GADTs Meet Their Match:

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

Georgios Karachalias
Ghent University, Belgium
georgios.karachalias@ugent.be

Tom Schrijvers
KU Leuven, Belgium
tom.schrijvers@cs.kuleuven.be

Dimitrios Vytiniotis
Simon Peyton Jones
Microsoft Research Cambridge, UK
{dimitris,simonpj}@microsoft.com

Abstract

For ML and Haskell, accurate warnings when a function definition has redundant or missing patterns are mission critical. But today's compilers generate bogus warnings when the programmer uses guards (even simple ones), GADTs, pattern guards, or view patterns. We give the first algorithm that handles all these cases in a single, uniform framework, together with an implementation in GHC, and evidence of its utility in practice.

Categories and Subject Descriptors D.3.2 [*Language Classifications*]: Applicative (functional) languages; D.3.3 [*Language Constructs and Features*]: Patterns

Keywords Haskell, pattern matching, Generalized Algebraic Data Types, OUTSIDEIN(X)

1. Introduction

Is this function (in Haskell) fully defined?

```
zip :: [a] -> [b] -> [(a,b)]
zip []     []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

No, it is not: the call (`zip [] [True]`) will fail, because neither equation matches the call. Good compilers will report missing patterns, to warn the programmer that the function is only partially defined. They will also warn about completely-overlapped, and hence redundant, equations. Although technically optional for soundness, these warnings are incredibly useful in practice, especially when the program is refactored (i.e. throughout its active life), with constructors added and removed from the data type (Section 2).

But what about this function?

```
vzip :: Vect n a -> Vect n b -> Vect n (a,b)
vzip VN        VN        = VN
vzip (VC x xs) (VC y ys) = VC (x,y) (vzip xs ys)
```

where the type `Vect n a` represents lists of length `n` with element type `a`. `Vect` is a Generalised Algebraic Data Type (GADT):

```
data Vect :: Nat -> * -> * where
  VN :: Vect Zero a
  VC :: a -> Vect n a -> Vect (Succ n) a
```

Unlike `zip`, function `vzip` is *fully* defined: a call with arguments of unequal length, such as (`vzip VN (VC True VN)`), is simply ill-typed. Comparing `zip` and `vzip`, it should be clear that only a *type-aware* algorithm can correctly decide whether or not the pattern-matches of a function definition are exhaustive.

Despite the runaway popularity of GADTs, and other pattern-matching features such as view patterns and pattern guards, no production compiler known to us gives accurate pattern-match overlap and exhaustiveness warnings when these features are used. Certainly our own compiler, GHC, does not; and nor does OCaml. In this paper we solve the problem. Our contributions are these:

- We characterise the challenges of generating accurate warnings in Haskell (Section 2). The problem goes beyond GADTs! There are subtle issues concerning nested patterns, view patterns, guards, and laziness; the latter at least has never even been noticed before.
- We give a type-aware algorithm for determining missing or redundant patterns (Sections 3 and 4). The algorithm is parameterised over an oracle that can solve constraints: both type constraints and boolean constraints for guards. Extending the oracle allows us to accommodate type system extensions or improve the precision of the reported warnings *without* affecting the main algorithm at all.

  The central abstraction in this algorithm is the compact symbolic representation of a set of values by a triple $(\Gamma \vdash u \triangleright \Delta)$ consisting of an environment Γ, a syntactic value abstraction u and a constraint Δ (Section 4.1). The key innovation is to include the constraints Δ to refine the set of values; for example $(x{:}\texttt{Int} \vdash \texttt{Just}\ x \triangleright x{>}3)$ is the set of all applications of `Just` to integers bigger than 3. This allows us to handle GADTs, guards and laziness uniformly.
- We formalise the correctness of our algorithm (Section 5) with respect to the Haskell semantics of pattern matching.
- We have implemented our algorithm in GHC, a production quality compiler for Haskell (Section 6). The new implementation is of similar code size as its predecessor although it is much more capable. It reuses GHC's existing type constraint solver as an oracle.
- We demonstrate the effectiveness of the new checker on a set of actual Haskell programs submitted by GHC users, for whom inaccurate warnings were troublesome (Section 7).

There is quite a bit of related work, which we discuss in Section 8.

ICFP'15, August 31 – September 2, 2015, Vancouver, BC, Canada
ACM. 978-1-4503-3669-7/15/08...$15.00
http://dx.doi.org/10.1145/2784731.2784748

2. The Challenges That We Tackle

The question of determining exhaustiveness and redundancy of pattern matching has been well studied (Section 8), but almost exclusively in the context of purely structural matching. In this section we identify three new challenges:

- The challenge of GADTs and, more generally, of patterns that bind arbitrary existential type variables and constraints (Section 2.2).
- The challenge of laziness (Section 2.3).
- The challenge of guards (Section 2.4).

These issues are all addressed individually in the literature but, to our knowledge, we are the first to tackle all three in a single unified framework, and implement the unified algorithm in a production compiler.

2.1 Background

Given a function definition (or case expression) that uses pattern matching, the task is to determine whether any clauses are missing or redundant.

Missing clauses. Pattern matching of a sequence of clauses is *exhaustive* if every well-typed argument vector matches one of the clauses. For example:

```
zip []     []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

`zip` is not exhaustive because there is a well-typed call that does not match any of its clauses; for example `zip [] [True]`. So the clause `zip [] (b:bs) = e` is *missing*.

Redundant clauses. If there is no well-typed value that matches the left hand side of a clause, the right hand side of the clause can never be accessed and the clause is *redundant*. For example, this equation would be redundant:

```
vzip VN (VCons x xs) = ....
```

Since the application of a partial function to a value outside its domain results in a runtime error, the presence of non-exhaustive pattern matches often indicates a programmer error. Similarly, having redundant clauses in a match is almost never intentional and indicates a programmer error as well. Fortunately, this is a well-studied problem[1, 14–16, 30, 33]: compilers can detect and warn programmers about these anomalies. We discuss this related work in Section 8.

However, Haskell has moved well beyond simple constructor patterns: it has overloaded literal patterns, guards, view patterns, pattern synonyms, and GADTs. In the rest of this section we describe these new challenges, while in subsequent sections we show how to address them.

2.2 The Challenge of GADTs

In recent years, Generalized Algebraic Data Types (GADTs, also known as guarded recursive data types [37], first-class phantom types [4], etc.) have appeared in many programming languages, including Haskell [23, 25], OCaml [10] and Ωmega [28]. Apart from the well-studied difficulties they pose for type inference, GADTs also introduce a qualitatively-new element to the task of determining missing or redundant patterns. As we showed in the Introduction, only a *type-aware* algorithm can generate accurate warnings.

Indeed, although GADTs have been supported by the *Glasgow Haskell Compiler* (GHC) since March 2006 [23], the pattern match check was never extended to take account of GADTs, resulting in many user bug reports. Although there have been attempts to improve the algorithm (see tickets[1] #366 and #2006), all of them are essentially *ad-hoc* and handle only specific cases.

This matters. GHC warns (wrongly) about missing patterns in the definition of `vzip`. Programmers often try to suppress the warning by adding a third fall-through clause:

```
vzip _ _ = error "Inaccessible branch"
```

That suppresses the warning but at a terrible cost: if you modify the data type (by adding a constructor, say), you would hope that you would get warnings about missing cases in `vzip`. But no, the fall-through clause covers the new constructors, so GHC stays silent. At a stroke, that obliterates one of the primary benefits warnings for missing and redundant clauses: namely, their support during software maintenance and refactoring, perhaps years after the original code was written.

Moreover, GADTs are special case of something more general: *data constructors that bind arbitrary existential type variables and constraints*. For example:

```
data T a where
  MkT :: (C a b, F a ~ G b) => a -> b -> T a
```

where `C` is a type class and `F` and `G` are type functions. Here the constructor `MkT` captures an existential type variable `b`, and binds the constraints `(C a b, F a ~ G b)`. In the rest of the paper we draw examples from GADTs, but our formalism and algorithm handles the general case.

2.3 The Challenge of Laziness

Haskell is a lazy language, and it turns out that laziness interacts in an unexpectedly subtle way with pattern matching checks. Here is an example, involving two GADTs:

```
data F a where        data G a where
  F1 :: F Int           G1 :: G Int
  F2 :: F Bool          G2 :: G Char

h :: F a -> G a -> Int
h F1 G1 = 1
h _  _  = 2
```

Given `h`'s type signature, its only well-typed non-bottom arguments are `F1` and `G1` respectively. So, is the second clause for `h` redundant? No! Consider the call `(h F2 ⊥)`, where ⊥ is a diverging value, or an error value such as `(error "urk")`. Pattern matching in Haskell works top-to-bottom, and left-to-right. So we try the first equation, and match the pattern `F1` against the argument `F2`. The match fails, so we fall through to the second equation, which succeeds, returning 2.

Nor is this subtlety restricted to GADTs. Consider:

```
g :: Bool -> Bool -> Int
g _    False = 1
g True False = 2
g _    _     = 3
```

Is the second equation redundant? It certainly *looks* redundant: if the second clause matches then the first clause would have matched too, so `g` cannot possibly return 2. The right-hand side of the second clause is certainly dead code.

Surprisingly, though, *it is not correct to remove the second equation*. What does the call `(g ⊥ True)` evaluate to, where ⊥ is a looping value? Answer: the first clause fails to match, so we attempt to match the second. That requires us to evaluate the first argument

[1] Tickets are GHC bug reports, recorded through the project's bug/issue tracking system: `ghc.haskell.org/trac/ghc`.

of the call, $\perp$, which will loop. But if we omitted the second clause, `(g` $\perp$ `True)` would return 3.

In short, even though the right-hand side of the second equation is dead code, the equation cannot be removed without (slightly) changing the semantics of the program. So far as we know, this observation has not been made before, although previous work [16] would quite sensibly classify the second equation as non-redundant (Section 8).

The same kind of thing happens with GADTs. With the same definitions for `F` and `G`, consider

```
k :: F a -> G a -> Int
k F1 G1 = 1
k _  G1 = 2
```

Is the second equation redundant? After all, anything that matches it would certainly have matched the first equation (or caused divergence if the first argument was $\perp$). So the RHS is definitely dead code; `k` cannot possibly return 2. But removing the second clause would make the definition of `k` inexhaustive: consider the call `(k F2` $\perp$`)`.

The bottom line is this: if we want to report accurate warnings, we must take laziness into account. We address this challenge in this paper.

2.4 The Challenge of Guards

Consider this function:

```
abs1 :: Int -> Int
abs1 x | x < 0     = -x
       | otherwise = x
```

This function makes use of Haskell's boolean-valued *guards*, introduced by "`|`". If the guard returns `True`, the clause succeeds and the right-hand side is evaluated; otherwise pattern-matching continues with the next clause.

