
Universiteit Utrecht

Faculty of Science

Dept. of Information and Computing Sciences


# Type error diagnosis for OutsideIn(X) in Helium


*Author*

J.J.F. Burgers
ICA-5545358

*Supervisor*

Dr. A. Serrano
Dr. J. Hage


June 25, 2019

**Abstract**

Rhodium is a modification to the OutsideIn(X) type inference framework, the type inferencer used in the GHC compiler. Rhodium allows for the usage of heuristics for advanced type error diagnosis. Rhodium is successfully implemented in the Haskell compiler Helium, which is known for its high quality type error messages. Using Rhodium, we extend the language with Generalized algebraic datatypes (GADTs). Rhodium has also support for type families and multi-parameter type classes, while preserving the high quality of the type error messages. We will discuss how the type inferencer uses type graphs and heuristics to generate these error messages.

# Contents


1 Introduction **5**
- 1.1 Relevance . . . 5
- 1.2 Summary of results . . . 6
- 1.3 Thesis overview . . . 6

2 Literature review **8**
- 2.1 Helium . . . 8
- 2.2 Generalized algebraic datatypes . . . 8
- 2.3 Type checking and inference . . . 9
- 2.4 Type error diagnosis . . . 12

3 Research questions **17**

4 OutsideIn(X) framework **18**
- 4.1 Introduction . . . 18
- 4.2 Simplification without local constraints . . . 19
- 4.3 Simplification with local constraints . . . 20
- 4.4 Solving constraints using OutsideIn(X) . . . 22
- 4.5 Instantiating the X . . . 22
- 4.6 Error collection . . . 23

5 Type graphs **24**
- 5.1 Introduction . . . 24
- 5.2 Type graphs . . . 24
- 5.3 Applying heuristics on type graphs . . . 25
- 5.4 Language-independent heuristics . . . 26
- 5.5 Language-dependent heuristics . . . 27

6 Rhodium: Type graphs for OutsideIn(X) **31**
- 6.1 Introduction . . . 31
- 6.2 Design requirements . . . 31
- 6.3 Type graphs . . . 32
- 6.4 Simplification of type graphs . . . 37
- 6.5 Solve result . . . 46
- 6.6 Summary . . . 47

7 Heuristics for Rhodium **49**
- 7.1 Collection of error paths . . . 49
- 7.2 Graph modifiers . . . 50
- 7.3 Application of heuristics . . . 51
- 7.4 Helium heuristics . . . 52
- 7.5 Residual constraint related heuristics . . . 57
- 7.6 GADT related heuristics . . . 59
- 7.7 Interaction between heuristics . . . 62
- 7.8 Summary . . . 66

8 Conclusion **67**

9 Future work **68**
9.1 Heuristics for language extensions . . . . . 68
9.2 Type inference for overloaded functions . . . . . 68
9.3 DSL support for Rhodium . . . . . 68
9.4 Optimizations to Rhodium . . . . . 68
9.5 Support for language extensions for Helium . . . . . 69
9.6 Specialized error messages for novice programmers . . . . . 69

# 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 [17]. As Haskell has such different use cases, the language has many different features that make the error messages confusing for people who just started functional programming [10]. 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 [35], type families [24] and GADTs [22] (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 once had to learn these new functionalities. Therefore, if something goes wrong with these advanced features, there should be specialized and clear error messages. Currently Helium [14] (Section 2.1) is one of the Haskell compilers that uses a number of techniques to improve type error messages, but these techniques are limited to a subset of the Haskell 2010 standard and lacked support for more advanced features that are not supported by the standard, let alone have specialized type error diagnosis for them.
This thesis focuses on the addition of GADTs to Helium, a feature outside the Haskell 2010 standard, using a modification of the OutsideIn(X) framework, which is a parametric type inference framework, also used in the GHC compiler. We also transferred the existing quality of error messages to the OutsideIn(X) framework and will discuss how we did this. OutsideIn(X) is a framework for type checking and inferencing, which will be discussed in detail in Section 6. We will extend OutsideIn(X) with type graphs, which is a data structure that allows for improved type error diagnosis, to allow support for type error diagnosis of 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 of that data structure, not the largest of two values. The error message provided by GHC, when written in GHCi, the interactive environment that comes 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 experienced 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 now state the type of *maximum* and the way it was used. It will also advice you that changing *maximum* to *max* might fix the problem, like in the error messages that Helium provides given the following code:

```
(4,5): Type error in variable
 expression                : maximum
   type                    : Ord a => [a] -> a
   expected type           :          Int -> Int -> b
 probable fix              : use max instead
```

## 1.2 Summary of results

We designed and implemented a new type inferencer for the Haskell compiler Helium. This type inferencer, which we called Rhodium, is a modified implementation of the type inferencer OutsideIn(X) with a parametric constraint solver. In Helium, we used an instantiation of $X$ that can handle GADTs, multi parameter type classes and type families, in addition to the features that were already available in Helium. It has support for the heuristics that exist in Helium by using type graphs to allow the heuristics to work on the type errors. We also added additional heuristics for the newly introduced GADTs to enable high quality error messages and to show that the heuristics can work on constructions like GADTs.
The name Rhodium was chosen as an homage to its direct predecessor, the OutsideIn(X) implementation with the name Cobalt [25]. To remain in the elements theme of Helium, Iridium [3] and Cobalt, we chose Rhodium. In the periodic system, the element Rhodium comes directly below Cobalt, indicating similar properties, which is appropriate as Rhodium is similar to Cobalt, but also has some unique aspects.

## 1.3 Thesis overview

In Section 2, we discuss the current state with regards to the literature, but we excluded a detailed discussion of OutsideIn(X) and type graphs, which will be discussed in Section 4 and Section 5 respectively. In Section

3, we discuss our research questions. In Section 6, we will discuss the design and implementation of Rhodium and in Section 7, the heuristics that work on Rhodium, as well as a detailed description about the Rhodium type error diagnosis process will be given. Finally, we will answer the research questions and conclude this thesis in Section 8, after which, in Section 9, we mention briefly some future work.

# 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. [4]:

> "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 5). 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[1] and type families. The frequency of usages of these extensions is reported by Hage [7], which indicates that they are indeed widely used. The backend of Helium, LVM [19], 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 [20] 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:

$$
\begin{array}{l}
\textbf{data}\ T\ a\ \textbf{where} \\
\quad T1 :: T\ Int \\
\quad T2 :: a \to T\ a \\
f\ T1 \quad\ \ = 0 \\
f\ (T2\ x) = x
\end{array}
$$

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

[1] GADTs were partially added as part of this thesis

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 inference described here and the implementation inside OutsideIn(X). OutsideIn(X) does not allow for type inference 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:

$$
\begin{aligned}
&\textbf{data}\ Expr\ a\ \textbf{where}\\
&\quad I \quad\quad :: Int \to Expr\ Int\\
&\quad B \quad\quad :: Bool \to Expr\ Bool\\
&\quad Plus \quad :: Expr\ Int \to Expr\ Int \to Expr\ Int\\
&\quad Equals :: Eq\ a \Rightarrow Expr\ a \to Expr\ a \to Expr\ Bool\\
&eval :: Expr\ a \to a\\
&eval\ (I\ i) \quad\quad\quad = i\\
&eval\ (B\ b) \quad\quad\quad = b\\
&eval\ (Plus\ x\ y) \quad = eval\ x + eval\ y\\
&eval\ (Equals\ x\ y) = eval\ x \equiv eval\ y
\end{aligned}
$$

The type checker ensures that the *eval* function can never fail due to the type checking. In this case, only expressions of type $Expr\ Int$ can be used to construct a $Plus$ expression. This knowledge is used in this case to prevent us from writing $Plus\ (B\ True)\ (B\ False)$, 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 $Equals$, we can use the $\equiv$ operator in the branch of $eval\ (Equals\ x\ y)$. It is noteworthy to add that this $Eq$ constraint only applies to the $Equals$ branch of the eval function. If we would have a general constructor of type $b \to Expr\ b$, 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 $Plus$ to a more general $Num\ a \Rightarrow Expr\ a \to Expr\ a \to Expr\ a$, if we would want to increase our expressiveness.
Other examples of GADTs are given by Sheard [32], Hinze et al. [16] and Peyton Jones et al. [22], who also provide a number of typing rules for a language which includes GADTs.

## 2.3 Type checking and inference

Type checking and inference 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 [2], that is an inference algorithm for the Hindley-Milner type system Hindley [15]. In this thesis, we 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 this thesis aims 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 and therefore a fixed order, 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 [18]. 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, namely having some bias in its type errors.

### 2.3.2 Constraint-based type inference

Constraint-based type inference 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, then 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 4) are the most relevant for this thesis, 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. [34].

- **Soundness**: Soundness of a type inferencer, as described by Sulzmann [33], is that any judgement made by the type inferencer can also be made in the logical system. Wright et al. [36] 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 $\tau$, the type inference rules will also show that $e$ can be given a type $\tau$. Note that $\tau$ 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 [10] that his type inferencer is 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 Damas and Milner [2] as the type $\tau$ such that that every possible type that can be given to that function is an instance of $\tau$. 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, neither of which is an instance of one another. Examples of these are given byVytiniotis et al. [34]. An example of a non-GADT program that might not have its principal type inferred is the following program:

```
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 tFype 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 [38]. The first reason is that there are constraints that make the types 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 $f = show \circ read$[2], where the type of $read$ is $Read\ a \Rightarrow String \to a$ and the type of $show$ is $Show\ a \Rightarrow a \to String$. 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.
It is possible to emulate Algorithm W, Algorithm M and a hybrid algorithm, Algorithm G [18], using a constraint based inference, using custom constraint ordering, as explained by Hage and Heeren [8]. 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.

### 2.3.3 TOP

TOP is a type inference framework, developed by Heeren [10]. 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 5), specialized heuristics and specialized rules for DSLs (Section 2.4.3). 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 occurs. Different constraint ordering, different constraint solvers and different heuristics are a few of these options. A *heuristic* is a function that can give an indication about the likeliness that an error has a specific cause, like too many arguments were applied.
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 additional information. TOP can use type graphs for its type inference.
Type graphs are a data structure to describe a constraint set, and which are further explained in Section 5. As TOP's type graphs are quite sophisticated, it does not use the type graphs for normal type inference, 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, a type graph is 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 X, which indicates that some parts of the framework can be specified by the language developers, allowing for new features which can be added

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

later on. The framework, theorized by Vytiniotis et al. [34], has a number of differences 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.

Local constraints are introduces whenever a pattern match on a GADT occurs. It will allow us to use the information provided by the type signature of the GADT constructor to make assumptions about the types involved in the function. Consider the following example, where the constructor $A$ has type $Int \to X\ Int$ and constructor $B$ has type $Bool \to X\ Bool$:

$f :: X\ a \to a$
$f\ (A\ x) = x + 1$
$f\ (B\ x) = x\ ||\ True$

In this case, the polymorphic type $X\ a \to a$ will change to the type $X\ Int \to Int$ during the first pattern match. Therefore, we introduce the constraint $a \sim Int$ which is placed in an existential constraint, as it should only hold for the constraints that are gathered from the first pattern match, but not for the constraints of the second pattern match.

The introduction of local constraints also imposes a problem, as it breaks a number of features we would like to see in a type inferencer. Firstly, they do not implicitly generalize let bindings, as it interferes with GADTs. This breaks certain **let**-bindings which rely on the generalization of the **let**-binding. Secondly, they lose the type principality, as they support GADTs and GADTs cause a loss of type principality. Thirdly, 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 $\mathcal{Q}; Q_{given}; \bar{a}_{touch} \mapsto\!\!\blacktriangleright^{solv} C_{wanted} \rightsquigarrow Q_{residual}; \theta$. In this case, the solver produces a set of residual constraints $Q_{residual}$ and a substitution $\theta$, if the provided 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 and available type families, 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 $\mapsto\!\!\blacktriangleright^{solv}$ is based on simplifying and verifying the result. The reason that there are residual constraints 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 constraints arise. In that case, the solver won't make the judgement of accepting or rejecting the result, but will give the residual constraints back. See also Section 4 for a detailed description of OutsideIn(X).

## 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 [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. [37] 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 thesis, have a soundness proof, which guarantees that an incorrect program will not be accepted. In case of GADTs, the loss of type principality can cause situations in which a guess-free type inferencer, which is a type inferencer that does not make any assumptions about types that is does not know anything about, cannot infer the type and therefore produces an error, even though the program is correct and adding a type signature would fix the problem. As mentioned, *guess-free type inferencer* is a type inferencer that does not make assumptions about the types involved. An example is if we have the constraint $Eq\ a$. If a type inferencer would add to the substitution $a \sim Int$, we could discharge the constraint. However, if we have no information whether $a$ is of type $Int$, the type inferencer makes the guess that $a$ is equal to $Int$. A guess-free type inferencer is a solver that does not guess about the types about which it has no information.

2. **Precise**: The error message should 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 message not too 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 be visible to the programmer in the error message. An example of this happens when the following code is checked by GHC:

   **data** $X\ a$ **where**
   $A :: Int \to X\ Int$
   $f\ (A\ x) = x$

   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 inference 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 call this a bias, as discussed in the section about Algorithm W (Section 2.3.1).

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 2.4.2).

### 2.4.2 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 [23] and by Haack and Wells [6]. 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 $f_1 = \lambda x_2 \rightarrow length\ (x_4\ \texttt{'a'} +\!\!+ x_5\ [True])$. 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 to the expression $[True]$ as the mistake, as they both infer the type of $x$ to be $Char \rightarrow [a]$. The problem with that approach is that if we swap the parameters of $+\!\!+$, the type of $x$ will be $[Bool] \rightarrow [a]$. In this case, there are 3 locations at which 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 et al. [21], where they do not 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.

### 2.4.3 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 [26].

#### Type class directives

Hage and Heeren [10, 11] 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 $Num\ (a \rightarrow b)$. 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 2*D* and 3*D*. Sometimes, the DSL designer wants an instance to be either 2*D* or 3*D*, 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 $Num\ (a \rightarrow b)$, 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 inferencer. 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[3]. 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 Heeren [10], Heeren et al. [13]. 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 Serrano and Hage [27, 30]. They 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

[3] 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.

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:

```
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 2.4.3, 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 made by customizing the constraints in the source code. Serrano and Hage [30] 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 (<*>):

$(<\!*\!>) ::$
$IfNot\ (p_1 \sim Parser\ s_1\ (a \to b))\ (TypeError$ `"First argument should be a parser"`$)\ ($
$IfNot\ (p_2 \sim Parser\ s_2\ a)\quad\ (TypeError$ `"Second argument should be a parser"`$)\ ($
$IfNot\ (p_3 \sim Parser\ s_3\ b)\quad\ (TypeError$ `"Second argument should be a parser"`$)\ ($
$IfNot\ (s_1 \sim s_2)\quad\ (TypeError$ `"Parser tokens are not equal"`$)$
$...))) \Rightarrow p_1 \to p_2 \to p_3$

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 constraint does not hold, the error message is reported.

### Reparation and blaming

Serrano [26], Serrano and Hage [31] 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 $C \rightsquigarrow \top \quad fix\ M$ 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 this thesis is extending the possible language constructions that Helium supports, while 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, which we will answer in this thesis.

**Research question 1.** *Is it possible to translate the heuristics available in Helium to the OutsideIn(X) framework, maintaining the existing quality 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 possible 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 this 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 OutsideIn(X) framework

## 4.1 Introduction

OutsideIn(X) is a type inference framework which was introduced by Vytiniotis et al. [34]. The X is a parameter that is passed to the framework and represents an external constraint solver that will be used during the type inference process. This means that OutsideIn(X) is a type inferencer for which a custom constraint solver can be used. This constraint solver can be used to introduce custom behaviour into the framework. GHC, one of the current Haskell compilers, uses an implementation of OutsideIn(X) for its type inferencer. GHC uses an instantiation of X that supports both type classes and type families. It also allows additional language extensions, which add custom constraint handling rules to the GHC constraint solver to enable these rules to make slight modifications to the type inference process. An example of this is a type checker for units of measure by Gundry [5]. It should be noted, that not all all features a language extensions could possibly want, like code generation, can be performed by extensions to the type system, but for this thesis, we are only interested in modification to the type inference process.

OutsideIn(X) is a constraint-based type inferencer. This means that OutsideIn(X) works on constraints that are gathered from the Abstract Syntax Tree (AST), which is a representation of the input source code. This gathering is done by traversing every node in the AST and converting that node into a set of constraints, which are combined into a single set of constraints. Consider the following program:

$f\ i = odd\ i$

If we were to convert this program into a set of constraints, the resulting constraints might look like the following constraint set:

| | | | |
|---|---|---|---|
| $f$ | $\sim$ | $i \to result$ | Function binding |
| $odd$ | $\sim$ | $input \to output$ | Application of $odd$ |
| $odd$ | $>$ | $Int \to Bool$ | Type signature of $odd$ |
| $input$ | $\sim$ | $i$ | Input of application |
| $result$ | $\sim$ | $output$ | Result of $odd$ and $f$ |

In this example, we will only consider a single set of constraints, ignoring touchable variables, given constraints or axioms that might be gathered during the gathering process and which are necessary for simplification.
After the constraints and any additional information are collected, the constraints are given to the type inferencer, OutsideIn(X) in this case. OutsideIn(X) simplifies the constraints. The simplification process might return an error, when an inconsistent set of constraints was passed. This error can be found when an constraint is either inconsistent, like $Int \sim Bool$ or when a constraint is residual. A *residual constraint* is a constraint that cannot be simplified further. Constraints that are included in the substitution are not considered residual. A common example of a residual constraint is a residual class constraint, like $Show\ a$. In this case, we cannot resolve the constraint any further and it is considered residual.

## 4.2 Simplification without local constraints

Figure 4.1: The definitions of types and constraints

| | | | | |
|---|---|---|---|---|
| Axioms | $\mathcal{Q}$ | ::= | $\forall[\bar{\alpha}].\ \xi \sim F\ \bar{\tau}$ | type families |
| | | \| | $\forall[\bar{\alpha}].\ Q \Rightarrow D\ \bar{\tau}$ | class axioms |
| Monotypes | $\tau$ | ::= | $\alpha$ | variables |
| | | \| | $\tau_1 \tau_2$ | type variable application |
| | | \| | A | constants |
| | | \| | $F\ \bar{\tau}$ | type families |
| Polytypes | $\sigma$ | ::= | $\forall\bar{\alpha}.\ Q \Rightarrow \tau$ | scoped types |
| Scoped constraints | $C$ | ::= | $Q$ | non-existential constraints |
| | | \| | $C_1 \wedge C_2$ | conjunction |
| | | \| | $\exists[\bar{\alpha}].\ (Q \supset C)$ | existential constraint |
| | | \| | $\epsilon$ | empty constraint |
| Constraints | $Q$ | ::= | $Q_1 \wedge Q_2$ | conjunction |
| | | \| | $\tau_1 \sim \tau_2$ | equality |
| | | \| | $\tau > \sigma$ | instantiation |
| | | \| | $D\ \bar{\tau}$ | class constraint |
| | | \| | $\epsilon$ | empty constraint |
| Type-family free monotypes | $\xi$ | ::= | $\tau$ where $\tau$ contains no type families | |
| Substitution | $\theta$ | ::= | $[\alpha \mapsto \tau]$ | |

The main process of type inference is done by simplification of the constraints, which are represented by $Q$ and $C$. The definition of $Q$ can be found in Figure 4.1. There are two types of constraints, given constraints and wanted constraints. *Given constraints* are constraints that were supplied by existing knowledge, like a type signature. For example, the constraint $Eq\ a$ is a given constraint if the type signature $Eq\ a \Rightarrow a \rightarrow a \rightarrow Bool$ is provided by the programmer. The *wanted constraints* are constraints that need to be verified. If we use the function *show* with the type signature $Show\ a \Rightarrow a \rightarrow String$, we have the wanted constraint $Show\ a$ that needs to be resolved. The OutsideIn(X) framework also requires a list of touchable variables. These *touchable variable* or *touchables* are variables that occur in the constraints and are allowed to be unified with a type. We will consider this constraint residual. There are cases in which we have a variable that we do not allow to unify with another type.
Consider the following program:

$f :: a \rightarrow a$
$f\ x = x\ ||\ True$

According to the type signature, $a$ should be a variable that cannot be unified with a type. Therefore, we consider the variable $a$ to be non-touchable and would reject the example program. We will have a constraint of the form $a \sim Bool$, where $a$ is non-touchable, that cannot be unified further. Other examples where touchable variables are used will be shown in Section 4.3.

