Universiteit Utrecht

Faculty of Science

Dept. of Information and Computing Sciences

Proposal: Type error diagnosis for OutsideIn(X) in Helium

Author

J.J.F. Burgers

Supervisor

Dr. J. Hage

Dr. A. Serrano Mena

March 4, 2019

1 Introduction

Haskell is a pure, lazy, statically typed, functional programming language which is designed to be a language suitable for teaching, research and applications [14]. As Haskell has such different use cases, the language has many different features that make it confusing for people who just started functional programming. Beside those who just started programming, Haskell can confuse many an experienced programmer, as the language supports many features for an experienced programmer, like ad-hoc polymorphism[30], type families[23] and GADTs[21] (Section 2.2). These features allow experienced programmers to program with a high degree of certainty in the correctness of their program, due to Haskell's static type system. Even though these features are usually used by experienced programmers, every experienced programmer was once a programmer who was still learning. Therefore, if something goes wrong with these advanced features, there should be specialized and clear error messages. Currently Helium[10] (Section 2.1) is one of the Haskell compilers that uses a number of techniques to improve the error messages, but these techniques are limited to a subset of the Haskell 2010 standard and lacks support for more advanced features that are not supported by the standard, let alone specialized type error diagnoses for them. My thesis will focus on adding GADTs, a feature outside the Haskell 2010 standard, to Helium using the OutsideIn(X) framework and to transfer the existing quality of error messages to the OutsideIn(X) framework. OutsideIn(X) is a framework for type checking and inferencing, which will be discussed in detail in Section 2.3.4. We will do this to allow support for type error diagnosis for more advanced features.

1.1 Relevance

Functional programming languages are difficult to learn, especially when an error occurs and it is not clear to the programmer what the error is, where the error originated and how to fix it. Take the example *maximum* 3 5, which is incorrect as *maximum* would require a data structure to compute the maximum, not two variables. The error message created by GHC, when written in GHCi, the interactive environment provided with GHC, is as follows:

```
<interactive>:1:1: error:
    * Non type-variable argument in the constraint: Ord (t1 -> t2)
      (Use FlexibleContexts to permit this)
    * When checking the inferred type
        it :: forall (t :: * -> *) t1 t2.
              (Num (t (t1 -> t2)), Num t1, Ord (t1 -> t2), Foldable t) =>
              t2
```

Even an experience programmer, would find such an error message confusing. It talks about *FlexibleContexts*, but enabling this language extension, does not make it better. With *FlexibleContexts* enabled, the error message says:

```
    * Could not deduce (Num t0)
      from the context: (Num (t (t2 -> t3)),
                         Num t2,
                         Ord (t2 -> t3),
                         Foldable t)
        bound by the inferred type for `it':
                   (Num (t (t2 -> t3)), Num t2, Ord (t2 -> t3), Foldable t) => t3
        at <interactive>:9:1-11
      The type variable `t0' is ambiguous
    * In the ambiguity check for the inferred type for `it'
      To defer the ambiguity check to use sites, enable AllowAmbiguousTypes
      When checking the inferred type
        it :: forall (t :: * -> *) t1 t2.
              (Num (t (t1 -> t2)), Num t1, Ord (t1 -> t2), Foldable t) =>
              t2
```

It talks about type variables that are introduced by the compiler, *Ord* instances for functions and ambiguous type variables. The simple mistake of confusing the functions *max* and *maximum* causes an unreadable type error which does not make clear to the programmer what mistake they made. There have already been improvements done in the area of error diagnosis. Helium for example, will state the type of *maximum* and the way it was used. It will also advice you that *max* might fix the problem. This is a much better approach, but unfortunately, Helium only supports a subset of the Haskell 2010 standard. Ideally, we would like to have the error messages with the quality that Helium supports, but with the support for more of the features that GHC supports and, ideally, the possibility of extending the amount of features while still having the same quality of error messages.

To enable this goal, we will try to enable the heuristics that allow Helium to give better error messages in the OutsideIn(X) framework. This framework is used by GHC to do type checking and inferencing and has the support for the advanced language features that we would need. Helium currently used the TOP framework, which gives its error messages using type graphs. This data structure is not build to handle the more advanced types of constraints that OutsideIn(X) requires to complete its type inferencing. This thesis will focus on transforming the Helium heuristics to OutsideIn(X), probably by creating a variant of type graphs that allows the Helium heuristics to work with an implementation of OutsideIn(X).

Section 2 will describe the current state of research with respect to Haskell, GADTs, OutsideIn(X) and type error diagnosis. Section 3 will lay out the research questions we will try to answer. After this, the approach (Section 4) and expected planning (Section 5) to answer these questions will finalize this proposal.

2 Literature review

2.1 Helium

Helium is a Haskell compiler, developed at Utrecht University. Helium is specifically designed and developed to have high quality error messages. As proclaimed by Gerdes *et al*[3]:

"Helium gives excellent syntax-error and type error messages"

This aim is achieved by a number of techniques, most notably the TOP framework (Section 2.3.3) and a large number of heuristics (Section 2.4.3). Helium supports most of the Haskell 2010 standard, excluding newtypes, records and strict fields for datatypes. Even without these features, Helium can be used to compile most regular programs, but it has no support for several widely-used extensions, like multi parameter type classes, GADTs and type families. The frequency of usages of these extensions is reported by Hage[5], which indicates that they are indeed widely used. The backend of Helium, LVM [16], is used to run Haskell programs and currently has support for all functionality that is supported by Helium.

2.2 Generalized algebraic datatypes

Generalized algebraic datatypes, commonly abbreviated as GADTs, are datatypes with extra information attached to the constructors. This is done by providing a type signature for the constructor, like $A :: Int \to T\ Int$ or $B :: Eq\ a \Rightarrow T\ a \to T\ a \to T\ Bool$, which both construct a data type of type $T :: (* \to *)$. GADTs are one of the many extensions to the official Haskell 2010 standard[17] implemented in GHC and enabled by way of an language extension. GADTs allow a programmer a finer control over their type checking, as the type checker can verify certain cases to be correct. Consider the following example:

```
data T a where
  T1 :: T Int
  T2 :: a -> T a
f T1     = 0
f (T2 x) = x
```

In the above example, the GHC type checker infers the type $T\ p \to p$, which would seem incorrect to somebody who is not familiar with GADTs. After all, the first branch of the function f clearly returns a 0, which would normally force the type to be $f :: T\ Int \to Int$, but in the case of GADTs, this type is correct, as for every possible argument $T\ p$, p is the result of the function. In the first branch, we know that the type of the argument has to be $T\ Int$, due to the type of the constructor $T1$, therefore, for that particular branch, there is only one possibility the result of the function can be. In the second case, the type variable a is immediately returned and therefore follows the type $T\ p \to p$. This problem with types is one of the complications that arise with GADTs. It is correct to annotate the function f with the more restrictive type $T\ Int \to Int$, but the type $T\ p \to p$ is more general and therefore the preferred type to infer. It should be noted that there is a difference between the described type inferencing described here and the implementation inside OutsideIn(X). OutsideIn(X) does not allow for type inferencing when no type signature is provided when

GADTs are involved. GHC does infer the correct type in some cases, but this is achieved by heuristics that are not part of the OutsideIn(X) framework.
Another example that is quite often used, is the case of an expression language that is used to ensure type safety of all the expressions. One version of such an expression language is as follows:

```
data Expr a where
  I   :: Int → Expr Int
  B   :: Bool → Expr Bool
  Add :: Expr Int → Expr Int → Expr Int
  EEq :: Eq a ⇒ Expr a → Expr a → Expr Bool

eval :: Expr a → a
eval (I i)       = i
eval (B b)       = b
eval (Add x y)   = eval x + eval y
eval (EEq x y)   = eval x ≡ eval y
```