It is clear to the reader that this function is exhaustive, but not so clear to a compiler. Notably, `otherwise` is not a keyword; it is simply a value defined by `otherwise = True`. The compiler needs to know that fact to prove that the pattern-matching is exhaustive. What about this version:

```
abs2 :: Int -> Int
abs2 x | x < 0  = -x
       | x >= 0 = x
```

Here the exhaustiveness of pattern-matching depends on knowledge of the properties of < and >=. In general, the exhaustiveness for pattern matches involving guards is clearly undecidable; for example, it could depend on a deep theorem of arithmetic. But we would like the compiler to do a good job in common cases such as `abs1`, and perhaps `abs2`.

GHC extends guards further with *pattern guards*. For example:

```
append xs ys
  | []     <- xs = ys
  | (p:ps) <- xs = p : append ps ys
```

The pattern guard matches a specified expression (here `xs` in both cases) against a pattern; if matching succeeds, the guard succeeds, otherwise pattern matching drops through to the next clause. Other related extensions to basic pattern matching include literal patterns and *view patterns* [9, 32].

All these guard-like extensions pose a challenge to determining the exhaustiveness and redundancy of pattern-matching, because pattern matching is no longer purely structural. Every real compiler must grapple with this issue, but no published work gives a systematic account of how to do so. We do so here.

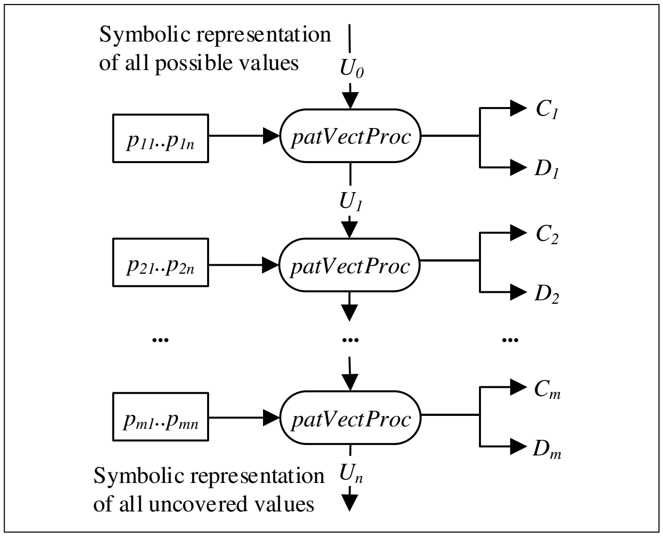

Figure 1: Algorithm Outline

3. Overview of Our Approach

In this section we describe our approach in intuitive terms, showing how it addresses each of the three challenges of Section 2. We subsequently formalise the algorithm in Section 4.

3.1 Algorithm Outline

The most common use of pattern matching in Haskell is when a function is defined using multiple *clauses*:

$$f\ p_{11}\ \ldots p_{1n}\ = e_1 \qquad \text{Clause 1}$$
$$\ldots$$
$$f\ p_{m1} \ldots p_{mn} = e_m \qquad \text{Clause } m$$

From the point of view of pattern matching, the function name "f" is incidental: all pattern matching in Haskell can be regarded as a sequence of clauses, each clause comprising a pattern vector and a right hand side. For example, a `case` expression also has multiple clauses (each with only one pattern); a Haskell pattern matching lambda has a single clause (perhaps with multiple patterns); and so on.

In Haskell, pattern matching on a sequence of clauses is carried out top-to-bottom, and left-to-right. In our function f above, Haskell matches the first argument against p_{11}, the second against p_{12} and so on. If all n patterns in the first clause match, the right hand side is chosen; if not, matching resumes with the next clause. Our algorithm, illustrated in Figure 1, works in the same way: it analyses the clauses one by one, from top to bottom. The analysis *patVectProc* of an individual clause takes a compact symbolic representation of the vector of argument values that are possibly submitted to the clause, and partitions these values into three different groups:

C The values that are *covered* by the clause; that is, values that match the clause without divergence, so that the right-hand side is evaluated.

D The values that *diverge* when matched against the clause, so that the right-hand side is not evaluated, but neither are any subsequent clauses matched.

U The remaining *uncovered* values; that is, the values that fail to match the clause, without divergence.

As illustrated in Figure 1, the input to the first clause represents all possible values, and each subsequent clause is fed the uncovered values of the preceding clause. For example, consider the function zip from the Introduction:

```
zip []     []     = []
zip (a:as) (b:bs) = (a,b) : zip as bs
```

We start the algorithm with $C_0 = \{\texttt{\_ \_}\}$, where we use "_" to stand for "all values". Processing the first clause gives:

$$\begin{aligned} C_1 &= \{\texttt{[] []}\} \\ D_1 &= \{\bot\ \texttt{\_},\ \texttt{[]}\ \bot\} \\ U_1 &= \{\texttt{[] (\_:\_)},\ \texttt{(\_:\_) \_}\} \end{aligned}$$

The values that fail to match the first clause, and do so without divergence, are U_1, and these values are fed to the second clause. Again we divide the values into three groups:

$$\begin{aligned} C_2 &= \{\texttt{(\_:\_) (\_:\_)}\} \\ D_2 &= \{\texttt{(\_:\_)}\ \bot\} \\ U_2 &= \{\texttt{[] (\_:\_)},\ \texttt{(\_:\_) []}\} \end{aligned}$$

Now, U_2 describes the values that fail to match either clause. Since it is non-empty, the clauses are not exhaustive, and a warning should be generated. In general we generate three kinds of warnings:

1. If the function is defined by m clauses, and U_m is non-empty, then the clauses are non-exhaustive, and a warning should be reported. It is usually helpful to include the set U_m in the error message, so that the user can see which patterns are not covered.
2. Any clause i for which C_i and D_i are both empty is redundant, and can be removed altogether.
3. Any clause i for which C_i is empty, but D_i is not, has an inaccessible right hand side even though the equation cannot be removed. This is unusual, and deserves its own special kind of warning; again, including D_i in the error message is likely to be helpful.

Each of C, U, and D is a set of *value abstractions*, a compact representation of a set of value vectors that are covered, uncovered, or diverge respectively. For example, the value abstraction `(_:_) []` stands for value vectors such as

```
(True:[])                []
(False : (True : []))    []
```

and so on. Notice in D_1, D_2 that our value abstractions must include $\bot$, so that we can describe values that cause matching to diverge.

3.2 Handling Constraints

Next we discuss how these value abstractions may be extended to handle GADTs. Recall vzip from the Introduction

```
vzip :: Vect n a -> Vect n b -> Vect n (a,b)
vzip VN        VN        = VN
vzip (VC x xs) (VC y ys) = VC (x,y) (vzip xs ys)
```

What do the uncovered sets U_i look like? Naively they would look like that for zip:

$$\begin{aligned} U_1 &= \{\texttt{VN (VC \_ \_)},\ \texttt{(VC \_ \_) \_}\} \\ U_2 &= \{\texttt{VN (VC \_ \_)},\ \texttt{(VC \_ \_) VN}\} \end{aligned}$$

To account for GADTs we add *type constraints* to our value abstractions, to give this:

$$\begin{aligned} U_1 = \{&\texttt{VN (VC \_ \_)} \triangleright (\texttt{n} \sim \texttt{Zero}, \texttt{n} \sim \texttt{Succ n2}) \\ &, \texttt{(VC \_ \_) \_} \triangleright (\texttt{n} \sim \texttt{Succ n2})\} \end{aligned}$$

Each value tuple abstraction in the set now comes with a type equality constraint (e.g. $\texttt{n} \sim \texttt{Succ n2}$), and represents values of the specified syntactic shape, *for which the equality constraint is satisfiable* at least for some substitution of its free variables. The first abstraction in U_1 has a constraint that is *unsatisfiable*, because `n` cannot simultaneously be equal to both `Zero` and `Succ n2`. Hence the first abstraction in U_1 represents the empty set of values and can be discarded. Discarding it, and processing the second clause gives

$$U_2 = \{\texttt{(VC \_ \_) VN} \triangleright (\texttt{a} \sim \texttt{Succ n}, \texttt{a} \sim \texttt{Zero})\}$$

Again the constraint is unsatisfiable, so U_2 is empty, which says that the function is exhaustive.

We have been a bit sloppy with binders (e.g. where is `n2` bound?), but we will tighten that up in the next section. The key intuition is this: *the abstraction $u \triangleright \Delta$ represents the set of values whose syntactic shape is given by u, and for which the type constraint Δ is satisfied.*

3.3 Guards and Oracles

In the previous section we extended value abstractions with a conjunction of type-equality constraints. It is straightforward to take the idea further, and add term-equality constraints. Then the final uncovered set for function abs2 (Section 2.4) might look like this:

$$U_2 = \{\texttt{x} \triangleright (\texttt{False} = \texttt{x<0}, \texttt{False} = \texttt{x>=0})\}$$

We give the details of how we generate this set in Section 4, but intuitively the reasoning goes like this: if neither clause for `abs2` matches, then both boolean guards must evaluate to `False`. Now, if the compiler knows enough about arithmetic, it may be able to determine that the constraint is unsatisfiable, and hence that U_2 is empty, and hence that `abs2` is exhaustive.

For both GADTs and guards, the question becomes this: *is the constraint Δ unsatisfiable?* And that is a question that has been *extremely* well studied, for many particular domains. For the purposes of this paper, therefore, we treat satisfiability as a black box, or oracle: the algorithm is parameterised over the choice of oracle. For type-equality constraints we have a very good oracle, namely GHC's own type-constraint solver. For term-level constraints we can plug in a variety of solvers. This modular separation of concerns is extremely helpful, and is a key contribution of our approach.

3.4 Complexity

Every pattern-checking algorithm has terrible worst-case complexity, and ours is no exception. For example, consider

```
data T = A | B | C
f A A = True
f B B = True
f C C = True
```

What values U_3 are not covered by `f`? Answer

```
{ A B, A C, B A, B C, C A, C B }
```

The size of the uncovered set is the square of the number of constructors in `T`. It gets worse: Sekar et al. [26] show that the problem of finding redundant clauses is NP-complete, by encoding the boolean satisfiability (SAT) problem into it. So the worst-case running time is necessarily exponential. But so is Hindley-Milner type inference! As with type inference, we hope that worst case behaviour is rare in practice. Moreover, GHC's current redundancy checker suffers from the same problem without obvious problems in practice. We have gathered quantitative data about set sizes to better characterise the problem, which we discuss in Appendix A.

