Type Directives in Elm

Falco Peijnenburg

MSc Thesis (ICA-3749002)

June 28, 2016

Universiteit Utrecht

Center for Software Technology
Dept. of Information and Computing Sciences
Utrecht University
Utrecht, the Netherlands

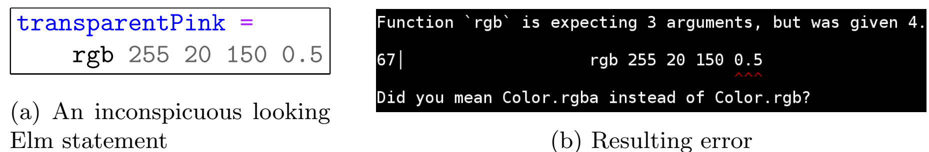
First supervisor:
dr. J. Hage
Second supervisor:
dr. B. Heeren

Introduction

Error messages in functional programming languages can at times be daunting. Generally, a compiler is very strict in the programs it does or does not consider valid. The errors thrown by compilers often reason about the types of expressions. These types can at times become rather involved, rendering the error messages harder to read. Elm is an example of such a strictly typed functional programming language, though despite this, Elm is famous for its nicely worded and understandable error messages.

The main author of Elm, Evan Czaplicki, has put great amounts of effort into getting Elm's type error messages where they are now. Some techniques involve suggesting alternatives for misspelled record fields, highlighting the difference of two conflicting types and displaying the source code containing the error without any reformatting. In this spirit, this thesis attempts to improve error messages even further. Improving an already good system of error messages is taken on from two directions: investigating the context of an error more thoroughly before throwing an error and letting experienced library programmers take control over error messages.

Let us take a look at some of the contributions of this thesis. In figure 1 we can see some color, transparent pink, being defined in terms of three integers and a float. Sadly, this code is incorrect. The error shows that `rgb` takes 3 arguments, but has been given 4. This error, its clear description of the problem and the red underlining of the fourth argument are part of Elm's famously understandable error messages. The "Did you mean" hint below the code is one of the major contributions of this thesis. This hint adds valuable information, as it tells us how this problem can be resolved. It appears to have some insight about the similarity between the functions `rgb` and `rgba` and has figured that the programmer might have confused the two.



(a) An inconspicuous looking Elm statement

(b) Resulting error

Figure 1: A confusion between the functions `rgb` and `rgba`.

Error messages that suggest corrections based on conceptually similar functions can be very helpful for programmers. The improvements do not stop there, however. Type directives give authors of functions fine tuning control over the hints added to the error messages that are shown when their functions are not

```
The 1st argument of function `const` is in conflict with the return type.

227|          const "str" True
              ^^^^^^^^^^^^^^^^
Function `const` is expecting the return value to be:

    String

But the context requires:

    Bool

Hint:
The author of function `const` gives the following explanation:
  It's the first parameter that gets returned, not the second.
```

Figure 2: An error message with a hint from the author of the function.

used correctly. This fine tuning control makes it possible for error messages to explain the problem as though they understand the meaning of the function. One example of such explanation can be seen in figure 2. The expression that caused this error is `if const "str" True then "foo" else "bar"`.

This thesis expands on earlier research done by Hage and Heeren [18] [14], the main authors of the Helium Haskell compiler along with van IJzendoorn, van Haaften and Leijen. The Helium compiler uses a graphing technique in combination with heuristics to find the best explanation for error messages. This is part of a continuing effort to improve error messages. The bar for these improvements is set by Yang et al [41], who have written a manifesto describing the requirements of good error messages. According to that manifesto, error messages are to be correct, precise, succinct, non-mechanical, source-based, unbiased and comprehensive. Type graphs are particularly adept at removing bias and being comprehensive. Along with Elm's focus on precise, succinct and source-based errors, compiler errors can be made a rather pleasant experience, rather than a bothersome weight on one's shoulders.

The next chapter describes the literature related to this thesis. The chapter after that will describe the research question. Chapter 3 argues for the relevance of the research to modern science, technology and society. Chapter 4 gives a brief overview of Elm's architecture. Chapter 5 gives a small overview of the contributions of this paper. Chapters 6, 7, 8 and 9 go in-depth into type graphs, siblings, interfaces and specialized type rules respectively. The last chapter concludes this thesis.

Chapter 1

Literature review

In 2013, Evan Czaplicki and Stephen Chong [9] published the paper that describes the essence of the Elm programming language. Elm was designed to easily create responsive graphical user interfaces using functional programming. Up until version 0.17, this involved functional reactive programming. Some design goals of Elm are to be simple, easy to learn, purely functional and to have a clean syntax. Lately, a great amount of effort has been put into improving Elm's error messages. Elm's current error messages (as of version 0.17) already help the beginning programmer a great deal, but as the types of a program get more involved, the type errors become more complicated. This is especially a problem in uses of libraries that encompass embedded Domain Specific Languages (eDSLs), which often create complicated type structures to create the ideal syntax. This thesis will focus on improving error messages generated by Elm. Specifically, by giving library/eDSL writers control over them by means of "Directives". This document presents some context on the knowledge required to do this.

In 1996, Hudak [21] stressed the importance of embedded Domain Specific Languages. Using a strongly typed host language will provide many benefits to the DSL. One benefit is that the DSL can leverage the compiler and type checker from the host language. The DSL can be modelled to represent the abstraction for a particular domain, while being able to interact with other DSLs. One problem though, is that the underlying types of DSLs can become quite involved, especially when the type system is abused to enforce certain properties upon the language. End users are often confronted with these complex types when an error is made. Ideally, DSL writers should be able to hide the complex types from those who do not need to learn about their intricacies.

We will start by focussing on Functional Reactive Programming. We describe several approaches: Functional Reactive Animation (Fran), Elm's first order FRP, Reactive-Banana and FRPNow. After that, we will delve deeper into type inferencing, particularly by describing Algorithms W, M and U_{AE}. Section 1.3 stresses the importance of type errors and describes some desired properties. Section 1.4 focusses on type error diagnosis and improving the quality of error messages. Several techniques are outlined: explanation systems, automatic reparation systems, error slicing, type graphs, counterfactual typing, type error debugging and type directives. Finally, the literature review is concluded.

1.1 Functional Reactive Programming

A reactive program is one that can continuously interact with its environment while running. Common examples of reactive programs are robotics, user interfaces and games, both of which should be able to handle input from an end user immediately and at any time. The pure nature of many functional programming languages has made reactivity somewhat awkward, since interaction with the outside world is forced into IO Monads or IO streams. The solution for that problem comes in the form of a new paradigm: Functional Reactive Programming (FRP). FRP allows the programmer to model the behaviour of their reactive program declaratively. We will now discuss various views on and implementations of FRP.

Fran

Elliott and Hudak's paper on Functional Reactive ANimation [11] is attributed to introducing the concept of FRP. It is a Haskell library that implements data structures and helper functions to allow programming in the FRP paradigm, specifically for animations. At the basis lie a set of mutually recursive data structures: Behavior and Event. A Behavior can be seen as holding a value that changes over time, e.g. mouse coordinates. Events are represented as tuples that contain the time when an event occurred and some value containing information about the Event. This allows a program to be declared to have some behavior until some event occurs, after which a different behavior can be assumed.

The interval based event system forces Fran to recalculate the entire interface multiple times per second to see if any events are generated and whether these events change anything in the interface. It also suffers from Haskell's lazy evaluation: one can take a snapshot of a Behavior value at any given time, but the value contained in that snapshot will not be calculated until it is needed. Since Haskell has not calculated at what exact point the value was to be stored, its lazy mechanism stores every observed value. This causes severe space leaks. Finally, Fran does not provide an easy means for FRP programs to perform IO actions.

