Heuristics-based Type Error Diagnosis for Haskell

The case of GADTs and local reasoning

Joris Burgers
Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
jorisburgers@live.nl

Jurriaan Hage
Dept. of Information and Computing Sciences
Utrecht University
The Netherlands
j.hage@uu.nl

Alejandro Serrano
47 Degrees/Utrecht University
The Netherlands
a.serranomena@uu.nl

ABSTRACT

Helium is a Haskell compiler designed to provide programmer friendly type error messages. It employs specially designed heuristics that work on a type graph representation of the type inference process.

In order to support existentials and Generalized Algebraic Data Types (GADTs) in Helium, we extend the type graphs of Helium with facilities for local reasoning. We have translated the original Helium heuristics to this new setting, and define a number of GADT-specific heuristics that help diagnose Helium programs that employ GADTs.

CCS CONCEPTS

• **Theory of computation** → **Type structures**; **Program analysis**; • **Software and its engineering** → *Functional languages*; *Abstract data types.*

KEYWORDS

type error diagnosis, generalized algebraic data types, type graphs, Haskell

ACM Reference Format:
Joris Burgers, Jurriaan Hage, and Alejandro Serrano. 2020. Heuristics-based Type Error Diagnosis for Haskell: The case of GADTs and local reasoning. In *IFL 2020: Proceedings of the 32nd Symposium on Implementation and Application of Functional Languages (IFL '20), September 2–4, 2020, Canterbury, United Kingdom.* ACM, New York, NY, USA, 11 pages. https://doi.org/10.1145/3462172.3462189

1 INTRODUCTION

Haskell has always been a hotbed of language and type system innovation, contributing to the popularization of many such features. The advantage of a rich type system is that the programmer can obtain many guarantees about the correctness of an implementation without having to resort to testing. But advanced type system features come at a price. One price is that when type inconsistencies arise, it is noticeably harder for the compiler to explain to the programmer what the inconsistency is, where it arises, how it might be resolved, all without revealing internal implementation details of the compiler. This hinders the uptake of these advanced features, leading to programmers avoiding them, and settling for fewer guarantees.

One such language feature is that of Generalized Algebraic Datatypes (GADTs for short), that allows the programmer to encode type information in the data type constructors of an algebraic data type. It is a popular feature of Haskell, in particular for encoding the type system of a deeply embedded domain-specific languages. A simple but typical example is:

```
data Expr a where
  LitInt  :: Int  → Expr Int
  LitBool :: Bool → Expr Bool
  Equals  :: Eq a ⇒ Expr a → Expr a → Expr Bool
```

where the *Equals* constructor encodes that it can only compare the equality of two subexpressions that have the same type *a*, that moreover is an instance of the *Eq* type class. The type inferencer will then forbid expressions such as *Equals* (*LitBool True*) (*LitInt* 1), because the arguments to *Equals* do not agree on the choice for *a*. Typical for GADTs, as compared to ordinary ADTs, is that the type variable *a* does not show up in the result of *Equals*, making it an *existential variable*.

Now, if we type check the following function (note that we have omitted the type signature),

```
lit (LitInt  x) = x
lit (LitBool x) = x
```

then GHC, the standard Haskell compiler, returns the type error message

```
* Couldn't match expected type 'p' with actual type 'Int'
    'p' is untouchable
      inside the constraints: a ~ Int
      bound by a pattern with constructor: LitInt :: Int -> Expr Int,
               in an equation for 'lit'
      at <interactive>:18:6-13
  'p' is a rigid type variable bound by
    the inferred type of lit :: Expr a -> p
    at <interactive>:(18,1)-(19,19)
  Possible fix: add a type signature for 'lit'
* In the expression: x
  In an equation for 'lit': lit (LitInt x) = x
* Relevant bindings include
    lit :: Expr a -> p (bound at <interactive>:18:1)
```

What is wrong with this message? First of all, the message introduces type variables such as *p* that are not part of the input program. It uses terminology, e.g., ~, rigid and untouchable, that are involved in the type inference process but of which the programmer should not be aware [27], and it provides an inferred type for *lit*, namely

IFL '20, September 2–4, 2020, Canterbury, United Kingdom

ACM ISBN 978-1-4503-8963-1/20/09…$15.00
https://doi.org/10.1145/3462172.3462189

Expr a → p, which is in fact not correct. Moreover, it produces a very similar message for the other branch of *lit*!

The `thesis_outsidein` branch of the Helium compiler (`https://github.com/Helium4Haskell/helium.git`) contains our implementation that instead returns the following message in which it reports that the problem is that a type signature is missing, and moreover it produces a type signature for *lit* as a hint which is consistent with the rest of the code:

```
(6,1), (7,1): A type signature is necessary for this definition
 function : lit
 hint     : add a valid type signature, e.g. (X a) -> a
```

In Section 2 we reiterate material on constraint-based type inferencing and GADTs. Our contribution starts in Section 3 where we discuss how to extend the type graphs of Helium to Rhodium graphs in order to cope with GADTs and local reasoning, mirroring the behavior of the OutsideIn(X) system, the basis of the type inference process of GHC. In Section 4, we discuss heuristics that work on Rhodium graphs to recognise GADT-related error patterns. Related work is discussed in Section 5. Additional technical details and examples can be found in [1].

2 CONSTRAINT-BASED TYPE INFERENCE

The earliest implementations of type inference for functional languages use a *direct* approach in which type inference is implemented by traversing the Abstract Syntax Tree (AST) and performing unifications on the fly, e.g., the classic $\mathcal{W}$ and $\mathcal{M}$ implementations of the Hindley-Milner type system [4, 10].

Later approaches often prefer a *constraint-based* approach, divided into two phases. In the first phase, the AST is traversed to gather *constraints* which must be satisfiable for the program to be well-typed. A dedicated *solver* then takes these constraints as input, checks their validity and returns types found for the inferred elements of the program. Pottier and Rémy [15] is the standard reference; many compilers have followed their lead.

Direct approaches to type inference usually have a *bias* with respect to type error reporting, due to the fixed order in which they traverse the AST. For example, if we are checking the expression $True \equiv {'a'}$ and we traverse arguments from left to right, the error will be found in the second argument. For that reason, constraint-based approaches are often the preferred approach for type error diagnosis: we can more easily solve constraints in different orders, and it is easy to consider alternative modified sets of constraints to figure out the best explanation for an error [6, 7, 18]. Given that the GHC dialect of Haskell has a constraint-based specification, constraint-based type inference is a natural choice [25].

In the remainder of this section we give a high-level overview of constraint-based type inference. We describe type checking for the λ-calculus with pattern matching defined in Figure 1. Our presentation is heavily influenced by OutsideIn(X) [25]; we omit some details for the sake of conciseness. In particular, the described λ-calculus does not have a **let** construct for local bindings, but of course our implementation does.

As usual in Hindley-Damas-Milner-based type systems, the types of variables and data constructors in an environment Γ may quantify over some variables, and thus are assigned a *type scheme*. In addition to quantified variables, type schemes may also request some *constraints* to hold at each use of the corresponding variables. The syntax of constraints is left open by the framework – hence the X in OutsideIn(X) –, we only require X to have a notion of equality between types, $\tau_1 \sim \tau_2$. In the case of GHC, X includes the theory of type classes and type families, so we can form type schemes such as $\forall a.Eq\ a \Rightarrow a \to a \to Bool$.