Types		
τ	$::= a \mid \tau_1 \to \tau_2 \mid T\,\overline{\tau} \mid \ldots$	Monotypes
$a, b, a', b', \ldots$		Type variables
T		Type constructors
Γ	$::= \epsilon \mid \Gamma, a \mid \Gamma, x : \tau$	Typing environment
Terms and clauses		
$f, g, x, y, \ldots$		Term variables
e		Expression
c	$::= \vec{p} \to e$	Clause
Patterns		
K		Data constructors
p, q	$::= x \mid K\,\vec{p} \mid G$	Pattern
G	$::= p \leftarrow e$	Guard
Value abstractions		
S, C, U, D	$::= \overline{v}$	Value set abstraction
v	$::= \Gamma \vdash \vec{u} \rhd \Delta$	Value vector abstraction
u, w	$::= x \mid K\,\vec{u}$	Value abstraction
Constraints		
Δ	$::= \epsilon \mid \Delta \cup \Delta$	
	$\mid Q$	Type constraint
	$\mid x \approx e$	Term-equality constraint
	$\mid x \approx \bot$	Strictness constraint
Q	$::= \tau \sim \tau$	Type-equality constraint
	$\mid \ldots$	other constraint

Figure 2: Syntax

4. Our Algorithm in Detail

4.1 Syntax

Figure 2 gives the syntax used in the formalisation of the algorithm. The syntax for types, type constraints and type environments is entirely standard. We are explicit about the binding of type variables in Γ, but for this paper we assume they all have kind $*$, so we omit their kind ascriptions. (Our real implementation supports higher kinds, and indeed kind polymorphism.)

A *clause* is a vector of patterns $\vec{p}$ and a right-hand side e, which should be evaluated if the pattern matches. Here, a "vector" $\vec{p}$ of patterns is an ordered sequence of patterns: it is either empty, written ϵ, or is of the form $p\,\vec{p}$.

A pattern p is either a variable pattern x, a constructor pattern $K\ \vec{p}$ or a *guard* G. We defer everything concerning guards to Section 4.4, so that we can initially concentrate on GADTs.

Value abstractions play a central role in this paper, and stand for sets of values. They come in three forms:

- A *value set abstraction* S is a set of value abstractions $\overline{v}$. We use an overline $\overline{v}$ (rather than an arrow) to indicate that the order of items in S does not matter.
- A *value vector abstraction* v has the form $\Gamma \vdash \vec{u} \rhd \Delta$. It consists of a vector $\vec{u}$ of syntactic value abstractions, and a constraint Δ. The type environment Γ binds the free variables of $\vec{u}$ and Δ.
- A *syntactic value abstraction* u is either a variable x, or is of the form $K\,\vec{u}$, where K is a data constructor.

A value abstraction represents a set of values, using the intuitions of Sections 3.1 and 3.2. We formalise these sets precisely in Section 5.

Finally, a constraint Δ is a conjunction of either type constraints Q or term equality constraints $x \approx e$, and in addition *strictness* constraints $x \approx \bot$. Strictness constraints are important for computing diverge sets for which we've used informal notation in the previous sections: For example $\{$`(_:_)` $\bot\}$ is formally represented as $\{\Gamma \vdash (x{:}y)\ z \rhd z \approx \bot\}$ for some appropriate environment Γ.

Type constraints include type equalities $\tau_1 \sim \tau_2$ but can also potentially include other constraints introduced by pattern matching or type signatures (examples would be type class constraints or refinements [24, 31]). We leave the syntax of Q deliberately open.

4.2 Clause Processing

Our algorithm performs an abstract interpretation of the concrete dynamic semantics described in the last section, and manipulates value vector abstractions instead of concrete value vectors. It follows the scheme described in Section 3.1 and illustrated in Figure 1. The key question is how *patVectProc* works; that is the subject of this section, and constitutes the heart of the paper.

Initialisation As shown in Figure 1, the algorithm is initialised with a set U_0 representing "all values". For every function definition of the form:

$$\begin{array}{l} f :: \forall\vec{a}.\tau_1 \to \ldots \to \tau_n \to \tau \\ f\ p_{11}\ \ldots\ p_{1n} = \ldots \\ \quad\ldots \\ f\ p_{m1}\ \ldots\ p_{mn} = \ldots \end{array}$$

the initial call to *patVectProc* will be with a singleton set:

$$U_0 = \{\vec{a}, (x_1{:}\tau_1), \ldots, (x_n{:}\tau_n) \vdash x_1\ \ldots\ x_n \rhd \epsilon\}$$

As a concrete example, the pattern match clauses of function `zip` of type $\forall ab.$`[`a`]` $\to$ `[`b`]` $\to$ `[`(a,b)`]` from Section 3.1 will be initialised with

$$U_0 = \{a, b, (x_1{:}\texttt{[}a\texttt{]}), (x_2{:}\texttt{[}b\texttt{]}) \vdash x_1\ x_2 \rhd \epsilon\}$$

Notice that we use variables x_i, rather than the underscores used informally in Section 3.1, so that we can record their types in Γ, and constraints on their values in Δ.

The main algorithm Figure 3 gives the details of *patVectProc*. Given a pattern vector $\vec{p}$ and an incoming set S of value vector abstractions, *patVectProc* computes the sets C, U, D of covered, uncovered, and diverging values respectively. As Figure 3 shows, each is computed independently, in two steps. For each value vector abstraction v in S:

- *Use syntactic structure:* an auxiliary function ($\mathcal{C}$,$\mathcal{U}$ and $\mathcal{D}$) identifies the subset of v that is covered, uncovered, and divergent, respectively.
- *Use type and term constraints:* filter the returned set, retaining only those members whose constraints Δ are satisfiable.

We describe each step in more detail, beginning with the syntactic function for covered sets, C.

Computing the covered set The function $\mathcal{C}\ \vec{p}\ v$ refines v into those vectors that are covered by the pattern vector $\vec{p}$. It is defined inductively over the structure of $\vec{p}$.

Rule [CNIL] handles the case when both the pattern vector and the value vector are empty. In this case the value vector is trivially covered.

Rule [CCONCON] handles the case when both the pattern and value vector start with constructors K_i and K_j respectively. If the constructors differ, then this particular value vector *is not* covered and we return $\varnothing$. If the constructors are the same, $K_i = K_j$, then we proceed recursively with the subterms $\vec{p}$ and $\vec{u}$ and the suffixes $\vec{q}$ and $\vec{w}$. We flatten these into a single recursive call, and recover the structure afterwards with *kcon* K_i, defined thus:

$$kcon\ K\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta) = \Gamma \vdash (K\ \vec{u})\ \vec{w} \rhd \Delta$$

where $\vec{u}$ matches the arity of K.

$$\boxed{\mathit{patVectProc}(\vec{p}, S) = \langle C, U, D \rangle}$$

$$\mathit{patVectProc}\,(\vec{p}, S) = \langle C, U, D \rangle \quad \text{where} \quad \begin{array}{l} C = \{ w \mid v \in S, w \in \mathcal{C}\,\vec{p}\,v,\ \vdash_{\mathrm{SAT}} w \} \\ U = \{ w \mid v \in S, w \in \mathcal{U}\,\vec{p}\,v,\ \vdash_{\mathrm{SAT}} w \} \\ D = \{ w \mid v \in S, w \in \mathcal{D}\,\vec{p}\,v,\ \vdash_{\mathrm{SAT}} w \} \end{array}$$

$$\boxed{\mathcal{C}\,\vec{p}\,v = C \ \text{ (always empty or singleton set)}}$$

$$\begin{array}{llll}
[\text{CNil}] & \mathcal{C}\ \epsilon & (\Gamma \vdash \epsilon \rhd \Delta) & = \{\, \Gamma \vdash \epsilon \rhd \Delta \,\} \\
[\text{CConCon}] & \mathcal{C}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta) & = \begin{cases} \mathit{map}\ (\mathit{kcon}\ K_i)\ (\mathcal{C}\ (\vec{p}\,\vec{q})\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta)) & \text{if } K_i = K_j \\ \varnothing & \text{if } K_i \neq K_j \end{cases} \\
[\text{CConVar}] & \mathcal{C}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = \mathcal{C}\ ((K_i\,\vec{p})\,\vec{q})\ (\Gamma' \vdash (K_i\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \quad \text{where } \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x{:}\tau_x) \in \Gamma \quad K_i :: \forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau \\
 & & & \qquad\quad \Gamma' = \Gamma, \vec{a}, \vec{y}{:}\vec{\tau} \\
 & & & \qquad\quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i\,\vec{y} \\
[\text{CVar}] & \mathcal{C}\ (x\,\vec{p}) & (\Gamma \vdash u\,\vec{u} \rhd \Delta) & = \mathit{map}\ (\mathit{ucon}\ u)\ (\mathcal{C}\ (\vec{p})\ (\Gamma, x{:}\tau \vdash \vec{u} \rhd \Delta \cup x \approx u)) \qquad \text{where } x\#\Gamma \quad \Gamma \vdash u : \tau \\
[\text{CGuard}] & \mathcal{C}\ ((p \leftarrow e)\,\vec{p}) & (\Gamma \vdash \vec{u} \rhd \Delta) & = \mathit{map\ tail}\ (\mathcal{C}\ (p\,\vec{p})\ (\Gamma, y{:}\tau \vdash y\,\vec{u} \rhd \Delta \cup y \approx e)) \qquad \text{where } y\#\Gamma \quad \Gamma \vdash e : \tau
\end{array}$$

$$\boxed{\mathcal{U}\,\vec{p}\,v = U}$$

$$\begin{array}{llll}
[\text{UNil}] & \mathcal{U}\ \epsilon & (\Gamma \vdash \epsilon \rhd \Delta) & = \varnothing \\
[\text{UConCon}] & \mathcal{U}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta) & = \begin{cases} \mathit{map}\ (\mathit{kcon}\ K_i)\ (\mathcal{U}\ (\vec{p}\,\vec{q})\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta) & \text{if } K_i = K_j \\ \{\, \Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta \,\} & \text{if } K_i \neq K_j \end{cases} \\
[\text{UConVar}] & \mathcal{U}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = \bigcup_{K_j} \mathcal{U}\ ((K_i\,\vec{p})\,\vec{q})\ (\Gamma' \vdash (K_j\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \quad \text{where } \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x{:}\tau_x) \in \Gamma \quad K_j :: \forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau \\
 & & & \qquad\quad \Gamma' = \Gamma, \vec{a}, \vec{y}{:}\vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_j\,\vec{y} \\
[\text{UVar}] & \mathcal{U}\ (x\,\vec{p}) & (\Gamma \vdash u\,\vec{u} \rhd \Delta) & = \textit{exactly like } [\text{CVar}]\textit{, with } \mathcal{U} \textit{ instead of } \mathcal{C} \\
[\text{UGuard}] & \mathcal{U}\ ((p \leftarrow e)\,\vec{p}) & (\Gamma \vdash \vec{u} \rhd \Delta) & = \textit{exactly like } [\text{CGuard}]\textit{, with } \mathcal{U} \textit{ instead of } \mathcal{C}
\end{array}$$