The type checker ensures that the *eval* function can never fail. In this case, only expressions of type *Expr Int* can be used to construct an *Add* expression. This knowledge is used in this case to prevent us from writing *Add* (*B True*) (*B False*), which would be possible to write without the usage of GADTs. It is also possible to add a class predicate to a particular branch. Because of the *Eq a* constraint in the context of *EEq*, we can use the $\equiv$ operator in the branch of *eval* (*EEq x y*). It is noteworthy to add that this *Eq* constraint only applies to the *EEq* branch of the eval function. If we would have a general constructor of type $b \rightarrow Expr\ b$, we could not use the $\equiv$ operator, as we would not have any information whether the constraint *Eq b* holds. It would of course be possible to change the constructor *Add* to a more general $Num\ a \Rightarrow Expr\ a \rightarrow Expr\ a \rightarrow Expr\ a$, if we would want to increase our expressiveness. Other examples of GADTs are given by Sheard [27], Hinze *et al*[13] and Peyton Jones et al. [21], who also provide number of typing rules for a language which includes GADTs.

2.3 Type checking and inferencing

Type checking and inferencing is the process of respectively verifying and inferring the types of a program. Usually, this is a combined process, in which the type system will check the types provides by the programmer and infer any other types that are not specified. There are a number of techniques, each with their respective advantages or disadvantages.

2.3.1 Algorithm W, Algorithm M and Algorithm G

Algorithm W is an algorithm, proposed by Damas and Milner [2], that is an inference algorithm for the Hindley-Milner type system[12]. In this proposal, we will assume knowledge of the Hindley-Milner type system. We will also not go into much detail of the working of Alogrithm W, as it is not that relevant for the problems the final thesis will aim to solve, yet go into some of the limitations of Algorithm W. The main problem with Algorithm W is that it infers types in a biased way, which results in a bias in its type error reporting. As Algorithm W is a bottom-up algorithm, it reports the first inconsistency it encounters during the solving process. The problem with this approach is that this might not be the type error that is most useful or accurate to the programmer. We would like to know why the type error occured and which parts contribute to the error. As Algorithm W reports the first type error it encounters, it has a bias towards some parts of the program, like a left-right bias. It also has the problem that it reports type errors relatively late. As Algorithm W was not designed with type error reporting in mind, but rather with the intention of analyzing programs effeciently, it is to be expected that other algorithms are more adapted to correctly reporting errors. Beside Algorithm W, there is also a top-down variant, called Algorithm M[15]. Algorithm M is interesting as it will report possibly different errors than in the case of Algorithm W, when checking the same program. In fact, it reports the errors earlier than Algorithm W, but it still suffers from the same problems as Algorithm W.

There are also hybrid algorithms, like Algorithm G[15]. It is possible to emulate these algorithms using a constraint based inferencing, using custom constraint ordering, as explained by Hage and Heeren[7]. They converted the AST to a constraint tree with the same structure. After this, they add a separate phase between the gathering and solving of these constraints. In this phase, the constraint tree is flattened to a list of constraints. This order of this lists is determined by the tree walk, how to flatten the tree, and specifies the behaviour of the type inferencer, that is, which constraint is blamed for an inconsistency, if there exists one.

2.3.2 Constraint-based type inferencing

Constraint-based type inferencing is the process of generating constraints, based on the given program, and then solving the generated constraints to produce the types that are present in the given program, as long as the program is type correct. Otherwise, a type error needs to be generated. This approach is explained by Aiken and Wimmers[1]. They state that if the type inferencer can produce a solution from the given constraints, the program is well-typed and that if no solution is found, that the program might crash, as it is not type correct. As there are a number of different type inference systems, where TOP (Section 2.3.3) and OutsideIn(X) (Section 2.3.4) are the most relevant for this proposal, there are a number of properties that we are interested in for type inference systems. These properties are soundness, completeness and principality of types, as taken from Vytiniotis *et al*[29].

- **Soundness**: Soundness of a type inferenceer, as described by Sulzmann [28], is that any judgement made by the type inferencer can also be made in the logical system. Wright and Felleisen[31] stated it as when well typed programs cannot cause type errors. This means that if it follows from the type inferencer that expression e has type τ, the type inference rules will also show that e can be given a type τ. Note that τ is not necessarily the only valid type of e. It follows from this that if the type inference rules rejects any program that is not well-typed, the inference algorithm can not find the program to be well-typed.

- **Completeness**. Heeren says in his PhD thesis[11] that his type inferencer to be then not only sound, but also complete. He says that if there is a substitution that satisfies the constraints, that a substitution will be found. If the completeness property is not present, the system is not guaranteed to find any valid substitution that exists and might therefore reject a program that has a valid substitution. This property holds specifically for a type inferencer and it says nothing about the type system for which the type inferencer is designed.

- **Principality of types**. The principal type of a function is explained by Damas an Milner [2] as the type τ such that that every possible type that can be given to that function is an instance of τ. For the identity function, the principal type is $a \to a$, because every possible type for that function is an instance of $a \to a$, such as $Int \to Int$ or $Bool \to Bool$. This property of principal types, that holds in Haskell 2010, is broken by the introduction of GADTs. In case of GADTs, it is possible to construct functions that have multiple valid types, but neither of these is an instance of one another. Examples of these are given by Vytiniotis *et al*[29]. An example of a non-GADT program that might not have its principal type inferred:

  ```
  f x = let
    a = f "a"
    in x
  ```

  Both GHC and Helium infer the type of f as $String \to String$, but when the type signature $f :: a \to a$ is provided, both compilers give f the type $a \to a$. Therefore, we can conclude that $a \to a$ is the principal type of this function. It should be noted that the loss of principal type for this particular function is not caused by Haskell's type system, but rather by these specific implementations.

There are two ways a set of constraints in a constraint-based system can be inconsistent and that the constraint-system can therefore not find a substitution, as explained by Zhang and Myers[33]. The first reason is that there are constraints that make the type system inconsistent. Usually, this means that a constraint, that was generated based on the source code of the program, should not have been there. This happens when a programmer writes an incorrect expression, like a list with elements with different types. An example of this is that the system simplifies the constraints to the form $Int{\sim}Bool$, which results in an error. The second inconsistency is the absence of necessary constraints. One such example would be a missing class predicate, like *Show a*. This could happen in the example $f = show \circ read$[1]. In this example, the inferencer should reject the conditions, as there is not one single possible instance for *Read a* or *Show a*, but many possible instances.

2.3.3 TOP

TOP is a type inferencing framework, developed by Heeren [11]. TOP is a constraint based type checker and inferencer that is used in the Helium compiler. TOP's main advantage is the improved quality of the error messages it can produce. It does this via a number of ways, which include type graphs (Section 2.4.2), specialized heuristics (Section 2.4.3) and specialized rules for DSLs (Section 2.4.5). TOP was developed for the Haskell 98 standard and lacks support for the features that are currently used in variants of Haskell, like GADTs, type families and multi parameter type classes. It does however have support for type classes and instances.
TOP has a number of options, that it can use to change which type error messages it produces in case a type error exists. Different constraint ordering, different constraint solvers and different heuristics are a few of these options. Beside the options that a programmer can specify, TOP also has a number of features that make it interesting for compiler developers. It allows for arbitrary information to be added to constraints. This information can then be used by the heuristics to determine the most likely cause of the type inconsistency. The ability to add arbitrary information gives a great amount of freedom to construct new heuristics, which might need some extra information. TOP can use type graphs for its type inferencing. Type graphs are a data structure to describe a constraint set, which are further explained in Section 2.4.2. As TOP's type graphs are quite sophisticated, it does not use the type graphs for normal type inferencing, but only when a type error is detected. It usually starts with a greedy solver, which tries to solve the constraints. If this solver fails, the type graphs are used to detect the exact reason for the type error.