The simplification process takes a set of given constraints $Q_g$, wanted constraints $Q_w$ and the touchables $\bar{\alpha}$ and produces a new set of given constraints, a new set of wanted constraints and a new set of touchables. It does this by applying the canonicalisation, interaction, simplification and top level react rules until none of the rules can be applied any more, after which the simplification is complete.

The simplification is done by canonicalisation, interaction, simplification and top level react rules. While canonicalisation and top level react take an atomic constraint and reduce it, the interaction and simplification rules both take two constraints at the time. The top level react rules also requires the axioms $\mathcal{Q}$ to function.

- Canonicalisation rules take a single constraint and simplify that single constraint to en equivalent form, like $a \to b \sim c \to d$ becomes $a \sim c \wedge b \sim d$ or $Int \sim Int$ becomes $\epsilon$. The canonicalisation rules can also introduce new touchable variables, return substitutions or return errors. An example of a constraint that produces an error is $Int \sim Bool$.
- The interaction rule takes either two given constraints or two wanted constraints and produces a set of new constraints, like $a \sim b \wedge c \sim a \to a$ becomes $a \sim b \wedge c \sim b \to b$. Notice how the original constraint is returned. The reason this is done is that the constraint $a \sim b$ might be necessary to interact with other constraints. If the constraint is not returned, only one reduction with the first constraint is possible.
- The simplification rule is another rule that takes two constraints, but this time one given constraint and one wanted constraint. This rule allows the simplification of the wanted constraint, without modifying the given constraint. It is therefore not necessary, in contrast to the interaction rule, to return the original constraint. If the given constraints is returned, the given constraint is added to the wanted constraints, which might be undesirable. An example of a useful application of the simplification rule is with the given constraint $a \sim Int$ and the wanted constraint $a \sim Int$, which would produces the new wanted constraint $Int \sim Int$, a constraint that can be reduced further by the canonicalisation rule.
- The last kind of rule is the top level react rule. This rule modifies an atomic constraint based on a set of axioms. This rule can be used to deal with for example type families or instances. Examples of axioms that deal with these constraints are $Eq\ Int$, $\forall a.\ Eq\ a \Rightarrow Eq\ [a]$ or $F\ Int \sim Bool$, where $Eq$ is a type class and $F$ is a type family.

The four rules above are all given by OutsideIn(X), where the X provides the behaviour of the rules depending on the given constraints. This means that new behaviour can be obtained by modifying the handling rules in X, without modifications to the OutsideIn(X) framework as a whole. A downside of this liberal approach to constraint handling is the lack of assumptions that can be made about the rules.
An example of an assumption we want to make, is that the interact rules only work on constraints that can actually interact with each other. The constraint $a \sim b$ will interact with $b \sim Int$, but usually not with the constraint $c \sim d$, as the constraint $c \sim d$ does not mention the variables $a$ or $b$. However, due to the parametric constraint solver X, there might be a situation that a constraint handling rule specified by X does allow these constraints to interact. As we cannot rule this situation out, we need to check all possible combination of constraints, including those combinations that seem unlikely. This might influence the performance, depending on the implementation.

## 4.3 Simplification with local constraints

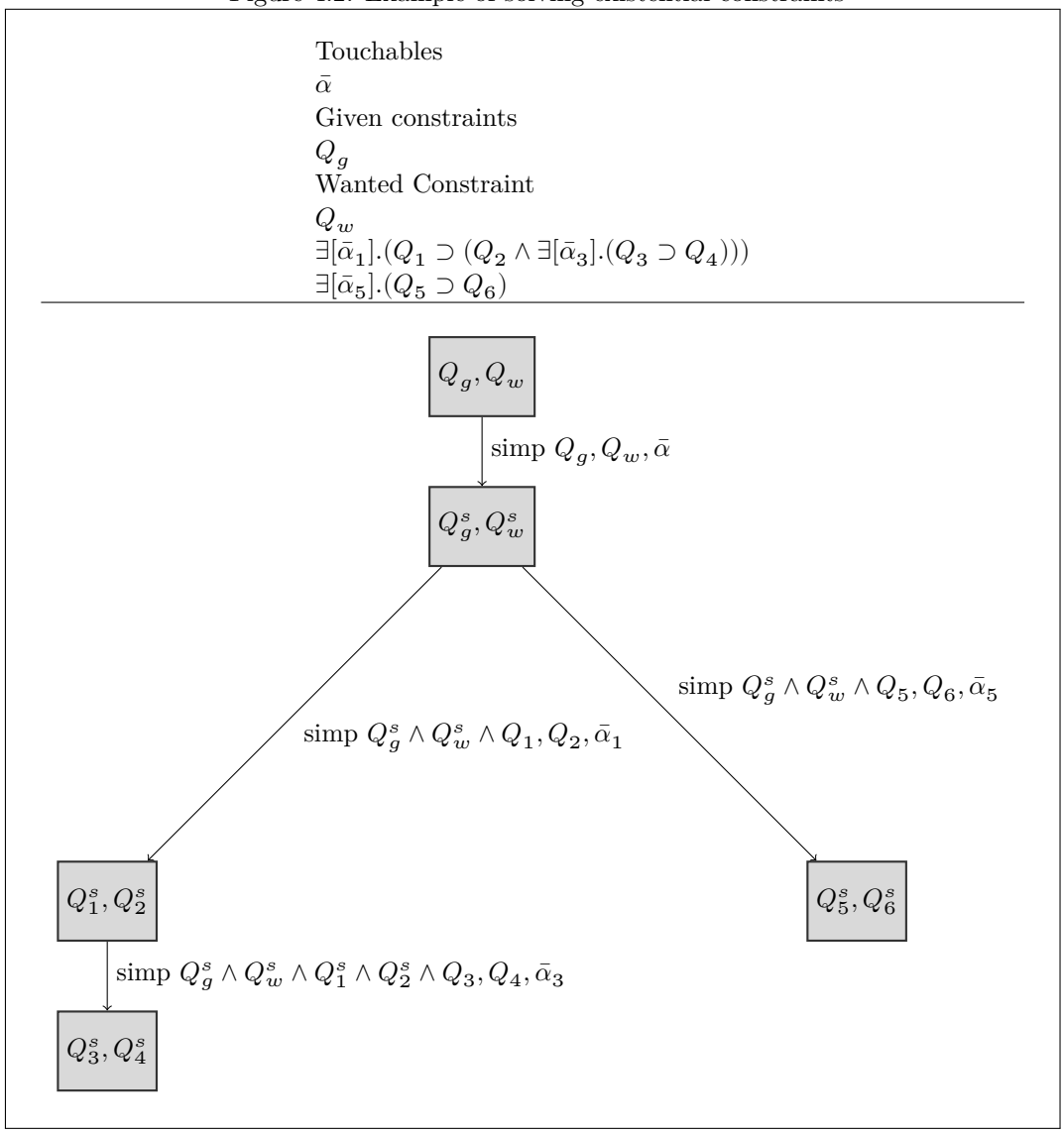
OutsideIn(X) allows for constraints that are only valid at a certain scope. We call these scoped constraints C, as defined in Figure 4.1. The constraint $\exists[\bar{\alpha}].\ (Q \supset C)$ describes constraints that are valid under a certain assumption. The constraint should be read as, the constraint $C$ is consistent under the assumption that the constraints $Q$ hold. $\bar{\alpha}$ describe all the variables that are touchable in that scope. These touchables are

collected from the variables that can be unified in the scope of the GADT, but do not include variables that occur outside the GADT. Consider the following example:

**data** $X\ a$ **where**
  $A :: (a \to b \to X\ a)$
$f :: X\ c \to c$
$f\ (A\ x\ y) = x$

In this example, the variable $x$, that has type $a$, is touchable, as it occurs in the result of the GADT type signature of $A$, but the variable $y$ is not touchable, as it does not occur in the result of the type signature of $A$. Even though $x$ is touchable, unifying $x$ with a constant like $Int$, produces an error. This is because the variable $c$ is considered non-touchable in this example.

Figure 4.2: Example of solving existential constraints



We solve constraints $C$ by splitting the constraints in two parts, a part without existential constraints and a part with existential constraints. First, the non-existential constraints are simplified according to the method described in Section 4.2. We will call these constraints $Q_2$. After this, we solve all the existential constraints one at a time. Assume we want to solve the constraint $\exists[\bar{\alpha}].\ (Q_1 \supset C)$. We assume we have the simplified constraints $Q_2$, which were simplified from the constraints that did not include existential constraints, both given and wanted. After this, we simplify the existential constraint, where $Q_1 \wedge Q_2$ are the given constraints and $C$ the wanted constraints, with $\bar{\alpha}$ as our only touchables. Notice that the existential constraint can have existential constraints inside it. If this occurs, we solve them using the same method. That is, solving all other constraints that do not have existential constraints, after which we solve the existential constraint, where the given constraints are all the constraints that were already solved, including those of other existential constraints. An example of the solving process can be seen in Figure 4.2. The constraints in the vertices are either those present at the beginning of the inference process or the result of the simplification process attached to the incoming edges. The resulting touchables of the simplification process are ignored in this example. The superscript $s$ denotes the simplified version of a set of constraints. Therefore, the notation "simp $Q_1^s, Q_2$" simplifies a set of constraints where $Q_1^s$, the simplified version of $Q_1$, are the given constraints and $Q_2$ are the wanted constraints.

## 4.4 Solving constraints using OutsideIn(X)

Now that we specified the way we are able to simplify these constraints, we need to formalise the solving process of OutsideIn(X). The main solving is done using the simplification process. As the simplification process can find inconsistencies, some of the type errors are reported by the simplification process. After this process, the residual constraints are collected. These are constraints that could not be resolved by the simplification and are not part of the resulting substitution. They might include equality constraints that contain non-touchable variables or class constraints that were not resolved. Only wanted constraints can be part of the residual constraints. The residual constraints might be original constraints, but they do not have to be. These might be simplified constraints that were found after simplification to be residual, like $Show\ a$ or $b \sim Int$ where $b$ is not touchable.

## 4.5 Instantiating the X

As explained, there are four kind of rules which make up the constraint solver X. Each of these has their own specific requirement, depending on whether they are given or wanted. These constraints can also be found in Figure 4.3.

Figure 4.3: OutsideIn(X) rules

| | |
|---|---|
| Canonicalisation | $canon[l](Q_1) = (\bar{\alpha}, \varphi, Q_2)_\perp$ |
| Interaction | $interact[l](Q_1, Q_2) = Q_3$ |
| Simplification | $simplify(Q_g, Q_1) = Q_2$ |
| Top level react | $topreact[l](Q, Q_1) = (\bar{\alpha}, Q_2)_\perp$ |

### 4.5.1 Canonicalisation

The canonicalisation rule is of the form $canon[l](Q_1) = (\bar{\alpha}, \varphi, Q_2)_\perp$, where $l$ is either $g$ or $w$, which stands for whether the constraint $Q_1$ is given or wanted. The result of the canonicalisation consists of three components.

The $\bar{\alpha}$ represents a set of new touchables that are introduced by the rule, $\varphi$ is a substitution that is applied to the constraints at the end and $Q_2$ is a conjunction of constraints, which might be empty. The canonicalisation rule might also fail. In case of the constraint $Int \sim Bool$, this constraint should fail and $\bot$ will be returned.

### 4.5.2 Interaction

The interaction rule is of the form $interact[l](Q_1, Q_2) = Q_3$, which represents the simplification of the atomic constraints $Q_1$ and $Q_2$ into the conjunction of constraints $Q_3$, which might be empty. The label $l$ has either the value $g$ or $w$ depending on whether the constraints are given or wanted. Notice that we cannot let two a given and a wanted constraint interact, we can only let constraints interact that are either both given or both wanted. $Q_3$ is added to either the given or the wanted constraints, depending on the label $l$, and the constraints in $Q_3$ may be used in other rules.

### 4.5.3 Simplification

Simplify rules are of the form $simplify(Q_g, Q_1) = Q_2$, where the given atomic constraint $Q_g$ simplifies the atomic wanted constraint $Q_1$, simplifying it to the conjunction of constraints $Q_2$, which might be empty. In all other rules, the constraints that are provided to the rules are removed from the simplification process. They can therefore be used only once. In the $simplify$ rule, only the wanted constraint $Q_1$ is removed, the given constraint can be used more than once. As the simplify rule always takes a single given and a single wanted constraint, there is no need to provide a label to the rule, as is required by other rules.

### 4.5.4 Top level react

The top level react rule is of the form $topreact[l](\mathcal{Q}, Q_1) = (\bar{\alpha}, Q_2)_\bot$ allows the axioms $\mathcal{Q}$ to to react with the atomic constraint $Q_1$, which is either a given or a wanted constraint. The label $l$ indicates whether the constraint $Q_1$ is given or wanted. The result of this rule is either a set of new variables that are touchable and a set of constraints $Q_2$ or an error, represented by $\bot$.

## 4.6 Error collection

The errors that can possibly originate from OutsideIn(X) are split into two parts. The simplest form is whenever the canonicalisation or top level react rule returns an error, represented by $\bot$. In this case, the simplification and therefore the inferencer fails. This happens at the moment the canonicalisation or top level react rule returns $\bot$. It is possible to continue with the simplification process after an error has been found. One advantage of this might be to detect additional errors.
The other case is whenever a constraint is considered residual. These residual constraints are collected at the end of each simplification step. Every constraint that is not a unification constraint is always a residual constraint. An example of such a residual constraint is $Num\ (a \to b)$. The other type of residual constraints are constraints that are unification constraints, but are not accepted in the substitution. They are rejected from the substitution, when the unification constraint is of the form $\tau_1 \sim \tau_2$, neither $\tau_1$ or $\tau_2$ is a variable that is touchable. The set of residual constraints is included in the result of the simplification process.

# 5 Type graphs

## 5.1 Introduction

A type graph is a data structure that allows a (possibly inconsistent) constraint set to be represented. This gives a type inferencer the advantage to inspect all the types to make an informed decision which part of the program to blame for a type error. Heuristics are used to properly blame the error, depending on the language that generated the constraints from which the type graph was constructed. The type graphs described in this thesis are based on the work by Heeren [10], Heeren et al. [13, 12]. Type graphs are a part of the TOP framework, which is the type inferencer that is used by the Helium compiler [14]. Helium also provide a large number of heuristics for detecting type errors. These heuristics are described in detail by Hage and Heeren [9]. A type graph is constructed from a set of constraints, like equality constraints, generalization constraints and instantiation constraints.

## 5.2 Type graphs

In this section, we will introduce the different shapes of type graphs and explain the ideas behind the error collection within a type graph, as well as introduce the notion of an error path.

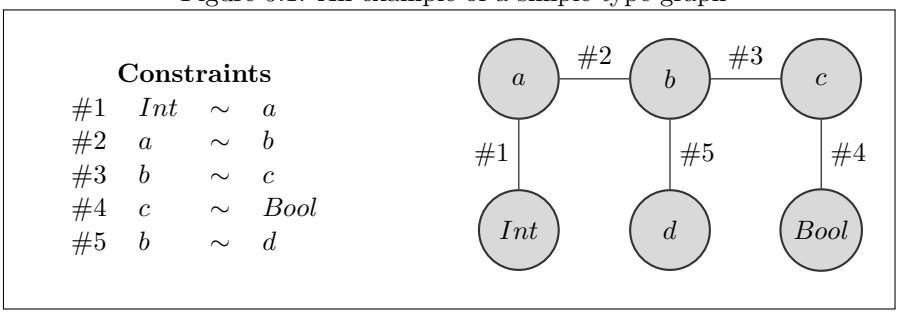
### 5.2.1 Simple type graphs

Type graphs consist of vertices and edges. In simple type graphs, edges represent equality between types and vertices represent simple types, which are either constants or variables. Types are denoted by the symbol $\tau$. An example of a type graph can be seen in Figure 5.1. In this case, the type graph has an inconsistency in the form of $Int \sim Bool$. At the moment, there are 4 constraints to blame, each of which has the same likeliness to be the cause of the error. To increase the likeliness of blaming the constraint that caused the inconsistency, we attach additional constraint information to the constraint, after which the heuristics can make an informed decision which constraint to blame.
The heuristics work on an error path. An *error path* consists of all the edges that are involved in the error. In the given example, the error path for the inconsistency $Int \sim Bool$ consists of the edges {#1, #2, #3, #4}. The edge #5 is not involved in the error and is therefore not included in the error path.

### 5.2.2 Extended type graphs

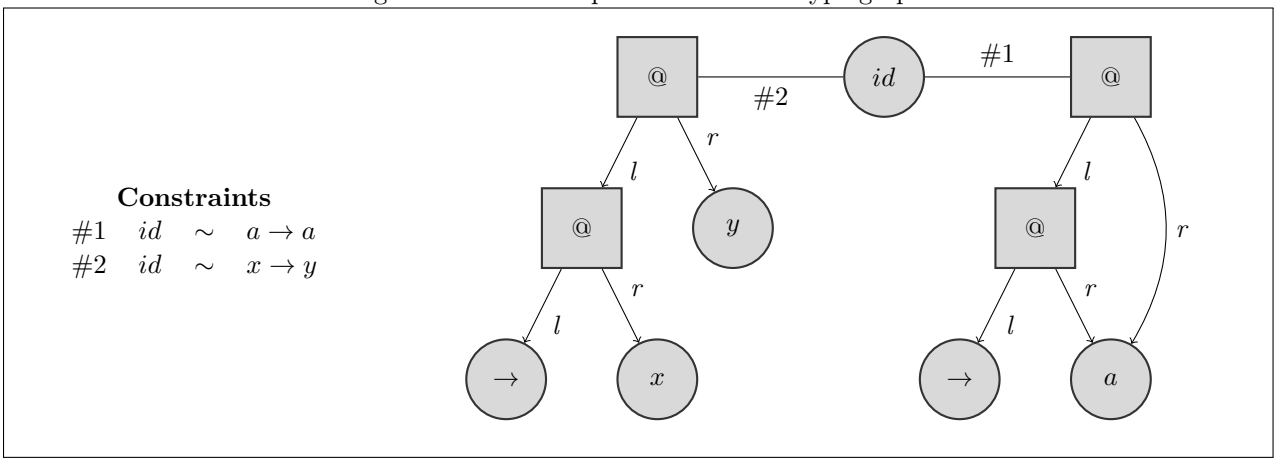
We can extend the simple type graphs from the previous section with type application. This will extend the types $\tau$ of type variables and constants with type application of the form $\tau_1\tau_2$, with the shorthand $\tau_1 \to \tau_2$ for $(((\to)\ \tau_1)\ \tau_2)$. Type application is visually represented by the square vertices with the `@` inside it. We will also abandon the convention that every vertex represents a type and every edge represents an equality. An vertex can either be a type variable, a constant of a type application. An edge can either be an equality or a

Figure 5.1: An example of a simple type graph




child edge.
Consider the example in Figure 5.2. The square vertices represent type application. The child edges, coming from the type application, are annotated either $l$ or $r$, representing respectively the left or the right child edge. In the given example, we can deduce that $a \sim x$ and $a \sim y$. From this, we can deduce that $x \sim y$, as we would expect in the case of the identity function. If some other constraints would specify that $x$ and $y$ should be different, like $x \sim A$ and $y \sim B$, this would cause an inconsistency.
Inconsistencies are found in type graphs by the introduction of derived edges. A *derived edge* is an edge that is created based on the equality between two vertices, but was not originally present in the constraints.
If we would have a type graph that described the constraints $a \sim A, a \sim B$, an equality edge would be created between the vertices $a$ and $A$ and between $a$ and $B$. After this, a derived edge is created between the vertices $A$ and $B$. As these constructors are different, but connected by a derived edge, we would notice this inconsistency, where two different constructors are connected by an edge, and an error would be reported.