$$\boxed{\mathcal{D}\,\vec{p}\,v = D}$$

$$\begin{array}{llll}
[\text{DNil}] & \mathcal{D}\ \epsilon & (\Gamma \vdash \epsilon \rhd \Delta) & = \varnothing \\
[\text{DConCon}] & \mathcal{D}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash (K_j\,\vec{u})\,\vec{w} \rhd \Delta) & = \begin{cases} \mathit{map}\ (\mathit{kcon}\ K_i)\ (\mathcal{D}\ (\vec{p}\,\vec{q})\ (\Gamma \vdash \vec{u}\,\vec{w} \rhd \Delta) & \text{if } K_i = K_j \\ \varnothing & \text{if } K_i \neq K_j \end{cases} \\
[\text{DConVar}] & \mathcal{D}\ ((K_i\,\vec{p})\,\vec{q}) & (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = \{\, \Gamma \vdash x\,\vec{u} \rhd \Delta \cup (x \approx \bot)\} \ \cup \ \mathcal{D}\ ((K_i\,\vec{p})\,\vec{q})\ (\Gamma' \vdash (K_i\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \quad \text{where } \vec{y}\#\Gamma \quad \vec{a}\#\Gamma \quad (x{:}\tau_x) \in \Gamma \quad K_i :: \forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau \\
 & & & \qquad\quad \Gamma' = \Gamma, \vec{a}, \vec{y}{:}\vec{\tau} \quad \Delta' = \Delta \cup Q \cup \tau \sim \tau_x \cup x \approx K_i\,\vec{y} \\
[\text{DVar}] & \mathcal{D}\ (x\,\vec{p}) & (\Gamma \vdash u\,\vec{u} \rhd \Delta) & = \textit{exactly like } [\text{CVar}]\textit{, with } \mathcal{D} \textit{ instead of } \mathcal{C} \\
[\text{DGuard}] & \mathcal{D}\ ((p \leftarrow e)\,\vec{p}) & (\Gamma \vdash \vec{u} \rhd \Delta) & = \textit{exactly like } [\text{CGuard}]\textit{, with } \mathcal{D} \textit{ instead of } \mathcal{C}
\end{array}$$

Figure 3: Clause Processing

Rule [CConVar] handles the case when the pattern vector starts with constructor K_i and the value vector with variable x. In this case we refine x to the most general abstraction that matches the constructor, $K_i\,\vec{y}$, where the variables $\vec{y}$ are fresh for Γ, written $\vec{y}\#\Gamma$. Once the constructor shape for x has been exposed, rule [CConCon] will fire to recurse into the pattern and value vectors. The constraint (Δ') used in the recursive call consists of the union of the original Δ with:

- Q; this is the constraint bound by the constructor K_i :: $\forall\vec{a}.Q \Rightarrow \vec{\tau} \rightarrow \tau$, which may for example include type equalities (in the case of GADTs).
- $x \approx K_i\ \vec{y}$; this records a term-level equality in the constraint that could be used by guard expressions.
- $\tau \sim \tau_x$, where τ_x is the type of x in the environment Γ, and τ is the return type of the constructor. This constraint will be useful when dealing with GADTs as we explain in Section 4.3.

Rule [CVar] applies when the pattern vector starts with a variable pattern x. This matches any value abstraction u, so we can proceed inductively in $\vec{p}$ and $\vec{u}$. However x may appear in some *guard* in the rest of the pattern, for example:

```
f x y | Nothing <- lookup x env = ...
```

To expose the fact that x is bound to u in subsequent guards (and in the right-hand side of the clause, see Section 4.6), rule [CVar] adds $x \approx u$ to the constraints Δ, and correspondingly extends Γ to maintain the invariant that Γ binds all variables free in Δ. Finally, $\mathit{map}\ (\mathit{ucon}\ u)$ prefixes each of the recursive results with u:

$$\mathit{ucon}\ u\ (\Gamma \vdash \vec{u} \rhd \Delta) = \Gamma \vdash u\,\vec{u} \rhd \Delta$$

Rule [CGuard] deals with guards: see Section 4.4.

Finally it is worth noting that the $\mathcal{C}\,\vec{p}\,v$ function always returns an empty or singleton set, but we use the full set notation for uniformity with the other functions.

Computing the uncovered and divergent sets The two other functions have a similar structure. Hence, we only highlight the important differences.

The function $\mathcal{U}\,\vec{p}\,v$ returns those vectors that are *not covered* by the pattern vector $\vec{p}$. When both the pattern vector and value vector are empty then (we have seen in the previous case) that the value vector is covered and hence we return $\varnothing$. In rule [UConCon] there are two cases, just as in [CConCon]. If the head constructors match ($K_i = K_j$), we simply recurse; but if not, the entire value vector abstraction is uncovered, so we return it. In case [UConVar] we take the union of the uncovered sets for all refinements of the variable x to a constructor K_j; each can lead recursively through rule [UConCon] to uncovered cases. To inform guards, we record the equality $x \approx K_j\ \vec{y}$ for each constructor. As in rule [CConVar] we also record a type constraint between the constructor return type and the type of x in Γ. (Section 4.3)

The function $\mathcal{D}\ \vec{p}\ v$ returns those vectors that diverge when matching the pattern vector $\vec{p}$. The empty value vector does not diverge [DNil]. The case for variables [DVar] is similar to previous cases. In the case of constructors in the head of the pattern vector as well as the value vector [DConCon] there is no divergence either – we either recurse when the constructors match or else return the empty divergent set. When the clause starts with constructor K_i, and the vector with a variable x, rule [DConVar] combines two different results: (a) the first result represents symbolically all vectors where x diverges; (b) the second result recurses by refining x to $K_i\ \vec{y}$. In the first case we record the divergence of x with a *strictness* constraint ($x \approx \bot$). For the second case, we appeal recursively to the divergent set computation (We give more details on the Δ' that we use to recurse in Section 4.3.)

Filtering the results with constraints Function $patVectProc$ prunes the results of $\mathcal{C}\ \vec{p}\ v$, $\mathcal{U}\ \vec{p}\ v$, and $\mathcal{D}\ \vec{p}\ v$ that are semantically empty by appealing to an oracle judgement $\vdash_{\text{SAT}} (\Gamma \vdash \vec{u} \rhd \Delta)$. In the next section we define "semantically empty" by giving a denotational semantics to a value vector abstraction $[\![v]\!]$ as a set of concrete value vectors.

The purpose of $\vdash_{\text{SAT}}$ is to determine whether this set is empty. However, because satisfiability is undecidable in general (particularly when constraints involve term equivalence), we have to be content with a decidable algorithm $\vdash_{\text{SAT}}$ that gives sound but incomplete approximation to satisfiability:

$$\not\vdash_{\text{SAT}} v \quad\Rightarrow\quad [\![v]\!] = \emptyset$$

In terms of the outcomes (1-3) in Section 3.1, "soundness" means

1. If we do not warn that a set of clauses may be non-exhaustive, then they are definitely exhaustive.
2. If we warn that a clause is redundant, then it definitely is redundant.
3. If we warn that a right-hand side of a non-redundant clause is inaccessible, then it definitely is inaccessible.

Since $\vdash_{\text{SAT}}$ is necessarily incomplete, the converse does not hold in general. There is, of course, a large design space of less-than-complete implementations for $\vdash_{\text{SAT}}$. Our implementation is explained in Section 6.

Another helpful insight is this: during constraint generation (Figure 3) the sole purpose of adding constraints to Δ is to increase the chance that $\vdash_{\text{SAT}}$ will report "unsatisfiable". It is always sound to omit constraints from Δ; so an implementation is free to trade off accuracy against the size of the constraint set.

4.3 Type Constraints from GADTs

Rules [CConVar], [UConVar], and [DConVar] record *type equalities* of the form $\tau \sim \tau_x$ between the value abstraction type (τ_x) and the return type of the appropriate data constructor each time (τ).

Recording these constraints in [CConVar] and [UConVar] is important for reporting precise warnings when dealing with GADTs, as the following example demonstrates:

```
data T a where
  TList :: T [a]
  TBool :: T Bool

foo :: T c -> T c -> Int
foo TList _ = ...
foo _ TList = ...
```

To determine C_2, the covered set from the second equation, we start from an initial singleton vector abstraction $U_0 = \{\Gamma_0 \vdash x_1\ x_2 \rhd \epsilon\}$ with $\Gamma = c, x_1{:}\texttt{T}\ c, x_2{:}\texttt{T}\ c$. Next compute the uncovered set from the first clause, which (via [UConVar] and [UVar]) is $U_1 = \{\Gamma_1 \vdash \texttt{TBool}\ x_2 \rhd \Delta_1\}$, where

$$\begin{array}{rcl} \Gamma_1 & = & \Gamma_0, a \\ \Delta_1 & = & (x_1 \approx \texttt{TBool}) \cup (\texttt{T}\ c \sim \texttt{T Bool}) \end{array}$$

Note the recorded type constraint for the uncovered constructor TBool from rule [UConVar]. Next, from U_1, compute the covered set for the second equation (via [CVar] and [CConVar]):

$$\begin{array}{rcl} C_2 & = & \mathcal{C}\ (\_\ \texttt{TList})\ (\Gamma_1 \vdash \texttt{TBool}\ x_2 \rhd \Delta_1) \\ & = & \{\Gamma_1, b \vdash \texttt{TBool TList} \rhd \Delta_2\} \\ & & \text{where } \Delta_2 = \Delta_1 \cup (x_2 \approx \texttt{TList}) \cup (\texttt{T}\ c \sim \texttt{T}\ [b]) \end{array}$$

Note the type constraint $\texttt{T}\ c \sim \texttt{T}[b]$ generated by rule [CConVar]. The final constraint Δ_2 is unsatisfiable and C_2 is semantically empty, and the second equation is unreachable. Unless [CConVar] or [UConVar] both record the type constraints we would miss reporting the second branch as redundant.

Rule [DConVar] also records term and type-level constraints in the recursive call. Indeed if the first case in that rule is deemed unsatisfiable by our oracle it is important to have a precise set of constraints for the recursive call to detect possible semantic emptiness of the result.

4.4 Guards

A major feature of our approach is that it scales nicely to handle *guards*, and other syntactic extensions of pattern-matching supported by GHC. We briefly reprise the development so far, adding guards at each step.

Syntax (Section 4.1). We begin with the syntax in Figure 2: a pattern p can be a *guard*, g, of the form $(p \leftarrow e)$. This syntax is very general. For example, the clauses of abs1 (Section 2.4) would desugar to:

```
x (True <- x<0)       -> -x
x (True <- otherwise) -> x
```

Notice that these *two*-element pattern vectors match against *one* argument; a guard $(p \leftarrow e)$ matches against e, not against an argument.