FRPNow

The FRPNow library, described by Atze van der Ploeg and Koen Claessen [37] modifies Fran's interface and semantics with the goal to solve the space leaks and to provide a means to have FRP programs interact with the outside world. To find space leaks, they define a notion called "*equality up to time-observation*" that can prove whether an FRP function allows its implementation to forget the past. After all, there need not be space leaks from remembering values from the past when the past can be forgotten.

IO actions are made possible by an **async** primitive, which fires off a given IO event in a Now monad and immediately returns an event that will fire when the IO action has finished. This means that actions in the Now monad are never blocked by IO actions. The results of the IO action can be processed when the event returned by **async** is fired.

While the space leaks and IO problems are solved, FRPNow still samples at a certain interval. Like Fran, the formal semantics of FRPNow are defined with continuous time in the formal syntax. In the implementation, this is approached by sampling at a constant interval. At every sample point the FRP code is executed to update the values stored in the Behaviors. This can cause trouble when the amount of time to recalculate the FRP code is larger than the sample interval. This problem becomes more and more apparent as the sample interval is chosen to be closer to 0.

Reactive-Banana

Reactive-Banana is an FRP library for Haskell written by Heinrich Apfelmus [1]. Reactive-Banana is a push-driven implementation of FRP, based on Elliot's paper [10]. Elliot described a new definition of FRP that only recomputes values when necessary. One important difference between Fran/FRPNow and Elliot's approach is that Elliot implements both continuous and discrete representations of time. This is done by splitting up the concept of behavior into a discrete part called "reactive value" and a continuous part called "time function". Events are changed to contain only reactive (discrete) values. The discrete representation of time makes this method much more efficient.

Elm

Czaplicki and Chong [9] took a different path for FRP by creating Elm. Elm is a purely functional language designed for FRP. It is focussed on the design of user interface, being strict in evaluation and event driven. Elm combines the concepts of Behaviors and Events into a single concept called Signals. Like Behaviors in Elliot's [10] work, Signals contain values that can change over time. Changes in Signals are triggered by outside events. By means of signal graphs, calculated at compile time, Elm can calculate which parts of an Elm program are affected by a change in a signal. This allows for efficient, event-based updates. The signal graph also allows the compiler to make Elm programs concurrent without the programmer having to change their code, but this is not currently implemented because concurrency is currently not practical in Elm's target language, Javascript.

IO inputs and outputs are delegated outside of Elm through a language construct called *ports*. Ports allow Elm to communicate with its host language (currently Javascript). A programmer can indicate that some task is to be performed by sending a value representing this task through a port. Some Javascript code will listen to this port and execute the task. When finished, it passes any results back to Elm. The downside of this approach is that Elm cannot perform the tasks itself and depends on its host language to do the heavy lifting. The benefits are, however, that Elm has full access to all of the capabilities of its host language. Besides, Elm can remain purely functional and requires no representation of the "real world", like Haskell does.

More about Elm can be read in Czaplicki's thesis [6].

1.2 Type Inferencing

One of the great benefits of the functional programming paradigm is that the compiler can automatically infer the types of programs, saving the programmer work in annotating expressions. Over the years a myriad of type inferencing algorithms have been described. In this section, we walk through the most notable type inferencing algorithms. We start with the earliest algorithms W and M and then continue with some of the improvements made upon them.

Algorithm *W* and *M*

The Hindley-Milner type system, also called Damas-Milner was first described by Hindley [20], and later rediscovered by Milner [28]. Its corresponding algorithm, called W, is one of the first algorithms that automatically infers the types of the typed lambda calculus. It infers types from the bottom up, unifying types on the go to find the most general types of expressions. Polymorphism is allowed, but only in `let` expressions.

Algorithm M is similar to algorithm W, but works top down. Algorithm M is a folklore algorithm, meaning that its discovery is not attributed to a single author. Lee and Yi [25] have proven that algorithm M generally finds type errors before algorithm W does. The same paper also proves that combining the two algorithms can give a strictly more precise location of the error than either of them can alone.

Removing Bias

One of the biggest problems of the traditional inference algorithms is the left-to-right bias. McAdam [27] describes an alteration of W and M in which the unification is modified to remove bias by treating subexpressions symmetrically. Another approach is described by Yang [40], who describes algorithms U_{AE} and IEI. The first removes left-to-right bias by typing subexpressions independently and unifying assumption environments at the top level of the AST. U_{AE} is more comprehensive than the traditional algorithms because it can explain that variables are used inconsistently at several locations in the program. However, U_{AE} suffers from the same problem as W, finding errors only in applications. When this happens, a switch is made to algorithm M to narrow down the error locations. This combination of W and M is what defines algorithm IEI.

Constraint-Based Inference

A type inferencing algorithm does two things: finding out how subexpressions are linked and finding a way to assign types such that all these links are respected. Traditional algorithms interweave these two processes into a single - greedy - algorithm, which unifies types and generates substitutions as soon as they are able. Over the years, the idea has arisen to separate these two into separate algorithms. This idea revolves around the concept of constraints. Instead of immediately generating substitutions, the inferencing algorithm generates constraints. After all constraints are generated, they are sorted and solved in a separate algorithm. This has the benefit of being able to remove bias and investigating multiple aspects of a single type error. Besides that, the split in responsibilities is a good example of the "separation of concerns" principle

One implementation of this idea can be found in ML, as described by Pottier [32], and more generally by Odersky et al. [29]. Another implementation of this idea can be found in Helium, a Haskell compiler built for students. This implementation is described by Heeren and Hage [15]. Constraints can be generated bottom-up or top-down. Heeren describes both approaches, whereas Pottier describes only bottom-up.

Extensible Records

Data structures are important in any real language. Elm has two kinds of data structures built into the language. The first is Algebraic Data Types (ADT), which is similar to Haskell's data types. The second and more interesting one is extensible records, as described by Leijen [26]. Extensible records contain values that can be indexed with some key. New keys and values can be added to the records, values of existing keys can be removed or changed and records can be indexed to retrieve specific values. All these operations are type safe, meaning that invalid record operations are detected at compile time. Elm implements this idea, but it does not support the addition or deletion of fields. These were supported until version 0.16 [7]. The features were removed to allow for more program optimisations and because it encouraged overly complex code.

Type Classes

Type classes were introduced by Philip Wadler and Stephen Blott [38]. Wadler and Blott argue for type classes to be the standard solution for ad-hoc polymorphism. Ad-hoc polymorphism, as opposed to parametric polymorphism, is defined for a finite set of types, and acts differently for each type. One example is the multiply operator `(*)` which has different behaviours for integers and floating numbers. The solution for type classes allows the same function to work on any type that defines the specific implementation of the function for that type. In the example, `(*)` works on any type that has an instance of the `Num` class. Ints and floats declare themselves as instances of the `Num` class by providing the unique implementations for the numerical functions and operators (including `(*)`).

In the compiler, programs with type classes and instances are translated into equivalent programs that lack those features. This is done by turning instances of type classes into dictionaries. The functions in the type classes are modified to take such dictionary as an extra parameter. This allows for a rather easy implementation of type classes in existing languages.

Qualified Types

Mark P. Jones describes the generic concept of qualified types [23]. Qualified types are a step between monotypes and parametrically polymorphic types. By assigning predicates to type variables, a programmer can define functions that are not restricted to monotypes, but are also not too generic. Type classes and extensible records are instances of qualified types. Specifically, the demand that a given type is an instance of a type class (as in, `Eq a => a -> a -> Bool`) are the predicates. Types that have an instance of a type class (e.g. `Eq Num`) fulfill the predicate.