Figure 5.2: An example of a extended type graph




## 5.3 Applying heuristics on type graphs

The main benefit of representing an inconsistent constraint set with a type graph is the ability to detect the cause of the inconsistencies. As said previously, having an error path that has no additional information is not that useful. Therefore, we apply heuristics that can access the additional information attached to the constraints to make an informed decision which edge to blame. Some of these heuristics can be provided by the type graph, but most will be language specific and need therefore be provided by the compiler that uses

the type graph, as the type graph is not designed to be language specific. This is also the case for TOP.
There are two types of heuristics that TOP supports to work on its type graphs. These are the filter heuristics and the voting heuristics. The filter heuristics is a heuristic that, given an error path, removes a number of edges from that path. These can be edges that should not be blamed, such as the constraint that says that the condition of an **if** statement should be of type *Bool*. Another filter heuristic might be a tiebreaker heuristics, that arbitrarily chooses an edge to blame if multiple edges are selected and there are no other heuristics left to apply. The filter heuristics have to take care to never remove all of the edges in the error path. Note that the filter heuristics only remove the edges from the error paths, which means that they will not be removed from the type graph.
The voting heuristics work on a different basis. Where the filter heuristics try to exclude certain edges from the error path, voting heuristics try to include certain edges in the error path. They do this by guessing the likeliness that an edge from the error path is the cause of the error, as every edge in the error path is still a candidate to be blamed. After all the heuristics have voted, the edges with the highest score are kept while all the other edges are removed. If none of the edges are selected by the voting procedure, all edges are kept and the next heuristic is tried. It should be noted that a number of voting heuristics are grouped in a heuristic group and that a heuristic group is treated as a single heuristic, just like the filter heuristic. This allows certain voting heuristics to vote together and compete against each other. In the case that a number of voting heuristics give the same score to multiple edges, all these edges are kept in the error path.
When all the heuristics are done, there should be a single edge left over in the error path. This edge is after this process blamed for the particular error and is then removed from the type graph. This reduces the number of errors and the next error, if any exists, is blamed using the heuristics. The removing of the edge that is found to be incorrect is done because the particular edge might be responsible for multiple inconsistencies. All of these inconsistencies disappear if the edge is removed and this prevents the system from blaming a single wrong piece of code in multiple errors. Consider the constraints $a \sim Int, a \sim b, b \sim Int, b \sim Bool$. In that case, if we detect that $b \sim Bool$ causes the error, we will remove the error and no errors are left in this type graph, even though two different error paths can be seen in the original constraint set. This process is continued until no errors are present in the type graph.

## 5.4 Language-independent heuristics

The language-independent heuristics are heuristics that do not depend on any constraint information and therefore do not depend on the programming language or its implementation.

### 5.4.1 First come, first serve heuristic

The first come, first serve heuristic, sometimes known as the tiebreaker heuristic is usually the last heuristic that is applied to the error path. This heuristics check whether the error path consists of a single edge. If this is not the case, the heuristic arbitrarily chooses an edge to blame. Having such a heuristic as the last heuristic is useful to make sure that there is only a single edge blamed for a single inconsistency in the type graph. As this is the last heuristic that should be applied, all other heuristics have already made their judgement and selected edges that they think are the cause of the mistake. Therefore, the edges that remain are edges that are likely closely related to the source of the error and are therefore all valid candidates.
One could argue that a tiebreaker heuristic should ideally not be necessary, as it is only called when the other heuristics are not sufficient to identify the exact cause of the error. On the other hand, in the cases that the tiebreaker heuristic is necessary, all the edges involved are causes of the error and each of them might be considered equally responsible for the inconsistency.

### 5.4.2 Participation heuristic

The participation heuristic is a heuristic that takes into account how often an edge occurs in all error paths to decide which edges to keep or which edges to discard. Let us assume that an edge was involved in a number of different error paths. This could be a strong indication that this particular edge was responsible for all of the error paths. We would therefore like to blame the edge that occurs more often than other edges. This increases the chance that an error is blamed which is involved in multiple errors, which in turn reduces the number of error messages that are reported about one particular program point. In the situation described above, it might seem that multiple error paths share a single edge. In practice, this is not the case. They usually share a number of edges, each of which is then kept to further identify the error. Other heuristics, which are language dependent, will then make sure that of the edges that remain, the error that is most likely for the inconsistency is blamed.

## 5.5 Language-dependent heuristics

The language dependent heuristics are heuristics that depend on the programming language from which the constraints originate. This means that the heuristics cannot come with the language-independent type inferencer, but have to be provided by the language specific compiler, as the type inferencer has no knowledge about the programming language that it is inferring the types for. The heuristics described in the following sections are all heuristics that are present in the Haskell compiler Helium, as described by Hage and Heeren [9].

### 5.5.1 Avoid unwanted constraint heuristics

There are a number of constraints that should never be blamed or are preferred to be blamed. Consider the following example:

$$
\begin{array}{l}
f :: \mathit{Bool} \\
f = \textbf{let} \\
\quad x = 3 \\
\quad \textbf{in}\ x
\end{array}
$$

A number of constraints are generated from this example. One of these is that the entire **let**-binding should have the same type as the body of the **let**-binding, the variable $x$ in this case, as we would never want to report to the user that the **let**-expression and the **let**-body should have a different type. Therefore, we would like them to be removed during the first heuristics. In Helium, they are removed by the avoid forbidden constraints heuristic.

#### Folklore heuristics

We also have the avoidance of so-called folklore constraints. These are constraints that are commonly accepted as right, such that the condition of an **if**-expression is of type *Bool*. The order in which these heuristics are applied matters greatly. For example, we want to exclude the forbidden constraints as soon as possible from the blaming process, as they should never be blamed for an error. However, the folklore constraints, we do later on in the blaming process, which allows for some of the specialized heuristics to vote on certain error messages and allowing them to reduce the error path.

#### Trust value heuristics

We can also assign trust values to constraints, with which we can prefer to blame constraints with a lower trust factor. An example of a constraint with a high trust value is a user provided type signature.
It is important to note that all these heuristics never remove all the edges, there will always some edge left in the error path after the heuristic is applied. If we would apply the folklore heuristic on an edge with only folklore constraints left, the original edge without modifications is returned, as the heuristic cannot reduce the error path in any meaningful way. In such a case, we apparently want to blame a folklore constraint and another heuristic tries to shrink the error path.

### 5.5.2 Sibling heuristics

*Siblings heuristics* are heuristics that will try to find a solution to an error by trying similar pieces of code, like changing a function from $foldr$ to $foldl$ or 3 to 3.0. There are 2 sibling heuristics present in Helium. The first one will try a literal sibling and the second one will try a function sibling.

#### Literal siblings

The literal sibling heuristic will try to explain errors in cases where a programmer typed the wrong literal, such as 3.0 of type *Float* instead of 3 of type $Int$ or `'a'` instead of `"a"`. It does this by specifically looking for cases in which an $Int$ and *Float* or *String* and *Char* are confused, as these are the only literals that Helium supports. If such a mistake is found, the programmer is recommended to change the type of the literal, as long as the type error originated from a literal.

#### Function siblings

The function sibling heuristic works on a similar basis as the literal sibling heuristic, but with the change that the candidate substitutions are defined outside of the heuristic. In the sibling heuristic, only four possible cases were addressed. In the case of the sibling function heuristic, the possible siblings are given by the programmer or compiler designer and can be passed to the heuristic as a parameter. Take the example of *maximum* 3 5, where *maximum* has the type $Ord\ a \Rightarrow [a] \rightarrow a$. This function is quite similar in both function and name to the function *max* with the type $Ord\ a \Rightarrow a \rightarrow a \rightarrow a$ and can therefore easily be confused. Therefore, the two functions are denoted as siblings and are subsequently tried by the function sibling heuristic. The heuristic will see a mistake that occurs in the function *maximum* and will try instead the function *max*. If this substituted function would be type correct in that place, the fix is recommended to the programmer. In Helium, many siblings are provided for functions originating from the Prelude, but any library or DSL developer can provide their own siblings. Common siblings include $foldr$ and $foldl$ or $(+\!\!+)$, $(+)$, $(\circ)$, which are all concatenation operators in different languages and might therefore be confused.

### 5.5.3 Argument heuristics

There are three heuristics with a very similar purpose. They are the application heuristic, the variable heuristic and the tuple heuristic. Each of them checks whether their language construction is used in a correct manner, with the correct types of its arguments, incorrect ordering of its arguments or that too few or too many arguments are provided. They work in similar fashion, but are of course specifically tailored to their respective language construction.

#### Application heuristic

The application heuristic deals with the application of functions. When an inconsistency occurs with an application, the application heuristic will try to determine the number of given arguments, the number of

wanted arguments and the order of the arguments. Take the example $foldr\ 0\ [1..10] > 0$. In this case, we miss the first argument of $foldr$, as we know that $foldr$ requires 3 arguments and inserting a first argument would remove the inconsistency. The knowledge that not any argument, but the first argument is missing, is quite difficult to come by. As we know we lack one argument, we will try to add a fresh type variable at every argument location and see whether that would fix the problem. If we would have more than one incorrect argument, it might be necessary to permute the argument to determine whether such a modification would remove the inconsistency. Other changes include removing an argument, reporting that the function requires no arguments, as it is not a function, like $True\ 3$ or reporting that too many arguments are given.

#### Tuple heuristic

The tuple heuristic works on similar basis, but will report errors regarding a tuple which is too short, too long or when its arguments need permutation. This can only be done when the expected type of the tuple is known, for example in the case of $fst\ (\texttt{'a'}, 3, True)$, in which we know that $fst$ requires a 2-tuple. In other cases, in which we do not know which tuple size or type is correct, we might report the more neutral message that two tuples do not match.

#### Variable heuristic

We also have the variable heuristic. This heuristic does roughly the same as the application heuristic, but only for functions which lack any arguments, but does require those arguments to be type correct. The reason this is done in a separate heuristic, is because the variable heuristic does not work on application constraints, as the application heuristic does. Application constraints, of the form $a \sim b \to c$, are not generated in a case such as $id :: Bool$. Therefore, the variable heuristic can give only one suggestion, namely that the programmer should insert an argument to the function.

### 5.5.4 Function binding heuristic

The function binding heuristic will verify that the number of arguments to a function binding do not exceed the number of arguments given by a type signature. See the following example:

$$f :: Int \to Int$$
$$f\ x\ y = x + 1$$

In this case, the function $f$ requires 1 arguments, according to its type signature, but two are given in its function binding. Therefore, we want to report that the function binding has more arguments than are given by its type signature. We do not report a lack of arguments, as this is perfectly acceptable in Haskell, for example in the following definition:

$$g :: [Int] \to [Int]$$
$$g = map\ (+1)$$

It might also be the case that the type signature indicates that no arguments need to be supplied. In such a case, an appropriate error message is also given.

### 5.5.5 Constraint from user

Constraints from users are a way that any programmer can customize the error message that are reported, which was proposed by Heeren et al. [13]. This is most useful when developing an embedded DSL, where

we want to make the error messages dependent on the domain. An example of such a customization is the following rule, which was described in Section 2.4.3:

```
     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";
```

Of course, a definition of (+++) should be part of the source code. We can know make sure that all the constraints are met and when a constraint is not met, we can report the appropriate error message.
The constraints of the user are handled by a heuristic that will try to see if the expression mentioned was involved. If this is the case, the constraints that are mentioned are checked, starting at the top. If the constraint matches, the heuristic will vote on that constraint and will attach the custom error message to the constraint. This custom error message is then shown to the user. This error message can be customized with values that can be later filled in, like the location of the expression, a pretty printed version of (certain parts of) the expression, expected types or the inferred types for variables. Serrano and Hage [28] have also shown ways to improve the quality of DSL error messages, based on the OutsideIn(X) system.

# 6 Rhodium: Type graphs for OutsideIn(X)

## 6.1 Introduction

In this chapter, we will explain the design of type graphs for OutsideIn(X). We will start by giving an introduction about how the framework works and what its features are. An implementation of this framework, with the name Rhodium, has been made and is currently working as a type inferencer for the Haskell compiler Helium. We will use the convention in this chapter that OutsideIn(X) refers to the original design of OutsideIn(X), as described by Vytiniotis et al. [34]. We refer to Rhodium when we want to indicate the type inferencer with the extension of type graphs.
Rhodium extends the ability of OutsideIn(X) in a few areas:

- **Single type inference process**: It makes sure that the type inference process is merged into a single inferencing process, instead of a recursive process in the case of OutsideIn(X). Every existential constraint starts a separate inferencer in OutsideIn(X) that uses the previous results as input (See Section 4.3). For optimal error diagnosis, we merged these separate steps into one single inference process.
- **Error path collection**: The system also has the ability to collect the error paths, the list of constraints that might contribute to the error, for problematic constraints. This is required when a constraint is found to be incorrect, as we want to retrieve the constraints that the incorrect constraint originated from. We do this by storing for every constraint a list of constraints that were involved in its creation. This creates a stack trace that we can retrace until we find the original constraints that led to the creation of the error.
- **Graph modifiers**: We have the ability to modify the constraints in case that an error is found. This process is similar to type graphs that remove an edge in case of an error. However, the new modifications are more powerful, as they also allow for introduction of new constraints, modification of constraints and changing whether variables are considered touchable. The usage of these modifications is explained in Chapter 7, as they closely relate to the type error diagnosis process.

## 6.2 Design requirements

We have requirements that influence the decisions made when designing the type inferencer. The main design requirement is the basis for this thesis. It is a strict requirement that the heuristics currently present in Helium should work on Rhodium. As these heuristics work on type graphs, we need to support such type graphs in Rhodium.
We also have a few additional design requirements that we would like to meet:

- **Heuristics for all language extensions**: We want to be able to add support for new heuristics that work on the other features OutsideIn(X) supports, like GADTs, and not limit ourselves to the heuristics already present in Helium.

- **Support for a parametric X**: Which features OutsideIn(X) supports is not known in advance, as the constraint solver X is parametric. This means that we have no information about which constraints are available and how these constraints need to be represented. Ideally, we want to make a type inferencer, with support for heuristics, independent of the constraint solver X. This means that we can not make any assumptions about how to represent the type graphs for the different types and constraints, as these are defined by the parameter X. It would be possible, and probably easier, to develop a type inferencer that supports heuristics for a specific X, as we can make assumptions about the shape of the constraints and the type graph. The reason that we do not limit ourselves to this, is that this would disallow any changes to the constraint solver X in the future, as well as the ability for custom extensions.
  Gundry [5] developed a specific language extension that allows units of measurement in types. This extension also requires extensions to the constraint solver X that is used in GHC. We would not be able to support such efforts when we limit ourselves to a specific constraint solver.
- **Similar to OutsideIn(X)**: We want Rhodium to have the same properties of type checking and inferencing as OutsideIn(X), that is, any constraint set is rejected by the type inferencer if and only if it is rejected by OutsideIn(X), for a given X.

## 6.3 Type graphs

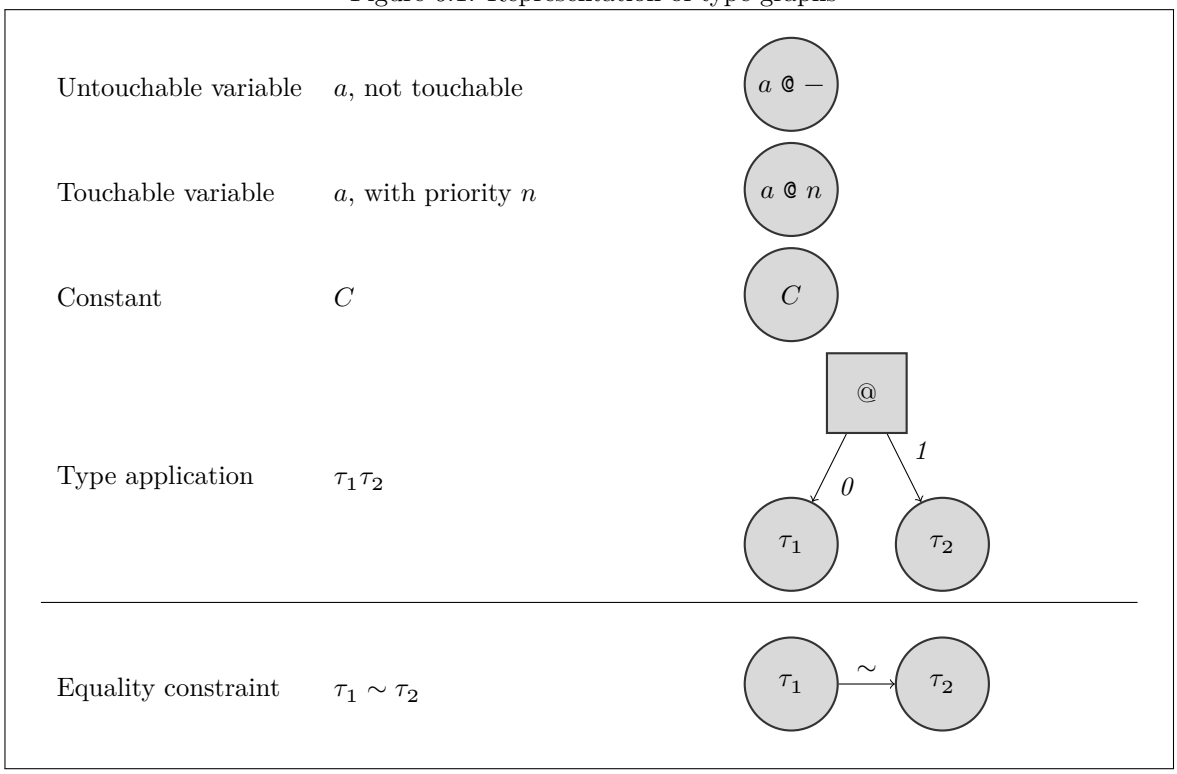
In this section, we will explain both the representation of type graphs in the type inferencer with a parametric X, as well as a concrete representation for the X that is used by Rhodium in Helium.

### 6.3.1 Representation of type graphs in Rhodium

The representation of type graphs is not standardised, as the shape of the type graphs depends on the available types in X. There are a number of common factors that every X shares, but there might also be differences between them. We will assume that every X has equality constraints, constants, type variables, that may or may not be touchable, and type application in some form. Even though we will give the representation for these constraints and types as they are defined in the implementation of Rhodium, it is the responsibility of the instantiation of X that the constraints and types are converted to their proper form. The main reason for this change is that there are multiple valid ways to represent a number of type constructions. Take the example of type application. This could either be the type $C\ \bar{\tau}$ or $\tau_1\tau_2$, for example this would represent $Either\ [A,\ B]$ or $((Either\ A)\ B)$. Each of these need a different representation and we therefore do not require one single representation.
Type variables and constants are represented in the same way as they are represented in TOP, as a single vertex. Type variables also have their touchability attached to it. This is either an integer, specifying the layer that the variable is touchable in, or $-$, which means that the variable is not touchable. Details about the layering and the touchability will be explained in Section 6.4.1. The representation of the type graphs of common types and constraints can be found in Figure 6.1. This figure also includes type application and equality constraints. Notice that type application has, contrary to type application in TOP, no left or right label, but rather a number. This is because type application is not limited to 2 child vertices, who are connected with type application edges, as is TOP. This will be used when dealing with the representation of type families in Section 6.3.2, but can also be used to represent a type that has the form $C\ \bar{\tau}$. In the case where there are two child vertices however, the label 0 can be read as the left child and the number 1 can be read as the right child.
Finally, we have constraint edges. This edge has a lot of properties. The largest change, compared to the type graphs in TOP, is the addition of the type of constraint that is represented. As every constraint needs to be able to be represented by an edge, we need to know the type of constraint, like unification or instantiation. Also notice that the edge between two types is directional. The reason for this is that constraints in OutsideIn(X)

are directional, that is $\tau_1 \sim \tau_2$ is not equal to $\tau_2 \sim \tau_1$.
Consider the example of the constraints $Num\ a, a \sim b, b \sim a$. We replace the constraint $Num\ a$ with $Num\ b$, as we have the constraint $a \sim b$. After this, we apply the constraint $b \sim a$, which results in the constraint $Num\ a$, which is the same as the input. This process would continue indefinitely.
Without this property, termination of the solving process can not be guaranteed. It therefore can happen that there are multiple edges between two vertices, both describing equality but in different directions. When the equalities are considered after the inferencing, a special case is created for equality edges and the direction of the constraint does not matter after the simplification, only during the simplification. After simplification, equality will work both ways, independent of the constraints.
Axioms are not represented in the type graph, as they will be considered unchangeable and should therefore change during the inferencing process and type error diagnosis process. It might be desirable that the axioms could change during the error diagnosis phase of the type inferencer, but this is currently not possible. However, this is not a fundamental limitation of the system and the system could be adapted to cope with such an addition. This would probably be desirable in cases of type error diagnosis with type families, where modifying the axioms might have a larger impact on the type inference process, but type error diagnosis for type families is outside the scope of this thesis.