GHC's pattern guards are equally easy to represent; there is no desugaring to do! However, the syntax of Figure 2 is more expressive than GHC's pattern guards, because it allows a guard to occur *arbitrarily nested inside a pattern*. This allow us to desugar literal patterns and view patterns. For example, consider the Haskell function

```
f ('x', []) = True
f _         = False
```

The equality check against the literal character 'x' must occur *before* matching the second component of the tuple, so that the call (f ('y',,$\bot$)) returns False rather than diverging. With our syntax we can desugar f to these two clauses:

```
(a (True <- a=='x'), [])  -> True
c                         -> False
```

Note the nested guard True <- a=='x'. It is not hard to see how to desugar view patterns in a similar way; see the extended version of this paper [11].

Clause processing (Section 4.2). It is easy to extend the clause-processing algorithm to accommodate guards. For example, equation [CGuard] in Figure 3 deals with the case when the first pattern in the pattern vector is a guard $(p \leftarrow e)$. We can simply make a recursive call to $\mathcal{C}$ adding p to the front of the pattern vector, and a fresh variable y to the front of the value abstraction. This variable y has the same type τ as e, and we add a term-equality constraint $y \approx e$ to the constraint set. Finally, the $map\ tail$ removes the guard value from the returned value vector:

$$tail\ (\Gamma \vdash u\, \vec{us}\ \triangleright\ \Delta) = \Gamma \vdash \vec{us}\ \triangleright\ \Delta)$$

That's all there is to it! The other cases are equally easy. However, it is illuminating to see how the rules work in practice. Consider again function `abs1` in Section 2.4. We may compute (laboriously) as follows:

$$
\begin{aligned}
U_0 &= \{v{:}\mathtt{Int} \vdash v \triangleright\, \} \\
U_1 &= \mathcal{U}\,(x\,(\mathtt{True} \leftarrow x{<}0))\ (v{:}\mathtt{Int} \vdash v \triangleright\,) \\
&= \text{(apply [UVar])} \\
&\quad\ map\ (ucon\ v)\ (\mathcal{U}\,(\mathtt{True} \leftarrow v{<}0)\ (v{:}\mathtt{Int} \vdash \epsilon\ \triangleright\ x \approx v)) \\
&= \text{(apply [UGuard])} \\
&\quad\ map\ (ucon\ v)\ (map\ tail \\
&\qquad (\mathcal{U}\,(\mathtt{True})\ (v{:}\mathtt{Int}, y{:}\mathtt{Bool} \vdash y\ \triangleright\ x \approx v, y \approx v{<}0)) \\
&= \text{(apply [UConVar]; the True/True case yields } \varnothing) \\
&\quad\ map\ (ucon\ v)\ (map\ tail\ (map\ (ucon\ y) \\
&\qquad (\mathcal{U}\,(\mathtt{True})\ (v{:}\mathtt{Int}, y{:}\mathtt{Bool} \vdash \mathtt{False} \\
&\qquad\qquad\qquad \triangleright\, x \approx v, y \approx v{<}0, y \approx \mathtt{False})) \\
&= \text{(apply [UConCon] with } K_i \neq K_j\text{, and do the } maps) \\
&\quad\ \{v{:}\mathtt{Int}, y{:}\mathtt{Bool} \vdash v\ \triangleright\ x \approx v, y \approx v{<}0, y \approx \mathtt{False}\}
\end{aligned}
$$

This correctly characterises the uncovered values as those v:`Int` for which $v{<}0$ is `False`.

4.5 Extension 1: Smarter Initialisation

In the previous section, we always initialised U_0 with the empty constraint, $\Delta = \epsilon$. But consider these definitions:

```
type family F a              data T a where
type instance F Int = Bool     TInt  :: T Int
                               TBool :: T Bool
```

Datatype `T` is a familiar GADT definition. `F` is a *type family*, or type-level function, equipped with an instance that declares `F Int = Bool`. Given these definitions, is the second clause of `f` below redundant?

```
f :: F a ~ b => T a -> T b -> Int
f TInt  TBool = ...
f TInt  x     = ...
f TBool y     = ...
```

Function `f` matches the first argument with `TInt`, yielding the local type equality `a` $\sim$ `Int`. Using this fact, together with the signature constraint `F a` $\sim$ `b` and the top-level equation `F Int = Bool`, we can deduce that `Bool` $\sim$ `b`, and hence the second clause is in fact redundant. In this reasoning we had to use the quantified constraint `F a` $\sim$ `b` from the signature of `f`. Hence the initial value abstraction U_0 for this pattern match should include constraints from the function signature:

$$U_0 = \{a, b, (x_1{:}\mathrm{T}\ a), (x_2{:}\mathrm{T}\ b) \vdash x_1\ x_2\ \triangleright\ \mathtt{F}\,\mathbf{a} \sim \mathbf{b}\}$$

4.6 Extension 2: Nested Pattern Matches

Consider this definition:

```
f [] = ...
f x  = ...(case x of { w:ws -> e })...
```

The clauses of `f` and those of the inner `case` expression are entirely disconnected. And yet we can see that both of the inner `case` expressions are exhaustive, because the `x` $=$ `[]` case is handled by the first equation.

Happily there is a principled way to allow the inner `case` to take advantage of knowledge from the outer one: *gather the constraints from the covered set of the outer pattern match, propagate them inwards, and use them to initialise U_0 for the inner one*. In this example, we may follow the algorithm as follows:

$$
\begin{aligned}
U_0^f &= \{a, v{:}[a] \vdash v \triangleright\, \} \\
U_1^f &= \{a, v{:}[a], v_1{:}a, v_2{:}[a] \vdash (v_1{:}v_2) \triangleright\, \} \\
C_2^f &= \{a, v{:}[a], v_1{:}a, v_2{:}[a], x{:}[a] \vdash (v_1{:}v_2)\ \triangleright\ x \approx v_1{:}v_2\}
\end{aligned}
$$

Propagate C_2^f inwards to the `case` expression. Now initialise the U_0^{case} for the `case` expression thus:

$$U_0^{case} = \{(\Gamma \vdash x\ \triangleright\ \Delta) \mid (\Gamma \vdash \vec{u}\ \triangleright\ \Delta) \in C_2^f\}$$

You can see that the Δ used for the inner `case` will include the constraint $x = v_1{:}v_2$ inherited from C_2^f, and that in turn can be used by $\vdash_{\mathrm{SAT}}$ to show that the `[]` missing branch of the `case` is inaccessible. Notice that U_0 many now have more than one element; until now it has always been a singleton.

The same idea works for type equalities, so that type-equality knowledge gained in an outer pattern-match can be carried inwards in Δ and used to inform inner pattern matches. Our implementation does exactly this and solves the existing GHC ticket #4139 that needs this functionality. (Caveat: our implementation so far only propagates type constraints, not term constraints.)

5. Meta-theory

In order to formally relate the algorithm to the dynamic semantics of pattern matching, we first formalise the latter as well as the semantics of the value abstractions used by the former.

5.1 Value Abstractions

As outlined in Section 3.1 a value abstraction denotes a set of values. Figure 4 formalises this notion.

As the Figure shows, the meaning of a closed value abstraction $\Gamma \vdash \vec{u}\ \triangleright\ \Delta$ is the set of all type-respecting instantiations of $\vec{u}$ to a vector of (closed) values $\vec{V} = \theta(\vec{u})$, such that the constraints $\theta(\Delta)$ are satisfied. The judgement $\models \Delta$ denotes the logical entailment of the (closed) constraints Δ; we omit the details of its definition for the sake of brevity.

A "type-respecting instantiation", or denotation, of a type environment Γ is a substitution θ whose domain is that of Γ; it maps each type variable $a \in \Gamma$ to a closed type; and each term variable $x{:}\tau \in \Gamma$ to a closed value V of the appropriate type $\vdash_v V : \tau$. The syntax of closed types and values is given in Figure 4, as is the typing judgement for values. For example,

$$
\begin{array}{l}
[\![\{\mathtt{a}, \mathtt{b}, \mathtt{x} : \mathtt{a}, \mathtt{y} : \mathtt{b} \vdash \mathtt{x}\ \ \mathtt{y}\ \triangleright\ \mathtt{a} \sim \mathtt{Bool}, \mathtt{b} \sim \mathtt{()}\}]\!] \\
= \{\ \ \mathtt{True}\ \ \mathtt{()}\ \ ,\ \ \mathtt{False}\ \ \mathtt{()}\ \ ,\ \ \bot\ \ \mathtt{()}\ \ , \\
\qquad \mathtt{True}\ \ \bot\ \ ,\ \ \mathtt{False}\ \ \bot\ \ ,\ \ \bot\ \ \bot\ \ \}
\end{array}
$$

5.2 Pattern Vectors

Figure 4 formalises the dynamic semantics of pattern vectors.

The basic meaning $[\![\vec{p}]\!]^{\theta}$ of a pattern vector $\vec{p}$ is a function that takes a vector of values $\vec{V}$ to a matching result M. There may be free variables in (the guards of) $\vec{p}$; the given substitution θ binds them to values. The matching result M has the form $\mathcal{T}$, $\mathcal{F}$ or $\bot$ depending on whether the match succeeds, fails or diverges.

Consider matching the pattern vector `x (True <- x > y)`, where `y` is bound to `5`, against the value `7`; this match succeeds. Formally, this is expressed thus:

$$[\![\mathtt{x\ (True\ \text{<-}\ x\ >\ y)}]\!]^{[\mathsf{y} \mapsto 5]}(7) = \mathcal{T}$$

For comparing with our algorithm, this formulation of the dynamic semantics is not ideal: the former acts on whole sets of value vectors (in the form of value abstractions) at a time, while the latter considers only one value vector at a time. To bridge this gap, $[\![\vec{p}]\!]$

$$\begin{array}{lll} \tau_c & ::= T\,\overline{\tau}_c \mid \tau_c \to \tau_c & \text{Closed Monotypes} \\ V, W & ::= K\,\vec{V} \mid \lambda x.e \mid \bot & \text{Values} \\ M & ::= \mathcal{T} \mid \mathcal{F} \mid \bot & \text{Matching Result} \\ \mathscr{S}, \mathscr{C}, \mathscr{U}, \mathscr{U} & ::= \overline{\vec{V}} & \text{Set of value vectors} \end{array}$$

Denotation of expressions

$$\boxed{E\llbracket e \rrbracket = V}$$

(definition omitted)

Denotation of value abstractions

$$\boxed{\llbracket S \rrbracket = \overline{\vec{V}}}$$

$$\llbracket S \rrbracket = \{\theta(\vec{u}) \mid (\Gamma \vdash \vec{u} \rhd \Delta) \in S,\ \theta \in \llbracket \Gamma \rrbracket,\ \models \theta(\Delta)\}$$

Denotation of typing environments

$$\boxed{\llbracket \Gamma \rrbracket = \bar{\theta}}$$

$$\begin{array}{lcl} \llbracket \epsilon \rrbracket & = & \{\epsilon\} \\ \llbracket x : \tau_c, \Gamma \rrbracket & = & \{\theta \cdot [x \mapsto V] \mid\ \vdash_v V : \tau_c, \theta \in \llbracket \Gamma \rrbracket\} \\ \llbracket a, \Gamma \rrbracket & = & \{\theta \cdot [a \mapsto \tau_c] \mid \theta \in \llbracket [a \mapsto \tau_c](\Gamma) \rrbracket\} \end{array}$$