Extensible records are an instance of qualified types with predicates that demand certain fields to be either present or lacking in a given record. This allows functions to work with records that contain *at least* some specific fields, but are allowed to contain more.

The paper describes the type rules for qualified types and extends algorithm W to infer qualified types.

1.3 Manifesto on Error Messages

Despite the fact that combining W and M gives more precise error locations, the algorithm is not ideal in providing error messages. Yang et al. [41] have written a manifesto which describes the properties of good error reports:

- Correct
  - Correct detection - Errors are reported when the program is not legal.
  - Correct reporting - The reported sites of the error contribute to the type conflict.
- Precise - Conflicting sites should be located in the smallest useful amount of source text.
- Succinct - Maximise useful information while minimising non-useful information.
- Non-mechanical - The error should not bother the programmer with information about the underlying error checking mechanism.
- Source-based - The error should speak in the terms of the programmer, not in terms of e.g. an intermediate core language.
- Unbiased - There should be no left-to-right (or similar) bias in the decision which sites contribute to the error.
- Comprehensive - All sites that contribute to a type conflict should be mentioned in the error.

Applying this manifesto on algorithm W shows that the algorithm is both correct and succinct, as it always gives the conflicting types when a program contains an error. It is, however, not precise, as algorithm W is known and proven [25] to only detect errors in function applications, which are sometimes not the precise locations of the error. Algorithm W's error messages are also mechanical in implicitly revealing the underlying unification algorithm. After all, an error describes how the inferencer attempted to unify a certain pair of types and failed to do so. Besides that, Algorithm W is biased from left to right and it is not comprehensive, since it stops at the first error it finds while traversing the program from top to bottom and left to right. Algorithm M and the combination of W and M make error messages more precise, but share all other properties with algorithm W.

1.4 Type Error Diagnosis

The type inferencing algorithms described earlier each work to improve error messages, but much more can be done. We will now describe type error diagnosing methods that extend existing inferencing algorithms. They describe various directions of research aimed at improving error messages.

Explanation Systems

To make type errors more comprehensive, Wand [39] devised a method to explain *why* the compiler comes to the conclusion that there is a type error. Based on algorithm W, it does this by having the unification algorithm keep track of the reasons for unifying type variables. Once a type conflict arises, both sides of the conflict will have a list of reasons as to why it was inferred. This helps the programmer in finding the cause of a type error. However, the method does trade comprehensiveness for succinctness, as the lists of reasons tend to be quite long. Too long even to be of any reasonable use to a programmer. Nevertheless, Wand has inspired other researchers to expand on the method.

Error Slicing

The focus of error slicing lies in reporting the right locations of an error. The essence consists of reporting all parts of a program that are related to a type conflict (called a "slice") as opposed to a single subtree or program point. An important property is that a slice correctly includes all of the parts of the program where the type error may be fixed while excluding the parts of the program for which no change can fix the type error.

Haack and Wells [12] implemented this concept for the Hindley-Milner type system, implemented using constraints. The generated constraints are mapped to program points. In case of a type error, a single minimal unsolvable constraint set is generated. The locations associated with the constraints in this set make up the slice. The method always returns a single slice, because calculating the full set of slices is expensive. Slices are pretty printed by printing the relevant parts of the slice while replacing everything between those relevant parts with dots. Sadly, due to inefficiencies in the constraint generation phase, constraint sizes tend to explode with program size.

A continuation of this work, (Rahli et al. [33]) consists of a type slicer tool for standard ML. It visualises the error slices by highlighting the parts in a text editor. It resolves the previous issue of exploding constraint sizes. Nevertheless, type slices tend to be rather big, since there tend to be many locations at which the type error can be fixed. This leaves a considerable amount of effort to find the solution to the programmer.

Pavlinovic et. al. [30] show that type inferencing can also be seen as an optimisation problem. The constraint based method assigns weights to generated constraints. These weights are decided by the compiler, which uses heuristics to determine which errors are more likely than others. In case of an error, the optimisation algorithm is run to find the maximal satisfiable subset of constraints. The complement of that set is then necessarily the minimum unsatisfiable set of constraints, which represents the error slice. The algorithm is quite expensive,

but its accuracy can be traded for speed. The method can also be used to create an interactive debugger, by letting the end user influence the weighing criteria.

Type Graphs

Some constraint-based type inferencing systems can be designed to allow equality constraints to be represented as a graph of types and their relations. This allows unbiased views on type conflicts through the ability to investigate the neighborhood (in a graph sense) of a type conflict. This is what Heeren [18] describes in the TOP framework, built for the Helium Haskell compiler. The constraint graph can be used to act as a constraint solver. When an error occurs, a number of heuristics can be applied to have the type graph point to a likely culprit. One major downside is that generating type graphs can be a quite expensive task.

Another use for type graphs is described by Zhang and Myers [42], who outline a method of analysing both satisfiable and unstisfiable constraints in a type graph using context-free grammars for a large subset of ML. Most interestingly, rather than a set of heuristics, it uses a Bayesian algorithm to rank the most likely explanation of an error. This Bayesian algorithm is trained with a corpus of common mistakes. Sadly, their graph structure is quite expensive to calculate. Besides, with a Bayesian ranking algorithm, the ranking of possible error causes might at times be suboptimal.

A continuation of this work is written using GHC's type inferencer and works as an extension to the SHErrLoc diagnostic tool [43]. It supports some of the more complicated structures of Haskell (like GADTs and type families) and is capable of counterfactual reasoning. For this, the constraint language is extended. That extension turns out to be too much for the context-free grammars to handle. To fix that, the graph is saturated and expanded. The constraints in this graph are checked for satisfiability. In case of unsatisfiable constraints, a Bayesian algorithm is invoked again to decide which part of the code is the culprit. The performance does seem to go down quickly as the lines of code increase (quadratic).

Counterfactual Typing

Counterfactual typing is a specific means of trying to find the right fix for type errors. Chen and Erwig [3] coined the term in a paper describing variational types for a simple lambda calculus. Variational types represent the type of an expression not as a single decision, but as a choice of multiple types. By passing and merging the variational types through the inferencer, a set of type choices is generated per subexpression. Possible fixes are then generated by enumerating combinations of choices for these options. It is defined for a simple lambda calculus with let-polymorphism. It also includes an extension to suggest adding/removing or swapping of function arguments. The method is reasonably fast, but the method seems to be unable to deal with errors caused by bad parenthesis placement.

An entirely different approach for counterfactual typing is outlined by [36], which uses the type checking algorithm as a black box. When an error occurs, it starts replacing subexpressions with holes until the error is resolved. Once a type correct program has been found, the types of the inserted holes are

requested from the type checker and given to the user as "expected type". The current implementation is for OCaml. Errors are always given in terms of expected and actual types. The method can collect multiple error messages, but will always show one at a time. The order in which errors are shown is not defined. Naturally, trying all possible locations to place holes is a very expensive operation, making this approach infeasible for real world usage.

Type Error Debugging

Besides generating better error messages, research has also focussed on helping programmers solve type errors interactively. By asking questions about the types of the program, an interactive debugger can use the programmer's view on the program to help them find the causes of type conflicts. Tsushima and Asai [35] devised a type debugger that extends OCaml's existing type inferencer. This has the benefit that the type debugger can easily be updated along with the type inferencer. Type debugging is done by asking the programmer whether the inferred types of certain expressions is correct. When the correct types for enough expressions are found, the source of the conflict is reported to the programmer, and a suggestion is made to fix it.