2.3.4 OutsideIn(X)

OutsideIn(X) is a constraint solving framework, which is parametric in the X, which indicates that some parts of the framework can be specified by the language developers, allowing for new features which can be added later on. The framework, theorized by Vytiniotis, Peyton Jones, Schrijvers and Sulzmann[29], has a number of changes with respect to other constraint solving frameworks. The largest change is that it allows for local constraints, that is, constraints that are only used and valid in certain parts of the program. This allows for features like GADTs, in which you want local constraints when pattern matching on such a GADT. Vytiniotis *et al* also gave an example instance of X, in which they added support for type classes, GADTs and type families. Their framework is currently used in the GHC compiler.

The introduction of local constraints also impose a problem, as it breaks a number of features we would like to see in a type inferencer. First, they do not generalize let bindings, as it interferes with GADTs. This breaks certain **let**-bindings which rely on the generalization of the **let**-binding. Second, they lose the principality of type, as they support GADTs and GADTs cause a loss of principality of type. Secondly, it loses the completeness property, as there are valid substitutions for certain functions, but the framework will not infer those, as there are multiple different valid possibilities. They argue however that neither of these problems are major problems, as they occur only in special situations and they can be fixed by providing a valid type signature to the part of the program that causes a problem.

The type solver judgement has the type $\mathcal{Q}; Q_{given}; \bar{a}_{touch} \vdash\!\!\blacktriangleright^{solv} C_{wanted} \rightsquigarrow Q_{residual}; \theta$. In this case, the

[1]In GHC, this program compiles, as there are defaulting rules for *Show* and *Read*

solver produces a set of residual constraints $Q_{residual}$ and a substitution θ, if the given constraints are correct. Otherwise, an error is returned. It creates the result from 4 parts, the top-level axioms $\mathcal{Q}$, which might contain things like available instances, the given constraints Q_{given}, which represent annotations like type-signatures, the touchable variables $\bar{a}_{touch}$, which are all type variables that can be unified, and the wanted constraints C_{wanted}, which are collected based on the given program. The judgement $\vdash\!\blacktriangleright^{solv}$ is based on simplifying and verifying the result. The reason that there are residual constrains returned beside the final substitution is that the possibility exists due to the local constraints that a constraint is not yet satisfied, but might be satisfied when other constrains arise. In that case, the solver won't make the judgement of accepting or rejecting the result, but will give the residual constraints back.

2.4 Type error diagnosis

Type error diagnosis is the process of explaining why a type error occurred and helping the programmer locate and fix the type error. This process usually has 3 phases, blaming, reparation and explanation. Blaming is deciding which parts of the program are responsible for the error, as explained by Serrano Mena[26]. The reparation phase tries to suggest ways to repair the error. This can be by removing or changing a constraint or adding a new constraint. The final phase is explanation, in which the error and possibly a solution for reparation is given to the programmer to explain why the program is not correct.

2.4.1 Properties of good error messages

Yang *et al*[32] gave 7 criteria which should be respected when designing error messages for a type inference system.

1. **Correct**: An error should only be given by if and only if the program is incorrect. This property is one of the few that is not open to interpretation and discussion, as what is considered a correct program is usually exactly defined. Most type systems, including those mentioned in this proposal, have a soundness proof, which guarantees that an incorrect program is not accepted. In case of GADTs, the loss of principality can cause situations in which a guess-free type inferencer cannot infer the type and therefore produces an error, even though the program is correct and adding a type signature would fix the problem. A guess-free solver is a solver that does not make assumptions about the types involved. An example is if we have the constraint $Eq\ a$. If a solver would make add to the substitution $a \sim Int$, we could discharge the constraint. However, if we have no information whether a is Int, the solver makes the guess that a is equal to Int. A guess-free solver is a solver that does not guess about the types about which it has no information.

2. **Precise**: Only include those parts that are relevant to the error. It follows from this that you would try to report the smallest possible location that is responsible for the error.

3. **Succinct**: This could be rephrased as: make the type error not to long and not too short. If the type error is too long, this would result in unnecessary reading and difficulty on the part of the programmer in figuring out what the actual problem is. On the other hand, the error should also not be too short. It should be clear that the error message "An error occurred somewhere in your program, sorry", is not sufficient for the programmer to actually find what error occurred. The ideal length is hard to justify and should ideally depend on the skill of the programmer, as novice programmers would require much more explanation than an experienced programmer who made a typo.

4. **Amechanical**: This means that the underlying process of the compiler should not influence the error message. An example of this happens when the following code is checked by GHC:

```
data X a where
  A :: Int → X Int
f (A x) = x
```

Part of the resulting error reads:

```
Couldn't match expected type `t' with actual type `Int'
       `t' is untouchable
         inside the constraints: t1 ~ Int
```

This example introduces two new type variables, t and t1, both of which were not present in the original code and show how the inferencer checks the type. Beside the new variables, the report says that t is untouchable. To understand what that means, the programmer should have knowledge about the underlying type inferencer and how it works.

5. **Source-based**: The error should be reported on the original source code that the programmer typed. This goes from extreme cases of changing the source, like rewriting a **do**-notation to a series of binds, but is also relevant for small changes (like changing the indentation of the code when printing).

6. **Unbiased**: The error should not depend on any particular order of inferencing of constraints. The example that Yang *et al* showed, $f\ x = (x\ 1, x\ True)$, should not point out either $x\ 1$ or $x\ True$ as a mistake, but report both and let the decision which one to blame to the programmer. If either one of these is reported without the other, we would call this a bias, as discussed in the section about Algorithm W.

7. **Comprehensive**: Ideally, every place the error occurred should be reported and not leave any relevant parts out. An type error slicer can be used to accommodate this (See Section 2.4.4).

2.4.2 Type graphs

Type graphs are an advanced data structure that is used by TOP to improve the quality of error messages by Hage and Heeren[6, 9, 11]. It is constructed from a set of equality constraints and applications of types. In a type graphs, the vertices are formed by type variables or applications of constructors (including the constructor $(\to)$). The edges between vertices represent a constraint between the vertices. The main advantage of a type graph is that it can represent a substitution that is inconsistent. Therefore, it can be used to try to change the type graph and make the substitution consistent. The decision which edges to remove is determined which error is reported by the solver. One downside of type graphs is their computational overhead. Therefore, the method TOP uses is it starts with a greedy solver, which calls the type graphs solver on the relevant part when it encounters an error.

The type graphs in TOP are created from a set of equality constraints. Every vertex in a type graph is either a type variable, like v, a type constant, like Int or a type application, like $v_0 v_1$. In this situation, a function $a \to b$ would be represented as $(((\to)\ a)\ b)$. Top uses 3 kinds of edges. The first is a directional edge, used for type application, going from the application to the child and are annotated with the l or r, depending on whether the child is on the left of the right or the right of the application. The other two edges are both undirected and both represent equality. The first is called the initial edge and goes between two vertices which have an equality constraint in the constraint set. The other edge is between two vertices which are equal, but not directly connected by an initial edge. Take the constraints $v_0 \sim v_1, v_1 \sim v_2$. In this case, there are three vertices, v_0, v_1 and v_2. There are two initial edges, one between v_0 and v_1 and one between v_1 and v_2. After this, there is also an implicit edge created between v_0 and v_2, because of the transitive property of equalities. If we would only consider a graph with only the initial and implicit edges, we would have a graph with a number of connected components, cliques, where all vertices should have the same type. This clique is called a equivalence group.