The constraint gathering judgement takes the form

$$\Gamma \vdash e : \tau \rightsquigarrow Q\ ,$$

which reads: in the environment Γ the expression e has type τ under the set of constraints Q. During constraint gathering some of the types are still unknown, so we introduce *unification variables* α to represent them. Finding the types these unification variables stands for, corresponds to the inference part of the solver. The rules for the judgment, given in Figure 2, are unsurprising. In the VAR rule the rigid type variables quantified in a type scheme are *instantiated*, that is, replaced with fresh unification variables. Pattern matching is described by the CASE rule: we need to find both the particular instantiation of the type constructor $\mathsf{F}\,\overline{\gamma}$ used by the scrutinee e, and the common return type β of all the branches.

The next step of the process is constraint *solving*, which is formulated as a rewriting relation on constraints [22, 25], turning the original constraints into a simpler *solved* set of constraints. For reasons of space we provide two example rules:

$$\mathsf{F}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{F}\ \rho_1\ \ldots\ \rho_n \rightsquigarrow \tau_1 \sim \rho_1 \wedge \cdots \wedge \tau_n \sim \rho_n$$
$$\mathsf{F}\ \tau_1\ \ldots\ \tau_n \sim \mathsf{G}\ \rho_1\ \ldots\ \rho_m \rightsquigarrow \bot, \quad \text{if } \mathsf{F} \not\equiv \mathsf{G}$$

The former rule shows how an equality check between two type constructors is decomposed (if they have the same name and the same number of arguments), while the latter shows that if the heads do not match, then a type error results (modeled by rewriting to $\bot$). In Section 2.1, we shall refine $\bot$ to capture some additional information.

2.1 Type graphs

If the constraint solver, applying the rules of the rewrite relation, terminates without finding any inconsistencies among the gathered constraints, the compiler pipeline continues with further analyses and optimizations, to eventually reach code generation. If an inconsistency is detected, we should explain the problem to the programmer by means of a type error message. We aim to make this message as informative as possible, and at the same time as concise as possible to prevent the programmer from being overwhelmed. In that case, we would like to know what are the *original* constraints which led to the problem; we can then link those constraints to the

Rigid type variables		$\ni$	$a, b, \ldots$
Type constructors		$\ni$	$\mathsf{F}, \mathsf{G}, \ldots$
Monotypes	τ, ρ	$::=$	$a \mid \mathsf{F}\,\overline{\tau}$
Constraints	Q	$::=$	$\top \mid Q_1 \wedge Q_2 \mid \tau_1 \sim \tau_2 \mid \ldots$
Type schemes	σ	$::=$	$\forall a.Q \Rightarrow \tau$
Term variables		$\ni$	$x, y, \ldots$
Data constructors		$\ni$	$K, \ldots$
Expressions	e	$::=$	$x \mid K \mid \lambda x \to e \mid e_1\ e_2$
			case e **of** $\overline{K\,\overline{x} \to e}$

Figure 1: Syntactic categories of λ-calculus with pattern matching

$$\begin{array}{llll} \text{Unification variables} & \ni & \alpha, \beta, \ldots & \text{Type variables} \quad v, \omega \quad ::= \quad a \mid \alpha \\ \text{Monotypes} & \tau & ::= \quad v \mid \ldots & \text{Environments} \quad \Gamma \quad ::= \quad \epsilon \mid \Gamma, x : \sigma \end{array}$$

$$\frac{x : \forall \overline{a}.Q \Rightarrow \tau \in \Gamma \qquad \overline{\alpha} \text{ fresh}}{\Gamma \vdash x : [\overline{a \mapsto \alpha}]\tau \rightsquigarrow [\overline{a \mapsto \alpha}]Q} \text{ VAR} \qquad \frac{\alpha \text{ fresh} \qquad \Gamma, x : \alpha \vdash e : \tau \rightsquigarrow Q}{\Gamma \vdash \lambda x \to e : \alpha \to \tau \rightsquigarrow Q} \text{ ABS}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 \rightsquigarrow Q_1 \qquad \Gamma \vdash e_2 : \tau_2 \rightsquigarrow Q_2 \qquad \alpha \text{ fresh}}{\Gamma \vdash e_1\, e_2 : \alpha \rightsquigarrow \tau_1 \sim (\tau_2 \to \alpha) \wedge Q_1 \wedge Q_2} \text{ APP}$$

$$\frac{\Gamma \vdash e : \tau_0 \rightsquigarrow Q_0 \qquad \beta, \overline{\gamma} \text{ fresh} \qquad K_i : \forall \overline{a}.\, \overline{\rho_i} \to \mathsf{F}\, \overline{a} \in \Gamma \qquad \Gamma, \overline{x_i : [\overline{a \mapsto \gamma}]\rho_i} \vdash e_i : \tau_i \rightsquigarrow Q_i}{\Gamma \vdash \textbf{case } e \textbf{ of } \overline{K_i\, \overline{x}_i \to e_i} : \beta \rightsquigarrow Q_0 \wedge \tau_0 \sim \mathsf{F}\, \overline{\gamma} \wedge \overline{Q_i \wedge \tau_i \sim \beta}} \text{ CASE}$$

Figure 2: Constraint gathering for λ-calculus with pattern matching

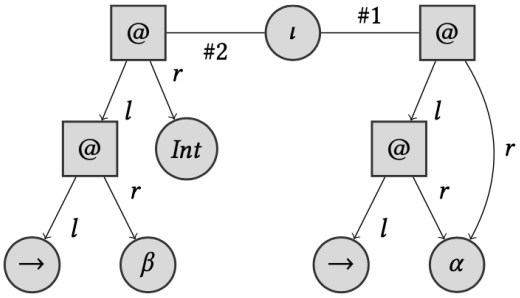

Figure 3: With type applications

program positions in which they were generated to construct an informative error message. A naïve solution to the problem of finding the problematic constraints is to include every constraint which has ever taken part in the rewriting path to the constraint. However, we can easily end up with too many constraints. Consider for example the set of three constraints $\alpha \sim [\beta] \wedge \alpha \sim \mathit{Maybe}\,\gamma \wedge \gamma \sim \mathit{Int}$ (although we call them sets, we combine the separate constraints with $\wedge$). Since the order of solving is not set in advance, we can first make the second and third constraints interact, leading to $\alpha \sim \mathit{Maybe\ Int}$, and only then discover that we have inconsistent ideas of what α should stand for. The naïve approach would flag the three constraints as problematic, but the third one is in fact not relevant type error.

Although alternative solutions exist to omit constraints that do not play a role in the type error (e.g., finding all minimal unsatisfiable constraint sets [6, 21]), in our work we maintain a data structure with all the constraints obtained during the solving process, that we can process later to figure out the problem. Such a data structure must be able to represent not only consistent, but also *inconsistent* sets of constraints. *Type graphs* [7] provide that functionality for the case of type equalities. Type graphs are part of the TOP framework, which is the type inference engine used by the Helium Haskell compiler [8, 9]. Figure 3 shows the type graph for $\{\#2\ \iota \sim \beta \to \mathit{Int}, \#1\ \iota \sim \alpha \to \alpha\}$. Vertices can have two shapes: circular vertices are used for (unification and rigid) type variables and type constructors; the special square vertex tagged with @ is used for type application. Following the usual convention, type application associates to the left and the arrow constructor is written infix, so $\beta \to \gamma$ is equivalent to $((\to)\,\beta)\,\gamma$. Each type variable only appears *once* in a type graph, so different references to α in Figure 3 point to the same node. Edges are either directed edges marked with l and r outgoing from a type application node @ representing the two arguments of @, or undirected edges representing a type equality marked with the constraint they originated from.