Well-typed values

$$\boxed{\vdash_v V : \tau_c}$$

$$\frac{}{\vdash_v \bot : \tau_c}\ \textsc{Bot} \qquad \frac{x : \tau_{c,1} \vdash e : \tau_{c,2}}{\vdash_v \lambda x.e : \tau_{c,1} \to \tau_{c,2}}\ \textsc{Fun}$$

$$\frac{\begin{array}{l} K :: \forall \overline{a}\,\overline{b}.Q \Rightarrow \overline{\tau} \to T\,\overline{a} \quad \models \theta(Q) \\ \theta = [\overline{a \mapsto \tau_{c_i}}, \overline{b \mapsto \tau_{c_j}}] \quad \vdash_v V_i : \theta(\tau_i) \quad (\forall i) \end{array}}{\vdash_v K\,\vec{V} : T\,\vec{\tau}_{c_i}}\ \textsc{Con}$$

Denotation of patterns

$$\boxed{\llbracket \vec{p} \rrbracket^{\theta} :: \vec{V} \to M}$$

$$\begin{array}{ll} \llbracket \epsilon \rrbracket^{\theta}(\epsilon) & = \mathcal{T} \\ \llbracket x\,\vec{p} \rrbracket^{\theta}(V\,\vec{V}) & = \llbracket \vec{p} \rrbracket^{[x \mapsto V] \cdot \theta}(\vec{V}) \\ \llbracket (p \leftarrow e)\,\vec{p} \rrbracket^{\theta}(\vec{V}) & = \llbracket p\,\vec{p} \rrbracket^{\theta}(E\llbracket \theta(e) \rrbracket\,\vec{V}) \\ \llbracket (K_i\,\vec{p})\,\vec{q} \rrbracket^{\theta}((K_j\,\vec{V})\vec{W}) & = \begin{cases} \llbracket \vec{p}\,\vec{q} \rrbracket^{\theta}(\vec{V}\,\vec{W}) & \text{, if } K_i = K_j \\ \mathcal{F} & \text{, if } K_i \neq K_j \end{cases} \\ \llbracket (K_i\,\vec{p})\,\vec{q} \rrbracket^{\theta}(\bot\,\vec{V}) & = \bot \end{array}$$

$$\boxed{\llbracket \vec{p} \rrbracket :: \overline{\vec{V}} \to \langle \overline{\vec{V}}_c, \overline{\vec{V}}_u, \overline{\vec{V}}_\bot \rangle}$$

$$\begin{array}{lcl} \llbracket \vec{p} \rrbracket(\mathscr{S}) & = & \langle \{\vec{V} \mid \vec{V} \in \mathscr{S} \text{ where } \llbracket \vec{p} \rrbracket^{\epsilon}(\vec{V}) = \mathcal{T}\} \\ & & , \{\vec{V} \mid \vec{V} \in \mathscr{S} \text{ where } \llbracket \vec{p} \rrbracket^{\epsilon}(\vec{V}) = \mathcal{F}\} \\ & & , \{\vec{V} \mid \vec{V} \in \mathscr{S} \text{ where } \llbracket \vec{p} \rrbracket^{\epsilon}(\vec{V}) = \bot\} \rangle \end{array}$$

Figure 4: Semantics of Value Abstractions and Patterns

lifts $\llbracket \vec{p} \rrbracket^{\epsilon}$ from an individual value vector $\vec{V}$ to a whole set S of value vectors. It does so by partitioning the set based on the matching outcome, which is similar to the behaviour of our algorithm.

5.3 Correctness Theorem

Now we are ready to express the correctness of the algorithm with respect to the dynamic semantics. The algorithm is essentially an abstraction of the dynamic semantics. Rather than acting on an infinite set of values, it acts on a finite representation of that set, the value abstractions. Correctness amounts to showing that the algorithm treats the abstract set in a manner that faithfully reflects the way the dynamic semantics treats the corresponding concrete set. In other words, it should not matter whether we run the algorithm on an abstract set S and interpret the abstract result $\langle C, U, D \rangle$ as sets of concrete values $\langle \mathscr{C}, \mathscr{U}, \mathscr{D} \rangle$, or whether we first interpret the abstract set S as a set $\mathscr{S}$ of concrete values and then run the concrete dynamic semantics on those.

This can be expressed concisely as a commuting diagram:

$$\begin{tikzcd} S \arrow[r, "patVectProc(\vec{p})"] \arrow[d, "\llbracket \cdot \rrbracket"'] & \langle C, U, D \rangle \arrow[d, "\llbracket \cdot \rrbracket"] \\ \mathscr{S} \arrow[r, "\llbracket \vec{p} \rrbracket"'] & \langle \mathscr{C}, \mathscr{U}, \mathscr{D} \rangle \end{tikzcd}$$

This diagram allows us to interpret the results of the algorithm. For instance, if we choose s to cover all possible value vectors and we observe that C is empty, we can conclude that no value vector successfully matches $\vec{p}$.

To state correctness precisely we have to add the obvious formal fine print about types: The behaviour of pattern matching is only defined if:

1. the pattern vector $\vec{p}$ is well-typed,
2. the value vector $\vec{V}$ and, by extension, the value set $\mathscr{S}$ and the abstract value set S are well-typed, and
3. the types of pattern vector $\vec{p}$ and value vector $\vec{V}$ correspond.

The first condition we express concisely with the judgement $Q; \Gamma \vdash \vec{p} : \vec{\tau}$, which expresses that the pattern vector $\vec{p}$ has types $\vec{\tau}$ for a type environment Γ and given type constraints Q.

For the second condition, we first consider the set of all values value vectors compatible with types $\vec{\tau}$, type environment Γ and given type constraints Q. This set can be compactly written as the interpretation $\llbracket S^* \rrbracket$ of the value abstraction $S^* = \{\Gamma, \vec{x} : \vec{\tau} \vdash \vec{x} \rhd Q\}$. Any other well-typed value vectors $\vec{V}$ must be contained in this set: $\vec{V} \in \llbracket S^* \rrbracket$. Similarly, $\mathscr{S} \subseteq \llbracket S^* \rrbracket$ and $\llbracket S \rrbracket \subseteq \llbracket S^* \rrbracket$

Finally, the third condition is implicitly satisfied by using the same types $\vec{\tau}$, type environment Γ and given type constraints Q.

Wrapping up we formally state the correctness theorem as follows:

Theorem 1 (Correctness)**.**

$$\begin{array}{r} \forall \Gamma, Q, \vec{p}, \vec{\tau}, S :\ \ Q; \Gamma \vdash \vec{p} : \vec{\tau}\ \wedge\ \llbracket S \rrbracket \subseteq \llbracket \{\Gamma, \vec{x} : \vec{\tau} \vdash \vec{x} \rhd Q\} \rrbracket \\ \Longrightarrow\ \ \llbracket patVectProc(\vec{p}, S) \rrbracket = \llbracket \vec{p} \rrbracket \llbracket S \rrbracket \end{array}$$

6. Implementation

This section describes the current implementation of our algorithm in GHC and possible improvements.

The pattern-match warning pass runs once type inference is complete. At this stage the syntax tree is richly decorated with type information, but has not yet been desugared. Warnings will therefore refer to the program text written by the user, and not so some radically-desugared version. Actually the pattern-match warning generator is simply called by the desugarer, just before it desugars each pattern match.

The new pattern match checker takes 504 lines of Haskell, compared to 588 lines for the old one. So although the new checker is far more capable, it is of comparable code size.

6.1 The Oracle

The oracle judgement $\vdash_{\text{SAT}}$ is treated as a black box by the algorithm. As long as it is conservative, any definition will do, even accepting all constraints. Our implementation does quite a bit better than that.

Type-level constraints For type constraints we simply re-use the powerful type-constraint solver, which GHC uses for type inference [25]. Hence, inconsistency of type constraints is defined uniformly and our oracle adapts automatically to any changes in the type system, such as type-level functions, type-level arithmetic, and so on.

Term-level constraints Currently, our oracle implementation for term-level constraints is vestigial. It is specialised for trivial guards of the form `True` and knows that these cannot fail. Thus only conjunctions of constraints of the form $y \approx \texttt{True}, y \approx \texttt{False}$ are flagged as inconsistent. This enables us to see that `abs1` (Section 2.4) is exhaustive, but not `abs2`. There is therefore plenty of scope for improvement, and various powerful term-level solvers, such as Zeno [29] and HipSpec [5], could be used to serve the oracle.

6.2 Performance Improvements

We have optimised the presentation of our algorithm in Section 4 for clarity, rather than runtime performance. Even though we cannot improve upon the asymptotic worst-case time complexity, various measures can improve the average performance a big deal.

Implicit solving The formulation of the algorithm in Section 4 generates type constraints for the oracle with a high frequency. For instance, rule [CConVar] of the $\mathcal{C}$ function generates a new type equality constraint $\tau \sim \tau_x$ every time it fires, even for Haskell'98 data types.

While there are good reasons for generating these constraints in general, we can in many cases avoid generating them explicitly and passing them on to the oracle. Instead, we can handle them immediately and much more cheaply. One important such case is covered by the specialised variant of rule [CConVar] in Figure 5: the type τ_x has the form $T\ \vec{\tau}_x$, where T is also the type constructor of the constructor K_i. This means that the generated type constraint $\tau \sim \tau_x$ actually has the form $T\ \vec{a} \sim T\ \vec{\tau}_x$. We can simplify this constraint in two steps. Firstly, we can decompose it into simpler type equality constraints $a_i \sim \tau_{x,i}$, one for each of the type parameters. Secondly, since all type variables $\vec{a}$ are actually fresh, we can immediately *solve* these constraints by substituting all occurrences of $\vec{a}$ by $\vec{\tau}_x$. Rule [CConVar] incorporates this simplification and does not generate any type constraints at all for Haskell'98 data types.

Incremental solving Many constraint solvers, including the OutsideIn(X) solver, support an incremental interface:

```
solve :: Constraint -> State -> Maybe State
```

In the process of checking given constraints C_0 for satisfiability, they also normalise them into a compact representation. When the solver believes the constraints are satisfiable, it returns their normal form: a *state* σ_0. When later the conjunction $C_0 \wedge C_1$ needs to be checked, we can instead pass the state σ_0 together with C_1 to the solver. Because σ_0 has already been normalised, the solver can process the latter combination much more cheaply than the former.

It is very attractive for our algorithm to incorporate this incremental approach, replace the constraints Δ by normalised solver states σ and immediately solve new constraints when they are generated. Because the algorithm refines step by step one initial value abstraction into many different ones, most value abstractions share a common prefix of constraints. By using solver states for these common prefixes, we share the solving effort among all refinements and greatly save on solving time. Moreover, by finding inconsistencies early, we can prune eagerly and avoid refining in the first place.