As said previously, a type graph can represent an inconsistent substitution. Such an inconsistency exists if there is a path along initial and implicit edges between two distinct constants, like Int and $Bool$. Such an error path can visit every vertex only once. The way to make the substitution consistent is to remove an edge in the error path until there are no more error paths. This will always result in a consistent substitution. The difficulty lies in deciding which edge (or constraint) to remove. It might seem optimal to always remove the least number of constraints, as this results in the least number of error messages, but this is not always the case. Some error messages are more likely to point out the actual cause of the error and in that case, we want to prefer them over less likely errors.

2.4.3 Heuristics for TOP

TOP is a general framework that can be used to infer types for any language, not just for Haskell. The problem with that approach is that TOP was not designed to work on specific language constructions. The solution is to add custom information to every edge in the type graph, specified by the compiler that uses TOP. This compiler can also provide a number of heuristics that use that specific type information to aid the type graph in making a decision which edge to remove. Beside these edges, these heuristics can also be used to try to repair the error and report the possible reparation as a hint. The heuristics that help to decide which edge to remove, can either give a score on how likely they judge the edge to be likely for the mistake or can exclude certain edges altogether. In the sections below, we will give a number of heuristics that are implemented in Helium, to give an overview of the possible heuristics. All these heuristics are described by Hage and Heeren[6].

Participation Ratio Heuristic

This heuristic can be used if an edge is part of multiple error paths. In the case that this happens, it is more likely that this constraint is responsible for the error, as we have a stronger indication that solving this particular error removes a number of problems. This idea is supported by Zhang and Myers[33]. Take the example for the constraints $[Int \sim \alpha, Int \sim \alpha, Int \sim \alpha, Char \sim \alpha, \beta \sim [\alpha]]$, which represents a list expression with three integers and one character. In this case, we could remove the three edge that involve the Int or remove one edge that says that α should be $Char$. As we have more evidence that the type of the list should be $[Int]$, we would like to remove the edge responsible for $Char \sim \alpha$. This heuristics is also responsible for limiting the number of errors that can occur at a single location in the source, but it is not the case that this heuristic will strictly prevent multiple error messages in a single location.

Trust Factor Heuristic

The trust factor heuristics assigns a trust value to every edge, based on how sure we can be that a constraint is right. There are constraints of which we can be certain they are correct and there are constraints who have a higher certainty value because of some indication. One such indication is a type signature provided by the programmer. We cannot rely on this type signature, but we can use it as an indication that the chance is higher that the mistake is in another constraint. In some cases, Hage and Heeren use **let** as an example, we want to be absolutely certain that the constraint that binds the type of the body of a **let** is the same as the type of the result of the entire **let** expression. This is a constraint that should never be blamed, as we know that this constraint is true. Other constraints that should not be questioned by TOP are the definitions of the Prelude, which are assumed to be defined correctly.

Sibling Function Heuristic

The sibling function heuristics tries to provide a hint by suggesting a different function that fixes the type error. Take the case of $maximum\ 3\ 5$, which is actually wrong, but it is not clear immediately to a programmer what is wrong with this expression. In this case, the expression should be $max\ 3\ 5$, as $maximum$ works on finding the maximum value in a data structure, not to choose between two values. The heuristic is given a list of siblings, which might be provided by the compiler developer or, in the case of Helium, by a file that any programmer or designer of a DSL can specify. This list is then used to check if a certain function in the expression can be changed to its sibling and whether that sibling would make the expression type correct. If this is the case, the sibling can be returned to the programmer as an additional hint. Another example of a set of siblings, beside $[maximum, max]$ is $[+, \circ, ++]$, which are all different versions of string concatenation in different languages and a programmer might get them confused at times. There also exists a heuristic that tries to change the type of literals, such as trying `'a'` instead of `"a"` or 3 instead of 3.0, to suggest fixes in a similar way to the sibling functions.

Permutation Heuristic

The Permutation Heuristic is an heuristic that tries to find a mistake when the correct arguments are supplied, but in the wrong order. If we would defined the functions $sum = foldr\ 0\ (+)$, we would like the compiler to suggest us to reorder the arguments, as the function $foldr$ has type $(a \to b \to b) \to b \to [a] \to b$. As there is a large number of possible ways the heuristic can reform the type and constraints, we want to limit the possible time or resources that can be spend repairing the type error. Algorithms for finding permutations or type isomorphisms are given by McAdam[18].

2.4.4 Type error slicing

Type error slicing is the process of determining which parts of the program contribute to the type error, as described by Schilling [22] and by Haack and Wells[4]. They each describe a system that can identify all possible locations that contribute to a type error. It does this not by working on the constraints, but directly working on the Haskell source code of the program. When a type error occurs, some parts of an expression are replaced by $\perp$ with the type $\forall a.a$, which always type checks. Schilling gave the example of the program $f_1 = \lambda x_2 \to length\ (x_4\ \texttt{'a'} + x_5\ [True])$. In this case, the type error occurs because x_2 is monomorphic and it is used with different parameters. The problem is that it is difficult to point out the location of the mistake. Both GHC and Helium point out the expression $[True]$ as the mistake, as they both infer the type of x to be $Char \to [a]$. The problem with that approach is that if we swap the parameters of $+$, the type of x will be $[Bool] \to [a]$. In this case, there are 3 locations that the function can be fixed, namely x_2, $x_4\ \texttt{'a'}$ and $x_5\ [True]$. The approach by Schilling is to replace an expression by $\perp$ and check if that type checks. If the type error is fixed by $\perp$, that location is recorded in the list of locations to be reported for this error. Another approach is by Pavlinovic, King and Wies[20], where they don't work on the direct source. They give every possible place that contributes to the error a weight and they minimize the weights, such that only errors that are relevant and require the smallest number of changes are reported. This approach could be used using a constraint based system.

2.4.5 Type error diagnosis for DSLs

Domain Specific Languages or DSLs are languages that are restricted to a specific domain. These DSLs can be embedded in existing programming languages, such as Haskell, in which case they are called embedded DSLs. The advantage of embedding in a DSL in an existing language is that functionality can be reused to create the DSL. A few functionalities that can be reused are features like code generation, optimization and, most important for this thesis, type checking. There is a problem with the type checking for DSLs. Usually, a DSL is embedded as a library. In such cases, if a type error occurs, the underlying implementation of the DSL is revealed. Ideally, an error message would be displayed, specifically designed for that specific error in the DSL. This process is described by Serrano Mena in his PhD thesis[26].

Type class directives

Hage and Heeren [11, 8] describe a number of type class directives that can be used to guide type errors for DSLs that use type classes. They propose four different type class directives, the never directive, the close directive, the disjoint directive and the default directive. The never directive indicates that a specific instance can never occur, such as $Num\ (a \to b)$. The close directive specifies that all the type classes for this specific instance are known and that no new instances can be provided. The disjoint directive indicates that a type can never be an instance of multiple classes at once, such as $Integral$ and $Fractional$ or $2D$ and $3D$. Sometimes,

the DSL designer wants an instance to be either *2D* or *3D*, but never both. In such cases, a custom error message, that is given with the directive can be generated to explain why this error occurs. In the case of $Num\ (a \to b)$, it could be "*A function can never be a instance of the Num class*" as error message.
Finally, we have the defaulting directive, which can be used to guide the type inferencing. In the case of $show\ [\,]$, it is unknown which instance of $Show$ should be chosen. If we had chosen $[\,] :: [Int]$, the result would be `"[]"`, but if we had chosen $[\,] :: String$, the result would be `""`. Both of these would be valid choices and therefore, we would like that the programmer state explicitly which instance to use[2]. In the default directive, we can give the type inferencer possible instances that it can try in order to remove the ambiguous predicate. This would be useful if would have a function $empty :: a$ in the class $DSLSet\ a$. If $empty$ was used without context from which to infer the type, the default directive could try to find a valid instance for $DSLSet$.

