GADTs Meet Their Match:

Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness

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

ACM Reference Format:
Sebastian Graf, Simon Peyton Jones, and Ryan G. Scott. 2020. GADTs Meet Their Match:: Pattern-Matching Warnings That Account for GADTs, Guards, and Laziness. *Proc. ACM Program. Lang.* 1, ICFP, Article 1 (January 2020), 24 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:

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

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

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

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

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

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/

Fig. 1. Definitions used in the text

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

- We characterise the nuances of coverage checking that not even the algorithm in [Karachalias et al. 2015] handles. We also identify issues in GHC's implementation of this algorithm.
- **Ryan:** Describe the "Overview Over Our Solution" section and "Formalism" sections.
- We have implemented **SYSNAME** in GHC. **Ryan:** More details.

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

2 THE PROBLEM WE WANT TO SOLVE

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

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

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

2.1 Guards

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-valued guard that evaluates its right-hand side if the expression in the guard returns *True*. The second guard is a *pattern guard* that evaluates its right-hand side if the pattern in the guard successfully matches. Moreover, a guard can have **let** bindings or even multiple

checks, as the third guard demonstrates. Note that the fourth guard uses *otherwise*, a boolean guard that is guaranteed to match successfully.

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:

$$
\begin{aligned}
&signum :: Int \to Int \\
&signum\ x \\
&\quad \mid x > 0 = 1 \\
&\quad \mid x == 0 = 0 \\
&\quad \mid x < 0 = -1
\end{aligned}
$$

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 commonly use cases for guards. For instance, take the following function that only uses pattern guards:

$$
\begin{aligned}
&not :: Bool \to Bool \\
&not\ b \\
&\quad \mid False \leftarrow b = True \\
&\quad \mid True \leftarrow b = False
\end{aligned}
$$

not is clearly equivalent to the following function that matches on its argument without guards:

$$
\begin{aligned}
&not' :: Bool \to Bool \\
&not'\ False = True \\
&not'\ True = False
\end{aligned}
$$

We would like our coverage checking algorithm to mark both *not* and *not′* as exhaustive, and **SYSNAME** does so. We explore the subset of guards that **SYSNAME** 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. Both Haskell and GHC offer patterns that are impossible to check in the general case. We consider three such patterns here: overloaded literals, view patterns, and pattern synonyms.

2.2.1 Overloaded literals. Numeric literals in Haskell can be used at multiple types by virtue of being overloaded. For example, the literal 0, when used as an expression, has $Num\ a \Rightarrow a$ as its most general type. The *Num* class, in turn, has instances for *Int*, *Double*, and many other numeric data types, allowing 0 to inhabit those types with little fuss.

In addition to their role as expression, overloaded literals can also be used as patterns. The *isZero* function below, for instance, can check whether any numeric value is equal to zero:

$$
\begin{aligned}
&isZero :: (Eq\ a, Num\ a) \Rightarrow a \to Bool \\
&isZero\ 0 = True \\
&isZero\ n = False
\end{aligned}
$$

Why does *isZero* require an *Eq* constraint on top of a *Num* constraint? This is because when compiled, overloaded literal patterns essentially desugar to guards. As one example, *isZero* can be rewritten to use guards like so:

```
isZero :: (Eq a, Num a) ⇒ a → Bool
isZero n | n == 0 = True
isZero n = False
```

Desugaring overloaded literal patterns to guards directly like this is perhaps not always desirable, however, since that can make the coverage checker's job more difficult. For instance, if the *isZero n = False* clause were omitted, concluding that *isZero* is non-exhaustive would require reasoning about properties of the *Eq* and *Num* classes. For this reason, it can be worthwhile to have special checking treatment for common numeric types such as *Int* or *Double*. In general, however, coverage checking patterns with overloaded literals is undecidable.