One downside of this approach is again the large size of subexpressions that tend to be involved in type conflicts. This forces the type debugger to ask many questions to the programmer, which can be tedious. This issue is less apparent in Chen and Erwig's Guided Type Debugging [4], which describes a type debugger based on counterfactual typing. Rather than asking whether the inferred type is correct, it asks what the programmer thinks the type of a subexpression should be. To make this question easier, only the types of simple subexpressions are asked. Counterfactual typing is used to find the minimal number of subexpressions that need their types verified to find the cause of the problem. This significantly reduces the number of questions that have to be asked.

Directives

Directives provide a means for library programmers to alter the type inferencing process. This allows the library programmer to hide complex internal types from the end user, which would otherwise be shown in type error messages. Another use is anticipating for pitfalls and building in hints that are shown when end users succumb to them. Heeren et al. [19] described several kinds of directives, and implemented them in the Helium compiler. These directives are all described in Heeren's PhD thesis [16].

One such kind is Siblings. Sibling functions are functions that are similar in functionality, but differ in types, e.g. the applicative functions <$ and <$>. Through siblings, a library author can indicate that these functions are likely to be mistaken for one another. When a function is involved in a type error, the compiler can try its siblings to see if they fit the expected type. When they do, a hint is generated that advises to use the alternative function.

A second kind, more general than siblings, is called "repair directives". Repair directives tell the compiler to attempt specific rewrites of faulty code. Repair directives can flip arguments, place parentheses or define siblings. Sadly, the more combinations of repair directives a compiler tries to apply, the slower

the type inferencer becomes. The compiler must thus be limited to only attempt a few combinations of repairs.

Through a different, but more powerful kind of directives, the library programmer can define specialised type rules for their library functions. By means of a domain specific language, the library writer overrides the inference rule for the function application of their specific function. The library author can define custom constraints and the order in which constraints are to be checked. Specialised type rules can have a large influence on the type inferencing process. This also means that library programmers could potentially make type error messages even less descriptive than what the compiler would generate on its own. Fortunately, specialised type rules are made sure not to render the type inferencer unsound. This is done by validating the specialised type rules against the type inferencer's judgement. Finally, the language that describes specialised type rules can be quite complex, and requires knowledge of typing rules to be written. It remains a challenge to find a syntax that is easy to use.

Heeren and Hage [17] also described a kind of directives for type classes. These directives can impose limits on instances of type classes: the *never* directive can state that a given type can never be an instance of a given type class (e.g. `Show (a -> b)`). The *close* directive can state that no new instances for a type class can be defined. The *disjoint* directive can force the instances of two type classes not to overlap. Lastly, a *default* directive allows the definition of a default instance when a type is ambiguous. A default instance of Show being String or Bool, for example, would make `show []` work correctly without having to annotate the type of the empty list.

A continuation of Heeren's work is described in Hage's plan [13] to implement type directives in UHC. The plan outlines how Haskell's many extensions provide a challenge for implementing type directives in a real world compiler.

Serrano Mena [34] is working on a generic specialised type rule language, currently implemented for GHC's `OutsideIn(X)` framework. The generic type rule language captures its most powerful abilities. Through a two-stage type checker, the language allows the library programmer to define advanced conditions on when a specialised type rule should apply. Since this complicates the language even further, more work is being put in defining a syntax that is clear and intuitive to use.

In a completely different approach, Christian [5] defined post-processing directives for the dependently typed Idris. Reflections, as they are called, can be used by library writers to rewrite errors before they are shown to the end user. Error rewrites are represented as a list of a recursive data type that contains text, the name of an expression, a term or a suberror. These can be programmed by writing a specially annotated function that takes an error and produces a maybe containing an error rewrite. This allows DSL writers to hide the implementations of their proofs. In the dependently typed setting, the combination of building proofs and customising the error messages can greatly aid the end user in understanding the cause of the problem. However, it is limited to rewriting the parts in the original error messages. The type checker cannot be prodded to give more information about the error. This forms a bigger restriction in non-dependently typed languages, in which proofs cannot be written as extensively as in dependently typed languages.

Finally, GHC version 8 will introduce a completely different kind of type directives [2]. Specifically designed for type level programming, a special Type-

Error type family will throw a custom error when it is reduced by the type inferencer. This approach is very specific for type family programming and type classes.

1.5 Conclusion

This literature review started by describing several methods of Functional Reactive Programming (FRP). FRP allow the programmer to create a program that reacts continuously with its environment. This can be done in a paradigm that is declarative and purely functional. A difference is shown between methods that treat time as continuous (Fran, FRPNow) and those that treat time as discrete (Elm).

A comparison is made between the type inferencing algorithms W, M and U_{AE}. W is bottom-up, while M is top-down. M shows generally shows better error messages than W, but both are biased. U_{AE} seeks to remove that bias. The same can be achieved by using a constraint based type inferencer, which separates the concerns of generating and solving typing constraints.

Reasearch on creating understandable type error messages go in many directions, from improving the way they are explained, to letting library programmers customise them. The methods each tackle different sets of problems that type errors can have, as described in the manifesto: incorrectness, unsuccinct, imprecise, mechanical, not source-based, biased or incomprehensive.

Chapter 2

Research question

Previous research [19] [17] has shown that directives can greatly improve the errors generated by the compiler. This research, however, applies only to the Helium Haskell compiler. Further research [34] is on-going about implementing directives in GHC's `OutsideIn(X)` framework. Again, this applies only to Haskell. The idea is to implement similar concepts in Elm. Since there are discussions [8] about extending Elm with type classes, it will be included in the research. The main research question is then whether the concept of directives is applicable to Elm extended with type classes.

Type classes raise the question about how they can be implemented in such way that the compiler can give nice error messages. The of type classes itself might be considered difficult to understand. Clear error messages might relieve the users of some of the complicated aspects. Perhaps the concept can be made more accessible to the end user with some heuristics built into the type inferencer.

Elm is chosen to be the target language for this research for several reasons. It is a purely functional language with a syntax similar to Haskell's. Unlike Haskell, it does not have Rank-N types, higher kinded polymorphism, GADTs or type families. It does have built-in primitives for Functional Reactive Programming (FRP), although those have been removed in version 0.17. While these constructs do not necessarily pose a challenge for implementing directives, they might provide an opportunity for finding different kinds of directives that have not been described before. Elm also supports records, similar to those described by Leijen [26]. These records have (limited) support for row polymorphism, which allows records to extend other records, inheriting their fields. Providing support for this in type graphs might be a challenge.

Related to that, is verifying the soundness of directives. The specialised type rules described by Heeren et al. [19] could render the type inferencer unsound if left unchecked. Heeren et al. state that specialised type rules can be verified against the existing type system to ensure that they indeed do not cause the inferencer to become unsound. This verification method needs to be ported to Elm. Perhaps even, in the spirit of Elm's clear error messages, this verification method should try to provide clear error messages to allow debugging. After all, the people writing those type rules are programmers too, and they benefit from nice error messages.

Finally, with a focus on beginning programmers, Elm's syntax is designed to

be as simple as possible. This philosophy will have an influence on designing the way library writers write directives. The motivation needs to exist to write them. This motivation might fade if they turn out to be too complicated and/or hard to write. A challenge exists in finding a balance between powerful directives and an elegant, easy to understand syntax. On one hand, a larger influence on type errors implies a larger capability of explaining the type errors of a library. This may, on the other hand, make the syntax of the directives too complicated. Besides, too much power over type error messages might create pitfalls that allow library programmers to write error messages that are even worse than what the compiler itself would come up with.