During the solving phase, the type graph is saturated with *derived edges*, which represent those equalities which are implied by the original set. In Figure 3 two derived edges would be present once the solver is finished: one between β and α, and another between *Int* and α.

An inconsistency in the case of type equalities arises from a constraint which equates two distinct type constructors, such as $\mathit{Int} \sim \mathit{Bool}$, or fails the occurs check, such as $a \sim [a]$. In the type graph such a problem is represented by a path between the two problematic elements, we call these *error paths*. Figure 3 does not have error paths, but it would have if we replace β by *Bool*.

Heuristics. An error path gives a set of constraints involved in an error, but in order to produce a concise error message we need to choose one of them as responsible. The choice should be made so that if the blamed constraint is removed, the type graph becomes consistent (if no other inconsistencies are present in the type graph). This is easy to check in the type graph by ensuring that no other path exists between the problematic vertices. However, we do not want to check every possible subset of constraints, and the choice may not be unique. For that reason, we define a set of *heuristics* to guide the search in the type graph.

Different heuristics work in different ways. Some of them filter out constraints which should not be blamed, other heuristics select a constraint to be blamed and assign it a weight, and then the one assigned the highest weight will be blamed.

Heuristics tend to differ in their specificity. Language-*independent* heuristics can be applied to any type graph, regardless of the programming language the type graph is used for. The participation heuristic assigns a higher weight to those constraints depending on how often they are part of an error path. Language-*dependent* heuristics employ knowledge of the underlying language, and which are the more plausible explanations for a programmer mistake. Because of their specificity and the subsequent specificity

of the error messages they can generate, they typically assign higher weights. In the Helium compiler there are heuristics such as "missing argument in an application", "missing components in a tuple", or "mistook $(+)$ for $(\#)$ in a function call".

2.2 Type inference for GADTs

Generalized Algebraic Data Types (or GADTs, for short) is a popular type system extension in Haskell [1] that extends ordinary ADTs, by allowing us to refine type information for particular constructors.

For the *Expr* datatype defined in the introduction, we can write a well-typed interpreter of type $Expr\ t \to t$.

$$
\begin{aligned}
&eval :: Expr\ t \to t \\
&eval\ (LitBool\ b) \quad = b \\
&eval\ (LitInt\ \quad i) \quad = i \\
&eval\ (Equals\ x\ y) = eval\ x \equiv eval\ y
\end{aligned}
$$

Note that we do not have to check at every step that the returned expression has the correct type, because this is statically enforced.

Following the rules in Figure 2, this code is not well-typed: for one, the rule CASE requires that the types of all branches coincide, while in this case the first branch returns a *Bool* and the second an *Int*. Second, the type signature of *eval* requires the function to be *polymorphic* in t. However, each of the three branches fixes one *concrete* t.

The key difference with pattern matching over a GADT is that each constructor may bring in *local* information. For example, by matching on *LitBool* we know that t can only be *Bool* in that branch. But that only works if the solver avoids mixing information local to different branches.

The language of constraints from Figure 1 cannot encode local information, so we extend our constraint language with *existentials* (see Figure 4). A constraint of the form $\exists\overline{\alpha}.\,(Q_1 \supset Q_2)$ represents a local scope in which a substitution for unification variables $\overline{\alpha}$ should be obtained, and where the *wanted* constraints Q_2 may use information from the *given* constraints Q_1. For the *eval* function, the constraint set will then be something like:

$$
\begin{aligned}
&\exists\overline{\alpha}.\,(t \sim Bool \supset \text{constraints from } LitBool \text{ branch}) \\
\wedge\ &\exists\overline{\beta}.\,(t \sim Int \supset \text{constraints from } LitInt \text{ branch}) \\
\wedge\ &\exists\overline{\gamma}.\,(t \sim Bool \supset \text{constraints from } Equals \text{ branch})
\end{aligned}
$$

The modified CASE* rule is responsible for harvesting the given constraints $Q_i^\star$ in each existential from the types of the data constructors matched upon. One small detail is that the OutsideIn(X) framework insists that the return type of each data constructor has the same form as for ADTs, that is, a type constructor applied to distinct type variables. The solution is to work around this restriction by using equality constraints. In other words, for the type checker the type of *LitBool* is actually:

$$LitBool :: \forall a.a \sim Bool \Rightarrow Bool \to Expr\ a$$

instead of the equivalent $Bool \to Expr\ Bool$.

The constraint solver also has to be extended to deal with local constraints. In the case of OutsideIn(X), this is done by moving from a simple rewriting relation $Q \rightsquigarrow Q'$ into a more complex form $Q_g; \overline{\alpha} \vdash Q_w \rightsquigarrow Q_r$, which represents that under local (given) information Q_g we can rewrite the (wanted) constraints Q_w into the simpler (residual) form Q_r, and only the variables $\overline{\alpha}$ should be treated as unifiable. Keeping track of the unifiable (or touchable) variables is important for maintaining scoping invariants that prevent information from one branch to infect the other. This rewriting relation is recursively called by the $\vdash^\star$ judgment from Figure 5: every time we go inside an existential, the set of given constraints grows. As a technical detail, each type checker has to define a notion of *solved form*: a set of constraints which is completely solved. In the case of type equalities, that means that every constraint in the residual set is of the form $\alpha \sim \tau$.

The purpose of our work is to combine type graphs, a data structure that has been found useful for explaining type errors, with the ability to deal with local information. The heuristics can then work on such type graphs to analyze type inconsistencies in the presence of GADTs, and generate suitable type error messages.

3 RHODIUM GRAPHS: TYPE GRAPHS WITH LOCAL CONSTRAINTS

This section introduces our extensions to type graphs so that they can be used to represent a type inference process in OutsideIn(X). From this point on we use the term OutsideIn(X) to refer to the original design described by Vytiniotis et al. [25], TOP to refer to the older implementation in the Helium compiler based on type graphs, and *Rhodium* (type) graphs to refer to the extended type graphs introduced in this paper. Due to space limitations, we cannot include all formalities in this paper and refer to [1, Chapter 6] for further details.

It makes sense for Rhodium graphs to be as backwards compatible as possible both with OutsideIn(X) and Helium. There is one problem: the formulation of OutsideIn(X) insists that local definitions are not implicitly generalized, while Helium follows the Hindley-Milner convention of generalizing every local binding as much as possible. We follow OutsideIn(X) in this, which means we sometimes reject programs that are accepted by Helium. These programs can easily be made type correct by adding the right type signatures for all let-polymorphic local definitions.

3.1 Representation of Rhodium type graphs

In this section we explain how constraints in OutsideIn(X) are translated into Rhodium type graphs. The main extension with respect to TOP is the need to represent *existential* constraints. Note that OutsideIn(X) is parametric, so each concrete implementation may add new sorts of vertices and edges to the type graph. In this section we focus on the parts shared by every possible X, namely types and equality constraints.