Figure 6.1: Representation of type graphs

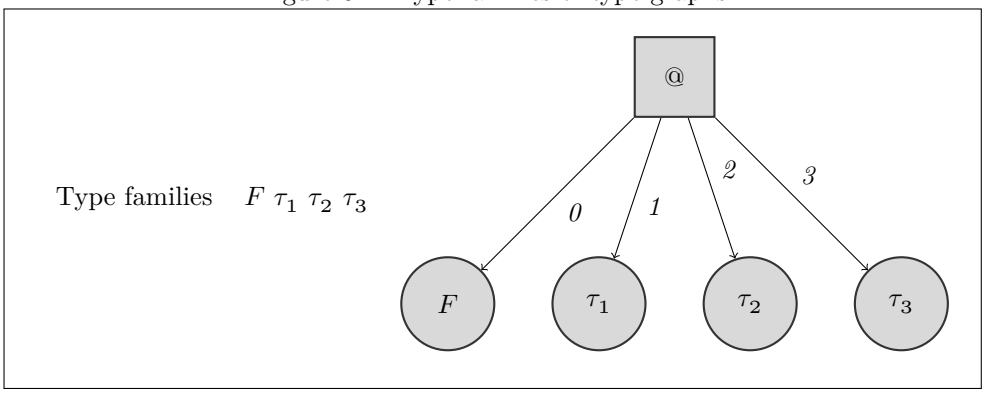



### 6.3.2 Conversion to type graphs for Helium

The conversion of constraints and types to their appropriate representation in the type graph is the responsibility of the constraint solver X, which has to be extended with this information. The representation given is the representation that is used by the Helium compiler. Note that the given representation is definitely not the only correct representation and that different choices can be made depending on the available types and

constraints. The types and constraints that are explained are defined in Figure 4.1 and are the types used by the Helium compiler. In the rest of this section, whenever we refer to the variable X, we mean the specific instantiation of X that is currently implemented in Helium.
The only addition to the types $\tau$ in X, compared to types shown in Figure 6.1 is the addition of type families. They are represented by the notation $F\ \bar{\tau}$, that is the type family $F$ with the type level parameters $\bar{\tau}$. Notice that the $\bar{\tau}$ does not have a fixed length. This is the main reason that type application edges, do not have a label like left and right attached to them, but rather a number describing their ordering. An example representation of the type family can be found in Figure 6.2. In cases that the type family requires no arguments, there is a single application vertex, a single application edge, going to a constant vertex. One situation where this might occur is the type synonym *String*, which is represented as a type family in the instantiation of Rhodium that Helium uses.

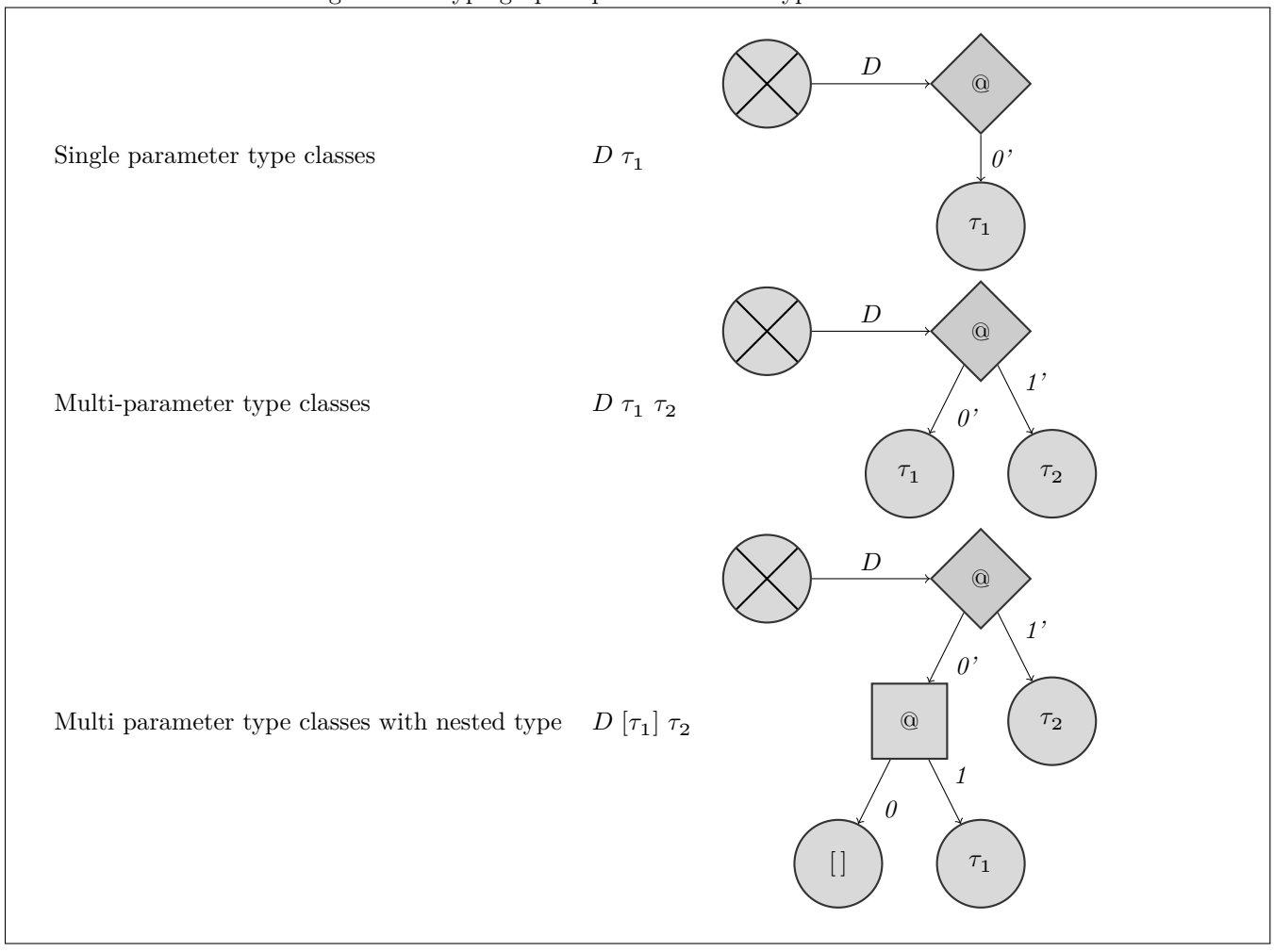
Figure 6.2: Type families of type graphs




Beside equality, we also have instantiation constraints and class constraints in Helium. Instantiation constraints, of the form $\tau > \sigma$, say that the type $\tau$ has to be an instantiation of the type $\sigma$. $\sigma$ is of the form $\forall\bar{\alpha}.\ Q \Rightarrow \tau$, where $\bar{\alpha}$ are the scoped variables and $Q$ are the constraints that need to be satisfied. The way this type is stored in the type graph is determined by the Helium specific X and depends on its form. If there set of variables $\bar{\alpha}$ is not empty, we store the type $\sigma$ in a single vertex. The main reason is that we do not want this type to interact with the other types and constraints in the type graphs, as some of them might be scoped. We will also quickly instantiate this type with fresh variables in the canonicalisation rule, after which the type will no longer be scoped and will be integrated into the type graph, as described above. In cases that there are no variables that are scoped, we can immediately integrate the type into the type graph. We convert $\tau$ to a type graph and add the constraints $Q$ to the type graph and integrate all these type graphs into the main type graph. For scoped constraints, this process happens as the canonicalisation rule instantiates the scoped variables $\bar{\alpha}$. For Helium, this happens the first time the canonicalisation rule is applied on the instantiation constraint is applied.
The final constraint we need to mention is the class constraint. Class constraints are of the form $D\ \bar{\tau}$. Even though Helium supports only single parameter type classes, our instantiation of X supports also multi-parameter type classes and we will therefore explain type graphs for multi parameter type classes. The situation described for multi parameter type classes is the same as for single parameter type classes. TOP does not put its class constraints in the type graph, but has a special case for these type classes. For Rhodium, we want to add type classes to the type graph, as it depends on the instantiation of X whether type classes are present, how they are represented and how they are dealt with.
The problem with type classes is that they do not connect two vertices with a type, as equality constraints do. However, we still want to adhere to the convention that constraints can only be attached to edges. We therefore introduce a new vertex, called a dead vertex. A *dead vertex* is a vertex that has no special value and does not represent a type, nor does it have any additional information attached to it. We can create a

constraint edge that connects the dead vertex with a constraint application vertex. A *constraint application vertex* is similar to an application edge, but it does not represent the application of types, but rather that types are applied to the constraint. They are connected via the same type edges as regular type application, and they use integers as an index. The difference between a type application vertex and a constraint application vertex is that a type application vertex stores a representation of the type that it represents, like $Int$ or $(\rightarrow)\ a$. Beside the internal representation, the place in the graph and their functions differ. Examples of type graph representations can be found in Figure 6.3. In the first example, we have a single parameter type class. Even though this is only a single parameter type class, we still use a constraint application vertex. In the second example, we have a multi parameter type class that has two types passed to it. In the final example, we see the type graph that is constructed from a type $D\ [\tau_1]\ \tau_2$, where the first parameter is a list with elements of type $\tau_1$. The difference between a type application edge and a constraint application edge can be seen in the shape of the vertex in the visual representation of the type graph, where the constraint application edge is represented by the diamond shaped vertex. A dead vertex is visualized by the crossed-out vertex. The constraint application edges are marked with primed labels.

Figure 6.3: Type graph representations of type classes



### 6.3.3 Existential constraints in type graphs

In the previous sections, we established how we represent different types and constraints as a type graph. There are two different approaches we could make with regards to existential constraints. As these constraints have nested constraints in them, they are a special case not considered before in type graphs and we would argue they do not have one single clear representation. Therefore we will consider two possible choices and discuss their advantages and disadvantages.
The first form we represent is related to the original inference process for OutsideIn(X). As described in Section 4.3, first all the constraints at top level are simplified, after which we recurse into the existential constraints. Such an approach is also possible for type graphs. In such a case, we create a type graph for all top level constraints, without the existential constraints. After we solved the type graph, for each existential constraint, we create a new type graph for the existential constraint and solve this new type graph, in which the constraints from top level are considered given constraints. This process continues recursively until all type graphs are solved. Although this process is similar to both the OutsideIn(X) framework, as it recursively handles the existential constraints, and to the TOP framework, that the type graphs only represent constraints that are all in scope, it has one major disadvantage. As all the type graphs are separate, we cannot describe interactions between different branches of the existential constraints. Consider the following example:

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

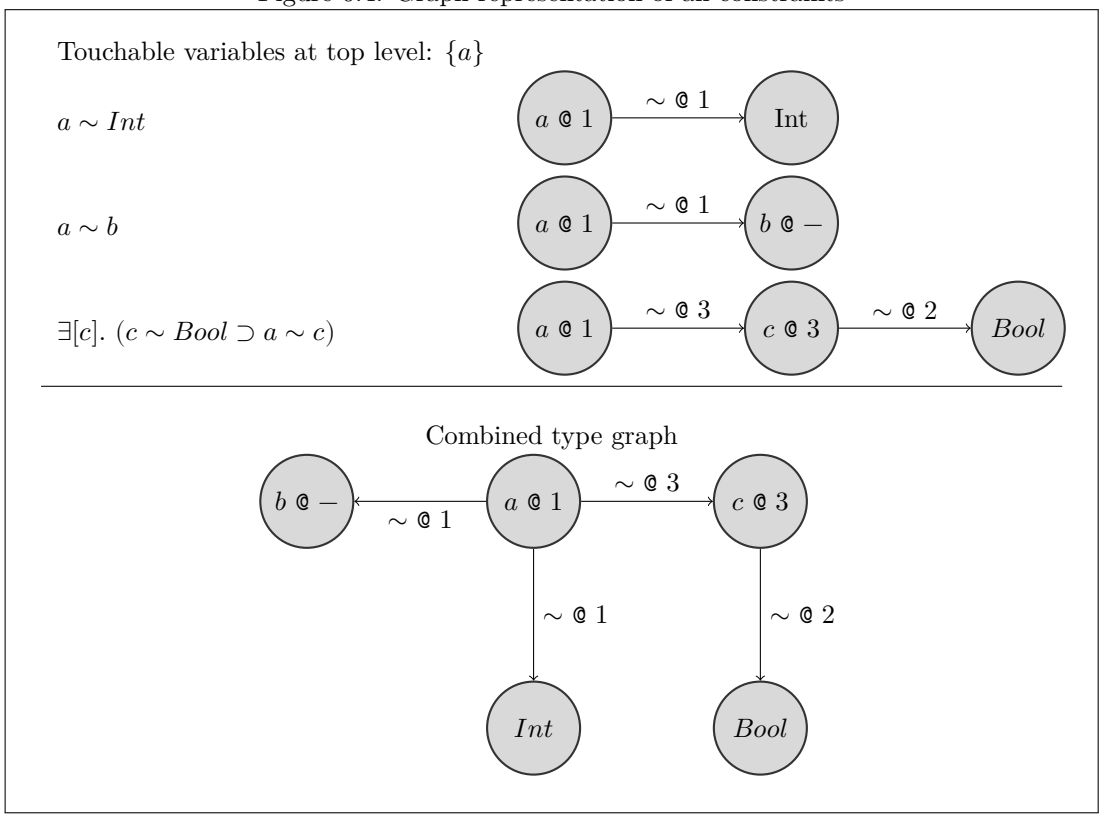
The problem in this case is that the type signature is probably incorrect. If we would have separate type graphs, we would see 3 errors, as each of the branches does not match the type signature. There is however no way to see that changing the type signature would resolve the error, as there is no interaction between the different branches, as each of them has their own existential constraint and therefore their own type graph.
Considering the previous example, we want to choose a representation that allows us to let the existential constraints interact with one another. We therefore opt to go for a representation where all the constraints, including the existential constraints, are united into a single type graph. To enable this, we will assign to all constraints a priority and a group. The priority indicates the depth that a constraint is at and the group will depend on which existential constraint is represented. If we have two existential constraints, both at top level, they have the same priority, but have a different group. If we have two existential constraints, where one existential constraint is nested in the other constraint, they have a different priority, but the group of the outer existential constraint is a suffix of the group of the inner existential constraint. Top level constraints have the lowest priority. The exact specification how the priorities and groups are represented and used in the simplification process is explained in Section 6.4.1.

### 6.3.4 Creating the entire type graph

Now that all the separate pieces of the type graph are known, we can convert a set of given constraints, wanted constraints and touchables into a single type graph. This process is relatively straight forward. We convert all of our constraints to their own separate type graph, as specified by the specified X. After this, we merge the type graphs, making sure that similar variables are combined. Consider the example in Figure 6.4. This example has three constraints, one of which is an existential constraint. Priorities of the edges are shown using

the @ symbol. The priority of the edge should not be confused with the priority of the vertex, as they are related but not necessarily equal.

Figure 6.4: Graph representation of all constraints




## 6.4 Simplification of type graphs

In the previous section, we established how we are going to represent type graphs for the constraints we are using. In this section, we show how to simplify the type graph. We will show how the simplification process works for the modified version of OutsideIn(X), Rhodium, in general and show the specific instantiation of X that Helium uses.

### 6.4.1 Priorities and groupings

Every constraint edge has a grouping and a priority attached to it. Given constraints always have an even priority, starting at 0. Wanted constraint have an odd priority, starting at 1. Consider the constraints $Q_g, Q_w \wedge \exists[\bar{\alpha}_1].\ (Q_1 \supset Q_2) \wedge \exists[\bar{\alpha}_2].\ (Q_3 \supset \exists[\bar{\alpha}_3].\ (Q_4 \supset Q_5))$. In this example, the given constraints $Q_g$ all have the priority 0. The wanted constraints $Q_w$ have priority 1. Inside the scope of the first existential, $Q_1$ have the priority 2 and $Q_2$ have priority 3. In the second existential, $Q_3$ also has the priority 2, as this is considered a given constraint in the second existential. There are no constraints in the second existential that have priority 3. For $Q_4$ and $Q_5$, we increase the priority again and give these constraints the priorities 4 and 5 respectively.
Beside the priority, every constraint edge also has a grouping. This grouping determines which constraints can interact with each other. A *group* is defined as an ordered list of group ids. How the group ids are

represented does not matter, but we will use the convention that the group ids are integers and that the top level constraints are in group 0. We will denote this as [0]. An existential will extend this group. It will take the previous group and will prepend a new unique group id. If we have an existential constraint at top level, the constraints will be extended with a unique group id, for example $[1, 0]$. A possible grouping for the constraints above would be:

- $Q_g = [0]$
- $Q_w = [0]$
- $Q_1 = [1, 0]$
- $Q_2 = [1, 0]$
- $Q_3 = [2, 0]$
- $Q_4 = [3, 2, 0]$
- $Q_5 = [3, 2, 0]$

The length of the group of a constraint with priority $p$ is equal to $\lfloor p/2 \rfloor + 1$. The group determines with which other constraints the constraint can interact. The interaction depends on whether the constraint is given or wanted. If a constraint is wanted, it can only interact with other wanted constraints and therefore, the group and priority has to match exactly. If the constraints are considered given, the suffixes of constraints have to match. Consider the situation that we want to let 2 given constraints interact. They can interact if they have the groups [0] and $[1, 0]$ or $[2, 0]$ and $[2, 0]$, but not in the case of $[2, 0]$ and $[3, 0]$, as they are not equal and neither constraint is a suffix of the other. The reason they are not considered equal is that if the first group is a suffix of the second group, this means that the second group is a group inside the first constraint group. The exact relation between the groups and the priorities will be discussed for canonicalisation, interaction, simplification and top level reaction in their appropriate sections, as the rules vary slightly between the four different rules. This grouping system is similar to the scoping system described by Serrano [26], Serrano and Hage [29]

### 6.4.2 Phasing

To keep the simplification process similar to the OutsideIn(X) framework, we want to simplify the constraints in phases. First, all the top level constraints are simplified and after that the existential constraints are simplified. We follow the same process in Rhodium. We start the simplification process with all the constraints with priority 0, that is, all the given constraints, after which we will continue with constraints with priority 1, the wanted constraints. Hereafter, all constraints with priority 2 are considered, these are the given constraints from the first layer of the existential constraints. We continue this process until we have simplified all layers. At every moment, we have a current priority that indicates at which priority we are at the current stage. We will never consider constraints that have a higher priority than the current priority. We will consider a constraint given if it has an even priority or the priority is smaller that the current priority. This is similar to OutsideIn(X) where existential constraints will consider all the other constraints that were simplified before as given.
The separation of the simplification between the given and wanted constraints, that is, we do not consider the wanted constraints until the given constraints are done, does not pose a risk that certain valid combinations cannot be combined. We can see from the four different rules, canonicalisation, interaction, simplification and top level react, that a wanted constraint can never influence a given constraint, only the other way around. We will also allow rules to work on a constraint with a priority that is strictly lower than the current priority, but we will consider this constraint given in such a situation. An example of this is the set of wanted constraints

$Q_w \wedge \exists[\bar{\alpha}].\ (Q_1 \supset Q_2)$. If the current priority is 3 and we are simplifying the set $Q_2$, we will consider the constraints $Q_w$ and $Q_1$ as given, as they both have a priority lower than 3, and allow them to interact with each other, similar to OutsideIn(X).
The simplification algorithm is as follows:

```
start:
apply canonicalisation rules
apply interaction rules
apply simplification rules
apply top level react rules
if (rules were applied) then
    goto start
else if (current priority < largest priority) then
    current priority += 1
    goto start
else
    done
```

Increasing the priority is similar to the way OutsideIn(X) first separates the existential constraints from the other constraints, after which the non-existential constraints are solved. The simplified non-existential constraints are then passed to the simplification of the existential constraints. This last process is ensured by considering all constraints with a lower priority, that is, all constraints that were already simplified, as given.