2.2.2 View patterns. View patterns are a GHC extension that allow arbitrary computation to be performed while pattern matching. When a value v is matched against a view pattern $f \to p$, 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 (x, xs)) = 1 + length xs
```

View patterns can be thought of as a generalization of overloaded literals. For example, the *isZero* function in **Ryan:** cite section can be rewritten to use view patterns like so:

```
isZero :: (Eq a, Num a) ⇒ a → Bool
isZero ((==) 0 → True) = True
isZero n = False
```

Just like with overloaded literals, view patterns desugar to guards when compiled. As a result, the coverage checker can cope with view patterns provided that they desugar to guards that are not too complex. For instance, **SYSNAME** would not be able to conclude that *length* is exhaustive, but it would be able to conclude that this reimplementation of *safeLast* is exhaustive:

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

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

```
pattern Nil :: Text
pattern Nil ← (Text.null → True)

pattern Cons :: Char → Text → Text
pattern Cons x xs ← (Text.uncons → Just (x, xs))

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

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

Intuitively, *Text.null* and *Text.uncons* are obviously 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: **Ryan:** Consider moving some of this code to the figure **Ryan:** There is an erroneous space between the ! and the *a*

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

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

The *SJust* constructor is strict in its field, and as a consequence, evaluating *SJust* ⊥ 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 (SJust _) = 1
```

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

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

Meta variables

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

Pattern Syntax

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

Fig. 2. Source syntax

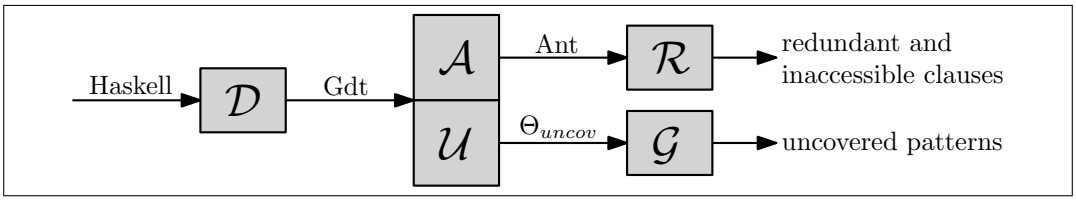


Fig. 3. Bird's eye view of pattern match checking

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

$v' :: Maybe\ Void \to Int$
$v'\ Nothing = 0$
$v'\ (Just\ !\ x) = 1$

The *Just* case in v' is unreachable for the same reasons that the *SJust* case in v is unreachable. Due to 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.

***2.3.2 Newtypes.* TODO:**

2.4 Term and type constraints

TODO:

2.5 A solved problem?

We are not the first to tackle the problem of coverage checking. Karachalias et al. [2015] develop a checking algorithm that handles many of the subtleties of GADTs, guards, and laziness mentioned earlier in this section. For the sake of brevity, we will refer to their algorithm as GMTM (an abbreviation of "GADTs Meet Their Match", the name of the corresponding paper).

Ryan: more details pls

Guard Syntax

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

Constraint Formula Syntax

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

Clause Tree Syntax

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

Fig. 4. IR Syntax

3 OVERVIEW OF OUR SOLUTION

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

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

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

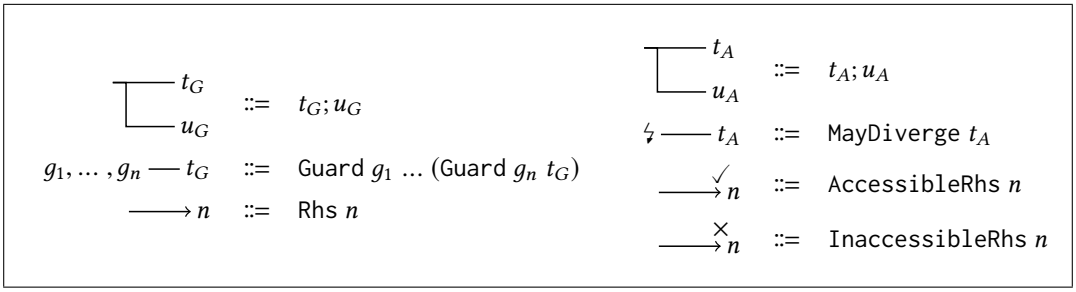


Fig. 5. Graphical notation

Function definitions in Haskell allow one or more *guarded right-hand sides* (GRHS) per syntactic *clause* (see fig. 2). 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.

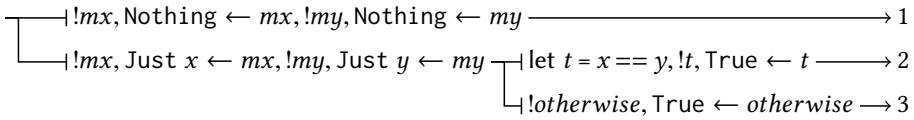
Compare that to matching on (*Just* 1) (*Just* 2): While matching against the first clause fails, the second matches x to 1 and y to 2. Since there are multiple GRHSs, each of them in turn has to be tried in a top-to-bottom fashion. The first GRHS consists of a single boolean guard (in general we have to consider each of them in a left-to-right fashion!) that will fail because $1 /= 2$. 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:

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

Transforming the first clause with its single GRHS is easy. But the second clause already 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 our algorithm desugars the source syntax to the following *guard tree* (see fig. 4 for the full syntax and fig. 5 the corresponding graphical notation):



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. Note that *otherwise* is an external identifier which we can assume to be bound to *True*, which is in fact how it defined.

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

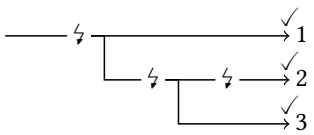
Pattern match checking works by gradually refining the set of reaching values **Ryan:** Did you mean to write "reachable values" here? "Reaching values" reads strangely to me. **SG:** I was thinking "reaching values" as in "reaching definitions": The set of values that reach that particular piece of 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* $\{\, (mx : \mathit{Maybe}\ a, my : \mathit{Maybe}\ a) \mid \checkmark \,\}$ (which is a Θ from fig. 4) . This set is gradually refined until finally we have $\Theta_{liftEq} := \{\, (mx : \mathit{Maybe}\ a, my : \mathit{Maybe}\ a) \mid \Phi \,\}$ as the uncovered set, where the predicate Φ is semantically equivalent to:

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

Every $\vee$ 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 ($\checkmark$) 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 $\lightning$ 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 $\lightning$), but we can certainly rule out $\mathit{otherwise} \approx \bot$ 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 $\lightning$ decorators are irrelevant.

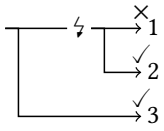
Perhaps surprisingly and most importantly, Grd with its three primitive guards, combined with left-to-right or top-to-bottom semantics in Gdt, is expressive enough to express all pattern matching in Haskell (cf. the desugaring function $\mathcal{D}$ in fig. 6)! We have yet to find a language extension 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: **SG:** I think this kind of detail should be motivated in a prior section and then referenced here for its solution.

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

Is the first 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!

Here is the corresponding annotated tree after checking:



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

3.3 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 pattern match checking **Ryan:** We inconsistently switch between calling it "pattern match checking" and "coverage checking" in the prose. We should adopt one term and stick with it. **SG:** I think I like "coverage checking" (after introducing it as "pattern match coverage checking" better, provided the term subsumes both exhaustivity and overlap 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 $\{ x : \mathit{Maybe\ Void} \mid x \not\approx \bot \wedge x \not\approx \texttt{Nothing} \}$. Specifically, the candidate *SJust y* (for fresh *y*) for *x* should be rejected, because there is no inhabitant for *y*! $\bot$ is ruled out by the strict field and *Void* 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 pattern match 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. 4) we introduced earlier.

As in the previous section, we divide this section into three main parts: desugaring, pattern match 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. 6 that wasn't already clear after reading 3.1.

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

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

f (*Just* (!*xs*, _)) *ys*@*Nothing* = 1
f *Nothing* *zs* = 2

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

$$\mathcal{D}\llbracket f\ (\mathit{Just}\ (!xs, \_))\ ys@\mathit{Nothing} = 1 \rrbracket = \mathit{Just}\ t_1 \leftarrow x_1, !t_1, (t_2, t_3) \leftarrow t_1, !t_2, \mathrm{let}\ xs = t_2, \mathrm{let}\ ys = x_2, !ys, \mathit{Nothing} \leftarrow ys \longrightarrow 1$$