Chapter 3

Relevance

Elm has a strong focus on understandable error messages. This is because it is designed for beginning programmers who typically do not have a thorough understanding of types and the specifics of the underlying type inferencer. While Elm manages to word errors nicely, most type errors reveal how the inferencer came across two types that it could not unify. Not only do these errors reveal how the underlying type inferencer does its work, the beginning programmers must reason about the types of subexpressions to know where the problem lies and what the cause is. While quite a bit of effort has gone into providing specific hints on error messages, the error messages may remain daunting for beginning programmers.

Knowing that directives have been proven to be very effective in the past [19], their improved error messages might prevent a beginning programmer from being overwhelmed by complicated error messages. Since the goal of directives is to explain type errors in terms of the domain of the eDSL, it might allow the programmer to gain a deeper understanding on how to work with the eDSL.

Type classes are particularly known for making error messages more complicated, although their presence makes a language more expressive. This research is also aimed at demonstrating that the inclusion of type classes in the language does not necessarily prove detrimental to type error messages. Perhaps one can have both type classes and nice error messages.

The benefit is not limited to beginning programmers. eDSLs can have complicated type structures, which means that debugging type errors can be a difficult task, even for a seasoned programmer. This can be especially annoying when type errors reveal a complex type infrastructure that the eDSL's interface would have otherwise hidden. Type directives allow the eDSL to continue hiding the complex types in the error. Since the writers of the eDSL know how its functions are to be used, errors in uses of the functions can be explained in terms of the supposed usage. This can save time even for the experienced programmer by allowing them to quickly identify the cause of the problem. With proper error messages, the underlying complexities of an eDSL need not be understood by the programmer.

Scientifically, the idea of directives has been discussed before. This thesis focuses on its applicability in a different functional programming language with different language features. Besides that, the intricacies of Elm might give rise to new perspectives on the concept. Directives could perhaps be demonstrated to

be generic and extendable to languages that contain more difficult programming concepts (type classes). This in turn could give directives a stronger appeal, providing an incentive for other functional language authors to look deeper into the concept of directives.

Chapter 4

Elm's architecture

This section gives a high level overview of the Elm compiler. This can be useful to get an idea of the context of this thesis. The architecture described in this section is not complete, and some parts are more detailed than others. This is because it serves only to highlight the aspects relevant to the subject of this thesis. This section starts by explaining the different components of the Elm compiler. After that, the architecture of the `elm-compiler` component is described in somewhat more detail, since all the work in this thesis is done in this package.

4.1 Elm components

Elm's toolset consists of several packages, all of which are written in Haskell. The source is hosted on GitHub in several repositories hosted by an organisation called "elm-lang". The most noteworthy packages are listed below:

- `elm-compiler`
- `elm-make`
- `elm-repl`
- `elm-reactor`
- `elm-package`
- `elm-platform`
- `core`
- `error-message-catalog`

The `elm-compiler` package holds the compiler library. This library contains all compiler related code from parsing up to generating the Javascript. The `elm-make` package contains an equally named executable, which is what end users use to compile a project. It manages the building process and uses `elm-compiler` for the actual compiling work of individual modules. The `elm-repl` package contains a program that can evaluate Elm expressions interactively,

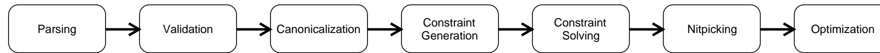


Figure 4.1: Compiling from source to optimized module

much like GHCi does for Haskell. `elm-reactor` hosts a simple webserver program that is capable of compiling Elm files and debugging them live in the web browser. `elm-package` contains Elm's package manager, with which one can install, publish and manage libraries written in and for Elm. The `elm-platform` package bundles the previously mentioned packages to create a unified set of tools for working with Elm. Not contained in the `elm-platform` is the `core` package, which contains the core libraries, similar to Haskell's Prelude. Finally, the `error-message-catalog` is a collection of faulty Elm programs designed to trigger exotic error messages in Elm. It is used for testing corner cases in the compiler.

4.2 Elm compiler

The `elm-compiler` deals with the compilation of a single module. The package acts as a library for `elm-make`, which leverages its functionality to compile projects. At the highest level, the `elm-compiler` provides two features. The first one consists of compiling a module from source code to a type checked and optimized module. The second one takes an optimized module and generates Javascript code. The focus of this architecture description will lie on the former feature.

Figure 4.1 shows the steps involved in compiling a module from source code to an optimized and type checked Abstract Syntax Tree (AST). Each step builds upon result of the previous step. The first step, parsing, builds a primitive AST that directly represents the source code. The validation phase checks for obvious errors, such as duplicate function definitions and unused type variables in Abstract Data Types (ADTs). The canonicalization phase resolves variable names and sorts the function definitions, creating binding groups of interdependent functions.

The constraint generation phase works in a top-down fashion. The constraints are very similar to those of Pottier et al. [32] in dealing with equality, instance and generalization. The constraint solver takes these constraints and finds the most general types. When the program is found to be type correct, last minute checks are performed in the nitpicking phase. These checks include making sure that there is a `main` function (when compiling the root module) and that this function has a valid type. The nitpicking phase also makes sure to throw an error when case expressions do not cover all possible cases. Elm is in this sense more strict than Haskell. Finally, the optimization phase performs some optimizations by transforming the AST. Optimizations include detecting tail calls and flattening let expressions.

During the parsing, validation, canonicalization, constraint solving and nitpicking phase, errors and warnings can be thrown. The compiling process halts immediately or at the end of the current phase when one or more errors are thrown. Warnings do not halt the compilation. Errors and warnings are pretty

printed and shown to the user in inverse order of being thrown, i.e. the first thrown error appears at the bottom of the output.

Chapter 5

Contributions: an overview

This chapter gives a high level overview of the contributions made by this thesis. Rather than providing an in-depth explanation of the inner workings of these techniques, it will focus on giving a small introduction to and a motivation for these techniques. It will also serve as an introduction for later chapters which will go further in depth. The concepts are described in no particular order. Where possible, the concepts will be described in terms of how an Elm programmer would experience them.

5.1 Type graphs

Type graphs allow for a more careful consideration of which error to throw when multiple error messages are available to describe the problem. As stated before, Elm's type inferencer works by first generating a set of constraints before solving them. These constraints together form a tree, which loosely follows the shape of the AST. Elm's constraint solver solves this tree of constraints by walking through it depth-first while maintaining some state that keeps track of types that have already been unified. If all goes well, this algorithm finishes with a data structure that gives every expression a type.

This way of solving constraints does, however, have one major downside: the algorithm immediately cries foul when asked to unify two types that cannot be unified (i.e. when a type error exists in the program). The very first constraint that demands two incompatible types to be equal will get the blame, and the error message tied to that constraint (attached during the constraint generation phase) will be the one that is thrown. Blaming the first bad constraint that the algorithm has come across introduces a "first come first serve" bias, which begs the question: what if it is more appropriate to blame a different constraint?

```
foo : List Float
foo = [1.0, "2.0", 3.0]
```

Figure 5.1: An example of a type incorrect statement

Observe figure 5.1. It contains a simple type conflict: all the values in the list have type `Float`, except for the third, which is a `String`. A simplified list

of constraints relevant to this type conflict is as follows:

1. The first item is a `Float` literal
2. The second item is a `String` literal
3. The third item is a `Float` literal
4. The first item has the same type as the second item
5. The second item has the same type as the third item

These constraints are listed in the order in which Elm solves them. The constraint solver will solve the first three constraints without complaining. At the fourth constraint, however, the solver is suddenly expected to unify a `Float` with a `String`. This is the first constraint where such conflict is found. As such, an error is thrown saying that the first and second items of the list do not match. While this error is not incorrect, one could argue that the list contains two `Float` values, and only one `String`. Not only that, the type annotation clearly states that the expression is a `List Float`. With all the evidence stacked against the literal being misplaced, surely it would be more appropriate to blame the second constraint?