### 6.4.3 Touchables

Touchable variables are another part of the simplification process, beside the phasing, that has changed with respect to the OutsideIn(X) framework. We will use the assumption that every variable introduced by an existential constraint only occurs in that existential constraint. In the example $C \wedge \exists[\bar{\alpha}].(Q_1 \supset C_1)$, the variables $\bar{\alpha}$ do not occur in $C$, but only in $Q_1$ and $C_1$. This means that we only need to verify that a variable is used in combination with the correct priority, not with the same group.
We define a variable to be touchable if the priority of the variable is larger or equal than the current priority. Variables with the priority $-$ are never touchable. The reason to define variables to be touchable when they are larger and equal, not just equal, is to allow given constraints, with an even priority, to access variables that were defined touchable with an odd priority. Consider the example $\exists[\bar{\alpha}].\ (Q \supset C)$ where the priority of $\bar{\alpha}$ is $n$ where $n$ is even. The priority of the constraints of $Q$ are $n$ and the constraints $C$ have the priority $n+1$. We want, when simplifying the constraints $C$, to be able to consider the variables $\bar{\alpha}$ touchable.
Allowing variables with a higher priority to be touchable does not pose a problem for the correctness of the type inferencer. This is because a constraint should never mention a variable that is not in scope or will be brought into scope by a new existential, like with the *canon* rule. We do also not consider constraints that have a higher priority than the current priority at any moment in the simplification process, which might cause problems with touchability.

### 6.4.4 Canonicalisation

The canonicalisation rule will canonicalize a single constraint into a set of new constraints and a set of new touchables. We will change the canonicalisation rule to the form $canon[l](Q_1) = (\bar{\alpha}, Q_2)_{\perp_l}$. Notice that we removed the substitution from the result, compared to OutsideIn(X). The main reason for the decision to not allow substitutions to be returned is the quality of the error messages. When we would allow substitutions that were to be applied to the type graph, this would not allow us to create reliable error paths. For example, if we would have a substitution like $[\alpha \mapsto A]$, when we apply this substitution, reverting back to the original

constraints will be difficult. We would also argue that the substitution is not required for the type inferencer. Both the Cobalt implementation by Serrano [25], as well as Rhodium, do not support substitutions, but still support all possible constructions we want to supports. The application of the canonicalisation rule is relatively straightforward. The *canon* rule is applied on an atomic constraint and the result is collected. This result can be one of three options. If the rule could not be applied, the constraint is marked as tried and the simplification process continues. The other options are either an error or the application of the rule. These 3 options are used by all of the 4 rules used in X. If an error occurred, of the form $\bot_l$ the constraint edge belonging to the constraint is marked as incorrect, with the addition of the label $l$. The edge will also be marked as tried for that specific rule, which means that it will not be used in any other rule.
The final option is when the *canon* rule could actually be applied to the constraint. In such a case, a few changes are made to the constraint and the rest of the type graph. The constraint is marked as resolved for the specific group. This means that with respect to the current group, the edge can no longer be considered usable in any other rule. The reason that we do not consider the constraint resolved for every group, but only the current group is to prevent the situation where different existential constraints interact with one another. However, due to suffix matching of groups, if a constraint is considered resolved in the group $[1, 0]$, it will also be considered resolved in group $[2, 1, 0]$, but not in a group like $[3, 0]$.
Beside marking the constraint as resolved, we would also like to apply the results of the *canon* rule to the type graph. There are two parts we need to add to the type graph, the constraints and the new touchable variables. For the constraints, we will use the same process as when we constructed the type graph. This means that we convert all constraints in $Q_3$ to their own separate type graph, which we will insert into the main type graph. All the new constraints need to have a group and a priority assigned. This will be the same group and priority as the constraint they were created from. When the constraints are combined with the type graph, we will mark every variable $\bar{\alpha}$ from the result to be touchable in the type graph. The priority of the touchable variable will be the same as the priority of the constraint, as they need to be touchable in the scope of the variable, but not above that. This will also guarantee that the priority of the new constraint is the same as the priority of the variables it introduces.
Helium has an implementation of the OutsideIn(X) canonicalisation rules. The canonicalisation rules can be found in Figure 6.5 and are based on Cobalt [25] and on the instantiation given by Vytiniotis et al. [34]. We also provided the labels that accompany the *canon* rule cases that produce an error. These error labels might be used by the heuristics or the error messages to determine where the error is coming from and what kind of error it is.
The canonicalisation rules use the *unfamily* function, which extracts type families from a type application and is strongly inspired by the function with the same name in Cobalt [25]. Vytiniotis et al. [34] call this procedure *flattening*. The *unfamily* rule is of the shape $\mathit{unfamily}(\tau) = (\bar{\alpha}, \xi, Q)$, where $\tau$ might contain a type family. Every type family in $\tau$ is replaced by a fresh type variable $\beta$, while the shape of the constraints remains the same. The constraint $Q$ contains a list of equalities of the form $\beta \sim F\ \bar{\tau}$ with a fresh $\beta$ for every type family $F\ \bar{\tau}$. The variable $\beta$ will also be a part of $\bar{\alpha}$. Take as an example $\mathit{unfamily}(F\ Int \to G\ Bool) = (\{\alpha, \beta\}, \alpha \to \beta, \alpha \sim F\ Int \wedge \beta \sim G\ Bool)$. The variables $\alpha$ and $\beta$ are freshly introduced and replace the occurrences of the type families $F$ and $G$.

### 6.4.5 Interaction

The interaction rules allows two constraints to interact with one another to create a new set of constraints. The interact rule is of the form $\mathit{interact}[l](Q_1, Q_2) = Q_3$. The atomic constraints $Q_1$ and $Q_2$ should either both be given or wanted. A constraint can not interact with itself. For type graphs, this means that the priorities of the constraints should be equal to one another and the current priority should be odd to consider both to be wanted. For a constraint to be given, we insist that both priorities are strictly smaller than the current priority. The other possibility for a constraint to be considered given is that both priorities need to be smaller or equal to the current priority, but in such a case the current priority must be even. This ensures

Figure 6.5: Canonicalisation rules

| | | | |
|---|---|---|---|
| $canon[l](\tau \sim \tau)$ | $=$ | $(\epsilon, \epsilon)$ | Reflection |
| $canon[l](T \sim T)$ | $=$ | $(\epsilon, \epsilon)$ | Equal constructors |
| $canon[l](T \sim S)$ | $=$ | $\bot_{\text{Incorrect constructors}}$ | Different constructors |
| $canon[l](\tau_1\tau_2 \sim T)$ | $=$ | $\bot_{\text{Incorrect constructors}}$ | Different number of arguments |
| $canon[l](T \sim \tau_1\tau_2)$ | $=$ | $\bot_{\text{Incorrect constructors}}$ | Different number of arguments |
| $canon[l](\tau_1\tau_2 \sim \tau_3\tau_4)$ | $=$ | $(\epsilon, \tau_1 \sim \tau_3 \wedge \tau_2 \sim \tau_4)$ | Equality of application |
| $canon[l](\tau_1 \sim \tau_2)$ | | | Lexicographically ordering |
| where $\tau_2 \prec \tau_1$ | $=$ | $(\epsilon, \tau_2 \sim \tau_1)$ | |
| $canon[l](tv \sim \tau_1\tau_2)$ | | | Type family in application |
| where $unfamily(\tau_i) = (\bar{\alpha}_i, \xi_i, Q_i)$ | $=$ | $(\bar{\alpha}_1\bar{\alpha}_2, \{tv \sim \xi_1\xi_2\} \cup Q_1 \cup Q_2)$ | |
| $canon[l](F\ \bar{\tau} \sim \tau')$ | | | Nested type families |
| where $unfamily(\tau_i) = (\bar{\alpha}_i, \xi_i, Q_i)$ | $=$ | $(\bar{\alpha}_1..\bar{\alpha}_n, \{F\ \bar{\xi} \sim \tau'\} \cup (\bigcup_i^n Q_i))$ | |
| $canon[l](tv \sim \xi)$ | | | Infinite type |
| where $tv \in fv(\xi)$ | $=$ | $\bot_{\text{Infinite type}}$ | |
| $canon[l](\tau_1 > \forall\bar{\alpha}.Q \Rightarrow \tau_2)$ | | | Instantiation |
| where $\bar{\beta}$ fresh | $=$ | $(\bar{\beta}, [\bar{\alpha} \mapsto \bar{\beta}](\{\tau_1 \sim \tau_2\} \cup Q))$ | |

that constraints with an even priority that have an equal priority to the current priority are considered given and can therefore interact with constraints that might have a lower priority.
Beside the correct priorities, we also need to ensure that two constraints that want to interact have the same group. If we would allow two constraints of different groups to interact, these constraint might come from two different existential constraints, which could lead to problems. Take the example $a \sim \beta \wedge \exists[\bar{\alpha}_1].\ (a \sim Int \supset C_1) \wedge \exists[\bar{\alpha}_2].\ (a \sim Bool \supset C_2)$. We allow the constraint $a \sim b$ and $a \sim Int$ to interact and the constraints $a \sim \beta$ and $a \sim Bool$ to interact, but not the constraints $a \sim Int$ and $a \sim Bool$, as they are in different existential constraints and therefore in different groups. The other constraint pairs from the example are in the same group, according to the suffix group matching we use.
After the constraints $Q_1$ and $Q_2$ have interacted, we get the constraint $Q_3$, according to $interact[l](Q_1, Q_2) = Q_3$. We insert the constraints $Q_3$ into the type graph the same way we did with the canonicalisation rule. However, we have the problem that these new constraints need to be given a priority and a group. Every constraint in $Q_3$ is given the same group and the same priority. The priority is the maximum of the priorities of $Q_1$ and $Q_2$, that is, it is placed at the same level as the deepest constraint. We give the constraints the group with the highest length. That is, due to the proportional relationship between the length of the group and the priority, the same group as the highest priority is coming from. When two constraints have a group of equal length, they must be identical, as we use a suffix matching to determine their equality.
Consider the following example: $Q_g, Q_w \wedge \exists[\bar{\alpha}_1].\ (Q_1 \supset Q_2) \wedge \exists[\bar{\alpha}_2].\ (Q_3 \supset Q_4)$. The given constraint $Q_g$ has priority 0 and the constraint $Q_w$ is of priority 1. They are both of group [0]. The constraints $Q_1$ and $Q_3$ have priority 2 and $Q_2$ and $Q_4$ have priority 3. The first existential we will give group $[1, 0]$ and the second existential has group $[2, 0]$.
In the first iteration, only constraints within $Q_g$ can interact. In the second iteration, both $Q_g$ and $Q_w$ constraints can interact, but only within their own constraint set. It is not allowed for a constraint from $Q_g$ and $Q_w$ to interact in the current iteration. The second iteration becomes more interesting. We allow the constraints from $Q_w, Q_g$ and $Q_1$ to interact with one another as given constraints. The resulting constraints will be given the group $[1, 0]$ and the priority 2. The same is allowed for the constraints $Q_w, Q_g$ and $Q_3$, but they get the group $[2, 0]$. In the final iteration, we also allow interaction in the constraint $Q_3$ and $Q_4$, but not with any constraint outside their own constraint group and therefore not with one another.

A constraint can only once be involved in an application of the interaction rule. This means that we need to mark a constraint that has served as a parameter to the interact rule, where the rule was applied, as resolved. This is to make sure the constraint cannot be used again. There is however one problem with this description. Take the example of the wanted constraints $a \sim Int \wedge \exists[].\ (a \sim Int \supset \epsilon) \wedge \exists[].\ (a \sim Bool \supset \epsilon)$. If we would interact the constraints $a \sim Int$ (at top level) and the constraint $a \sim Int$ (from the first existential constraint) with one another and mark both as resolved, the top level constraint cannot be interacted with again. In such a case, the constraint $a \sim Int$ (at top level) and $a \sim Bool$ do not interact, as $a \sim Int$ is already marked as resolved and no error occurs, even though this set of constraints should be rejected. Therefore, we need to resolve a constraint only within a certain group.
We do this by marking for which group a constraint is resolved. We use the suffix equality of the groups to ensure this. If a constraint is marked resolved within the group $[0]$, it is considered resolved in the group $[0], [1, 0]$ and $[2, 0]$. If a group is resolved for the group $[1, 0]$, it is considered unresolved in the group $[2, 0]$ and can therefore be safely used in an interaction or any other rule.
We provide a possible instantiation for the *interact* rule, and in fact the instantiation that Helium uses, in Figure 6.6. Beside these rules, the instantiation of X for interaction for Rhodium can also provide a function that determines the candidate constraints. A *candidate constraint* is a constraint that, given an initial constraint, might allow the *interact* rule to be successfully applied on the first constraint and a constraint from the candidate constraints. By default, every constraint in the type graph is considered a candidate constraint and it is up to the interact rule to make a stricter selection to determine which constraint can successfully interact and which may not interact. The possible advantage for a type graph of such a function is the ability to look only at the neighbours of a vertex in the type graph. Consider the example of the constraint $a \sim b$. We know that only constraints that mention either $a$ or $b$ could possibly interact with this rule. Because we have a single vertex for every variable, we can look at the neighbours of the vertex representing $a$ or $b$ to determine which constraints could stand a chance to interact with the constraint $a \sim b$. Note that it might be that the variable $a$ is part of a constraint like $c \sim a \to b$ and it might be that more vertices need to be traversed in order to determine an accurate set of candidate constraints.
Due to limitations in the representation of type graphs, it requires $O(m)$ time to get all neighbours of a vertex, where $m$ is the number of edges. Therefore Helium uses the default behaviour to consider every constraint a candidate constraint. If a different representation could be chosen, where every vertex can obtain in constant time its vertices in constant time, this would probably increase the speed of the type graph and would make the time required to simplify the type graph less dependent upon the number of constraints.
Another optimization is done in the form of memorizing which rules have been applied for every constraint. This means that a constraint or an exact constraint combination will never be tried twice. This requires that every rule has the property that given the same constraint as input, it produces the same result. If a rule rejects some constraints, these constraints will not be checked for that specific rule again. Modified versions of the constraint will be tried though. It can be seen in Figure 6.6 that it is quite common that a constraint

Figure 6.6: Interaction rules

$$
\begin{array}{lcl}
interact[l](\tau_1 \sim \tau_2, \tau_1 \sim \tau_2) & = & \tau_1 \sim \tau_2 \\
interact[l](tv \sim \xi_1, tv \sim \xi_2) & = & tv \sim \xi_1 \wedge \xi_1 \sim \xi_2 \\
interact[l](tv_1 \sim \xi_1, tv_2 \sim \xi_2) & & \\
\quad \text{where } tv_1 \in fv(\xi_2) & = & tv_1 \sim \xi_1 \wedge tv_2 \sim [tv_1 \mapsto \xi_1]\xi_2 \\
interact[l](tv \sim \xi, D\ \bar{\xi}) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) & = & tv \sim \xi \wedge D\ [tv \mapsto \xi]\bar{\xi} \\
interact[l](D\ \bar{\xi}, D\ \bar{\xi}) & = & D\ \bar{\xi} \\
interact[l](tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & tv \sim \xi_1 \wedge F\ [tv \mapsto \xi_1]\bar{\xi} \sim [tv \mapsto \xi_1]\xi_2
\end{array}
$$

returns its input constraints again. We will see that this actually produces a problem when collecting the error paths, as these are determined by the rules that are applied. More detail will be given in Section 7.1, but for now it should be sufficient to know that an error path for an edge is based on the edges that were used in the creation of the constraint. We recursively backtrack through the graph until all edges in the error path are constraints that were originally provided to the type inferencer.
Consider the constraints $a \sim b$, $a \sim Int$ and $a \sim Bool$. In such a case, the constraint involved in the inconsistency are $a \sim Int$ and $a \sim Bool$. However, if we would apply the interaction rules to this set of constraints, we get a different error path. Consider the inferencing in Figure 6.7. The application is done from top to bottom and uses only the second *interact* rule as given in Figure 6.6. When we look at the constraint $Bool \sim Int$, we can use the backtracking over the constraints to determine which constraints were involved. We find that the constraints $[\#0, \#1, \#2, \#3]$ are involved in creating this constraint. After extracting all the original constraints, we find that the constraints $a \sim b$, $a \sim Int$ and $a \sim Bool$ were involved in creating the inconsistent constraint. Even though technically, $a \sim b$ was used in the creation of the inconsistent constraint, we do not want to include this constraint in the error path, as the inconsistency remains when the constraint would not be present.
The solution we used is to handle constraints from an interaction rule that are identical to the input constraints, in a different way. If we find for the rule $interact[l](Q_1, Q_2) = Q_3$ that either $Q_1$ or $Q_2$ is in the conjunction $Q_3$, that specific constraint is treated differently. We will focus on the case that $Q_1$ is in the conjunction $Q_3$, (the other cases are identical). If the constraint $Q_1$ is equal to one of the constraints returned by the *interact* rule, it is not marked as resolved and the new constraint is not added to the type graph. The implication of this is that all other constraints will not be based on the constraints $Q_1$ and $Q_2$ coming from $Q_3$, but still reference the old $Q_1$ constraint. Consider the modified example in Figure 6.8 that uses this new rule. We can now see that, even though $interact(\#10, \#11)$ produces both $a \sim Int$ and $b \sim Int$, we will not consider a constraint based on $a \sim Int$ to be influenced by $a \sim b$. Therefore, the modified rule reduces the size of the error paths and will prevent the inclusion of edges that were unrelated to the error. This modification does not prevent all unrelated edges to be included in the error path, but reduces their number of which the type graph can be certain that they are not involved in the creation of the inconsistency.

Figure 6.7: Overly conservative error path due to *interact* rule

| Edge id | constraint | based on | created by |
|---|---|---|---|
| #0 | $a \sim Int$ | [] | original |
| #1 | $a \sim b$ | [] | original |
| #2 | $a \sim Bool$ | [] | original |
| #3 | $a \sim Int$ | $[0, 1]$ | $interact(\#0, \#1)$ |
| #4 | $b \sim Int$ | $[0, 1]$ | $interact(\#0, \#1)$ |
| #5 | $a \sim Bool$ | $[2, 3]$ | $interact(\#2, \#3)$ |
| #6 | $Bool \sim Int$ | $[2, 3]$ | $interact(\#2, \#3)$ |

### 6.4.6 Simplification

The simplify rule is similar to the interact rule, as it takes two constraints and returns a new set of constraints. The major difference with the interaction rule is that the simplification rule takes a given and a wanted constraint and returns a new set of wanted constraints. The form of the simplification constraint is $simplify(Q_1, Q_2) = Q_3$, where $Q_1$ and $Q_2$ are atomic constraints. $Q_1$ will be the given constraint and $Q_2$ will be the wanted constraint. For $simplify$, the same rules with regards to grouping and priority apply that

Figure 6.8: Modified error path due to *interact* rule

| Edge id | constraint | based on | created by |
|---|---|---|---|
| #10 | $a \sim Int$ | [] | original, $interact(\#10, \#11)$ |
| #11 | $a \sim b$ | [] | original |
| #12 | $a \sim Bool$ | [] | original, $interact(\#12, \#10)$ |
| #13 | $b \sim Int$ | $[10, 11]$ | $interact(\#10, \#11)$ |
| #14 | $Bool \sim Int$ | $[10, 12]$ | $interact(\#12, \#10)$ |

apply to the interact rule. This means that the priority of $Q_1$ will be lower than the current priority and $Q_2$ will be equal to the current priority. They also need to be in the same group and the current priority should be odd.
As the inference process proceeds one layer at the time, we will approach a simplification from the perspective of the wanted constraint. For the implementation of Rhodium, we require beside the *simplify* rule, a function to select a number of candidate edges. This function acts similar to the function that returns the candidate constraints for the *interact* rule. This functionality is not used in Helium, as Helium uses the default candidate function which returns all constraints. One possible optimization that could be made is to return only given constraints, that is, constraints that are considered given under the current priority.
When the simplification rule is successfully applied, the resulting constraints $Q_3$ are converted to type graphs and are inserted into the main type graph. Beside the new constraints, the constraint $Q_2$ is marked as resolved, so that it can no longer be used in simplification. The constraint $Q_1$ however is not marked as resolved, as a given constraint can be used multiple times in the simplify rule.
The instantiation for the *simplify* rule that is used by Helium can be found in Figure 6.9. These rules are fairly similar to the instantiation that is given for the instantiation of X by Vytiniotis et al. [34].