$$\mathcal{D}\llbracket f\ \mathit{Nothing}\ zs = 2 \rrbracket = !x_1, \mathit{Nothing} \leftarrow x_1, \mathrm{let}\ zs = x_2 \longrightarrow 2$$

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

Note how our naïve desugaring function generates an abundance of fresh temporary variables. In practice, the implementation of $\mathcal{D}$ can be smarter than this by looking at the pattern (which might be a variable match or @-pattern) when choosing a name for a variable.

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

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

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

Fig. 6. Desugaring Haskell to Gdt

4.2 Checking Guard Trees

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

Operations on Θ

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

Checking Guard Trees

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

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

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

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

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

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

Fig. 7. Pattern match checking

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

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

Pattern guard semantics are important for $\mathcal{U}$ and bang guard semantics are important for $\mathcal{A}$. But what about let bindings? They are in fact completely uninteresting to the checking process, but making sense of them is important for the precision of the emptiness check involving $\mathcal{G}$. Of course,

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 ∇

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

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

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. 8. Generating inhabitants of Θ via ∇

"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 **SG:** TODO: Reference CoreMap/semantic equality extension, and possibly an extension for linear arithmetic or boolean logic.

4.3 Generating Inhabitants of a Refinement Type

The key function for the emptiness test is $\mathcal{G}$ in fig. 8, which generates a set of patterns which inhabit a given refinement type 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. 9 as is, because we might have $x \approx \bot$ 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 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. 8), 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.

C is the function that breaks down a Φ into multiple ∇s, maintaining the invariant that no such ∇ is $\times$. At the heart of 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 kernel of our algorithm: Figure 9 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

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. 9. Adding a constraint to the inert set ∇

SG: TODO reference CoreMap 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. 10. 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.

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

Test if x is inhabited considering ∇

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

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

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

Find data constructors of τ

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

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

Instantiate x to data constructor K

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

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

Fig. 10. Inhabitance 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. 10. In contrast to the emptiness test in fig. 8, this one focuses on a particular variable and works on a ∇ rather than a much higher-level Θ.

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

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

