM, because they are specific to how Δ is represented (solved, canonical type constraints in particular). That's of limited value for other implementations and not a conceptual improvement.

7 RELATED WORK

7.1 Comparison with GADTs Meet Their Match

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

7.1.1 GMTM does not consider laziness in its full glory. The formalism in Karachalias et al. [2015] incorporates strictness constraints, but these constraints can only arise from matching against data constructors. GMTM does not consider strict matches that arise from strict fields of data constructors or bang patterns. A consequence of this is that GMTM would incorrectly warn that v (**Ryan:** Cite the section!) 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 the *safeLast* function from **Ryan:** Cite the section!, will cause GMTM to emit erroneous warnings.

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 pattern synonyms (section 5.4) demonstrates. While GMTM's term oracle could be improved to accomplish the same thing, it is unlikely to be as straightforward of a process as extending $\oplus_\varphi$.

7.2 Comparison with similar coverage checkers

7.2.1 Structural and semantic pattern matching analysis in Haskell. Kalvoda and Kerckhove [2019] implement a variation of GMTM that leverages an SMT solver to give more accurate coverage warnings for programs that use guards. For instance, their implementation can conclude that the *signum* function from **Ryan:** Which section? 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?

7.3 Other related work

Ryan: Fill me in! Some possible candidates:

- GMTM (mine their Related Work)
- Maranget's work [Maranget 2007]; tries to account for laziness, but wrongly so
- Compare to "Elaborating dependent (co)pattern matching" [Cockx and Abel 2018], which is essentially GADTs MTM with more type foo going on
- Compare to refinement types
- Sestofts negative constraints [Sestoft 1996]
- OCaml: *SMaybe Void*

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.*

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 âĂŹ73).* 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

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

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