Figure 6.9: Simplify rules

$$
\begin{array}{lcl}
simplify(tv \sim \xi_1, tv \sim \xi_2) & = & \xi_1 \sim \xi_2 \\
simplify(D\ \bar{\tau}, D\ \bar{\tau}) & = & \epsilon \\
simplify(tv \sim \xi, D\ \bar{\xi}) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) & = & D\ [tv \mapsto \xi]\bar{\xi} \\
simplify(tv_1 \sim \xi_1, tv_2 \sim \xi_2) & & \\
\quad \text{where } tv_1 \in fv(\xi_2) & = & tv_2 \sim [tv_1 \mapsto \xi_1]\xi_2 \\
simplify(F\ \bar{\xi} \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & = & \xi_1 \sim \xi_2 \\
simplify(tv \sim \xi_1, F\ \bar{\xi} \sim \xi_2) & & \\
\quad \text{where } tv \in fv(\bar{\xi}) \text{ or } tv \in fv(\xi_2) & = & F\ [tv \mapsto \xi_1]\bar{\xi} \sim [tv \mapsto \xi_1]\xi_2
\end{array}
$$

### 6.4.7 Top level reaction

The top level react rule is of the form $topreact[l](Q_1) = (\bar{\alpha}, Q_2)$ where $Q_1$ is an atomic constraint that is passed as a parameter. The result consists of a set of new fresh variables $\bar{\alpha}$ and the new set of constraints $Q_2$. The main function of top level react is to let the axioms, that are provided to the type inferencer, be applied to the constraints. The axioms that we will discuss are the type family axiom, of the form $\forall[\bar{\alpha}].\ \xi \sim F\ \bar{\tau}$ and the class axioms $D\ \bar{\tau}$. These axioms are also the axioms that are available in the instantiation for X used by Helium. As discussed before, the axioms are not part of the type graphs and are represented separately. It is

up to the *topreact* rule to deal with these axioms in their entirety, as the *topreact* rule might need to look at all the axioms before choosing which axiom to apply. This means that they will not provided one at a time, but as a list. It is then up to the *topreact* rule to determine how to deal with that list.
The application of the *topreact* rule is very similar to the other rules, especially the *canon* rule. The constraints $Q_2$ are converted to type graphs and inserted into the main graph. The variables $\bar{\alpha}$ are made touchable in the type graphs. The constraint $Q_1$ will be marked as resolved when *topreact* could be applied.
The instantiation of X in Helium will react on two different types of constraint. We will first discuss the situation of $topreact[l](D\ \bar{\tau})$. This reacts with an axiom of the form $\forall[\bar{\alpha}].\ Q \Rightarrow D\ \bar{\tau}$. From now on, we will use the notation $\forall[\bar{\alpha}].\ Q \Rightarrow D_Q\ \bar{\tau}_Q$ to denote a class axiom and its different parts and distinguish them from the constraint that the axiom might react with. We look through the list of axioms to see if there is an axiom where $D = D_Q$. If we are certain that the class names match, we need to verify that $\tau \sim \tau_Q$ where the variables $\bar{\alpha}$ are touchable. This condition is checked by a separate call to the type inferencer. In this case, we use the type graph implementation, but there is no requirement to do this, as we are only interested in whether the types are equal, not why they would not be equal if that were to be the case. If this constraint matches, we obtain a substitution $\theta$. We return the constraints $Q$ with the substitution $\theta$ applied. No new touchable variables are returned from the application of this axiom.
The resulting constraints $Q_2$ are converted into a set of type graphs, each of which is then merged into the main type graph by merging the vertices that already existed in the type graph. After this, the new touchables are marked in the type graph with the correct priority to ensure their touchability.
Consider the following example of the constraint $Show\ [Int]$ and the available axioms $Show\ Int$ and $\forall[\alpha].\ Show\ \alpha \Rightarrow Show\ [\alpha]$. We can first apply the second axiom, which tries to solve $[\alpha] \sim [Int]$ where $\alpha$ is touchable, which results in the successful substitution $[\alpha \mapsto Int]$. We therefore return the substitution applied to the constraint $Q$, which is $Show\ \alpha$ in this case. Therefore, the resulting of the application of the *topreact* rule is $(\epsilon, Show\ Int)$. After this, we can apply the first axiom in a new call to *topreact* to the constraint $Show\ Int$. In that case, we try the unification of $Int \sim Int$, which succeeds and returns an empty substitution. As the axiom $Show\ Int$ should be read as $\forall[].\ \epsilon \Rightarrow Show\ Int$, we return $(\epsilon, \epsilon)$, which completes the simplification of the constraint $Show\ [Int]$.
The other case is the axiom that has the form $\forall[\bar{\alpha}].\ \xi_Q \sim F_Q\ \bar{\tau}_Q$. We will try to match this axiom to a constraint of the form $F\ \bar{\xi} \sim \xi$. We will use a similar situation as with the class axioms. We will first try to match $F_Q = F$. If this is the case, we will try to unify $\bar{\tau}_Q \sim \xi$ where the variables $\bar{\alpha}$ are touchable. If this is successful, the resulting substitution $\theta$ is returned. The result of the call to *topreact* is the constraint set $\theta\xi_Q \sim \xi$, where we applied the substitution $\theta$ to the condition of the axiom $\xi_Q$.

### 6.4.8 Error labels

We already showed that the *canon* rules can return custom error labels in case of an error. The main usage is to give additional information to the heuristics and the code that displays the error message which type of error occurred. It is up to the designer of the instantiation of X to choose an appropriate list of error messages. A number of common labels are provided, like a label for infinite types or for mismatched constructors. These labels do not have any value within the framework and are only provided for the ease of use of the instantiation of X. There is one error label which is attached by the framework and has a special value. This is the label for residual constraints. These residual constraints are not detected by a rule like *canon*, but are left over at the end of the inference process. The residual error label therefore indicates that an error was detected by the simplifier, rather than a compiler specific rule.

### 6.4.9 Residual constraints

After each phase, where we increase the current priority, we determine which constraints are residual. These constraints need to be marked as residual to ensure an error is reported for each of them. The marking

of residual constraints does not need to occur at the end of each phase, it also can be done after all the simplification is done. However, it is easier from an engineering standpoint to ensure that all constraints are marked at the end of the simplification of each process.
Whenever we speak about the current priority, this is the priority of the phase that was last simplified, before the increasing of the priority. The collection of residual constraints only happens when the current priority is odd and higher than 1. That the current priority needs to be odd is to make sure that a given constraint will never be considered residual, as given constraints have an even priority. We only consider constraints if the current priority is larger than 1 as we only want to mark constraints coming from an existential constraint during this phase. Wanted constraints that have a priority equal to 1 are collected in another phase, when we create the resulting substitution.
We will mark every constraint that is not a unification constraint to be a residual constraint. For the unification constraint to be residual, it should be considered residual under the current priority. For this, we need to check the variables involved in the current constraint. As long as the priority of the variable involved in the unification constraint is larger or equal to the current priority, the constraint is considered not residual. In other cases, the constraint is considered residual and is marked as incorrect. This means that if we have the constraint $\alpha \sim Int$ with priority 3, at the end of the simplification phase where the current priority is 3, this constraint is checked whether we should mark it as residual. If the priority of the variable $\alpha$ is larger or equal than 3, the constraint is not residual. Otherwise, the constraint is marked as residual.

## 6.5 Solve result

The result of the simplification is the solve result. The solve result consists of 5 parts. We include all the touchables that are in scope at top level, including those that were introduced during the simplification. Beside the touchables, we include a simplified version of the given constraints, the resulting substitution, the simplified type graph and the errors that were found during the type inference process. The resulting type graph can be used when heuristics want to test a certain change in the type graph and inspect the type graph afterwards. The errors that are included in the solve result might either be the errors diagnosed by the type error diagnosis process or might be those directly found by the type graph without any explanation. This depends on the options that were passed to the simplifier. Although the amount and the type of the errors might change depending on whether the flag to disable type error diagnosis is provided, whether or not any errors were found will remain the same. This means that the type error diagnosis flag only improves the quality of the errors, but does not change whether errors were found. When the type error diagnosis flag is passed, the type error diagnosis, described in Chapter 7, is disabled.

### 6.5.1 Touchables

The touchables are collected from the type graphs, as well as the touchables that were provided to Rhodium. Every variable with a priority that is smaller or equal than 1 is put in the set of touchables, which is equal to all variables that are touchable at top level, as the priorities 0 and 1 correspond to the given and wanted constraints respectively. We also include the touchables that were passed as a direct parameter to Rhodium, as they might not be included in the type graph. This happens when a variable is mentioned in the list of touchables, but is not present in any constraint. Only variables that exist in a constraint will be present in the type graph and therefore we need to include the other variables in the touchables as well.
The other possibility to have a variable be included in the list of resulting touchables is to provide a flag to ignore touchability. This flag should never be used when the correctness of the type inference process matters, as it can accept certain incorrect constraint sets. The flag is used by a number of heuristics that want to know the types of variables inside an existential constraint. The flag will cause Rhodium to consider every variable touchable and therefore include these variables in the resulting substitution. With this flag

enabled, the resulting substitution of a constraint set like $Q_w \wedge \forall[a, b](a \sim Int \supset b \sim Bool)$ would include $[a \mapsto Int, b \mapsto Bool]$, even though these variables where not considered touchable at top level.

### 6.5.2 Substitution

The resulting substitution, of the form $[\alpha \mapsto \tau]$, is created from the touchables that were described in the previous section. We collect all unification constraints that are of the shape $\alpha \sim \tau$ with priority 1 where $\alpha$ is in the set of touchables. If the flag to ignore touchablility is passed, the check that whether $\alpha$ is touchable is ignored, as well as the check that the priority should be equal to 1. The collected unification constraints are then split into the two parts of the equality and are then returned in the form of a substitution.
We do not give any guarantees about the correctness of the substitution if the type graph is inconsistent. It might be that the substitution is of the form $[\alpha \mapsto Int, \beta \mapsto Bool]$ even when the constraint $\alpha \sim \beta$ was present in the original constraints.

## 6.6 Summary

Rhodium translates the recursive simplification of OutsideIn(X) into a single process with priorities attached to the constraints. Constraints are simplified in order of their priority, starting with the lowest priority that represent the given constraints. Every constraint has a separate group, which ensures that only constraints that are allowed to interact with one another, are given the possibility to interact with one another. The simplification is done using the normal canonicalisation, interaction, simplification and top level react rules that OutsideIn(X) uses.
Consider the example with the wanted constraints $Num\ a \wedge a \sim Bool \wedge \exists[b].\ (b \sim Int \supset a \sim b) \wedge \exists[c].\ (c \sim Bool \supset a \sim c)$, where the variable $a$ is touchable at top level and no axioms or given constraints are present. The following steps are taken by Rhodium to get the final result:

1. Rhodium makes a type graph of all the constraints, based on the constraint solver X that is specified. We will give the first and second existential constraint the groups $[1, 0]$ and $[2, 0]$ respectively.

2. We start the simplification process with priority 0, but as there are no given constraints, that have priority 0, we increase the priority to 1.

3. For priority 1, we allow two constraints, $Num\ a$ and $a \sim Bool$, to interact with one another. This results, with the usage of the *interact* rule, in the constraints $Num\ Bool$ and $a \sim Bool$. As these constraints can not be simplified further, we increase the current priority to 2.

4. With a current priority of 2, we will consider the given constraints of the existential constraints. These constraints could interact with the constraints of priority 1, but not with one another. Even though they are allowed to interact based on the priority and grouping rules, no rule could be applied because of the shape and the variables of the constraint. We therefore increase the current priority to 3.

5. With priority 3, we will allow all constraints in the same group to interact with one another. The constraints $a \sim b$ and $a \sim c$ are the only constraints that currently are considered wanted, all other constraints are considered given. We can therefore only use the $simplify$ rule, as these constraints cannot interact with one another due to them having different groups. After a number of applications of the $simplify$ rule, we have the constraints $Bool \sim Int$ and $Bool \sim Bool$, both with priority 3, but with different groups. The constraint $Bool \sim Int$ has group $[1, 0]$, while $Bool \sim Bool$ has group $[2, 0]$. The constraint $Bool \sim Bool$ can be simplified further by the canonicalisation rule, while $Bool \sim Int$ is marked as an error by the canonicalisation rule.

6. We have no constraints that can be simplified further and the current priority, 3, is equal to the maximum priority. Therefore, we construct the solve result from the type graph.

7. The solve result will consist of two errors, the residual constraint $Num\ Bool$ and the inconsistent constraint $Bool \sim Int$. If no errors were found, this constraint set would result in the substitution $[a \mapsto Bool]$, as this is the only unification constraint with a priority smaller or equal to 1 that contains a touchable variable.

# 7 Heuristics for Rhodium

In this chapter, we explain the type error diagnosis process of Rhodium in Helium, both the design and the application of heuristics. We also discuss the heuristics that are available in Helium, both those that were designed for TOP and those heuristics newly introduced.

## 7.1 Collection of error paths

We want to collect all the constraints that are involved in the creation of every error. We assume that the simplification of Rhodium is completed and that all constraints that are incorrect are marked with a separate label. These incorrect constraints might be of the form $Int \sim Bool$ with the label `Incorrect constructors` or the constraint $a \sim Int$ with the label `Residual constraint` or another label, depending on the type of the error. The `Residual constraint` label is given to a constraint by Rhodium, where every other label is supplied by one of the rules, usually by the canonicalisation rule.
The path we want to construct will be based on the incorrect constraint and the simplified type graph. The creation of every path is independent of any other and does not modify the graph in any way. The error path consists of two parts, an error specific part and a path specific part. The error specific part consists of the incorrect edge, the incorrect constraint, like $Int \sim Bool$, and the label that was attached to the error. The path specific part consists of all the constraints that might be involved in the creation of the error. As mentioned in our explanation of Rhodium, every constraint that is created from one or more constraints, records from which constraint it was created. We can use this information to construct all the constraints that were involved in the simplification process that led to the creation of the constraint that caused the problem. Let us assume that an incorrect constraint with id #10 was created from the constraints #7 and #8. After this, we check the constraints #7 and #8. For each of these constraints, we check whether they were originally provided or were created during the simplification. If they were original, they are included in the error path. If they are not original, we continue the lookup process for the new constraint, in this case the constraint #7 or #8 and we look at the constraints they were based on. At the end, we have a set of constraint where none of these constraints was created by a rule during the simplification process and therefore they were all original.
The path part of the error paths, consists of a set of constraints. We record some additional information for every constraint. Beside the constraint, we include the edge the constraint originated from and the constraint information that every constraint should have. This constraint information was attached to the constraint at the moment of collecting. Finally, we attach a graph modifier to every constraint. This graph modifier will be used to change the type graph in such a way that the error will be resolved, see also Section 7.2. The collection of the error paths is done by collecting all the incorrect constraint edges and converting every incorrect constraint edge into its appropriate path.

## 7.2 Graph modifiers

A graph modifier is a function that, as the name implies, modifies the graph. This graph modifier is used to modify the graph after a constraint is blamed for a particular mistake. By default, every constraint is given the default modifier. This default modifier has the same behaviour as the TOP framework after an edge has been blamed, namely the removal of the edge that is deemed responsible for the error. When the graph modifier is not changed by the heuristics, this means that every constraint that is blamed will be removed by the graph modifier.
The reason for introducing a separate graph modifier instead of removing every edge is the diversity of the language constructions that can be represented by the type graph. In TOP, only inconsistencies were represented by the type graph, any other mistake was not represented by the type graph.
Consider the following example where the type signature of the function $f$ is missing:

**data** $X\ a$ **where**
$A :: Int \to X\ Int$
$B :: Bool \to X\ Bool$
$f\ (A\ i) = odd\ i$
$f\ (B\ b) = b$

There is not a single constraint that is collected from this program that could be blamed for this mistake, which is the lack of a type signature that was necessary. In such a case, we want a graph modifier that would actually add the correct type signature to the function $f$. Consider also the following example where a type class predicate is missing from the type signature:

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

In this case, we would have two residual constraints of type $Show\ a$. In the case that we just remove constraints, we should have to remove both $show\ x$ and $show\ y$, which would result in two different error messages. However, we would like to add the predicate $Show\ a$ to the type signature of $g$, which would also fix the problem. Therefore, the graph modifier can be used for specific errors to remove the error in a more accurate way.
A graph modifier is a function that takes a constraint, an edge, the constraint information from the constraint and the type graph. It modifies the type graph in such a way that the number of errors is reduced. The constraint that is passed as its argument is the constraint that is actually inconsistent, like $Int \sim Bool$. The constraint information gives information about the constraint that is actually blamed and the edge is the edge that is blamed. This edge has to be an original edge.
Every constraint in a path has its own graph modifier attached to it. By default, the graph modifier to remove the edge that is blamed, is attached to every constraint. The heuristics can decide for every constraint they blame whether to return the graph modifier that was already attached to the constraint or create a new graph modifier. This graph modifier is only applied at the end of the application of the heuristics when the constraint was blamed. More details about the application of the graph modifiers can be found in Section 7.3. A number of graph modifiers has been created, both in the implementation of Rhodium and in Helium. Of course, a heuristic can create its own graph modifier. A few examples of graph modifiers are:

- **Remove edge modifier**: This is the same modifier that is implicitly applied in TOP. It removes the single constraint that it is given without changing the rest of the type graph.
- **Add residual constraint modifier**: If a residual constraint is found, like in the situation of $show\ [\,]$,

we will blame that constraint and will add the residual constraint to the type graph.

- **Remove edge and type signature**: When an application is found to be incorrect, we want to remove both the application edge and the type signature. Take the example of $True + 3$. In that case, we have the constraints $+ \sim a \to b \to c \wedge + > Num\ a \Rightarrow a \to a \to a$, if we would only remove the unification constraint, we would still be left with the instantiation constraint. This constraint then causes an error as it has an residual constraint $Num\ a$. This constraint will remove the application edge, as well as the accompanying type signature.
- **Add type signature**: When dealing with GADTs, every function that pattern matches on a GADT needs a type signature. When a type signature is missing, we produce a type error. In certain cases, we can recommend a type signature and want to add this recommended type signature to the type graph.

## 7.3 Application of heuristics

The application of heuristics is relatively similar to the application of heuristics in TOP. This means that after all paths are collected from the type graph, as described in Section 7.1, a single path is chosen on which all the heuristics are applied in order. After all heuristics are applied, a single edge is chosen which is henceforth blamed for that particular error. This blamed constraint will be used, in combination with the constraint information attached to the blamed constraint, and the error label to construct an appropriate error message. This process is repeated until all errors in the type graph are resolved.
The process starts with selecting an incorrect edge which we will use for the blaming of the error. Ideally, we would like to have a path that consists of a single constraint after all heuristics are done. This means that the heuristics were able to put the blame of the error upon a single constraint. We will always choose the constraint with the longest error path, as that path has the most constraints that could be blamed. This increases the chance that we manage to blame an edge that is responsible for multiple errors and we therefore are able to remove multiple errors by blaming one specific constraint. This is also related to the usage of the participation heuristic. After we have selected the longest error path, we need to apply the heuristic on that particular path. This application happens in the order that the heuristics were given to the type inferencer. After each application of a heuristic, we might reduce the error path, such that we have a path that is of similar size or has a smaller size than the previous error path.
If we are left with more than one error, we arbitrarily choose the first constraint from the reduced error path. It should never happen that the heuristics cause the error path to become empty, as we would be unable to put the blame on a particular constraint.