Variables, types, and constraints. There are multiple valid ways to represent a type in a type graph. Take for example the type *Either A B*. We can choose to represent type application as a binary operator, viewing the type as $(Either\ A)\ B$, or as an n-ary application in which the type constructor receives a list of argument types, hence viewing the type as $Either\ [A, B]$. In Rhodium, we follow the former design and use a special vertex for type application @, as depicted in Figure 6. Because Rhodium also supports type families, and these occur only in fully applied form, Rhodium does allow a vertex that represents a type family to have more than two children. For consistency reasons, the labels r and l that we saw in Section 2.1

[1] 15% of packages in LTS 13.6 (https://www.stackage.org/lts-13.6) use GADTs, and over 40% of the 30 most depended upon packages on Hackage.

$$\text{Constraints } Q ::= \exists \overline{\alpha}.\, (Q_g \supset Q_w) \text{ where } Q_g \text{ contains no existentials} \mid \ldots$$

$$\frac{\Gamma \vdash e : \tau_0 \rightsquigarrow Q_0 \qquad K_i : \forall \overline{a}\overline{b_i}.\, Q_i^\star \Rightarrow \overline{\rho_i} \to \mathsf{F}\,\overline{a} \in \Gamma \qquad \overline{\delta}_i = \mathsf{fuv}(\tau_i, Q_i) - \mathsf{fuv}(\Gamma, \overline{\gamma}) \\ \beta, \overline{\gamma} \text{fresh} \qquad \Gamma, \overline{x_i : [\overline{a \mapsto \gamma}]\rho_i} \vdash e_i : \tau_i \rightsquigarrow Q_i}{\Gamma \vdash \textbf{case } e \textbf{ of } \overline{K_i\, \overline{x}_i \to e_i} : \beta \rightsquigarrow Q_0 \wedge \tau_0 \sim \mathsf{F}\,\overline{\gamma} \wedge \overline{\exists \delta_i.\, ([\overline{a \mapsto \gamma}]Q_i^\star \supset Q_i \wedge \tau_i \sim \beta)}}\ \textsc{case}^\star$$

Figure 4: Constraint gathering for λ-calculus with GADT pattern matching

$$\frac{\begin{array}{l} Q_s = \{Q \in Q_w \mid Q \text{ is not existential}\} \\ Q_g; \overline{\alpha} \vdash Q_s \rightsquigarrow Q_r \\ \text{for each } \exists \overline{\beta}.\, (Q_g' \supset Q_w') \text{ in } Q_w\text{:} \\ \quad Q_g \wedge Q_r \wedge Q_g'; \overline{\beta} \vdash^\star Q_w' \rightsquigarrow Q_r' \\ \quad Q_r' \text{ is in solved form} \end{array}}{Q_g; \overline{\alpha} \vdash^\star Q_w \rightsquigarrow Q_r}$$

Figure 5: Skeleton of a solver for existential constraints

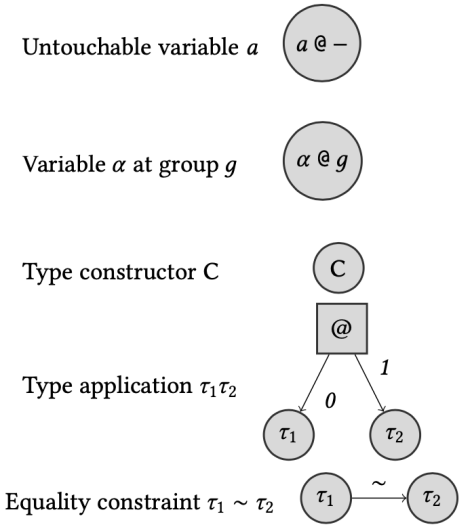

Figure 6: Representation of type graphs

have been replaced by the numbers 0 and 1. Apart from this detail, the treatment of type families in Rhodium follows [25].

Type variables and constructors inherit their representation from TOP. In the case of type variables we annotate the vertex with its *touchability*, which governs when a type variable can be unified. As depicted in Figure 6, a variable may be completely *untouchable* – also known as rigid or Skolem; these arise from checking polymorphic types – or *touchable* at a given group. As we shall discuss later, groups are used to track which constraints may interact with each other once existential constraints enter the picture.

The last element in our type graphs are *constraint edges*. This is an important design decision in Rhodium: every constraint in the system must be represented by an edge. In the simplest case of only type equalities, this representation is quite natural: we connect the two types which should be equal. But in contrast to TOP, type equalities in Rhodium are directional, that is, $\tau_1 \sim \tau_2$ is not represented in the same way as $\tau_2 \sim \tau_1$. The reason is that OutsideIn(X) requires an ordering to guarantee termination in a specific step of the solving process (more concretely, during *orientation*). Other than that, type equality edges are interpreted as undirected.

Relation to the type graphs of TOP. The original type graph implementation of Helium also deals with *instantiation* constraints of the form $\tau > \sigma$, representing that τ is an instantiation of a type scheme σ in order to deal with let-polymorphism. However, one of the design decisions in OutsideIn(X) is not to implicitly generalize **let** definitions. This makes instantiation constraints redundant, since we can generate new fresh instances of the programmer-provided type scheme during constraint gathering. In Rhodium we have taken an intermediate position: we do represent instantiation constraints explicitly in the type graph, but we readily turn them into equality constraints at the beginning of solving. Due to the invariants in OutsideIn(X) we can do this once and for all. The reason for this choice is two-fold. First, it opens the door to extensions of OutsideIn(X) such as GI [19], which introduce higher-rank and impredicative types. Second, future heuristics might want to return a different message depending on whether an inconsistent constraint arose from an instantiation constraint, or not.

Existentials. Pattern matching on GADTs introduces existential constraints during gathering, as described in Section 2.2. Supporting them leads to quite substantial changes to type graphs when compared to those of TOP. The most important issue is that an existential constraint may contain other constraints, and we need to represent this nesting in our type graphs. We consider two possible choices and discuss their advantages and disadvantages.

The first possibility is to avoid the use of existentials and nesting in type graphs. In this scenario, everytime we recurse using the $\vdash^\star$ judgment from Figure 5, we create a completely new type graph with the given constraints and the new simple constraints, and then proceed to solve it. This has the advantage of being simple, because we can be sure that all constraints in the graph may freely interact with each other. However, it makes type error diagnosis harder, since we cannot look at the interaction between different existential branches.

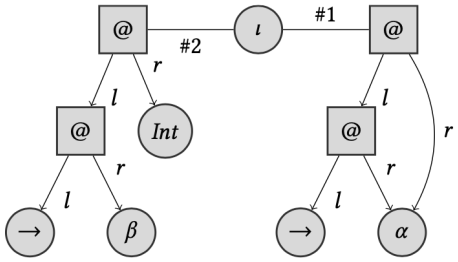

Figure 7: Rhodium type graph for $\alpha \sim Int \wedge \alpha \sim b \wedge \exists\gamma.(\gamma \sim Bool \supset \alpha \sim \gamma)$

Consider the following example:

```
data Expr a where
  I :: Int  → Expr Int
  B :: Bool → Expr Bool
  A :: a    → Expr a

f :: Expr a → Bool
f (I x)  = x
f (B b)  = if b then 3 else 5
f (A _) = 7
```

This code is ill-typed. The most probable cause is that the type signature of f is not correct; we can fix the problem by replacing *Bool* by *Int*. If each branch of f would lead to a separate type graph, we would in fact find three errors, because neither branch is consistent with the type signature.

In the interest of good error diagnosis, we prefer a representation that allows a more holistic view. Therefore we have chosen to integrate all constraints into a *single* type graph. However, this means we have to provide a means to decide which pairs of constraints may interact, otherwise the local reasoning that we need to deal with existentials is lost.

For this reason, we assign to each type variable and each constraint edge a *group*, which tells us to which existential each constraint belongs, and whether a constraint is a given or wanted constraint. In this paper we use numbers to represent groups, starting with 0 for top-level given constraints, 1 for the top-level wanted ones, increasing these numbers as we go into existential constraints. We are careful to maintain two invariants: (1) if an existential constraint is part of another constraint, then its group identifier is higher than that of its parent, and (2) given constraints are always assigned an even number, and wanted constraint always have an odd identifier.