Specialized type rules

Helium allows for specialized type rules to be added. Specialized type rules are custom rules that are checked by the type inferencer whether they hold at compile time. Because they are not coded into the type inferencer by a DSL designer, but provided externally, they can be checked for soundness and completeness before they are used. The following example is based on the test *typeerrors/Strategies/Duplicated1.hs* from the Helium compiler.
This code introduces the (+++) operator, which can be used to add two variables of type Int.

$(+\!\!+\!\!+) :: Int \to Int \to Int$
$x +\!\!+\!\!+ y = x + y$

The following specialized type rule can be defined for Helium, to ensure that both sides are in fact of type Int. If for example, variable x is not an Int, instead of a generic error message, our custom error message will be shown.

```
  x :: a ; y :: b ;
  -------------------------
        x +++ y :: Int;;

 a == Int : "Left hand side should be an Int"
 b == Int : "Right hand side should be an Int";
```

It is also possible to add rules rules involving more complicated type rules, such as the following example, based on an example by Hage, Heeren and Swierstra[11, 9]. This example describes custom typing rules for the (<$>) operator.

```
 a == Parser s1 x1 : "Variable on the left is not a Parser"
 b == Parser s2 x2 : "Variable on the right is not a Parser"
 s1 == s2 : "The tokens of the parsers are not the same type"
```

In this case, the first rule is checked. If this gives an error, the error message is shown, otherwise the next rule is checked. Therefore, if the rule `s1 == s2` is checked, we know that both `a` and `b` are parsers and therefore, we can specialize our error message on that.

Error customization

This section is based on work by Hage and Serrano Mena[25, 24]. He gave a number of improvements over existing type errors. One of these is the difference between **Type 'a' does not support equality** and **Type '[a]' does not support equality**. If we have a missing class predicate $Eq\ [a]$, but we have the axiom $Eq\ a \Rightarrow Eq\ [a]$, we would like to report the first error message, as giving an instance for $Eq\ a$ would resolve the problem. This error message could be improved further, as we could show why we would want the instance $Eq\ a$. The error message would read:

[2]In this case, Haskell has defaulting rules for a number of default classes including *Show* and therefore, this expression would not be rejected, as *Show* is a special case

```
There is no instance for `Eq a`
needed for the instance `Eq [a]`
```

This error message is extensible when further derivations need to be made. It is up to the type inferencer to keep track of these derivations. This system could be extended for type equalities, to show to the programmer why two types need to be equal according to the compiler. If we would have the parser example from Section 2.4.5, we would be able to produce an error that said:

```
Tokens `Char` and `String` are not the same
needed for `Parser Char x`
       and `Parser String x`
```

In this case, it is clear to the programmer why the two types should coincide and the programmer can investigate what the exact problem is. Other possibilities of customizing error messages can be done by customizing the constraints in the source code. Serrano Mena developed a system in which he allows DSL developers to give custom error messages in the types of a function. This is done by manipulating the GHC constraints in the Haskell source code, for example in a type signature. See the following example, in which we give the type signature for the parser combinator $(<*>)$:

$$\begin{array}{ll}
\multicolumn{2}{l}{(<*>) ::} \\
\quad IfNot\ (p_1{\sim}Parser\ s_1\ (a \to b)) & (TypeError\ \texttt{"First argument should be a parser"})\ (\\
\quad IfNot\ (p_2{\sim}Parser\ s_2\ a) & (TypeError\ \texttt{"Second argument should be a parser"})\ (\\
\quad IfNot\ (p_3{\sim}Parser\ s_3\ b) & (TypeError\ \texttt{"Second argument should be a parser"})\ (\\
\quad IfNot\ (s_1{\sim}s_2) & (TypeError\ \texttt{"Parser tokens are not equal"}) \\
\quad ...))) \Rightarrow p_1 \to p_2 \to p_3 &
\end{array}$$

One benefit of this technique over the heuristics discussed before is that the error messages can be given without using separate files or massive changes to the compiler. One example of manipulating these constraints is the $IfNot$ construct. This construct is given 3 parts, a constraint to check, an error message and a constraint to continue with if the first constraint holds. If the first constrain does not hold, the error message is reported.

Reparation and blaming

Serrano Mena[26] gave several techniques on trying to repair type errors. One of these techniques is called transformations, in which certain parts of the program are transformed, following the pattern $C_1, ..., C_n \rightsquigarrow C'_1, ..., C'_n \quad fix\ H$. This means that if the constraints $C'_1, ..., C'_n$ fixes a type error in the constraints $C_1, ..., C_n$, H will be reported as a fix. One such example is given as: $fn \sim a \to b \to c \rightsquigarrow fn \sim (a, b) \to c$ fix `arguments need to be uncurried`. In this case, if changing the function fn from $a \to b \to c$ to $(a, b) \to c$ fixes the problem, the given error message will be reported. There is also a transformation of the form $C \rightsquigarrow \top \quad fixM$ that can be used when removing a constraint fixes the type error. It is also possible to give a rank to a transformation, as we would prefer to choose a more specific transformation over a generic one, such as a removal. There are also other transformations possible, such as defaulting and using annotations to differentiate different functions with the same type signature. This would be useful in the case of (||) and (&&), as both have the type $Bool \to Bool \to Bool$, but the error message might be different.

3 Research questions

The main goal of my thesis will be extending the possible language constructions that Helium will be able to support, while still maintaining the same quality of error messages for the constructions that are already present. Ideally, we would also like to demonstrate the extended framework with heuristics for GADTs to show the practical usages of the improved framework. For this, we give a number of research questions.

Research question 1. *Is it possible to translate the heuristics available in Helium to the OutsideIn(X) framework, maintaining the existing qualify of type error messages?*

As these heuristics work on type graphs, we need to design a framework, both theoretical and as a prototype, that allows type graphs to work on the constraints of OutsideIn(X) and that provides the same guarantees in terms of soundness as the OutsideIn(X) framework. Ideally, we would like to let the heuristics work on these type graphs without any changes, but we expect to be making some small changes to these heuristics to let them work within the new framework, like working with touchable variables. It might also be possible that some heuristics will not work any more, because they rely on constraints or situations that are not available in OutsideIn(X).

Research question 2. *How can causes of type errors involving GADTs be discovered and explained by heuristics?*

Extending the existing heuristics to work on OutsideIn(X) is the main goal of my thesis, but we would like to show that the framework works with the new GADTs that were not available previously. By showing this, we also give a demonstration that the type graphs work with local constraints in the OutsideIn(X) framework, which is not possible in TOP.

4 Approach

4.1 OutsideIn(X) implementation in Helium

To enable the implementation of the Helium heuristics in OutsideIn(X), it is necessary to have an implementation of OutsideIn(X) work as a type inferencer for Helium. The OutsideIn(X) implementation Cobalt[19] has been implemented into Helium as part of this proposal to enable the research questions into the type error diagnosis.