7. Evaluation

Our new pattern checker addresses the three challenges laid out in Section 2: GADTs, laziness, and guards. However in our evaluation, only the first turned out to be significant. Concerning laziness, none of our test programs triggered the warning for a clause that is irredundant, but has an inaccessible right hand side; clearly such cases are rare! Concerning guards, our prototype implementation only has a vestigial term-equality solver, so until we improve it we cannot expect to see gains.

For GADT-rich programs, however, we do hope to see improvements. However, many programs do not use GADTs at all; and those that do often need to match over all constructors of the type anyway. So we sought test cases by asking the Haskell `libraries` list for cases where the authors missed accurate warnings for GADT-using programs. This has resulted in identifying 9 hackage packages and 3 additional libraries, available on GitHub.[2]

We compared three checkers. The baseline is, of course, vanilla GHC. However, GHC already embodies an *ad hoc* hack to improve warning reports for GADTs, so we ran GHC two ways: both with (GHC-2) and without (GHC-1) the hack. Doing so gives a sense of how effective the *ad hoc* approach was compared with our new checker.

For each compiler we measured:

- *The number of missing clauses (M).* The baseline compiler GHC-1 is conservative, and reports too many missing clauses; so a lower M represents more accurate reporting.
- *The number of redundant (R) clauses.* The baseline compiler is conservative, and reports too few redundant clauses; so a higher R represents more accurate reporting.

The results are presented in Table 1. They clearly show that the ad-hoc hack of GHC-2 was quite succesful at eliminating unnecessary missing pattern warnings, but is entirely unable to identify redundant clauses. The latter is where our algorithm shines: it identifies 38 pattern matches with redundant clauses, all of them catch-all cases added to suppress erroneous warnings. We also see a good reduction (-27) of the unnecessary missing pattern warnings. The remaining spurious missing pattern warnings in *accelerate* and *d-bus* involve pattern guards and view patterns; these can be eliminated by upgrading the term-level reasoning of the oracle.

Erroneous suppression of warnings We have found three cases where the programmer has erroneously added clauses to suppress warnings. We have paraphrased one such example in terms of the `Vect n a` type of Section 1.

```
data EQ n m where
  EQ :: n ~ m => EQ n m

eq :: Vect n a -> Vect m a -> EQ n m -> Bool
eq  VN        VN        EQ = True
eq (VC x xs) (VC y ys) EQ = x == y && eq xs ys
eq VN        (VC _ _)  _  = error "redundant"
eq (VC _ _)  VN        _  = error "redundant"
```

This example uses the `EQ n m` type as a witness for the type-level equality of `n` and `m`. This equality is exposed by pattern matching on

[2] https://github.com/amosr/merges/blob/master/stash/Lists.hs
https://github.com/gkaracha/gadtpm-example
https://github.com/jstolarek/dep-typed-wbl-heaps-hs

$$\begin{array}{llll}
[\textsc{CConVar'}] & \mathcal{C} \quad ((K_i\,\vec{p})\,\vec{q}) \quad (\Gamma \vdash x\,\vec{u} \rhd \Delta) & = & \mathcal{C}\,((K_i\,\vec{p})\,\vec{q})\,(\Gamma' \vdash (K_i\,\vec{y})\,\vec{u} \rhd \Delta') \\
 & & & \text{where} \quad \vec{y}\#\Gamma \quad \vec{b}\#\Gamma \quad (x{:}T\,\vec{\tau}_x) \in \Gamma \quad K_i :: \forall \vec{a}, \vec{b}.Q \Rightarrow \vec{\tau} \rightarrow T\,\vec{a} \\
 & & & \qquad\quad \theta = [\vec{a} \mapsto \vec{\tau}_x] \quad \Gamma' = \Gamma, \vec{b}, \vec{y}{:}\theta(\vec{\tau}) \\
 & & & \qquad\quad \Delta' = \Delta \cup \theta(Q) \cup x \approx K_i\,\vec{y}
\end{array}$$

Figure 5: Specialised Clause Processing

Table 1: Results

		GHC-1		**GHC-2**		**New**	
Hackage Packages	**LoC**	**M**	**R**	**M**	**R**	**M**	**R**
accelerate	$11,393$	11	0	9	0	8	14
ad	$1,903$	2	0	0	0	0	6
boolsimplifier	256	10	0	0	0	0	0
d-bus	$2,753$	45	0	42	0	16	1
generics-sop	$1,008$	0	0	0	0	0	3
hoopl	$2,147$	33	0	0	0	0	3
json-sop	393	0	0	0	0	0	2
lens-sop	280	2	0	0	0	0	2
pretty-sop	27	0	0	0	0	0	1
Additional tests	**LoC**	**M**	**R**	**M**	**R**	**M**	**R**
lists	66	1	0	0	0	0	3
heterogeneous lists	38	0	0	0	0	0	2
heaps	540	3	0	0	0	0	1

`EQ`. Hence, the third and fourth clauses must be redundant. After all, we cannot possibly have an equality witness for `Zero ~ Succ n`. Yes, we can: that witness is $\bot$ :: `EQ Zero (Succ n)` and it is not ruled out by the previous clauses. Indeed, calls of the form `eq` `VN` (`VC` $x\,xs$) $\bot$ and `eq` (`VC` $x\,xs$) `VN` $\bot$ are not covered by the first two clauses and hence rightly reported missing. The bottoms can be flushed out by moving the equality witness to the front of the argument list and matching on it first. Then the first two clauses suffice.

GHC tickets With our new algorithm we have also been able to close nine GHC tickets related to GADT pattern matching (#3927, #4139, #6124, #8970) and literal patterns (#322, #2204, #5724, #8016, #8853).

8. Related Work

8.1 Compiling Pattern Matching

There is a large body of work concerned with the *efficient compilation* of pattern matching, for strict and lazy languages [13, 15, 17, 18]. Although superficially related, these works focus on an entirely different problem, one that simply does not arise for us. Consider

```
f True  True  = 1
f _     False = 2
f False True  = 3
```

In a strict language one can choose whether to begin by matching the first argument or the second; the choice affects only efficiency, not semantics. In a lazy language the choice affects semantics; for example, does `f` ($\bot$, `False`) diverge, or return `2`? Laville and Maranget suggest choosing a match order that makes `f` maximally defined [15], and they explore the ramifications of this choice.

However, Haskell does not offer this degree of freedom; it fixes a top-to-bottom and left-to-right order of evaluation in pattern match clauses.

8.2 Warnings for Simple Patterns

We now turn our attention to *generating warnings* for inexhaustive or redundant patterns. For simple patterns (no guards, no GADTs) there are several related works. The most closely-related is Maranget's elegant algorithm for detecting missing and redundant (or "useless") clauses [16]. Maranget recursively defines a predicate that determines whether there could be any vector of values v that matches pattern vector $\vec{p}$, without matching any pattern vector row in a matrix P, $U_{req}(P, \vec{p})$, and answers both questions of exhaustiveness (query $\mathcal{U}_{req}(P, \_)$) and redundancy (query $\mathcal{U}_{req}(P^{1..(j-1)}, \vec{p}_j)$ where $P^{1..(j-1)}$ corresponds to all previous clauses). Our algorithm has many similarities (e.g. in the way it expands constructor patterns) but is more incremental by propagating state from one clause to the next instead of examining all previous clauses for each clause.

Maranget's algorithm does not deal with type constraints (as those arising from GADTs), nor guards and nested patterns that require keeping track of Δ and environment Γ. Finally the subtle case of an empty covered set but a non-empty divergent set would not be treated specially (and the clause would be considered as non-redundant, though it could only allow values causing divergence).

Krishnaswami [12] accounts for exhaustiveness and redundancy checking as part of formalisation of pattern matching in terms of the focused sequent calculus. His approach assumes a left-to-right ordering in the translation of ML patterns, which is compatible with Haskell's semantics.

Sestoft [27] focuses on compiling pattern matches for a simply-typed variant of ML, but his algorithm also identifies inexhaustive matches and redundant match rules as a by-product.

8.3 Warnings for GADT Patterns

OCaml and Idris both support GADTs, and both provide some GADT-aware support for pattern-match checking. No published work describes the algorithm used in these implementations.

OCaml When Garrigue and Le Normand introduced GADTs to the OCaml language [10], they also extended the checking algorithm. It eliminates the ill-typed uncovered cases proposed by OCaml's original algorithm. However, their approach does not identify clauses that are redundant due to unsatisfiable type constraints. For instance, the third clause in `f` below is not identified as redundant.

```
type _ t = T1 : int t | T2 : bool t

let f (type a) (x: a t) (y: a s) : unit =
  match (x, y) with
   | (T1, T1) -> ()
   | (T2, T2) -> ()
   | (_,  _)  -> ()
```

Idris Idris [2, 3] has very limited checking of overlapping patterns or redundant patterns.[3] It does, however, check coverage, and will use this information in optimisation and code generation.

[3] Edwin Brady, personal communication

ML variants Xi. [34–36] shows how to eliminate dead code for GADT pattern matching – and dependent pattern matching in general – for Dependent ML. He has a two-step approach: first add all the missing patterns using simple-pattern techniques (Section 8.2), and then prune out redundant clauses by checking when typing constraints are un-satisfiable. We combine the two steps, but the satisfiability checking is similar.

Dunfield's thesis [7, Chapter 4] presents a coverage checker for Stardust [8], another ML variant with refinement and intersection types. The checker proceeds in a top-down, left-to-right fashion much like Figure 1 and uses type satisfiability to prune redundant cases.

Neither of these works handles guards or laziness.

8.4 Total Languages

Total languages like Agda [22] and Coq [19] must treat non-exhaustive pattern matches as an *error* (not a warning). Moreover, they also allow overlapping patterns and use a variation of Coquand's dependent pattern matching [6] to report redundant clauses. The algorithm works by splitting the context, until the current neighbourhood matches one of the original clauses. If the current neighbourhood fails to match all the given clauses, the pattern match is non-exhaustive and a coverage failure error is issued. If matching is inconclusive though, the algorithm splits along one of the blocking variables and proceeds recursively with the resulting neighbourhoods. Finally, the `with`-construct [22], first introduced by McBride and McKinna [20], provides (pattern) guards in a form that is suitable for total languages.

The key differences between our work and work on dependent pattern matching are the following: (i) because of the possibility of divergence we have to take laziness into account; (ii) current presentations of `with`-clauses [20] do not introduce term-level equality propositions and hence may report inexhaustiveness checking more often than necessary, (iii) our approach is easily amenable to external decision procedures that are proven sound but do not have to return proof witnesses in the proof theory in hand.