```
data T = MkT !T
f :: SMaybe T → ()
f SNothing = ()
```

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

which leads to constraints $y \approx MkT\ z, z \not\approx \bot$, 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

5.1 Long Distance Information

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

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

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

The pattern match checking algorithm 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 $\mathit{Just}\ 15$ in the outer match.

In fact, the checking algorithm 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 $\{\, x : \mathit{Maybe\ Int} \mid \checkmark \,\}$.

5.2 Empty Case

As can be seen in fig. 2, Haskell function definitions need to have at least one clause. That leads to an awkward situation when pattern matching on empty data types, like *Void*:

$$
\begin{array}{l}
\mathit{absurd} :: \mathit{Void} \to a \\
\mathit{absurd}\ x \quad\ = \bot \\
\mathit{absurd}\ (!x) = \bot
\end{array}
$$

SG: lhs2TeX chokes on wildcards and will format $\mathit{absurd}\ !\ x$ 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, our checking algorithm will mark the second option as having an inaccessible RHS!

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

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

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

Although we did not give the syntax of **case** expressions in fig. 2, 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:

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

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 Identifying Semantically Equivalent Expressions

SG: Or just "View Patterns" for a catchier title? **TODO:**

5.4 Pattern Synonyms

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

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

SG: For pattern-match 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.

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

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

Knowing that the definitions of P and Q completely overlap, we can see that Q will cover all values that could reach P, so clearly P is redundant. A sound approximation to that would be not to warn at all. And that's reasonable, after all we established in section 2.2.3 that reasoning about pattern synonym definitions is undesirable.

But equipped with long distance information from the scrutinee expression, the checker would mark the *first case alternative* as redundant, which clearly is unsound! Deleting the first alternative would change its semantics from returning 1 to returning 2. In general, we cannot assume that arbitrary pattern synonym definitions are disjunct. That is in stark contrast to data constructors, which never overlap.

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

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

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

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

5.5 `COMPLETE` pragmas

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

We have ⊢Inst from fig. 10 currently making sure that this `COMPLETE` set is in fact inhabited. We also have ⊢NoCpl that handles the case when we can't find *any* `COMPLETE` set for the given type (think $x : Int \to Int$). The obvious way to generalise this is by looking up all `COMPLETE` sets attached to a type and check that none of them is completely covered:

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

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

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

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

Note that checking against multiple `COMPLETE` sets so frequently is computationally intractable. We will worry about that in section 6.

5.6 Literals

The source syntax in fig. 2 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

TODO:

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

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

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

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

Fig. 11. Fast pattern match checking

5.8 Strictness

TODO:

SG: Treat type information as an extension?

6 IMPLEMENTATION

The implementation of our algorithm 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, completeness checks involving GADTs need a type-checked program, so the only reasonable phase to run the Pattern match checker is between type-checking and desugaring to GHC Core, a typed intermediate representation lacking the connection to source syntax. We perform pattern match 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. 7, 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 C on what would have been the set of reaching values.

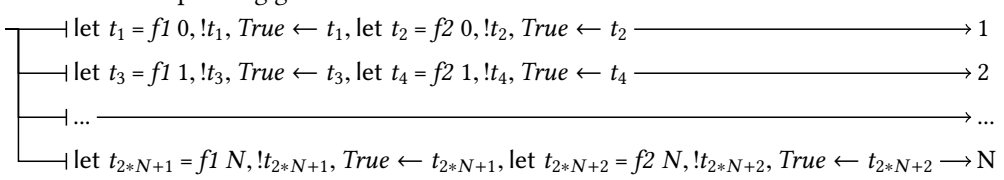
You can see the resulting definition in fig. 11. The readability of the interleaving of both functions is clouded by unwrapping of pairs. Other than that, all references to Θ were replaced by a vector of ∇s. $\mathcal{UA}$ requires that these ∇s are non-empty, i.e. not $\times$. This invariant is maintained by adding φ constraints through $\dot{\oplus}_\varphi$, which filters out any ∇ that would become empty. All mentions of $\mathcal{G}$ are 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, pattern match checking is just a static analysis pass without any effect on the produced binary! Consider the following example:

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

Here's the corresponding guard tree:



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

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

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

the ∇ falling through from the *f2* match (which could additionally diverge when calling *f2*). This approach seemed far to complicated for us to pursue.

Instead, we implement *throttling*: We limit the number of reaching ∇s to a constant. Whenever a split would exceed this limit, we continue with the original reaching ∇ (which as we established is a superset, thus a conservative estimate) instead. Intuitively, throttling corresponds to *forgetting* what we matched on in that particular subtree.

Throttling is refreshingly easy to implement! Only the last clause of $\mathcal{UA}$, where splitting is performed, needs to change:

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

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

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

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

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

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.

SG: We could talk about when adding a type constraint, we only need to perform the inhabitation check on a subset of all variables. Namely those that aren't obviously of plain old ADT type. But the implementation doesn't currently do that hack, so it's a bit of a moot point.

SG: We should talk about how we efficiently represent residual COMPLETE sets. And maybe how we represent Delta in general.

REFERENCES

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

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