The implementation is developed alongside the already existing TOP framework, which means that for most programs, the existing TOP implementation will be able to verify the results produce by OutsideIn(X). The problem with this approach is the different techniques used by the frameworks, which means that in some situations, the systems will produce different results for the same program. This difference extends so far that some programs might be rejected by one type inferencer while being accepted by the other type inferencer. As at the moment, the OutsideIn(X) implementation supports a subset of the types supported by TOP, but in future (when more advanced features, like GADTs, are added), the OutsideIn(X) framework will accept programs that the TOP framework will not be able to handle.
The biggest change between the two implementations at the moment is the issue of **let**-generalization. Consider the following example:

```
f x =
  let
    y z = z
  in (y x, y)
```

As TOP generalizes **let**-bindings, the resulting type of f is $a \to (a, b \to b)$, as the definition of y is separate from the usage of y in the **let** body. OutsideIn(X) does not generalize **let**-bindings, as explained by Vytiniotis *et al*[29]. The resulting type of y is $t \to t$, where t is not quantified, but the same type as the parameter x, such that x is also of type t. The result is that the type of this function in the OutsideIn(X) implementation is $a \to (a, a \to a)$, as the identity function only works on types of a. If we would add another function binding of the form $a = y\ 3$, even without every using a, this would force the type of y to be $Int \to Int$ and thus the inferred type of f would be $Int \to (Int, Int \to Int)$. This is also the reason that programs might be rejected by OutsideIn(X). If we would add a third **let**-binding like $b = y$ `'a'`, this would give as an inconsistency, as the type inferencer has to unify $Char$ and Int. It is still possible to generalize a **let**-binding in the OutsideIn(X) implementation and get the original behaviour from the type inferencer. This can be done by giving an explicit type signature for the **let**-binding, such as $y :: a \to a$. This would make sure that the function y would be generalized. Using this type signature, the resulting type would be $a \to (a, b \to b)$, regardless of whether the mentioned additional **let**-bindings $a :: Int$ and $b :: Char$ were added or not.

For the implementation of OutsideIn(X), we followed the basic overall structure as TOP. This basically means that we traverse the AST using an attribute grammar (uuagc[1]) and collect the constraints and other information necessary for the solver. The OutsideIn(X) solver wants 4 pieces of information. The first is the list of

[1]http://hackage.haskell.org/package/uuagc

axioms, which are passed as an inherited attribute through the entire AST. The axioms contain information about the available instances. The second piece of information are the given constraints. These are not passed through the AST, as these are only constructed at the location of solving. The other constraints, the wanted constraints, are however collected throughout the AST, as almost every node adds some constraints. Most of these constraints are instantiation constraints or unification constraints. Finally, we collect the type variables that we can touch during the unification process, touchables for short. These touchables are collected at all places where variables are introduced that can be unified. An example of a variable which cannot be touched and therefore be unified, are variables in explicit type signatures, like the type signature $a \to b$. In this case, if we would make a and b touchable, we could unify them with another type, like Int and this would not hold with the general type that is provided, that would work on any a and b.

For the solving phase, we modified the Binding Group Analysis (BGA) as proposed by Heeren[11]. The main reason for the modification is the lack of a generalization constraint. In **let**-bindings we solved this by removing generalization all together. However, this is not desired for top level declarations, as these are often required to be polymorphic. Therefore, we need to infer the type for every top-level declaration before we can use it. To get the type of the polymorphic function, we need to infer the type from the constraints and that means that we need to solve the binding group for that particular binding. As this is the case for every binding group, we infer the type of every binding group at top level and add the inferred type to our known type signatures. These type signatures can then be used to check the usage of top level bindings in other situations. We need one final solving phase, as the solving of a binding group might result in new constraints. Therefore, we collect all the residual constraints from all the solving processes, as well as all the new constraints that involve the type signatures we just inferred. All these constraints, as well as the substitutions are put through one final solving phase, resulting in the final judgement about our program. Therefore, if we have n top-level bindings, we solve constraints $n + 1$ times, once for every top-level binding group. After that, we solve one final time with the residual constraints and the resulting substitution as given constraints, as these already have been found to be correct.

During the answering of the research question, the described implementation will be changed and updated. One most notable feature that is not yet supported in Helium is the support for GADTs. Beside those, *Monad* definitions and the related **do**-notation are also not supported, as Cobalt currently does not support type variable application, which is necessary for *Monad* definitions. This will hopefully be resolved during my thesis. Finally, there are a few minor constructions currently not implemented in Helium, such as the negation pattern and enumerations, but these will hopefully be supported before the hand-in of my thesis.

4.2 Type variable application in OutsideIn(X)

In order to achieve the previous abilities of Helium, we would like to support type variable application of the form $f\ a$, where both f and a are type variables. Currently, the OutsideIn(X) implementation we are using, Cobalt by Serrano Mena[19], does not support type variable application. As this feature allows sophisticated features, like functors and monads, we would like to support this. Therefore, we will make an effort to implement type variable application into Cobalt. We will assume that the given program is kind correct, something which is guaranteed by the kind checking in Helium. Basically, we would like to support constraints of the type $f\ a \sim g\ b$ which becomes $f \sim g$ and $a \sim b$. Both f and g will be required kind $* \to *$ and a and b need to be of kind $*$. This will also allow us to remove the explicit handling of $(\to)$ and consider that a regular constructor. This would in theory open support for instances like **instance** *Functor* $((\to)\ a)$.

4.3 Research questions

In these sections, we will describe the desired approach of answering the research questions as laid out in Section 3.

4.3.1 Type graphs in OutsideIn(X)

The heuristics that currently exist in Helium are developed for the type graphs in TOP. The logical approach is to implement type graphs for OutsideIn(X), as the heuristics we try to get working with OutsideIn(X) are designed for type graphs. To be able to do this, we need to extend the type graphs to be able to work with the additional constraints OutsideIn(X) gives. The major changes come from the extensions OutsideIn(X) allows, most notably local constraints. These local constraints have to be fitted into the type graphs in a way that allows them to function correctly, but still allows the heuristics to do their work. Ideally, we would want to have a model for type graphs that works on the OutsideIn(X) framework, where the X is still parametrized and the type graphs would actually use that specific X. However, we will settle for a concrete instance for X that is given by Vytiniotis *et al*[29], if that seems more feasible that a generic solution. Regardless of which version is implemented, we still want to guarantee the same properties as OutsideIn(X) with respect to soundness, completeness and principality of type. Even though the first type checking will be done by Cobalt and the type graphs will only be used to detect which error has occurred, rather than if an error has occurred, we want to be able to use the type graphs as an independent system. This means that the type graphs cannot assume that it has an incorrect program, but must be able to type check correct programs. This includes returning a substitution and a set of residual constraints.

4.3.2 GADT heuristics

Giving an implementation of type graphs in OutsideIn(X) that is equivalent to the variant that already exists in TOP is not relevant. We want to show the extended possibilities that the new framework will provide and therefore, we want to develop one or more heuristics that are specifically designed to explain type errors caused by the introduction of GADTs. The main structure and functionality of these heuristics are part of the research that will be done, but we will lay out a few examples that cause an error, including the error message GHC[2] provides and a modified version we aim to achieve. Some of these error messages might already be supported by existing heuristics or we might be able to achieve the custom results by modifying existing heuristics. The type errors are ordered by priority, with the highest priority, that is, the error message we would like to implement, first.