Figure 7 depicts the Rhodium type graph that represents

$$\alpha \sim Int \wedge \alpha \sim b \wedge \exists\gamma.(\gamma \sim Bool \supset \alpha \sim \gamma)\,.$$

At group 1, the top level wanted constraints, the only touchable variable is α, so it is marked as such. Each separate constraint outside the existential is represented as an edge with this group identifier. Inside the existential, $\gamma \sim Bool$ is a given constraint, and thus it is assigned an even group 2 (higher than 1). The innermost wanted constraint is assigned a higher, odd group, 3. Note that the group of a type variable is not related in general to the groups of the constraint edges that point to it, but rather to the specific existential in which the variable is introduced.

3.2 The solving of Rhodium type graphs

Fueled by the above discussion, we change the solving process of OutsideIn(X) from one in which solving processes are spawned recursively when an existential is encountered, to a single iterative process in Rhodium. We employ groups attached to the constraints to ensure that only constraints that are allowed to, may interact with one another. Other than that, solving is performed using the usual rules that each implementation of OutsideIn(X) we know of uses. However, because we apply our rules to Rhodium type graphs instead of to constraints, below we provide some details of the rewriting process.

Groups and accessible sets. Recall that every constraint edge is assigned to a group, which represents the most deeply nested existential in which that constraint lives. To emulate local reasoning, we employ this information to decide when an interaction between two constraints may take place. Take for example the graph in Figure 7: the constraint $\alpha \sim Int$ should always be allowed to interact with other constraints, since it resides at top-level. The given constraint $\gamma \sim Bool$ (with group 2) should be visible in the wanted part of that existential, in group 3.

To decide for a given (current) group g which constraints may be employed during solving, we introduce the notion of *accessible set*, the set of groups g may interact with. The accessible set for a constraint is built starting with its group, and then adding all the ancestor existential groups until we reach top level. Take for example the set of constraints, in which constraints in Q_n are assigned group n:

$$Q_1 \wedge \exists\overline{\alpha}_1.(Q_2 \supset Q_3) \wedge \exists\overline{\alpha}_2.(Q_4 \supset Q_5 \wedge \exists\overline{\alpha}_3.(Q_6 \supset Q_7))$$

The accessible set of Q_6 is $\{1, 4, 5, 6\}$: those are the other groups (including itself) it may interact with. Note that in particular the accessible set of Q_6 does *not* contain 2 or 3, since those constraints are in other existential branches. This mechanism is similar to the scoping mechanism described by Serrano [17].

The solving process traverses each group in a similar fashion to the one described in Section 2.2 for the OutsideIn(X) framework. We start by considering the top level constraints, and then recurse into the existentials. The use of increasing natural numbers as identifiers for groups gives us a simple method to know at every point which constraints may be considered. Since we maintain the invariant that the group of a constraint is always higher than that of its parents in the existential structure, it is enough to start with the constraints at group 0 (the top level given ones), and then increase the current group until all have been considered.

Translating solving rules to the setting of type graphs. Although organized somewhat differently, the Rhodium type graph solver follows OutsideIn(X) faithfully, using a rewriting relation like OutsideIn(X) does. However, since we work on Rhodium type graphs, and not on constraints, we must reflect the result of applying a rewrite rule back into the type graph.

In the case of a *canonicalization* rule, which rewrites a single constraint, the type graph solver first selects a constraint edge in

edge	constraint	created by
#0	$a \sim Int$	original
#1	$a \sim b$	original
#2	$a \sim Bool$	original
#3	$a \sim Int$	interact(#0, #1)
#4	$b \sim Int$	interact(#0, #1)
#5	$a \sim Bool$	interact(#2, #3)
#6	$Bool \sim Int$	interact(#2, #3)

Figure 8: Overly conservative error path

edge	constraint	created by
#0	$a \sim Int$	original, interact(#0, #1)
#1	$a \sim b$	original
#2	$a \sim Bool$	original, interact(#2, #0)
#7	$b \sim Int$	interact(#0, #1)
#8	$Bool \sim Int$	interact(#2, #0)

Figure 9: Modified error path

the current group to which a canonicalization rule is applicable. Then it executes one step of the rewriting relation, producing a new set of constraints which should be added to the type graph. Special care should be taken here: the new constraints and new *touchable* variables have to be in the same group as the considered constraint. The former ensures that canonicalization rules respect the nested existential structure, the latter are necessary to deal correctly with type families [25].

One important difference between the representation of a set of constraints in a purely syntactic manner, as done by the OutsideIn(X) formalization, and our type graphs, is that in the former case a rewritten constraint is removed from the current set, whereas in the latter all the constraints created during the process are retained. To avoid infinite rewriting, once a rewriting rule has been applied to a constraint, that constraint is marked as *resolved*, and will not take part in further simplification.

In the case of *interaction* rules, two constraints interact with one another to create a new set of constraints. In order to guarantee correctness, we need to ensure that the constraints can interact safely. In particular, given a constraint Q in a group n, it may only interact with constraints of which the group number belongs to Q's accessible set.

In general, an interaction rule has the form $Q_1, Q_2 \rightsquigarrow Q_3$. We insert the constraints Q_3 into the type graph the same way we did with canonicalization rules, assigning them to the current group, and mark both Q_1 and Q_2 as resolved, at the deepest existential level of the two, as it should be.

One common scenario in a rewriting system for type inference is that some of the constraints in Q_1 and Q_2 may be returned as part of Q_3. In that case we need to ensure that only the new constraints are introduced in the type graph, otherwise error reporting may suffer. Take, for example, the constraints $a \sim Int \wedge a \sim b \wedge a \sim Bool$. In Figure 8 all the constraints from an interaction are added to the type graph, whereas in Figure 9 only the new ones are added. The latter describes more precisely the solving process, and thus leads to more precise heuristics. As result, we may need to unmark some of the constraints as resolved, if they are present again in the new set produced by the rewriting rule.

Errors. If a constraint rewrite returns $\bot$, no constraint is added to the type graph. Instead, the edge is marked as *inconsistent* preventing it from taking part in any further solving, although the solving process will continue. In addition, we may attach an *error label* to each inconsistent edge. For example, $Int \sim Bool$ may be labelled with INCORRECT CONSTRUCTORS, or $a \sim [a]$ with INFINITE TYPE. These labels can be employed by the heuristics used for type error diagnosis later on (Section 4).

Residual constraints. Once we have finished applying rewriting rules to the constraints in a group there might be some constraint edges which remain unmarked as resolved. However, a non-empty set of leftover constraints does not necessarily mean that the original program contains an error. To decide this we need some further post-processing. This post-processing may be either performed at the end of the simplification of each group, or at the very end of the solving process.

First of all, there are constraints such as $Eq\ \alpha$ which we always expect to mark as resolved. In this case, not having done so means that an instance for α was not found in the given constraints or the axioms, and we should report this fact as a type error. The error label we assign to these constraints is RESIDUAL CONSTRAINT.

For the case of equality constraints like $\alpha \sim Int$ the distinction is subtler. Some of those equality constraints correspond to parts of the final *substitution* that the solving process produces; those are the ones of the form $\alpha \sim \tau$ which satify that (1) its group g corresponds to a wanted set, and (2) the type variable α is also introduced in that same group. If condition (1) is not satisfied, the constraint is simply ignored, but if (1) holds but (2) does not, the constraint represents inconsistent information and it is marked as an error with label VARIABLE ESCAPE. Note that this pair of conditions is a safe over-approximation of when a set of equality constraints represents a correct substitution; real implementations such as GHC implement a "variable floating" rule which is less strict yet still safe [19].