This more careful consideration of the evidence is precisely what a type graph is designed for. A type graph, when drawn, shows how the types of an expression are connected by constraints. This gives a holistic view on type conflicts. Algorithms on the type graph can identify the constraints involved in a type conflict and, through heuristics, decide which constraint will get the blame.

5.2 Siblings

Some pairs of functions are conceptually similar, but differ in type. Some of these are famous for being mixed up. Think, for instance, about `foldl` and `foldr`, both folding in a different direction. These functions are famous for often being mixed up, especially by beginning programmers. It would be nice if the compiler would give a hint when the two are mixed up: "Did you mean foldl instead of foldr?". The compiler would know which pairs of functions are similar because it is explicitly told so by some programmer. Typically this programmer would be the author of either or both of the functions. Sibling directives are the means by which this is done.

The use of sibling directives is not restricted to functions in the core set of libraries, although many examples can be found there:

1. `(+)` versus `(++)`, one signifying addition, the other concatenation of lists.
2. `(|>)` versus `(<|)`, the former being function application, the latter inverse function application.
3. `(<<)` versus `(>>)`, both function composition, but with the results flowing either to the right or the left.
4. `curry` versus `uncurry`, for currying and uncurrying functions respectively.

```
sibling ++ resembles +

foo = 1.0 ++ 1.0
```

(a) Code

```
=================================== ERRORS ===================================

-- TYPE MISMATCH ------------------------------------------------------ Main.elm

`Basics.++` is being used in an unexpected way.

28| foo = 1.0 ++ 1.0
          ^^^^^^^^^^
Based on its definition, `Basics.++` has this type:

    appendable -> appendable' -> appendable''

But you are trying to use it as:

    Float -> Float -> a

Hint: Did you mean Basics.+ instead of Basics.++?

Detected errors in 1 module.
```

(b) Resulting error

Figure 5.2: A sibling statement along with a faulty expression and its error message.

For library developers siblings can be a simple, yet powerful tool to make their libraries easier to use. The provisional syntax for defining siblings is simple. Figure 5.2 shows an example sibling and a statement where the wrong function is used. The error on the right is unaffected by the sibling in everything but the hint at the bottom. The small, yet powerful hint suggests that (+) could be used instead. This hint would have been left out if the sibling statement did not exist.

5.3 Interfaces

Interfaces (i.e. type classes) provide a simple means for ad-hoc polymorphism. The main reason to implement them in Elm is to investigate how well they work together with the other techniques described in this thesis. Nevertheless, some care was taken to make them simple and easy to use. As a result, interfaces are very similar to type classes in Haskell 98, though with a different syntax.

```
interface Eq a where
    equals : a -> a -> Bool
    notequals : a -> a -> Bool

implement Eq for Bool where
    equals l r =
        case (l, r) of
            (True, True) -> True
            (False, False) -> True
            _ -> False

    notequals l r = not (equals l r)

identity : a -> Bool | Eq a
identity a = equals a a
```

Figure 5.3: An example interface, implementation and function.

Figure 5.3 shows an example interface and implementation (i.e. instance).

The syntax is somewhat borrowed from Idris. The choice for this syntax is made for several reasons. The first is that the word "class" is used in object oriented (OO) programming languages for a significantly different concept. The word "interface" is also used in OO languages, but interfaces in OO languages have much more in common with type classes than classes do. This will make it easier for programmers in shifting from an OO to a functional paradigm. The same applies to the word "instance", although the word "implementation" has no real OO counterpart.

Secondly, this wording allows the code to be read more naturally: `implement Eq for Bool where` forms an almost proper English sentence while remaining reasonably concise. This also applies to the syntax for qualifiers in type annotations, implementation headers and interface headers. Take, for example the type of `identity` in Figure 5.3. The type annotation can be read out loud as "a to Bool, with Eq a". This may make it easier to talk about programs in a person to person conversation.

5.4 Specialized type rules

```
checkMaybe : Maybe a -> a -> Bool | Eq a
errors for checkMaybe maybe val where
  constrain maybe
  constrain val

  unify maybe with Maybe a_1
    because The first argument has to be a Maybe.

  unify val with a_2

  unify a_2 with a_1
    because The second argument must match the thing in the Maybe.

  check Eq a_1
    because Eq is needed to test equality.

  unify return with Bool

  constrain return

checkMaybe maybe val =
    case maybe of
        Nothing -> False
        Just x -> isis x val
```

Figure 5.4: A function definition with type rules.

Type error messages are indifferent to the conceptual meaning of functions they reason about. A type error can tell a user that a function expected an argument of one type, and got an argument with an incompatible type. It

cannot, however, tell the user *why* the function needs the argument to have a certain type, nor can it identify, let alone explain common mistakes in the usage of functions. This information is not available to the compiler, since it cannot reason about the meaning of code. It does, however, exist in the mind of the author of a function.

The author of a (library) function has made the conscious decision to give the function a certain type. This same author, whether by speculative insight or through feedback, can foresee common mistakes in uses of the function. It would be beneficial for end users of the function if this information was somehow made available to them. What if a type error message included a note from the author of the function explaining the error in terms of the meaning of the function? When done right, such explanations could give the end user valuable insight into their mistakes. When done wrong, the note under the error might make no sense, but one would still have the type error message. Specialized type rules are the mechanism through which a function author can supply this information to the compiler.

Through specialized type rules, the author of a function can add explanations to the type constraints that are created when the function is used. This way, whenever a type constraint is blamed for a type error, the explanation of the author will be added as a hint. On top of that, the author can change the order in which constraints are checked. This gives the function author some control over which constraint is more likely to be blamed for an error.

Figure 5.4 shows a function with type rules. The function `checkMaybe` returns whether the value held by the `Maybe` equals the second argument. Note that the type rules exist between the function's type annotation and its definition. This is the only place where such type rules are allowed, as it makes sure that only the author of a function can define type rules.

Line 2 shows the header of a set of type rules. It holds the function name and it names all the parameters. These parameter names, along with a special parameter name "return", can be used in the type rules. Under the header, the constraint rules can be read from top to bottom. There are three different type rules: constrain, unify and check. The constrain rule, seen on lines 3, 4 and 19, tells the compiler to generate the constraints for an argument (or the return value). When omitted, they will be inserted automatically.

Unify type rules, seen on lines 6, 9, 11 and 17, generate equality constraints. The unify constraint on line 6, for example, states that the first argument (referred to as `maybe`) must be unified with `Maybe a_1`. The reason for this type rule is given on the next line. This reason is optional. When provided, the error message linked to the constraint will show the reason as a hint.

At this point it is useful to note that the type variables `a_1` and `a_2` are numbered. They refer to the type variables in the type annotation. The variable `a_1` refers to the first occurrence of `a` in the type annotation, `a_2` refers to the second occurrence of `a`, etc. Using these numbered variables is obligatory. Not only does it make the type rules more readable, the compiler uses the information to generate the error messages. More detail on that will be given in later sections.