Inaccessible pattern

The variable x in the function f should be of type Int, which follows from the type signature of A. We would like to show that, due to the type signature provided, pattern matching on the constructor A is not allowed.

```
data X a where
  A :: Int → X Int
  B :: X a
f :: X Bool → Bool
f (A x) = x
```

The GHC error message

```
Error2.hs:6:5: error:
    * Couldn't match type `Bool' with `Int'
      Inaccessible code in
        a pattern with constructor: A :: Int -> X Int,
        in an equation for `f'
    * In the pattern: A x
      In an equation for `f': f (A x) = x
```

[2]GHC version 8.0.2

The proposed Helium error message

```
(6:4) Pattern is not accessible
  pattern                : A x
  given pattern type     : Int  -> X Int
  inferred pattern type  : Bool -> X Bool
```

Missing predicate

In this case, the predicate *Show b* is missing. Normally, it would be possible to add such a predicate to the type signature of the function where it was used, but as there are no type variables in the type signature of $f :: (X \to String)$, that is not possible in this case. Therefore, we need to add the predicate to the constructor, such that the constructor becomes $A :: Show\ b \Rightarrow b \to X$ and the predicate *Show b* is available when pattern matching on A.

```
data X where
  A :: b → X
f (A x) = show x
```

The GHC error message

```
Error7.hs:4:11: error:
    * No instance for (Show b) arising from a use of `show'
      Possible fix:
        add (Show b) to the context of the data constructor `A'
    * In the expression: show x
      In an equation for `f': f (A x) = show x
```

The proposed Helium error message

```
(4:11): Missing instance (Show b)
  expression    : show x
  type          : b
  doesn't match : Show b => b
  hint: Add (Show b) to the data constructor A
```

Type error in branch

The variable x in function f is of type *Int* and can therefore not be used in combination with the boolean or-operator. The main problem with the GHC error message is that it reports two times at the exact same location that it expects both an *Int* and a *Bool* at the same location. We would like one single error message that states what the expected type is and that it does not match the inferred type.

```
data X a where
  A :: Int → X Int
  B :: X a
f :: X Int → Int
f (A x) = x || True
```

The GHC error message

```
Error3.hs:6:11: error:
    * Couldn't match expected type `Bool' with actual type `Int'
    * In the first argument of `(||)', namely `x'
      In the expression: x || True
      In an equation for `f': f (A x) = x || True

Error3.hs:6:11: error:
    * Couldn't match expected type `Int' with actual type `Bool'
    * In the expression: x || True
      In an equation for `f': f (A x) = x || True
```

The proposed Helium error message

```
(6:13): Type error in infix application
  expression        : x || True
  operator          : ||
    type            : Bool -> Bool -> Bool
    does not match  : Int  -> Bool -> Int
```

Type signature and case do not match

As the **case** alternatives are correct in the function f, we want to blame the constraint that infers the type of x to be of type $Y\ Int$, as the type signature shows. We do not blame a particular part of the function, letting the programmer put blame on either the type signature or the case expression, which might need to be changed.

```
data X a where
  A :: Int → X Int
  B :: a → X a
data Y a where
  C :: a → Y a
f :: Y Int → Int
f x = case x of
  A z → z
  B y → y
```

The GHC error message

```
Error5.hs:10:5: error:
    * Couldn't match expected type `Y Int' with actual type `X t0'
    * In the pattern: A z
      In a case alternative: A z -> z
      In the expression:
        case x of {
          A z -> z
          B y -> y }

Error5.hs:11:5: error:
    * Couldn't match expected type `Y Int' with actual type `X Int'
    * In the pattern: B y
      In a case alternative: B y -> y
      In the expression:
        case x of {
          A z -> z
          B y -> y }
```

The proposed Helium error message

```
(9,13): Type error in scrutinee of case expression
expression          : case x of {A z -> z; B z -> z}
    term            : x
    type            : Y Int
    does not match  : X a
```

Bound to a rigid type variable

We declare the function f to be of type $Y\ Int \rightarrow c$, which implies that c is polymorphic. However, due to the body of f, we find that b is equal to c, but the type of b is not polymorphic. The type variable b is bound by the way the constructor C was used and is therefore an existential type, which does not match with the universal type of the type variable c.

```
data Y a where
  C :: b → Y Int
f :: Y Int → c
f (C x) = x
```

The GHC error message

```
Error6.hs:5:11: error:
    * Couldn't match expected type `c' with actual type `b'
      `b' is a rigid type variable bound by
        a pattern with constructor: C :: forall b. b -> Y Int,
        in an equation for `f'
        at Error6.hs:5:4
      `c' is a rigid type variable bound by
        the type signature for:
          f :: forall c. Y Int -> c
        at Error6.hs:4:6
    * In the expression: x
      In an equation for `f': f (C x) = x
    * Relevant bindings include
        x :: b (bound at Error6.hs:5:6)
        f :: Y Int -> c (bound at Error6.hs:5:1)
Failed, modules loaded: none.
```

The proposed Helium error message

```
(4, 11): Type signature is too general
expression          : x
  type              : c
  doesn't match     : b
  in constructor    : C :: b -> Y Int
The type variable c is too general, restrict the variable to be the same as the type b.
```

Existential type escapes

In the type signature of C, we declare the constructor to be of type $b \rightarrow Y\ Int$, which allows for anything of type b to create something of type $Y\ Int$. The problem with the function f is that we return this variable, which could be anything. If we would not reject this code, it would be valid to write something like

$f\ (C\ 3) :: Char$, in which we would require the result of the call to f to be of type $Char$, even though the resulting type would be of type Int. As the type b is not visible in the signature $Y\ Int$, the type of b is not allowed to escape the scope of the function f.

```
data Y a where
  C :: b → Y Int
f (C x) = x
```

The GHC error message

```
Error10.hs:4:11: error:
    * Could not deduce: b ~ t
      from the context: t1 ~ Int
        bound by a pattern with constructor: C :: forall b. b -> Y Int,
                 in an equation for `f'
        at Error10.hs:4:4-6
      `b' is a rigid type variable bound by
        a pattern with constructor: C :: forall b. b -> Y Int,
        in an equation for `f'
        at Error10.hs:4:4
      `t' is a rigid type variable bound by
        the inferred type of f :: Y t1 -> t at Error10.hs:4:1
    * In the expression: x
      In an equation for `f': f (C x) = x
    * Relevant bindings include
        x :: b (bound at Error10.hs:4:6)
        f :: Y t1 -> t (bound at Error10.hs:4:1)
```

The proposed Helium error message

```
(4, 11): Can't find a type for type variable b from constructor C
expression          : x
  type              : forall. b
  doesn't match     : exists. b
  in constructor    : C :: b -> Y Int
The type in the constructor represents an existential type, which cannot be used
    as a universal type in the expression
```

Multiple valid type signatures

In the following example, the function f requires a type signature, as there are multiple valid type signatures that this piece of code could have, all with different behaviour. We will not give a list of possible type signatures as a hint, as there could be very large amount of valid type signatures and we would like to report them all for the sake of completeness. It might also be impossible for certain types to infer the type signature.

```
data X a where
  A :: Int → X Int
  B :: X a
f (A x) = x
```

The GHC error message

```
Error1.hs:5:11: error:
    * Couldn't match expected type `t' with actual type `Int'
```

```
        `t' is untouchable
          inside the constraints: t1 ~ Int
          bound by a pattern with constructor: A :: Int -> X Int,
                   in an equation for `f'
          at Error1.hs:5:4-6
      `t' is a rigid type variable bound by
        the inferred type of f :: X t1 -> t at Error1.hs:5:1
      Possible fix: add a type signature for `f'
    * In the expression: x
      In an equation for `f': f (A x) = x
    * Relevant bindings include f :: X t1 -> t (bound at Error1.hs:5:1)
```

The proposed Helium error message

```
(5, 1): Cannot infer type of function binding of function f
  hint: Add a valid type signature
```

5 Planning

Below is the approximate planning for my thesis. The dates above each tasks will be the planned date that the task is due to be completed and behind that date is the amount of weeks available for that particular task. The final deadline for my thesis is Monday, 17 June 2019.