We close this section with an elaborate example to illustrate the solving process. Consider the wanted constraint

$$Num\ \alpha \wedge \alpha \sim Bool \wedge \exists \beta.\ (\beta \sim Int \supset \alpha \sim \beta) \wedge \exists \gamma.\ (\gamma \sim Bool \supset \alpha \sim \gamma),$$

where the variable α is touchable at top level and no axioms or given constraints are present. (1) Rhodium makes a type graph of all the constraints, based on the constraint solver X that is specified. Groups are assigned as usual: even for given constraints, odd for wanted constraints. (2) We start the solving process for group 1. There we allow two constraints, $Num\ \alpha$ and $\alpha \sim Bool$, to interact with one another. This results in the constraints $Num\ Bool$ and $\alpha \sim Bool$, but only the former is added to the type graph, since the latter was already there. As these constraints can not be simplified further, we mark $Num\ Bool$ as residual, and we increase the current group to 2. (3) With a current group of 2, we consider the given constraints of the first existential. These constraints can interact with the constraints of group 1, but not with one another. Because of their particular shape, no interaction rule applies, and

we increase the current group to 3. (4) Within group 3, the constraint $\alpha \sim \beta$ is considered wanted. There are no more constraints in that group, but the constraint may interact with both $\alpha \sim Bool$ (group 1) and $\beta \sim Int$ (group 2), leading to $Bool \sim Int$. This is an inconsistent constraint, and it is marked as such. (5) We repeat the process with the other existential constraint. In this case the wanted constraint is first turned into $Bool \sim Bool$, which then disappears by a canonicalization rule. Thus, we have no residual constraints in this group.

4 HEURISTICS FOR GADTS

In this section, we focus on the heuristics defined specifically for diagnosing type incorrect code that involves GADTs, and provide examples of type error messages provided by our implementation. We have also re-implemented many heuristics that were present in Helium previously and that worked on the simpler type graphs in TOP [1].

4.1 How heuristics are applied

After constraint solving within Rhodium has terminated, some constraints may have been marked as an error (using a specific error label). For example, a constraint $Int \sim Bool$ will have the label INCORRECT CONSTRUCTORS, and $a \sim Int$ may have the label VARIABLE ESCAPE.

Given a single error constraint Q and the simplified type graph, we then determine the error slice associated with Q. This *error slice* consists of all the constraints that may have contributed to the problem. As mentioned previously, every constraint keeps track of how it was created: either it was generated from the program directly – an *original* constraint – or it is the result of a constraint solving step applied to some constraints, each of which keeps track of how it was created. By iteratively traversing the history of each constraint we construct the set all the constraints involved in the simplification process that led to the creation of Q. From this error slice we consider only those which were generated directly from the program, that is, the original ones. These constraints come with additional information obtained during gathering, e.g., the syntactic construct that generated the constraint, and the source location for that construct.

The input to the next step of this process is composed of pairs, where each pairs consists of an error constraint edge (which includes the error label attached to it) and the corresponding error slice. Each of these pairs is considered one by one. In each case, the goal is to reduce the error slice to a single original constraint, which is then *blamed* for the particular error. We do so by applying *heuristics* to the error slice. Even though heuristics consider only one error slice as target for reduction, they may query all the other error slices for additional information.

Rhodium provides quite a number of heuristics that are applied in sequence. Every application of a heuristic may reduce but never increase the error slice. If after running all heuristics more than one constraint remains, we choose the first constraint.

As in Helium, Rhodium supports two kinds of heuristic: filter heuristics and voting heuristics. A *filter heuristic* deletes constraints from the error slice, implying that those original constraints should *not* be blamed. An example of such a constraint is one that models that the condition of an **if**-expression should have type *Bool*. For the expression **if** 3 **then** 2 **else** 1 we expect a message that blames the use of 3 where a *Bool* is expected, and not a message that insists we should not demand an expression of type *Bool* in the condition.

A filter heuristic may delete any number of constraints from the slice, as long as the outcome is not the empty set, implying that no constraint can be blamed. Typically, if a filter heuristic observes that all constraints in the slice have the property it is designed to remove, it will in fact not delete any constraints in the hope that other heuristics can make a better choice.

The *voting heuristic* is essentially a collection of selectors. A *selector* is especially designed to recognize certain well-known error patterns, for example that the components of a pair occur in the wrong order. If it recognizes such a pattern, it returns the constraint to be blamed for the mistake, and a weight that indicates how likely it is that this is the cause of the inconsistency. If it does not recognize such a pattern, the heuristic will not participate in the voting heuristic.

After all selectors have made their choice, if any, all constraints with the highest weights assigned to them by a selector remain in the error slice and all others are deleted. The process then continues, if necessary, by considering any further heuristics.

The choice for a constraint to blame is not the only output of the process. Whenever a heuristic assigns the blame to a constraint, it also attaches a so-called *graph modifier* to that constraint that describes how the graph needs to be adapted to continue with the solving process. The default graph modifier is to delete the edge to which the blamed constraint was attached; this is the only graph modifier present in TOP, but we found we had to supply other options.

For example, a common type error is forgetting to add a particular constraint to the type signature of a function:

$g :: a \to a \to String$
$g\ x\ y = show\ x +\!\!+ show\ y$

In this case, we have two residual constraints of type *Show a*. If we may only remove constraints, we have to remove both *show x* and *show y* resulting in two very similar error messages. However, in Rhodium we employ a heuristic that blames a constraint that was found to be missing, and employs a graph modifier that *adds* the missing predicate $Show\ a$ to the type signature of g, so that inference may continue. The type error message will come with a hint to the programmer to add the predicate to the type of g.

Our implementation provides a number of graph modifiers. Beyond the default modifier, and the modifier that adds a residual constraint, Rhodium employs two others. Consider the example of $True + 3$. In that case, we have the constraints $\alpha \sim \beta \to \gamma \to \delta \wedge \alpha > \forall a.Num\ a \Rightarrow a \to a \to a$, where α represents the type of the function $(+)$. If we only remove $\alpha \sim \beta \to \gamma \to \delta$, we are still left with the instantiation constraint, which then causes an error as it has a residual constraint $Num\ a$. This graph modifier therefore removes both the application edge, as well as the accompanying type signature. The fourth modifier can add a type signature to a function. Indeed, every function that pattern matches on a GADT must have a type signature. When a type signature is missing, we produce a type error. In certain cases, we can recommend a type

```
data Expr a where
  LitInt  :: Int  → Expr Int
  LitBool :: Bool → Expr Bool

g :: Expr Int → Int
g (LitInt  x) = x
g (LitBool y) = y
```

Figure 10: Unreachable pattern example

signature computed from the GADT pattern matches, and this modifier essentially allows us to add this recommended type signature to the type graph so that inference may continue.

4.2 Heuristics for GADTs

We now consider the type errors that can occur whenever GADTs are introduced. We describe a number of heuristics which deal with new error scenarios introduced by this language feature.

Missing constraint in GADT constructor. One of the main features of GADTs is the ability to introduce existential variables which do not exist outside of the scope of that constructor:

```
data X where
  A :: b → X

f :: X → String
f (A x) = show x
```

The type of the variable x is not mentioned in the data type X, so in this case we cannot add the constraint to type signature of the function. The *missing constraint heuristic* is aware of this fact, and produces the following error message:[2]

```
(5,11): Missing class constraint in type signature
 function          : show
   declared type   : Show a => a -> String
 class constraint  : Show b
 hint : add the class constraint to the type signature
        from the GADT constructor, defined at (2,4)
```

As part of the type error we provide the constraint that needs to be added, in this case the type class constraint *Show b*, and the location of the constructor to which the constraint should be added.