### 7.3.1 Filter heuristics

The filtering heuristic is a type of heuristics that allows the heuristic to remove certain constraint from the error path. A filter heuristic is given a list of constraints, including constraint information, and determines which constraints are still possible constraints to blame. A common application of a filtering heuristic is to remove constraints that should not be blamed at a certain time. An example of this is the case of the expression **if** $x$ **then** $x$ **else** 3. In such a case, we do not want to blame the constraint that says that the condition of the **if**-statement, $x$ in this case, needs to be equal to *Bool*.
The filtering heuristic needs to make sure that it never returns a empty path. In such situations, an error would occur as the type error diagnosis does not know which constraint to blame. It is therefore common design, when filtering on a certain property, if every constraint in the provided error path has that property, to keep all constraints in the resulting error path. Assume that after a number of heuristics have been applied, we find that there are two constraints left that have a certain property. If we were to apply a heuristic that would not allow a constraint with that particular property to be blamed, there is no constraint left to blame.

In this situations, both constraints need to be returned and the next heuristics may be able to make a better informed choice which constraint to blame. It is the responsibility of the filtering heuristic to do this.
In the returned error path there is also a unique graph modifier included for every returned constraint. The graph modifier is used when the particular constraint is blamed, at the end of the application of the heuristics. Every heuristic has the choice to either use the graph modifier already present with the heuristic or to create a new graph modifier. An example might be that a tie breaker heuristic has two constraints, both with a graph modifier attached. In such a case, the tie breaker heuristic will pass the graph modifiers on that were already present. There are also cases in which the heuristic will decide to alter the existing graph modifier for a certain edge. The decision whether to use the provided modifier will be used or a new graph modifier will be created can be made on a constraint by constraint basis and does not need to be the same for every constraint in the constraint path.

### 7.3.2 Voting heuristics

The voting heuristic is a type of heuristic that let the heuristic vote on the likeliness of a certain error. A voting heuristic consists of a list of selectors. A *selector* is a function that can attach a weight to a constraint and is designed to detect a specific error. The higher the weight, the higher the certainty that the heuristic has identified the error. It might also be the case that the selector does not find the error for which it was designed. In such cases, no weight is returned and the constraint will not participate in the voting. All selectors inside a voting heuristic vote against one another and the constraints with the highest weights are chosen to remain in the error path. All other constraints are then removed from the error path. It is also possible that no selector can identify the problem. In such a case, the error path remains the same and the next heuristic will be applied. The voting selectors have, similar to the filtering heuristic, the ability to return a graph modifier. This graph modifier can be used to remove the specific error that the heuristic detected.
Technically, in Helium, there is only one voting heuristic. All heuristics that are described are in fact selectors in a single voting heuristic. This allows them to vote against one another and ensures that the best choice is made. Even though they should be called selectors, for the sake of consistency, we will to continue to call all voting selectors voting heuristic, whenever we describe a specific heuristic.

## 7.4 Helium heuristics

In this section, we will describe the heuristics that were already present in Helium and how they were converted to work with Rhodium. In Section 5.4 and Section 5.5 we gave an overview of some of the heuristics that were already present and how they work. We will mostly focus on how these heuristics were adapted to work with Rhodium. Whenever we refer to heuristics in Helium, we mean the heuristics that were already in Helium and work on TOP. If we mention the heuristics in Rhodium, we refer to the heuristics that were designed and implemented for this thesis, even though these heuristics are a part of Helium and not a part of the designed type inferencer, merely of the instantiation of X thereof.

### 7.4.1 First come, first serve heuristic

In Rhodium, there is no longer a first come, first serve heuristic, which makes sure that there is always a single constraint to blame. This behaviour is build in to the inferencer, where we always get the first constraint in the path to blame after the heuristics have all been considered. In most cases, the resulting error path will be a singleton, but on occasion, there might be more than one constraint left over after the blaming process is done. For debugging purposes, it is recorded in the logging facility of Rhodium whether there were some constraints not resolved by heuristics and therefore need to be removed by the first come, first serve mechanism. It is also reported which constraints are removed. The first come, first serve heuristic can still be implemented in Rhodium. In that case, the behaviour that there is a path with a length larger than 1, will never be triggered.

### 7.4.2 Participation heuristic

The participation heuristic will determine whether a constraint occurs in other path as well as the current paths. In Rhodium, this is accomplished by a filtering heuristics. This heuristic will request all paths from the current type graph. From these paths, we will create a participation map, which counts how often every constraint occurs in all the error paths. If there are constraints that occur in multiple paths, we want to try to blame these constraints using the other heuristics.

### 7.4.3 Avoid unwanted constraint heuristics

The heuristics that will prevent certain constraints to be blamed, are mostly unchanged in their functionality. These heuristics will remove constraints from the error path based on the properties of the constraint information. Therefore, ignoring implementation details, these heuristics remained the same in comparison to Helium. The heuristics can be split into two categories. The first category is the categories that filter simply based on whether a property is present, like the avoid forbidden constraints heuristic, which will prevent certain parts of the program from being blamed. An example of this could be the following program:

$$
\begin{aligned}
&f :: Int \to Bool \\
&f\ x = \textbf{let} \\
&\quad y = x \\
&\quad \textbf{in}\ y
\end{aligned}
$$

During the gathering phase, a constraint is gathered that says that the type of the entire **let**-binding is equal to the body of the **let**-binding, $y$ in this case. We never want to blame this constraint, as it is not something the programmer can change. Such properties do not depend on the shape of the type graph or on the shape of the constraint, but only on the properties of the constraint information attached to the constraint.
The other category of heuristic that avoid certain unwanted constraints are constraints that work on either a minimal or maximal weight. They determine either the lowest or highest weight, based on the constraint and the constraint information and choose all the constraints that are either the highest or lowest value, depending on which type of heuristic it is. An example of this is the avoid trusted constraints heuristic. This heuristic will try to blame the constraints that have the lowest trust value, which was given during the gathering phase. One such example of a trust value is a user provided type signature or a type signature that is coming from a standard library. This type signature has a certain trust value. If there is a constraint with a lower trust value, we prefer to blame this heuristic. However, if there is a constraint we trust more that the type signature, and which has therefore a higher trust value, we keep the type signature constraint in the error path.

### 7.4.4 Sibling heuristics

There are two types of sibling heuristics, the literal sibling heuristic and the function sibling heuristic. The function sibling heuristic needs a list of siblings, which need to be passed to the heuristic. These siblings, fore example ($foldr, foldl$) are defined in a separate file and represent that the functions are similar in usage that they might be confused. Consider the following example:

$$f = Nothing\ 3$$

The constructor $Nothing$ has no arguments and causes an error to occur. The error that is created by Rhodium is as follows:

```
(1,5): Type error in constructor
 expression                : Nothing
```

```
    type                    : Maybe a
    expected type           : Int -> b
  probable fix              : use Just instead
```

In the last line of the error message, it can be clearly seen that the sibling heuristic detected that a replacement with the constructor *Just* would solve the issue. To be able to detect this, it was provided to the heuristic that the constructors *Nothing* and *Just* might be confused.
Another common confusion is the difference between the functions *lines* and *unlines*. Therefore, we are able to suggest these functions as siblings as well. Consider the following error and corresponding error message, where the heuristic detected that replacing the function *lines* by the function *unlines* would actually solve the issue:

$g = lines\ [\texttt{"a"}, \texttt{"b"}, \texttt{"c"}]$

```
(1,5): Type error in variable
 expression                 : lines
   type                     : String   -> [String]
   expected type            : [String] -> a
 probable fix               : use unlines instead
```

Their can be multiple possible candidates, each of which will be tried if necessary. The function *zip* has 3 siblings in the `Prelude.type` file, which is the file that specifies the siblings for the Helium Prelude. The *zip* siblings, according to `Prelude.type`, are *zip*3, *unzip* and *unzip*3. If the function *zip* occurs in an error path, each of these siblings is tried and only if one of them fits with the expected types, this function will be reported. If multiple siblings fit, all of these will be reported as candidates.
Beside the ability to detect the siblings of functions, Helium and Rhodium have the ability to detect siblings literals. In these cases, whenever a literal is misused, it might be replaced with a similar literal. Consider the following code and error message, where an *Int* and a *Float* are both used with the $(*)$ operator:

$h = 3 * 0.5$

```
(1,5): Type error in literal
 expression                 : 3
   type                     : Int
   expected type            : Float
 probable fix               : use a float literal instead
```

In this case, a probable fix is reported which is similar to the one used in the function heuristic. The major difference is that in this case, it is advised to use an float instead of a concrete function. The replacement will only be suggested when the type of the literal is actually coming from a literal, not when a variable representing the same type is inspected.
This error message differs slightly from the error message that was previously present in Helium. Where Rhodium suggests to use a float literal, Helium would advise to change the literal 0.5 to an integer. In Rhodium, the change is made to only recommend the change from a float to an integer if the float is a number that can actually be converted to an integer. This is the case in the case of the float 7.0, but not in the case of 3.5. 7.0 can be converted to 7 while keeping the same value. That is not possible for the float 3.5.
In the following example, we show that the heuristic can deal with the difference between a *String* and a *Char*. As a *String* is represented as a type family in Rhodium, this shows that the heuristic is not limited to types without type families, but can recreate the behaviour also for literals of type *String*:

$shout :: Show\ a \Rightarrow a \to String$
$shout\ x = show\ x +\!\!+ \texttt{'!'}$

```
(2,21): Type error in literal
 expression                : '!'
   type                    : Char
   expected type           : String
 probable fix              : use a string literal instead
```

### 7.4.5 Application heuristics

There are 3 different heuristics we will consider to be application heuristics, as they all fulfil a similar purpose. These heuristics are the application heuristic, variable heuristic and tuple heuristic. All of these heuristics try to change the number of arguments in any way to try to discover the cause of the error.
We will first discuss the application heuristic in Rhodium. Consider the following Haskell program and corresponding error message, where we swapped the operands of the (:) operator:

$$f = [\,] : 3$$

```
(1,8): Type error in infix application
 expression                : [] : 3
 constructor               : :
   type                    : [a] -> Int -> b
   does not match          : c   -> [c] -> [c]
 probable fix              : swap the two arguments
```

It can be seen that the error was properly detected as it gives the advice to swap the two arguments. This advice is coming from the application heuristic, that tried to find if there was a permutation of the arguments to the function (:) that would unify with the type signature of (:).
It is also possible for an application heuristic to detect when an argument needs to be inserted. Consider the following example where we forgot to apply the argument $xs$ to the function $map$:

$$g :: [Int] \to [Int]$$
$$g\ xs = map\ (+1)$$

```
(2,8): Type error in application
 expression                : map (+ 1)
 term                      : map
   type                    : (Int -> Int) -> [Int]
   does not match          : (a   -> b  ) -> [a] -> [b]
 probable fix              : insert a second argument
```

All places are checked, so in case that the first argument is missing, like if the function $g$ would be $g\ xs = map\ xs$, the application would notice that inserting a first argument would be able to unify with the provided type signature of $map$.
Another possible error message that Rhodium supports is the detection of too many arguments that are passed to a function, as can be seen in the following program and error message, where we misused the concat function to concatenate two lists.

$$h :: [Int] \to [Int] \to [Int]$$
$$h\ xs\ ys = concat\ xs\ ys$$

```
(2,11): Type error in application
 expression                : concat xs ys
```

```
  term                      : concat
    type                    : [Int] -> [Int] -> [Int]
    does not match          : [[a]] -> [a]
  because                   : too many arguments are given
```

A special case is provided in case that too many arguments were provided when 0 arguments were expected. In that case, the programmer is told that the function they were using is not actually a function.
The application heuristic is capable of detecting when inserting arguments might be a possible fix for the problem. This requires however that there is an application constraint present, which is of the form $f \sim a \to b$. Their are cases in Haskell where we can have a function that should require an arguments, but no arguments were provided. Because no arguments were provided, there is not an application constraint gathered and therefore the application heuristic can not be used. We therefore use the variable heuristic. The usage of the variable heuristic can be seen in the following example:

$$i :: Int$$
$$i = id$$

```
(2,5): Type error in variable
  expression                : id
    type                    : a -> a
    expected type           : Int
  probable fix              : insert one argument
```

The variable heuristic is only capable of detecting when an arguments needs to be inserted, as that is the only error that can occur, as too many arguments or permutation of arguments cannot happen without the presence of an application constraint.
The last heuristic we will consider is the tuple heuristic. This heuristic will detect permutations and too few or too many elements in tuples. Consider for example the following example, were we switched the two elements of the tuple:

$$j :: a \to (a, Int)$$
$$j\ x = (0, x)$$

```
(2,7): Type error in tuple
  expression                : (0, x)
    type                    : (Int , a  )
    expected type           : (a   , Int)
  probable fix              : re-order elements of tuple
```

In this case, similar to the permutations of the application heuristic, the programmer is advised to reorder the elements of the tuples, according to the provided type signature.
We can also have the situation where a 2-tuple and a 3-tuple do not match. In the following example the function $fst$ expects an argument that is a 2-tuple, but gets a 3-tuple. This error is detected and reported to the programmer:

$$k = fst\ \$\ head\ [(1, 2, 3)]$$

```
(1,17): Type error in tuple
  expression                : (1, 2, 3)
    type                    : (Int, Int, Int)
    expected type           : (a, b)
  because                   : a 2-tuple does not match a 3-tuple
```

### 7.4.6 Function binding heuristic

The function binding heuristic is a heuristic that can detect a single possible error, namely that the number of patterns does not agree with the number of arguments given in the type signature. In Haskell, it is perfectly fine to give fewer arguments in the function binding than specified in the type signature, as long as the resulting function is still type correct. It is however not acceptable to provide more arguments than were provided in the type signature (ignoring the usage of type synonyms). Consider the example below:

$f :: Int \to Int$
$f\ x\ y = x$

```
(1,1): Type error in explicitly typed binding
 definition                : f
   declared type           : Int -> Int
   inferred type           : a   -> b -> a
 because                   : the function binding has 2 patterns, but its type signature
                             allows at most 1
```

In this case, the number of arguments for the function binding does not match the number of arguments that needed to be given according to the type signature. Therefore, the heuristic explained how many patterns were expected and how many were given.
The heuristic works by looking at the function binding constraints and the type signature constraint. It counts the number of arguments between them and determines if there is a difference between the number of arguments.

### 7.4.7 Constraint from user

The constraint from user heuristic is the only heuristic that is enabled in all situations in Helium, but is not working in the current implementation of Rhodium. The heuristic, that takes custom constraints and creates custom error messages for certain functions or operators, is implemented, but the infrastructure in Helium has not been converted to deal with the new types of Rhodium. Even though this is mostly an engineering question, as we have currently found no indication that this heuristic should not work, we cannot guarantee that the heuristic will work, as we cannot test its implementation. The part of Helium that is not converted to work with the Rhodium constraints is the part that inserts the custom constraint into the constraint information that will be attached to the function for which the custom rules were specified.

## 7.5 Residual constraint related heuristics

A number of errors were handled by special cases in the inferencer in TOP. For Rhodium, we decided to create special heuristics for these cases. These heuristics are language specific and therefore part of the Helium compiler. The incorrect constraints are constraints that are considered to be residual by the Rhodium type inferencer, but are not involved in GADTs.

### 7.5.1 Type signature too general

A type signature that is too general occurs when the programmer annotated a function with a type signature, like $a \to a$ that is more general than the type of the function that was inferred, like $Int \to Int$. An example of which is the following program and error message:

$f :: a \to b \to c$
$f\ x\ y = [x, y]$

```
(1,1): Type signature is too general
 function                  : f
   declared type           : a -> b -> c
   inferred type           : d -> d -> [d]
 hint                      : try removing the type signature
```

In this case, the definition of the function requires that $a$ and $b$ should be equal and that $c$ should be a list of type $[a]$ or $[b]$. The inferred type of the function $f$ is $d \to d \to [d]$, which is more specific than the provided type signature $a \to b \to c$.
The heuristic works by detecting when a constraint is found residual, as this indicates that some untouchable variable, like $a$, $b$ or $c$ is unified with a constant, an application or another untouchable variable. In such cases, we blame the appropriate type signature. This process also works for type signatures in **let**-bindings that are too general.

### 7.5.2 Missing predicate

There are a number of possible mistakes that can occur, all but one are discussed in this section. The one we will not discuss here has to do with predicates that occur when a GADT is involved. This situation will be discussed in Section 7.6.1.
As Helium supports type classes and instances, we want to be able to correctly explain any missing predicates that occur in overloaded functions and operators. Even though Rhodium supports multi parameter type classes, Helium does not and the heuristics that are implemented do solely support single parameter type classes. Therefore, every example given and the situation described will assume that only single parameter type classes can occur.

The first example we will discuss is actually an error that does not occur in Helium, but does occur in Rhodium. In Rhodium, whenever a predicate needs to be used, it is required to be explicitly mentioned. In the following example, we require an instance for $Num$, but this is not explicitly given, as no type signature was mentioned:

$f\ xs\ ys = sum\ (xs +\!\!+ ys)$

```
(1,1): Type error in overloaded function
 function                  : f xs ys = sum (xs ++ ys)
   inferred type           : [a] -> [a] -> a
   arising from            : sum
 problem                   : a is not an instance of class Num
 hint                      : add a type signature to the function
```

The error message has been designed specifically for this situation. Ideally, we want to infer these types including the type signatures, but this is not in the scope of this thesis. We therefore require type signatures for functions that use a predicate. With some slight modifications to the type inferencer, it should be possible to automatically infer the type signature of functions that use predicates.
It might also be the situation that a type signature is provided, but that there are constraints missing from the provided type signatures. Consider the following example:

$g :: [a] \to [a] \to a$
$g\ xs\ ys = sum\ (xs +\!\!+ ys)$

```
(1,1): Missing class constraint in type signature
 function                  : g
   declared type           : [a] -> [a] -> a
 class constraint          : Num a
 hint                      : add the class constraint to the type signature
```

In this case, the constraint $Num\ a$ should be added to the type signature, as indicated by the error message. Another case is the situation that we are missing the instance for a concrete type. We will have this case the predicate describes an constructor or an application, like $Num\ Bool$ or $Eq\ [a]$. An example of such a situation and the error message Rhodium produces is as follows:

$$h = True + False$$

```
(1,10): Type error in overloaded function
 function                  : +
   type                    : Num b => b    -> b    -> b
   used as                 :          Bool -> Bool -> a
 problem                   : Bool is not an instance of class Num
 hint                      : valid instances of Num are Int and Float
```

This heuristic will only report the missing heuristic after they are simplified. If we were to require an instance of $Eq\ [a \to a]$, the error would report a missing instance $Eq\ (a \to a)$, as the conversion from $Eq\ [b]$ to $Eq\ b$ can be made.
The final instance can occur whenever there is a residual predicate, but the type variable in the residual predicate cannot be traced back to an origin in the function binding or type signature. In that situation, like in the following example, the predicate is considered ambiguous and an explicit type signature is necessary to determine which instance needs to be chosen:

$$i = show\ [\,]$$

```
(1,5): Don't know which instance to choose for Show
 function                  : show
   type                    : Show a => a   -> String
   used as                 :           [b] -> c
 hint                      : write an explicit type for this function
                                 e.g. (show :: [Int] -> String)
```

In Helium, the code in the example does not create an error, as there is a defaulting mechanism that chooses a default type whenever a certain class is considered ambiguous. One of these classes that has this defaulting mechanism is the *Show* class. This default mechanism is not built into Helium nor Rhodium and therefore the code produces an error.

## 7.6 GADT related heuristics

We will now consider the errors that can occur whenever GADTs are introduced. There are 3 separate heuristics created for this purpose, as well as an addition to the missing predicate heuristic, which requires a special case for the missing predicate in a GADT constructor.