11 February - 1 week

We will need an implementation of GADTs to work with Helium. As Helium currently has no support for GADTs, we will need to add this support. For this task, we will add support up to type checking. This means the parsing and static checks for GADTs, as well as modifying the environments that exists to save the constructors.

25 February - 2 weeks

In this task, we will add support for type checking GADTs in the already existing Cobalt implementation. We expect this addition only needs to happen in the constraint gather phase, as Cobalt already has support for the constraints that are required by type checking.

11 March - 2 weeks

Changing the Cobalt implementation to work with type variable application will be done at this point. This would likely result in changing some of the inference rules and extensive testing whether the system still works. This would only require small changes to Helium, as Helium already supports type variable application. When time permits, we would also like to implement a few features that were still missing from Helium, due to the lack of type variable application in Cobalt. The largest of these changes involves the **do** notation.

25 March - 2 weeks

We will start on a separate implementation for type graphs for OutsideIn(X). Our aim for this 2 week period is having a type graph that can work on OutsideIn(X) equality constraints. This system will ignore all other constraints for the moment, but we will keep the other constraints in mind during the development process. We will try to reuse parts of the implementation of the TOP implementation, but the final system will be independent of TOP.

8 April - 2 weeks

Having finished the basis implementation, we will extend the framework with the other constraints that OutsideIn(X) supports, like classes, instances, instantiation and type families. We will however not yet consider local constraints and will only consider constraints that are in scope everywhere.

22 April - 2 weeks

We will add support for the local constraints, that is, constraints that are valid in some scope, but are not valid everywhere. This will be the major improvement, as this allows for features like GADTs to be supported by the type graphs. It will also constitute a major change in the working of type graphs, as we need to take the scope of constraints into account.

29 April - 1 week

The current heuristics probably need some changes to support the improved type graphs. We will make these changes and validate that the system still works as expected, that is, behave the same or equivalent to the old Helium compiler. At this stage, we won't consider any improvements like GADTs, but only consider programs that worked with Helium before.

13 May - 2 weeks

We will change the existing heuristics to work with the addition of GADTs and make sure the error messages they produce are still appropriate and correct when GADTs are involved. We will not yet consider heuristics that are specific for GADTs, just adapt existing heuristics for other cases to work when GADTs are involved.

27 May - 2 weeks

We will investigate which error messages for GADTs, as laid out in our approach, are not yet supported and we will try to design heuristics to ensure that the appropriate error message are displayed.

10 June - 2 - weeks

We will verify that the system works as before and also works with the addition of GADTs. We will also finalize any tasks that are not completed during this period, as we planned some extra time, when settling the deadlines for my thesis, for unforeseen circumstances.

17 June - 1 weeks

We complete our thesis and make sure every part of our thesis is ready for delivery, which include both the written thesis, as well as an implementation of the described work.

Bibliography

[1] Alexander Aiken and Edward L Wimmers. Type inclusion constraints and type inference. In *Proceedings of the conference on Functional programming languages and computer architecture*, pages 31–41. ACM, 1993.

[2] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212. ACM, 1982.

[3] Alex Gerdes, Bastiaan Heeren, Johan Jeuring, and L Thomas van Binsbergen. Ask-elle: an adaptable programming tutor for haskell giving automated feedback. *International Journal of Artificial Intelligence in Education*, 27(1):65–100, 2017.

[4] Christian Haack and Joe B Wells. Type error slicing in implicitly typed higher-order languages. In *European Symposium on Programming*, pages 284–301. Springer, 2003.

[5] Jurriaan Hage. Domain specific type error diagnosis (domsted), 2014.

[6] Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In *Symposium on Implementation and Application of Functional Languages*, pages 199–216. Springer, 2006.

[7] Jurriaan Hage and BJ Heeren. Ordering type constraints: A structured approach, 2005.

[8] Bastiaan Heeren and Jurriaan Hage. Type class directives. In *International Workshop on Practical Aspects of Declarative Languages*, pages 253–267. Springer, 2005.

[9] Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Scripting the type inference process. In *ACM SIGPLAN Notices*, volume 38, pages 3–13. ACM, 2003.

[10] Bastiaan Heeren, Daan Leijen, and Arjan van IJzendoorn. Helium, for learning haskell. In *Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71. ACM, 2003.

[11] Bastiaan J Heeren. *Top quality type error messages*. Utrecht University, 2005.

[12] Roger Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the american mathematical society*, 146:29–60, 1969.

[13] Ralf Hinze, Andres Löh, and Bruno C d S Oliveira. “scrap your boilerplate” reloaded. In *International Symposium on Functional and Logic Programming*, pages 13–29. Springer, 2006.

[14] Paul Hudak, Simon Peyton Jones, Philip Wadler, Brian Boutel, Jon Fairbairn, Joseph Fasel, María M Guzmán, Kevin Hammond, John Hughes, Thomas Johnsson, et al. Report on the programming language haskell: a non-strict, purely functional language version 1.2. *ACM SigPlan notices*, 27(5):1–164, 1992.

[15] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):707–723, 1998.

[16] Daan Leijen. Lvm, the lazy virtual machine. Technical report, Citeseer, 2002.

[17] Simon Marlow et al. Haskell 2010 language report. *Available online http://www. haskell. org/(May 2011)*, 2010.

[18] Bruce J McAdam. Repairing type errors in functional programs. 2002.

[19] Alejandro Serrano Mena. Cobalt. https://github.com/serras/cobalt, 2015.

[20] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Practical smt-based type error localization. *ACM SIGPLAN Notices*, 50(9):412–423, 2015.

[21] Simon Peyton Jones, Dimitrios Vytiniotis, Stephanie Weirich, and Geoffrey Washburn. Simple unification-based type inference for gadts. In *ACM SIGPLAN Notices*, volume 41, pages 50–61. ACM, 2006.

[22] Thomas Schilling. Constraint-free type error slicing. In *International Symposium on Trends in Functional Programming*, pages 1–16. Springer, 2011.

[23] Tom Schrijvers, Simon Peyton Jones, Manuel Chakravarty, and Martin Sulzmann. Type checking with open type functions. *ACM Sigplan Notices*, 43(9):51–62, 2008.

[24] Alejandro Serrano and Jurriaan Hage. Type error customization in ghc: Controlling expression-level type errors by type-level programming. In *Proceedings of the 29th Symposium on Implementation and Application of Functional Programming Languages*, page 2. ACM, 2017.

[25] A Serrano Mena and Jurriaan Hage. Context-dependent type error diagnosis for functional languages, 2016.

[26] Alejandro Serrano Mena. *Type Error Customization for Embedded Domain-Specific Languages.* PhD thesis, Utrecht University, 2018.

[27] Tim Sheard. Languages of the future. *ACM SIGPLAN Notices*, 39(12):119–132, 2004.

[28] Martin Sulzmann. A general framework for hindley/milner type systems with constraints. 2000.

[29] Dimitrios Vytiniotis, Simon Peyton Jones, Tom Schrijvers, and Martin Sulzmann. Outsidein (x) modular type inference with local assumptions. *Journal of functional programming*, 21(4-5):333–412, 2011.

[30] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad hoc. In *Proceedings of the 16th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 60–76. ACM, 1989.

[31] Andrew K Wright, Matthias Felleisen, et al. A syntactic approach to type soundness. *Information and computation*, 115(1):38–94, 1994.

[32] Jun Yang, Greg Michaelson, Phil Trinder, and JB Wells. Improving polymorphic type error reporting. 2000.

[33] Danfeng Zhang and Andrew C Myers. Toward general diagnosis of static errors. In *ACM SIGPLAN Notices*, volume 49, pages 569–581. ACM, 2014.