More generally, the "missing constraint" heuristic works in two phases. The heuristic first tries to introduce the missing constraint as part of the local definition, e.g., in its type signature. For example, a type signature of the function $Y\ a \rightarrow String$ would not be incorrect if the predicate *Show a* were to be added, so we prefer this over adding a constraint to the constructor. The main reason for this choice is that changing a constructor has arguably a larger impact than modifying a type signature, as the latter only requires the constraint to be satisfied whenever the *function* is called, not every single time the *constructor* is used. Only if the heuristic detects that it is impossible to add the constraint in a local definition, it will suggest changing the constructor itself.

Unreachable pattern. Within a GADT, knowing the type of the scrutinee of a pattern match can make certain pattern matches inaccessible. Take for example the function g defined over a simplified version of the data type in the introduction in Figure 10. In this case, the type signature of g only allows values of type *Expr Int* as argument. As a result, the case of constructor *LitBool* can never happen, since it requires a value of type *Expr Bool*. This causes an inconsistent constraint of the shape $Int \sim Bool$ in the type inferencer.

The *unreachable pattern* heuristic detects that the inconsistency is caused due to a pattern match that does not match the provided type signature and provides an appropriate error message:

```
(7,4): Pattern is not accessible
 Pattern                 :  LitBool y
   constructor type      :  Bool -> Expr Bool
   defined at            :  (3,4)
   inferred type of      :  a -> Expr Int
   pattern
 hint :  change the type signature, remove the branch
         or change the branch
 possible type signature :  (Expr b) -> b
```

The error message specifies the type of the constructor, the inferred type of the branch, as well as the location of the definition of the constructor. Note that the heuristic also suggests a type signature that would allow the pattern match to be kept. This type signature is based on the most general type that can be derived from all of the individual branches. After this, the type signature is tested against the type graph to verify that it indeed resolves the error and does not introduce any other problems. Only when the type signature would resolve the error, it is recommended to the programmer. In all other cases, only the hint is provided, without mentioning the possible type signature.

Missing GADT type signature. As discussed by Vytiniotis et al. [25], once GADTs are introduced in the language, the principal types property is lost. This means that there could be *multiple* valid type signatures no two of which are instances of each other. As a result, functions dealing with GADTs require a type signature.

A very strict policy would require providing a type signature for *every usage* of a GADT, making the detection of not providing a GADT type signature a static check, but we decided against that. The reason is that in many cases we can use the information in the type graph to infer a possible type for the function. The process to determine this type signature is very similar to the process described for inferring type signatures for unreachable patterns.

If we take the code from Figure 10 and drop the type signature for g, then a type signature that would resolve the error is inferred and reported to the programmer:

```
(5,1), (6,1): A type signature is necessary for
              this definition
 function : g
 hint : add a valid type signature, e.g. (Expr a) -> a
```

The error message provides the possible type $(Expr\ a) \rightarrow a$ as a suggestion to the programmer.

Non-unifiable GADT variables. As discussed earlier, one key issue to sound checking and inference of code using GADTs is keeping track of which type variables can be unified at each moment. In fact, some of those are rigid and may *never* be unified with another type unless a given constraint assumes so.

[2] Some error messages have been re-formatted to fit withing the page limits, but no text has been changed from the produced output of our implementation.

Consider the following example, where we unify the variable x of type b with the type *Bool*, but the variable b is an existential introduced by the constructor A, hence forbidding b to unify with anything:

$$\textbf{data } X \textbf{ where}$$
$$A :: b \to X$$
$$f :: X \to Bool$$
$$f\ (A\ x) = x\ ||\ True$$

Our implementation produces the following error message, stating that the variable cannot be unified. In addition to the error message itself, it also gives the original constructor, as well as the location at which it is defined:

```
(5,1): Cannot unify variable in function binding
 function binding         : f (A x) = x
   existential type       : b
   cannot be unified with : Bool
   constructor            : b -> X
   defined at             : (2,4)
```

This heuristic works on residual constraints of the shape $a \sim b$ where a is a non-touchable variable (be it rigid or coming from a different group) and b can be any type. We can tell from the type graph whether a is coming from a pattern match and whether that variable shows up in the result of the pattern match. For example, the variable d is not an existential in a constructor of type $c \to d \to Z\ d$, so in that case this heuristic does not apply.

4.3 What we left out

When moving to a new solver, we should be careful not to lose Helium's type error messages for plain Haskell 98 programs (insofar Helium supports those). Therefore, we have transplanted all heuristics on vanilla type graphs to Rhodium, so that for programs without GADTs we can expect to obtain the same type error messages. Another class of examples we cannot discuss due to space limitations are the ones that show that our heuristics behave well when multiple heuristics may interact. Examples of both these kinds can be found in a separate document [1, Sections 7.4 and 7.7].

5 RELATED WORK

Type error slicers present the programmer with information about *all* possible program points which contribute to the detected inconsistency. Skalpel [16] implements type error slicers for Standard ML, supporting advanced SML features like modules, which are somewhat related to GADTs in Haskell. The advantage of slicing is that the actual location that causes the problem is highlighted, a disadvantage is that many others locations are highlighted as well.

Because type error slices can be large, many researchers prefer to blame one, or a few constraints. For example, SHErrLoc [28] uses a graph-based structure to encode the solving process, and then ranks the likeness of a constraint being to blame using a Bayesian model. Their work considers type error reporting for modern Haskell, including local hypotheses. Chen and Erwig [2] explain type errors in Haskell programs using counter-factual typing, a version of variational typing in which they keep track of the different types that an expression may take. Although computationally somewhat costly, they can propagate type inconsistencies from one binding group to another. Pavlinovic et al. [12] achieve something similar by using an iterative deepening approach, in which the body of a binding is inlined in its usage site if a conflict is detected between both. This allows the inferencer to blame a location in the body of a (type correct) function if an application of that function is type incorrect, at the expense of repeatedly calling an SMT solver with a growing set of constraints. These papers perform only *error localization*.

In our work, we employ specialized heuristics that recognize type error patterns by examining a type graph. When we detect such a pattern, we not only know the location, but we can also explain about the pattern we detected, and for some patterns, even give a clue on how to fix the problem, inspired by [7]. Erwig [5] also uses graphs to represent the type inference problem for a given program, mostly as a replacement for the standard algorithms. Helium uses a straightforward constraint solver until a type error is discovered. Then it builds a type graph from the constraints of the binding group that faield to type, and applies heuristics to discover which constraint(s) are to blame.

Whenever the type system is extended, e.g., with type class information, extensions typically need to be made to the type graphs to represent these faithfully. The main technical contribution of this paper, is the design of a type graph structure that can represent constraint sets generated by OutsideIn(X), allowing us to represent local reasoning in type graphs. Type graphs were extended with type classes and row types in the setting of Elm [13], and Weijers et al. [26] uses heuristics to diagnose security type errors.

Some authors use a more complicated structure to diagnose type errors: [14] and [24] expose the trace of the type checker to the programmer, and Chitil [3] defines an explanation graph for Hindley-Miler type systems, which summarizes the information involved in type checking.

Pointwise GADTs [11] have been developed with better type error reporting in mind, by excluding pathological cases which are hard to explain. Others have used abduction to infer a common type for all branches in a GADT [20, 23]. In this case, reasoning is performed within a more complex framework, which is harder to explain to the programmer.