### 7.6.1 Missing predicate in GADT constructor signature

The behaviour of a missing predicate is implemented as a part of the missing predicate heuristic. It tries to insert the predicate in the type signature of the constructor, if the predicate cannot be inserted in a local definition, like the type signature of the function. If, 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, we would prefer this over adding a predicate to the constructor. The main reason for this choice is that adding a predicate to the constructor has arguable a larger impact than adding a predicate to the local type signature, as this would only require the predicate to be satisfied whenever it is actually used, not whenever the constructor is used.
An example of a situation where the predicate can only be added to the constructor is in the following example:

**data** $X$ **where**
  $A :: c \rightarrow X$
$f :: X \rightarrow String$
$f\ (A\ x) = show\ x$

In this case, the type of the variable $x$ is not mentioned in the data type $X$, so we can not add the predicate to the function. See the following error message:

```
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)
```

We provided the class that needs to be added and the location of the type signature to which the predicate should be added.

### 7.6.2 Unreachable pattern

Within a GADT, the type signature can make certain pattern matches not accessible. An example of this can be seen below:

**data** $X\ a$ **where**
  $A :: Bool \rightarrow X\ Bool$
  $B :: Int \rightarrow X\ Int$
$g :: X\ Int \rightarrow Int$
$g\ (A\ x) = x$
$g\ (B\ y) = y$

In this case, we said that the type signature of $g$ should only allow arguments of type $X\ Int$. In the constructor $A$, we require a parameter of type $X\ Bool$, which cannot be passed to the function. This causes a 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                   : A x
   constructor type        : Bool -> X Bool
```

```
  defined at              : (2,4)
  inferred type of        : a -> X Int
  pattern
hint                      : change the type signature, remove the branch or change the branch
possible type signature   : (X b) -> b
```

The type of the constructor and the inferred type can be found in the pattern, as well as the location of the definition of the GADT. Notice that a possible type signature is provided that would allow the pattern match to be kept. This type signature is based on the most general type that can be made 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 a possible type signature. The type signature that was provided by the programmer is not taken into account when coming up with a new type signature.

### 7.6.3 Missing GADT type signature

GADTs require a type signature, as there are multiple valid type signatures for most GADTs. We could have made the decision to make the detection of not providing a GADT type signature a static check, but we decided not to do that. If we would have done the static check that every GADT would require a type signature without any type checking being performed and we would not be able to use the type graph to infer a possible type for the GADT. As can be seen in the following example, a type signature is missing, but a type signature that would resolve the error is actually inferred and reported to the programmer. The process to determining this type signature is very similar to the process described for the inferencing of type signatures for unreachable patterns:

**data** $X\ a$ **where**
  $A :: Bool \to X\ Bool$
  $B :: Int \to X\ Int$
$g\ (A\ x) = x$
$g\ (B\ x) = x$

The error message provides the possible type $(X\ a) \to a$ as a suggestion, but other type signatures might also be possible. Therefore, we keep the type signature as a hint, not as a mandatory fix that has to be followed.

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

### 7.6.4 Non-unifiable GADT variables

Some variables inside a GADT cannot be unified in certain cases, as they are existential variables. Consider the following example:

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

In this case, we unify the variable $x$ of type $b$ with the type $Bool$, but the variable $b$ is coming from the type

signature $b \to X$, which does not allow $b$ to unify with *Bool*. The following error message, which Rhodium produces given the program above, shows that this variable cannot be unified.

```
(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)
```

It shows that the existential type $b$ cannot be unified with the type *Bool*. We also give the original constructor, as well as the location at which it is defined. This heuristics works on a residual constraints of the shape $a \sim b$ where $a$ is the non-touchable variable and $b$ can be any type. We can see in the type graph if the $a$ is coming from a pattern match and whether that variable shows up in the result of the pattern match. If the variable is from the type signature in the pattern match, but not in the resulting type of the pattern match, this heuristic is applied. An example of a variable that occurs in a pattern match, but not in the result is the variable $c$ is the type signature $c \to d \to Z\ d$.

## 7.7 Interaction between heuristics

In this section, we take a look at the interaction between different errors, and between different errors which might interfere with one another.

### 7.7.1 Interaction between existing and new heuristics without GADTs

There were a couple checks, for example that the type signature of a function is not too general, that were performed by dedicated pieces of code in TOP, as opposed to being detected in the type graph and resolved by a heuristic. Consider the following example, in which both a too general type signature is provided, as well as a type error that occurs:

$f :: a \to b$
$f\ x = 3\ ||\ True$

```
(2,7): Type error in infix application
 expression                : 3 || True
 operator                  : ||
   type                    : Bool -> Bool -> Bool
 left operand              : 3
   type                    : Int
   does not match          : Bool
```

In the resulting error message, only the type error is indicated. This error message is identical to the type error that is given by TOP for this specific example. In this case, the type signature is too general heuristic did not ever contribute to the type error diagnosis process, as it could not do anything with the first constraint that was given, which was $Bool \sim Int$. The inferencer detected also the residual constraint $b \sim Bool$, but this error was resolved by the resolving of the type error, indicating that the type inferencer, in combination with the heuristics, is capable of resolving multiple problems with a single solution.
In some cases, the type signature being too general does occur at the same time as another error. Consider the following example, in which we switched the elements of the tuple. Notice how the type signature does

not match with the result of the function:

$$g :: a \to b \to (a, b)$$
$$g\ x\ y = (y, x)$$

Even though both the heuristics regarding tuples and too general type signatures did fire, because of the weight of the different heuristics, the error message regarding the tuples was chosen. We would argue that this is a better choice than reporting that the type signature is too general, as re-ordering the elements of the tuple fixes the type signature that was provided, while the error message that the type signature was too general would not have offered any suggestions on how to resolve this error. The error message that Rhodium produces is as follows:

```
(2,9): Type error in tuple
 expression                : (y, x)
   type                    : (b , a)
   expected type           : (a , b)
 probable fix              : re-order elements of tuple
```

There is also an error in which there are 2 heuristics that both act on different pieces of code. Consider the following example and corresponding error messages:

$$f :: a \to (Bool, a)$$
$$f\ x = \textbf{let}$$
$$\quad y = 3\ ||\ True$$
$$\quad \textbf{in}\ (y, \texttt{"a"})$$

```
(1,1): Type signature is too general
 function                  : f
   declared type           : a -> (Bool , a     )
   inferred type           : b -> (c    , String)
 hint                      : try removing the type signature

(3,8): Type error in infix application
 expression                : 3 || True
 operator                  : ||
   type                    : Bool -> Bool -> Bool
 left operand              : 3
   type                    : Int
   does not match          : Bool
```

In this case, 2 errors occur. This is because both errors are unrelated, something which the type inferencer and the heuristic are aware of. We would actually argue that Rhodium is better than TOP in this case, as TOP only produces the second error and is not capable of combining different type of errors together, like a type inconsistency and a type signature that is too general.

### 7.7.2 Interaction between heuristics with GADTs

We are also interested in interactions between the heuristics and the GADTs, especially when errors occur that have nothing to do with the GADT. Consider the following example, where we have a small expression language and an evaluation function:

$$
\begin{aligned}
&\textbf{data } Expr\ a \textbf{ where}\\
&\quad I :: Int \to Expr\ Int\\
&\quad B :: Bool \to Expr\ Bool\\
&\quad Plus :: Expr\ Int \to Expr\ Int \to Expr\ Int\\
&eval :: Expr\ a \to a\\
&eval\ (I\ x) = x\\
&eval\ (B\ b) = b\\
&eval\ (Plus\ x\ y) = x + y
\end{aligned}
$$

Notice how we forgot to evaluate the arguments of the plus constructor. As there are two places where we have forgotten the evaluate, there are two error messages that are reported:

```
(10,19): Type error in variable
 expression               : x
   type                   : Expr Int
   expected type          : Int

(10,23): Type error in variable
 expression               : y
   type                   : Expr Int
   expected type          : Int
```

Even though the $(+)$ operator is overloaded, the type inferencer will not say that we need to given in instance of $Num\ (Expr\ a)$, as this would not solve the error. From the type signature, we can infer that the resulting type of this branch should be of type $Int$ and therefore, the result of the operator $(+)$ should be an $Int$, which results in the mismatch between the expected type $Int$ as the result and the arguments, which are of type $Expr\ Int$.
We can compare the resulting error message from Helium with the error message that is given by GHC[1] for this program:

```
Comparison1.hs:10:19: error:
    • Couldn't match type ‘Expr Int’ with ‘Int’
      Expected type: a
        Actual type: Expr Int
    • In the expression: x + y
      In an equation for ‘eval’: eval (Plus x y) = x + y
   |
10 | eval (Plus x y) = x + y
   |                   ^^^^^
```

In this case, GHC only produces one error message, namely that the type of `Expr Int` does not match the type `Int`. In the expected type however, it says that its expected type is `a`.
We would argue that in this example, we would prefer two error messages as opposed to one. As none of the arguments $x$ and $y$ is accepted as an argument for the function $(+)$, we want to blame them both individually. Even if we would change the body to the function $eval\ (x+y)$, we would lack an instance of $Num\ (Expr\ Int)$.
We can also extend the expression language and the evaluation function, introducing some mistakes in the new branches of the evaluation function. Consider the following example:

[1] GHC 8.2.2

$$
\begin{aligned}
&\textbf{data } Expr\ a \textbf{ where}\\
&\quad I :: Int \to Expr\ Int\\
&\quad B :: Bool \to Expr\ Bool\\
&\quad Plus :: Expr\ Int \to Expr\ Int \to Expr\ Int\\
&\quad Equals :: Eq\ a \Rightarrow Expr\ a \to Expr\ a \to Expr\ Bool\\
&\quad Max :: Expr\ Int \to Expr\ Int \to Expr\ Int\\
&eval :: Expr\ a \to a\\
&eval\ (I\ x) = x\\
&eval\ (B\ b) = b\\
&eval\ (Plus\ x\ y) = eval\ x + eval\ y\\
&eval\ (Equals\ x\ y) = eval\ x\ y\\
&eval\ (Max\ x\ y) = maximum\ (eval\ x)\ (eval\ y)
\end{aligned}
$$

There are two unrelated errors, one in the branch that checks the equality of expressions and the other is the almost traditional mistake that is used in this thesis, the confusion of *max* and *maximum*. The following error message is reported by Rhodium:

```
(12,21): Type error in application
 expression               : eval x y
 term                     : eval
   type                   : Expr a -> Expr a -> Bool
   does not match         : Expr b -> b
 because                  : too many arguments are given

(13,18): Type error in variable
 expression               : maximum
   type                   : Ord a => [a] -> a
   expected type          :          Int -> Int -> Int
 probable fix             : use max instead
```

The error message identified both errors correctly and was not confused about the presence of a predicate in the constructor of *Equals*. It also correctly identified the return type of the incorrect usage of *eval* which is reported as *Bool*, due to the type signature of *eval*.
We can compare this error message to the error message that GHC[2] produces for this program:

```
Comparison2.hs:12:21: error:
    • Could not deduce: a1 ~ (Expr a1 -> Bool)
      from the context: (a ~ Bool, Eq a1)
        bound by a pattern with constructor:
                   Equals :: forall a. Eq a => Expr a -> Expr a -> Expr Bool,
                 in an equation for ‘eval’
        at Comparison2.hs:12:7-16
      ‘a1’ is a rigid type variable bound by
        a pattern with constructor:
          Equals :: forall a. Eq a => Expr a -> Expr a -> Expr Bool,
        in an equation for ‘eval’
        at Comparison2.hs:12:7-16
      Expected type: Expr a1 -> a
```

[2]GHC 8.2.2

```
        Actual type: a1
    • The function ‘eval’ is applied to two arguments,
      but its type ‘Expr a1 -> a1’ has only one
      In the expression: eval x y
      In an equation for ‘eval’: eval (Equals x y) = eval x y
    • Relevant bindings include
        y :: Expr a1 (bound at Comparison2.hs:12:16)
        x :: Expr a1 (bound at Comparison2.hs:12:14)
   |
12 | eval (Equals x y) = eval x y
   |                     ^^^^^^^^

Comparison2.hs:13:27: error:
    • Couldn't match type ‘Int’ with ‘t0 (Int -> Int)’
      Expected type: t0 (Int -> a)
        Actual type: Int
    • In the first argument of ‘maximum’, namely ‘(eval x)’
      In the expression: maximum (eval x) (eval y)
      In an equation for ‘eval’:
          eval (Max x y) = maximum (eval x) (eval y)
   |
13 | eval (Max x y) = maximum (eval x) (eval y)
   |
```

Similar to the Helium error message, two error messages are given by GHC for the two errors that the program contains. In the first error message, it blames `a1 ~ (Expr a1 -> Bool)`. We would argue that this error message is worse because it introduces new variables, like `a1`. It also mentions a context `(a ~ Bool, Eq a1)`, which we never provided.
In the second error message, it mentions that it could not match `Int` with `t0 (Int -> Int)`, which is inconsitent with the next line in the error message, that says the expected type is `t0 (Int -> a)`. Nowhere in the error message is the type variable `t0` introduced, neither is mentioned that `maximum` should have gotten less arguments.

## 7.8 Summary

A number of heuristics have been shown in this section. These heuristics can be divided into two separate categories. We first described the heuristics that were already implemented for type graphs, either in Helium or in TOP. These heuristics have been adapted to work on the type graphs that Rhodium supports. We showed that these heuristics also work when more advanced language features were involved, like GADTs.
Beside the already existing heuristics, we also designed and implemented a few heuristics that were not yet available for type graphs. Some of these heuristics were for errors that were previously not detected using heuristics, like a too general type signature. Other heuristics were designed for new language features, like GADTs, that were previously not supported by type graphs.

# 8 Conclusion

This thesis has shown that, in answer to our first research question, as posed in Section 3, it is possible to transfer the quality of error message currently available in Helium to a type inferencer that is capable of handling GADTs and other extensions to the Haskell 2010 standard. We did this by developing the type inferencer Rhodium. The examples that use Rhodium show in our opinion that the quality of error messages remained the same as the error messages that were already available in Helium.
The type inferencer Rhodium has the capability to be extended with other language constructs due to the parametric constraint solver X. This is because the type inferencer and the type graphs are developed independent of the constraint solver X and the available types. This modular approach allows us to implement new constructions into Helium without the need to redesign the type inferencer.
We showed that it is also possible to develop heuristics that improve the quality of error message of GADTs, in answer to our second research question. We added a number of heuristics that were specifically designed to deal with the addition of GADTs, as well as developing some heuristics for a number of errors that were previously detectable, but not by special heuristics, but rather by the type inferencer itself. These heuristics allow us to report error messages that we consider to be of a higher quality than the error messages in GHC.
We feel that this work demonstrates that the addition to of GADTs does not pose a problem from the viewpoint of type error diagnosis. The errors when no GADTs are involved have a quality that we consider equal to the quality of the error messages of TOP, even when a type inferencer is used that is capable of handling GADTs. When GADTs are involved, the error messages are of sufficient quality to allow programmers to easily spot any errors that were detected. These error messages should give enough information to repair the mistake.

# 9 Future work

## 9.1 Heuristics for language extensions

Having a type inferencer that has an implementation for OutsideIn(X), where the X is parametric, it could be possible to investigate whether it is possible to develop heuristics for other language constructions that might be available. The X in Rhodium already supports multi parameter type classes and type families, but Helium does not support these constructions. Research into the quality of these other construction might be interesting and useful.

## 9.2 Type inference for overloaded functions

Currently, Rhodium cannot infer the types of functions that have an overloaded type, for example the function $plus\ x\ y = x+y$, as the type of this function would be $Num\ a \Rightarrow a \rightarrow a \rightarrow a$. We would like the type inferencer to be able to infer the types of these functions.

## 9.3 DSL support for Rhodium

Heeren [10] developed a number of directives that can be used in combination with DSLs to increase their error diagnosis. These directives, the never directive, the disjoint directive, the close directive and the defaulting directive, could be useful to implement in Rhodium. We expect the defaulting directive to be the hardest of all of these directives, as this directive could make a program that is rejected by the simplifier correct again. Beside the directives, the constraint from user heuristics is also useful when developing error messages for DSLs.

## 9.4 Optimizations to Rhodium

Rhodium has been developed as a proof of concept implementation and is therefore not optimized for large scale usage. We would therefore recommend that research is done in optimizing the simplification and error diagnosis process. Possible optimizations might include reducing the number of simplifications necessary, allowing only a part of the graph to be reset whenever the graph is modified and using a simplified solver without type graphs when no type error diagnosis is needed. Ideally, this simplified graph could use the same instantiation of X as the type graph solver.

## 9.5 Support for language extensions for Helium

The X in Rhodium could possible support language extensions other than those already discussed in this thesis. A general framework for developing these extensions, including research whether it is possible to develop heuristics for these extensions and how they interact with the other extensions and their heuristics, would be useful.

## 9.6 Specialized error messages for novice programmers

With the extension of the possible language constructions, it becomes harder to give clear error messages for all types of programmers. We would therefore propose a system to indicate to the compiler the level of knowledge that the current programmer has. Based on this knowledge, the error messages might be more personalized. Consider the type $Foldable\ t \Rightarrow (a \to b \to b) \to b \to t\ a \to b$ of the function $foldr$. If we were to allow such a type signature in the Helium Prelude, any novice programmer needs to know about type variable application, type classes and the *Foldable* class in order to understand this type signature, let alone the meaning of the error message the type signature is involved in. If we could know that an error message would be displayed for a novice programmer, we could display the type in an error to be $(a \to b \to b) \to b \to [a] \to b$, by defaulting the type variable $t$ to $[\,]$. Other simplifications in error messages might also be useful, depending on the skill of the programmer. This could allow for complicated function definitions in the Helium Prelude, without compromising the clarity of error message for novice programmers.

### 9.6.1 New heuristics for Haskell 2010 features

There are a number of possible heuristics that might be useful to implement into Helium, either for TOP or for Rhodium. The first heuristic deals with the addition and removal of so-called ticks, that turn a function into an operator. An example of this would be $a\ `div`\ b$, which can be confused with $a\ div\ b$. The heuristic could try to change a constraint of the form $a \sim div \to b \to result$ to $div \sim a \to b \to result$. The removing of ticks, where the ticks were provided accidentally, is also an option.
The other heuristic would be a variant of the siblings heuristic, but would use the available variables as siblings. For example if a programmer typed the variable $x$, which caused a type error, this heuristic might use the variable $y$, which is also in scope, to see if this fixes the type error. This would require however knowledge which variables are in scope at every moment.

# 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] Ivo Gabe de Wolff. The Helium Haskell compiler and its new LLVM backend, 2019.

[4] 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.

[5] Adam Gundry. A typechecker plugin for units of measure: Domain-specific constraint solving in GHC Haskell. In *ACM SIGPLAN Notices*, volume 50, pages 11–22. ACM, 2015.

[6] 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.

[7] Jurriaan Hage. Domain specific type error diagnosis (DOMSTED), 2014.

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

[9] 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.

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

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

[12] Bastiaan Heeren, Jurriaan Hage, and S Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms.

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

[14] 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.

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

[16] 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.

[17] 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.

[18] 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.

[19] Daan Leijen. LVM, the lazy virtual machine. Technical report, Citeseer, 2002.

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

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

[22] 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.

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

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

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

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

[27] Alejandro Serrano and Jurriaan Hage. Context-Dependent Type Error Diagnosis for Functional Languages, 2016.

[28] Alejandro Serrano and Jurriaan Hage. Type error diagnosis for embedded DSLs by Two-Stage specialized type rules. In *European Symposium on Programming*, pages 672–698. Springer, 2016.

[29] Alejandro Serrano and Jurriaan Hage. Constraint handling rules with binders, patterns and generic quantification. *Theory and Practice of Logic Programming*, 17(5-6):992–1009, 2017.

[30] 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.

[31] Alejandro Serrano and Jurriaan Hage. A compiler architecture for domain-specific type error diagnosis. *Open Computer Science*, 9(1):33–51, 2019.

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

[33] Martin Sulzmann. A general framework for Hindley/Milner type systems with constraints. 2000.

[34] 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.

[35] 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.

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

[37] Jun Yang, Greg Michaelson, Phil Trinder, and JB Wells. Improving Polymorphic Type Error Reporting. 2000.

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