8.5 Verification Tools

ESC/Haskell. A completely different but more powerful approach can be found in *ESC/Haskell* [38] and its successor [39]. ESC/Haskell is based on preconditions and contracts, so, it is able to detect far more defects in programs: pattern matching failures, division by zero, out of bounds array indexing, etc. Although it is far more expressive than our approach (e.g. it can verify even some sorting algorithms), it requires additional work by the programmer through explicit pre/post-conditions.

Catch. Another approach that is closer to our work but retains some of the expressiveness of ESC/Haskell is the tool *Catch* [21] Catch generates pre- and post-conditions that describe the sets of incoming and returned values of functions (quite similarly to our value abstraction sets). Catch is based on abstract interpretation over Haskell terms – the scope of abstract interpretation in our case is restricted to clauses (and potentially nested patterns). A difference is that Catch operates at the level of Haskell Core, GHC's intermediate language [40]. The greatest advantage of this approach is that this language has only 10 data constructors, and hence Catch does not have to handle the more verbose source Haskell AST. Unfortunately, at the level of Core, the original syntax is lost, leading to less comprehensive error messages. On top of that, Catch does not take into account type constraints, such as those that arise from GADT pattern matching. Our approach takes them into account and reuses the existing constraint solver infrastructure to discharge them.

Liquid types. Liquid types [24, 31] is a refinement types extension to Haskell. Similarly to ESC/Haskell, it could be used to detect redundant, overlapping, or non-exhaustive patterns, using an SMT-based version of Coquand's algorithm [6]. To take account of type-level constraints (such as type equalities from GADTs) one would have to encode them as refinement predicates. The algorithm that we propose for computing covered, uncovered, and diverging sets would still be applicable, but would have to emit constraints in the vocabulary of Liquid types.

9. Discussion and Further Work

We presented an algorithm that provides warnings for functions with redundant or missing patterns. These warnings are accurate, even in the presence of GADTs, laziness and guards. Our implementation is already available in the GHC repository (branch `wip/gadtpm`). Given its power, the algorithm is both modular and simple: Figure 3 is really the whole thing, apart from the satisfiability checker. It provides interesting opportunities for follow-on work, such as smarter reasoning about term-level constraints, and exploiting the analysis results for optimised compilation.

Acknowledgments

We are grateful to Edwin Brady for explaining Idris' behaviour, and to Jacques Garrigue and Jacques Le Normand for explaining OCaml's behaviour. We would also like to thank Nikolaos Papaspyrou for his contribution in the early stages of this work; and Gabor Greif, Conor McBride, and the ICFP referees for their helpful feedback. This work was partially funded by the Flemish Fund for Scientific Research (FWO).

References

[1] L. Augustsson. Compiling pattern matching. In *Proceedings of the 1985 Conference on Functional Programming and Computer Architecture*, 1985.

[2] E. Brady. Programming and reasoning with algebraic effects and dependent types. In *Proceedings of the 18th ACM SIGPLAN International Conference on Functional Programming*, ICFP '13, pages 133–144, New York, NY, USA, 2013. ACM. .

[3] E. Brady. Idris, a general-purpose dependently typed programming language: Design and implementation. *Journal of Functional Programming*, 23:552–593, 9 2013. .

[4] J. Cheney and R. Hinze. First-class phantom types. Technical report, Cornell University, 2003.

[5] K. Claessen, M. Johansson, D. Rosén, and N. Smallbone. Automating inductive proofs using theory exploration. In M. P. Bonacina, editor, *CADE*, volume 7898 of *Lecture Notes in Computer Science*, pages 392–406. Springer, 2013.

[6] T. Coquand. Pattern matching with dependent types. In *Proceedings of the Workshop on Types for Proofs and Programs*, 1992.

[7] J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Aug. 2007. CMU-CS-07-129.

[8] J. Dunfield. Refined typechecking with Stardust. In *Proceedings of the 2007 Workshop on Programming Languages Meets Program Verification*, PLPV '07, pages 21–32, New York, NY, USA, 2007. ACM. .

[9] M. Erwig and S. Peyton Jones. Pattern guards and transformational patterns. In *Proceedings of the 2000 Haskell Symposium*. ACM, 2000.

[10] J. Garrigue and J. L. Normand. Adding GADTs to OCaml: the direct approach. In *Workshop on ML*, 2011.

[11] G. Karachalias, T. Schrijvers, D. Vytiniotis, and S. P. Jones. GADTs meet their match (extended version). Technical report, KU Leuven, 2015. URL `http://people.cs.kuleuven.be/~george.karachalias/papers/gadtpm_ext.pdf`.

[12] N. R. Krishnaswami. Focusing on pattern matching. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 366–378, New York, NY, USA, 2009. ACM. .

[13] A. Laville. Comparison of priority rules in pattern matching and term rewriting. *J. Symb. Comput.*, 11(4):321–347, May 1991. .

[14] F. Le Fessant and L. Maranget. Optimizing pattern-matching. In *Proceedings of the 2001 International Conference on Functional Programming*, 2001.

[15] L. Maranget. Compiling lazy pattern matching. In *Proceedings of the 1992 ACM Conference on LISP and Functional Programming*, LFP '92, pages 21–31, New York, NY, USA, 1992. ACM. .

[16] L. Maranget. Warnings for pattern matching. *Journal of Functional Programming*, 17:387–421, 2007.

[17] L. Maranget. Compiling pattern matching to good decision trees. In *Proceedings of the ACM Workshop on ML*, 2008.

[18] L. Maranget and P. Para. Two techniques for compiling lazy pattern matching. Technical report, 1994.

[19] The Coq development team. *The Coq proof assistant reference manual*. LogiCal Project, 2004. URL `http://coq.inria.fr`. Version 8.0.

[20] C. McBride and J. McKinna. The view from the left. *Journal of Functional Programming*, 14(1):69–111, 2004.

[21] N. Mitchell and C. Runciman. Not all patterns, but enough: An automatic verifier for partial but sufficient pattern matching. In *Proceedings of the First ACM SIGPLAN Symposium on Haskell*, Haskell '08, pages 49–60, New York, NY, USA, 2008. ACM. .

[22] U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Department of Computer Science and Engineering, Chalmers University of Technology, SE-412 96 Göteborg, Sweden, September 2007.

[23] S. Peyton Jones, D. Vytiniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In *Proceedings of the Eleventh ACM SIGPLAN International Conference on Functional Programming*, ICFP '06, pages 50–61, New York, NY, USA, 2006. ACM. .

[24] P. M. Rondon, M. Kawaguci, and R. Jhala. Liquid types. In *Proceedings of the 2008 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 159–169, New York, NY, USA, 2008. ACM. .

[25] T. Schrijvers, S. Peyton Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *Proceedings of the 14th ACM SIGPLAN International Conference on Functional Programming*, ICFP '09, pages 341–352, New York, NY, USA, 2009. ACM. .

[26] R. C. Sekar, R. Ramesh, and I. V. Ramakrishnan. Adaptive pattern matching. *SIAM J. Comput.*, 24(6):1207–1234, Dec. 1995. ISSN 0097-5397. .

[27] P. Sestoft. ML pattern match compilation and partial evaluation. In O. Danvy, R. Glück, and P. Thiemann, editors, *Partial Evaluation*, volume 1110 of *Lecture Notes in Computer Science*, pages 446–464. Springer Berlin Heidelberg, 1996. .

[28] T. Sheard. Languages of the future. In *In OOPSLA '04: Companion to the 19th annual ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications*, pages 116–119. ACM Press, 2004.

[29] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. pages 407–421. Springer-Verlag Berlin, 2012. .

[30] P. Thiemann. Avoiding repeated tests in pattern matching. In G. Filé, editor, *3rd International Workshop on Static Analysis*, number 724, pages 141–152, Padova, Italia, Sept. 1993.

[31] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for Haskell. In *Proceedings of the 19th ACM SIGPLAN International Conference on Functional Programming*, ICFP '14, pages 269–282, New York, NY, USA, 2014. ACM. .

[32] P. Wadler. Views: A way for pattern matching to cohabit with data abstraction. In *Proceedings of the 14th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '87, pages 307–313, New York, NY, USA, 1987. ACM. .

[33] P. Wadler. Efficient compilation of pattern matching. In S. Peyton Jones, editor, *The implementation of functional programming languages*, pages 78–103. Prentice Hall, 1987.

[34] H. Xi. Dead code elimination through dependent types. In *Proceedings of the First International Workshop on Practical Aspects of Declarative Languages*, PADL '99, pages 228–242, London, UK, 1998. Springer-Verlag.

[35] H. Xi. *Dependent Types in Practical Programming*. PhD thesis, Carnegie Mellon University, Sept. 1998.

[36] H. Xi. Dependently typed pattern matching. *Journal of Universal Computer Science*, 9:851–872, 2003.

[37] H. Xi, C. Chen, and G. Chen. Guarded recursive datatype constructors. In *Proceedings of the 30th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '03, pages 224–235, New York, NY, USA, 2003. ACM. .

[38] D. N. Xu. Extended static checking for Haskell. In *Proceedings of the 2006 ACM SIGPLAN Workshop on Haskell*, Haskell '06, pages 48–59, New York, NY, USA, 2006. ACM. .

[39] D. N. Xu, S. Peyton Jones, and K. Claessen. Static contract checking for Haskell. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '09, pages 41–52, New York, NY, USA, 2009. ACM. .

[40] B. A. Yorgey, S. Weirich, J. Cretin, S. Peyton Jones, D. Vytiniotis, and J. P. Magalhães. Giving Haskell a promotion. In *Proceedings of the 8th ACM SIGPLAN Workshop on Types in Language Design and Implementation*, TLDI '12, pages 53–66, New York, NY, USA, 2012. ACM. .

A. Set Size Statistics

As we discussed in Section 3.4, our algorithm has exponential behaviour in the worst case. Nevertheless, we expect this behaviour to be rare in practice. To confirm this expectation, we put our implementation to the test by collecting statistics concerning the size of sets C and U our algorithm generates for the packages of Section 7:

Maximum size of C/U	Pattern Matches	(%)
$1-9$	8702	97.90%
$10-99$	181	2.04%
$100-2813$	5	0.06%

Since there was significant variance in the results, we divided them into three size groups. Out of 8888 pattern matches checked in total, almost 98% of the generated and processed sets have a size less than 10. In fact, the vast majority (over 95%) have size 1 or 2.

The percentage of sets with size between 10 and 99 is 2.04%. We believe that this percentage is acceptable for types with many constructors and for pattern matches with many arguments.

Last but not least, we encountered 5 cases (only 0.06%) with extremely large sets ($\geq$ 100 elements). All of them were found in a specific library[4] of package *ad*. As expected, all these involved pattern matches had the structure of function `f` from Section 3.4:

```
data T = A | B | C
f A A = True
f B B = True
f C C = True
```

Notably, the most extreme example which generated an uncovered set of size 2813, matches against two arguments of type `T` with 54 data constructors, a match that gives rise to 3025 different value combinations!

[4] Library Data.Array.Accelerate.Analysis.Match.