6 CONCLUSION AND FUTURE WORK

We have extended Helium with GADTs and achieving good error diagnosis for a number of classes of inconsistent programs, as compared to GHC. We have extended Helium type graphs in order to model local reasoning in the type graph and defined GADT specific heuristics to help diagnose problems that involved GADTs. This work is a major step in our endeavour to achieve good error diagnosis for advanced, but often used Haskell language extensions, including type class extensions, type families and higher-ranked types.

REFERENCES

[1] Joris Burgers. 2019. Type error diagnosis for OutsideIn(X) in Helium. https://dspace.library.uu.nl/handle/1874/382127.
[2] Sheng Chen and Martin Erwig. 2014. Counter-factual Typing for Debugging Type Errors. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Diego, California, USA) *(POPL '14)*. ACM, New York, NY, USA, 583–594. https://doi.org/10.1145/2535838.2535863
[3] Olaf Chitil. 2001. Compositional Explanation of Types and Algorithmic Debugging of Type Errors. In *Proceedings of the Sixth ACM SIGPLAN International*

Conference on Functional Programming (Florence, Italy) *(ICFP '01)*. ACM, New York, NY, USA, 193–204. https://doi.org/10.1145/507635.507659

[4] Luis Damas and Robin Milner. 1982. Principal Type-schemes for Functional Programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Albuquerque, New Mexico) *(POPL '82)*. ACM, New York, NY, USA, 207–212. https://doi.org/10.1145/582153.582176

[5] Martin Erwig. 2006. Visual Type Inference. *J. Vis. Lang. Comput.* 17, 2 (April 2006), 161–186. https://doi.org/10.1016/j.jvlc.2005.04.004

[6] María García de la Banda, Peter J. Stuckey, and Jeremy Wazny. 2003. Finding All Minimal Unsatisfiable Subsets. In *Proceedings of the 5th ACM SIGPLAN International Conference on Principles and Practice of Declaritive Programming* (Uppsala, Sweden) *(PPDP '03)*. ACM, New York, NY, USA, 32–43.

[7] Jurriaan Hage and Bastiaan Heeren. 2007. Heuristics for Type Error Discovery and Recovery. In *Proceedings of the 18th International Conference on Implementation and Application of Functional Languages* (Budapest, Hungary) *(IFL'06)*. Springer-Verlag, Berlin, Heidelberg, 199–216. http://dl.acm.org/citation.cfm?id=1757028.1757040

[8] Jurriaan Hage and Bastiaan Heeren. 2009. Strategies for Solving Constraints in Type and Effect Systems. *Electron. Notes Theor. Comput. Sci.* 236 (April 2009), 163–183. https://doi.org/10.1016/j.entcs.2009.03.021

[9] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. 2003. Constraint based type inferencing in Helium. In *Workshop Proceedings of Immediate Applications of Constraint Programming*, M.-C. Silaghi and M. Zanker (Eds.). Cork, 59 – 80.

[10] Oukseh Lee and Kwangkeun Yi. 1998. Proofs About a Folklore Let-polymorphic Type Inference Algorithm. *ACM Trans. Program. Lang. Syst.* 20, 4 (July 1998), 707–723. https://doi.org/10.1145/291891.291892

[11] Chuan-kai Lin and Tim Sheard. 2010. Pointwise Generalized Algebraic Data Types. In *Proceedings of the 5th ACM SIGPLAN Workshop on Types in Language Design and Implementation* (Madrid, Spain) *(TLDI '10)*. ACM, New York, NY, USA, 51–62. https://doi.org/10.1145/1708016.1708024

[12] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2015. Practical SMT-based Type Error Localization. In *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming* (Vancouver, BC, Canada) *(ICFP 2015)*. ACM, New York, NY, USA, 412–423. https://doi.org/10.1145/2784731.2784765

[13] Falco Peijnenburg, Jurriaan Hage, and Alejandro Serrano. 2016. Type Directives and Type Graphs in Elm. In *Proceedings of the 28th Symposium on the Implementation and Application of Functional Programming Languages, IFL 2016, Leuven, Belgium, August 31 - September 2, 2016*. 2:1–2:12. https://doi.org/10.1145/3064899.3064907

[14] Hubert Plociniczak. 2013. Scalad: An Interactive Type-level Debugger. In *Proceedings of the 4th Workshop on Scala* (Montpellier, France) *(SCALA '13)*. ACM, New York, NY, USA, Article 8, 4 pages. https://doi.org/10.1145/2489837.2489845

[15] François Pottier and Didier Rémy. 2005. The Essence of ML Type Inference. In *Advanced Topics in Types and Programming Languages*, Benjamin C. Pierce (Ed.). MIT Press, Chapter 10, 389–489.

[16] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2017. Skalpel: A constraint-based type error slicer for Standard ML. *J. Symb. Comput.* 80, P1 (May 2017), 164–208. https://doi.org/10.1016/j.jsc.2016.07.013

[17] Alejandro Serrano. 2018. *Type Error Customization for Embedded Domain Specific Languages*. Ph.D. Dissertation. Universiteit Utrecht, The Netherlands.

[18] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *Proceedings of the 25th European Symposium on Programming Languages and Systems - Volume 9632*. Springer-Verlag New York, Inc., New York, NY, USA, 672–698. https://doi.org/10.1007/978-3-662-49498-1_26

[19] Alejandro Serrano, Jurriaan Hage, Dimitrios Vytiniotis, and Simon Peyton Jones. 2018. Guarded Impredicative Polymorphism. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Philadelphia, PA, USA) *(PLDI 2018)*. ACM, New York, NY, USA, 783–796. https://doi.org/10.1145/3192366.3192389

[20] Vincent Simonet and François Pottier. 2007. A Constraint-based Approach to Guarded Algebraic Data Types. *ACM Trans. Program. Lang. Syst.* 29, 1, Article 1 (Jan. 2007). https://doi.org/10.1145/1180475.1180476

[21] Peter J. Stuckey, Martin Sulzmann, and Jeremy Wazny. 2006. Type Processing by Constraint Reasoning. In *Programming Languages and Systems*, Naoki Kobayashi (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 1–25.

[22] Martin Sulzmann, Gregory J. Duck, Simon Peyton Jones, and Peter J. Stuckey. 2007. Understanding Functional Dependencies via Constraint Handling Rules. *J. Funct. Program.* 17, 1 (Jan. 2007), 83–129. https://doi.org/10.1017/S0956796806006137

[23] Martin Sulzmann, Tom Schrijvers, and Peter J Stuckey. 2008. Type inference for GADTs via Herbrand constraint abduction. https://lirias.kuleuven.be/retrieve/10888

[24] Kanae Tsushima and Kenichi Asai. 2013. An Embedded Type Debugger. In *Implementation and Application of Functional Languages*, Ralf Hinze (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 190–206.

[25] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. 2011. OutsideIn(x): Modular Type Inference with Local Assumptions. *J. Funct. Program.* 21, 4-5 (Sept. 2011), 333–412. https://doi.org/10.1017/S0956796811000098

[26] Jeroen Weijers, Jurriaan Hage, and Stefan Holdermans. 2014. Security type error diagnosis for higher-order, polymorphic languages. *Science of Computer Programming* 95 (2014), 200 – 218. https://doi.org/10.1016/j.scico.2014.03.011 Selected and extended papers from Partial Evaluation and Program Manipulation 2013.

[27] Jun Yang, Greg Michaelson, Phil Trinder, and J. B. Wells. 2000. Improved Type Error Reporting. In *In Proceedings of 12th International Workshop on Implementation of Functional Languages*. 71–86.

[28] Danfeng Zhang, Andrew C. Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. Diagnosing Type Errors with Class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. ACM, New York, NY, USA, 12–21. https://doi.org/10.1145/2737924.2738009