Finally, `check` rules marry the concepts of type rules and interfaces. Check type rules represent the demand that there exists some implementation of some interface. In this case, it demands `a_1` to carry the Eq qualifier. This restricts the types that `a_1` can be unified with. Like with `unify` rules, a reason can

optionally be given to explain why an implementation is required.

The type rules in figure 5.4 only apply to calls to `checkMaybe` where all arguments are provided. Since functions in Elm are curried, it is possible to call the function with fewer than two arguments. Sadly, the hints in the type rules would make no sense in such cases. Curried uses of the function might even have different pitfalls than fully saturated calls to the function. If the author of the function feels this is the case, separate sets of errors can be defined for curried versions of the function. In the above example, one can imagine an extra set of type rules that starts with `errors for checkMaybe maybe`, with `maybe` referring to the first argument and `return` referring to the rest of the function (i.e. `a -> Bool`).

Figure 5.5 shows an example of an error in the use of the `checkMaybe` function defined above. The error in figure 5.5b shows the error that would have been thrown if the type rules were not given. Figure 5.5c shows the error from the type rules. The error itself is slightly different as a result of the implementation (this will be explained in chapter 9), but most importantly: the hint at the bottom tells us the cause of the error in human terms. It is the hint provided by the author of the function, seen on line 12 of figure 5.4.

In the end, specialized type rules are not meant to be used by beginning programmers. They require some knowledge and insight about constraints and how a type inferencer generally works. For the writers of libraries, specialized type rules can be a very powerful weapon to make their libraries easier to use. A simply worded hint under a complicated type error message could perhaps make the difference between spending a lot of time decrypting an error message and (almost) immediately understanding the mistake.

```
foo =
  checkMaybe
    (Just True)
    "bar"
```

(a) An error in the use of `checkMaybe`

```
The 2nd argument to function `checkMaybe` is causing a mismatch.

275|    checkMaybe (Just True) "bar"
                               ^^^^^
Function `checkMaybe` is expecting the 2nd argument to be:

    Bool

But it is:

    String

Hint:
I always figure out the type of arguments from left to right. If an argument is acc
eptable when I check it, I assume it is "correct" in subsequent checks. So the prob
lem may actually be in how previous arguments interact with the 2nd.
```

(b) The error without type rules.

```
The 1st and 2nd arguments of function `checkMaybe` conflict with one another.

273|    checkMaybe (Just True) "bar"
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Function `checkMaybe` is expecting the 2nd argument to be:

    Bool

But it is:

    String

Hint:
The author of function `checkMaybe` gives the following explanation:
  The second argument must match the thing in the Maybe
```

(c) The error with type rules

Figure 5.5: The type errors in the use of a function with and without specialized type rules.

Chapter 6

Type graphs

Type graphs have been described in great detail in chapter 7 of Heeren's PhD thesis [16]. It is recommended to read sections 7.1, 7.2 and optionally 7.3 before continuing this chapter. The principles of the type graphs implemented in Elm are exactly the same as the ones described by Heeren. After a short summary, we will focus on the enhancements made to the concept of type graphs that are not described by Heeren.

As a short summary of type graphs, let us take a look at figure 6.1. It is a drawing of the type graph that represents the expression `not "foo"`. The black lines indicate a type hierarchy, the numbered purple lines with arrows are constraints and the red dashed line shows the type conflict.

Let us first take a look at the cyan encircled tree. This represents the type of the `not` function, which is `Bool -> Bool`. One can see how this type forms a tree when writing the type in infix notation (i.e. `(->) Bool Bool`), then noticing the associativity: `(((->) Bool) Bool)` before making that explicit in a binary tree. The `app` nodes are explicit applications of types to parameters.

From the type of `not`, one constraint goes out to the rest of the graph. Constraints connect what would otherwise be islands of types. Through the constraints and the hierarchy of types, one can reason about which types should be equal to which. The list below describes what each constraint conceptually means. This meaning is stored in the error message tied to the constraints.

1. Node 5 must be an instance of the type represented by node 0.
2. Node 8 must represent a function with the same arity as node 5.
3. The return type of the function.
4. The first argument of the function.
5. A String literal was filled in here.

Leaving out the purple constraint arrows, one can find a tree with the same shape as the one representing `not` starting at vertex 8. The biggest difference is that this tree appears to describe `a -> a`, since vertices 10 and 12 are type variables. This tree represents the shape of the expression, which is a function call with one argument. This shape is linked through constraints #1 and #2. Since the type of variables and the "instance" constraints that go along with

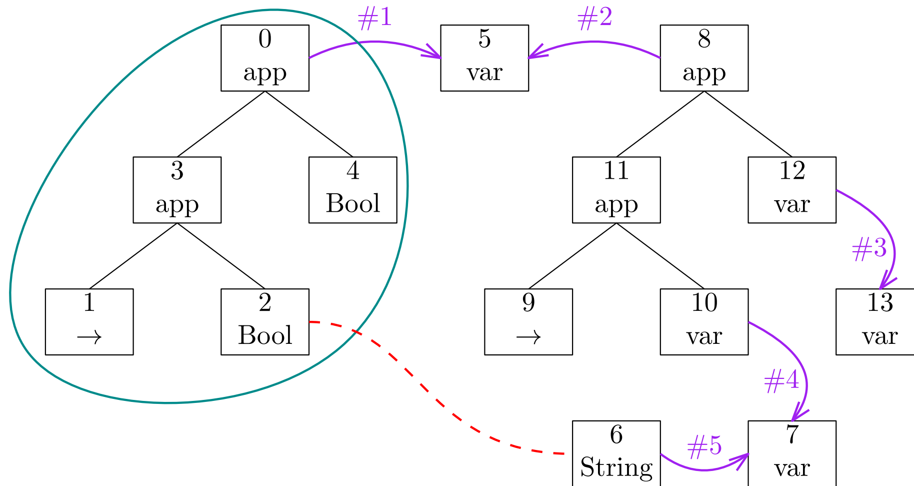


Figure 6.1: The type graph of `not "foo"`

them are inserted during the solving process, node 5 acts as a placeholder until the type is inserted. This is a minor detail, but it explains why there is a path of two constraints between nodes 0 and 8.

Node 7 is the placeholder for the type of the first argument. Constraint #4 links this placeholder to the first argument of the function. Constraint #5 links it to the type of the `String` literal that was filled in there. Finally, constraint #3 links the placeholder for the return type of the function to node 12, which we already know represents the return type of the function. If this expression were to live inside another expression (e.g. an `if`-statement or as an argument of another function), node 13 would be linked to some type in from that context.

Now that we know how the type graph represents the type of the expression, we can reason about what has gone wrong. Constraints #1 and #2 demand that node 0 must represent the same type as node 8. This means that the left children (i.e. nodes 3 and 11) of both nodes must hold equal types. The same applies to the right children of those nodes (i.e. 2 and 10). Constraint number #4 states that node 7 must hold the same type as node 10, which means it must also hold the same type as node 2. Constraint #5 does this for node 6. This means that nodes 2, 10, 7 and 6 must hold the same type. The conflict here is that node 2 holds a `Bool` while node 6 holds a `String`.

Now that we know which types are in conflict, we can start thinking about which constraint is best to blame. A careful reader might note that all constraints but constraint #3 are mentioned in the previous paragraph. One can argue that cutting any of those four constraints would break the link between nodes 2 and 6. This means that all four constraints say something about the type error. Despite this, it does not make equal sense for some of those constraints to get the blame. After all, if constraint #1 were to get the blame, then the error would be that somehow `not` does not have the type `Bool -> Bool`. Constraint #4, on the other hand, appears to be the right one to blame. After all, it says that the first argument of the function has the wrong type.

How this decision is made depends on heuristics. Currently, there are currently three heuristics implemented. These heuristics are called “Share in error

paths", "Constraint number" and "Trust factor". This thesis will not describe these heuristics, instead they are described by Hage and Heeren [14] and sections 8.1, 8.2 and 8.3 of Heeren's PhD thesis [16].

6.1 Mixing two constraint solvers

Like the type graphs described by Heeren, the type graphs in Elm are more expensive than the built-in constraint solver. Luckily, type graphs need only be constructed when there is a type error that needs to be investigated. A module without type errors will be compiled without the type graph ever being invoked. When the built-in constraint solver comes across a type error, a type graph is created to investigate the type conflict. The investigation will focus on the context of the constraint that caused the type error. To explain this, let us first delve deeper into Elm's constraint generation and the built-in solver.

As mentioned in chapter 4, the canonicalization phase sorts the definitions and let-expressions to create an ordering of binding groups. Definitions in these binding groups depend either on other definitions in the same binding group or definitions in higher binding groups. This vertically sorted tree of interdependent function definitions is the basis for constraint generation. As a result, the generated tree of constraints tightly follows the structure of the tree of binding groups. This makes solving the constraints easier, since the solver can be sure that the types inside a binding group will not change when recursing into a deeper binding group. Conversely, the constraints inside a binding group do affect the types inside of it. Naturally, when a type conflict arises from any of the constraints, the binding group containing this constraint is the subject of investigation.

Unfortunately for type graphs, Elm's built-in constraint solver works with a state in IO which is maintained through destructive updates. As such, the types in a binding group are partially resolved when the solver stops to report an error. Because of the destructive updates, one cannot go back to the unsolved and untouched binding group for reexamination. The solution is to manually create a copy of the solving state, which is updated when recursing into a binding group. This copy is examined when a type conflict occurs.

After the type graph has examined a type conflict and thrown the error(s) it has deemed most appropriate to throw, the built-in solver can continue with the remaining binding groups like it would as if the type graph never existed. At the end of the constraint solving process, the errors are pretty printed and shown to the user.

6.2 Error path expansion

When there is a conflict between two types, a path of at least one constraint must exist that states that these two types must be equal. Since the internal representation of the type graph does not explicitly store all edges (see section 7.2.1 of Heeren's thesis), some searching needs to be done to unravel the path that connects the two conflicting vertices in the type graph. When found, this path contains valuable information: the equality constraints that together cause the type conflict.

Internally, all nodes that should represent the same type are grouped in so called "equivalence groups". In Figure 6.1, for example, nodes 2, 10, 7 and 6 are stored in the same equivalence group. This equivalence group also stores the constraints #4 and #5 and has a reference to the parents of nodes 2 and 10.

Finding type conflicts in this representation is relatively easy: find two nodes in the same equivalence group that cannot be unified. As such, the conflict between nodes 2 and 6 is quickly found. Earlier, in the introduction of this chapter, we identified constraints #1, #2, #4 and #5 as contributing to this conflict. Now we shall take a look at how the search algorithm would identify those same constraints.

The starting point of the algorithm is the pair of conflicting vertices. In our example, these are nodes 2 and 6. The goal is to find a path from one node to the other. The means to find this path is a breadth-first search (BFS). Since the edges in the graph are symmetric, it does not matter from which of the nodes the search is started.

```
IterationInfo = II
  { current :: Vertex
  , goal :: Vertex
  , path :: Path
  , treeWalkStack :: [ChildSide]
  }

expand :: IterationInfo -> [IterationInfo]
expand ii =
  walkOverConstraintEdge ii ++
  walkUpParent ii ++
  walkDownChild ii

bfs :: [IterationInfo] -> [Path]
bfs successes =
  let
    nextIteration :: [IterationInfo]
    nextIteration = concatMap expand successes

    finished :: IterationInfo -> IterationInfo
    finished inf =
        current inf == goal inf &&
        null (treeWalkStack inf)

    anyFinished :: [IterationInfo]
    anyFinished = filter finished nextIteration
  in
    case anyFinished of
        [] -> bfs nextIteration
        xs -> map path xs
```

Figure 6.2: High level BFS algorithm

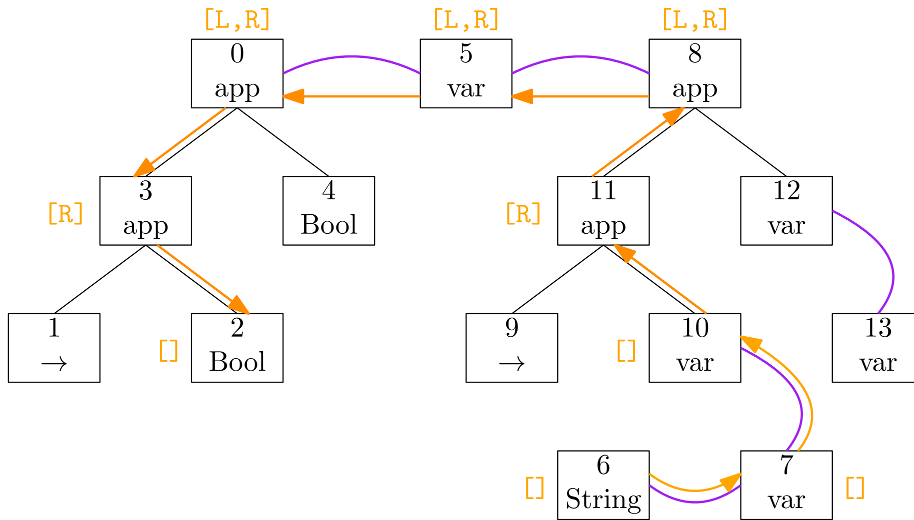


Figure 6.3: BFS in action

The BFS algorithm, shown in figure 6.2, works with a list of successes: throughout the algorithm, a list is maintained containing information on the paths currently being searched. This information includes the current vertex, the goal vertex, the path constructed thus far and a stack of child sides that will be explained later. In every iteration, this list of successes is expanded: for every path currently being searched, all next possible steps are generated. This is done in the `expand` function. If any of the successes has reached the destination, then the algorithm stops and returns the paths that have been found.

The algorithm is shown in action in figure 6.3. The orange arrows represent the actions of the BFS algorithm. There are three ways to expand a path. The first and simplest way is walking over a constraint edge. After all, the constraint states that the two nodes must represent equal types. This can be seen in the first two iterations, going from node 6 to 7 and from node 7 to 10. The second and third ways are walking up and down the type hierarchy. The reasoning for that is as follows: when two type applications are equal, both left children and both right children must be equal too. In figure 6.3, nodes 0 and 8 must be equal. This equality trickles down to the children: nodes 3 and 11 must be equal, nodes 4 and 12, nodes 1 and 9, etc.

When going up the tree, one must remember to at one point go back down the tree. Specifically, one must get back down in the reverse order of getting up. In the example, one go up the right edge of the tree from node 10 to node 11. From node 11, one go up the left edge to get to node 8. This means the reasoning is now about equalities between types that contain the originally conflicting type in its left and then right child. At one point one must go down a left and right child to get back to the originally conflicting type. In the example this happens from nodes 0 to 3 and nodes 3 to 2.

The orange lists containing `R` or `L` drawn next to the nodes display the `treeWalkStack`. This `treeWalkStack` is for keeping track of the edges that have been walked up. When walking up an edge, the side of the edge (`R` or `L`) is

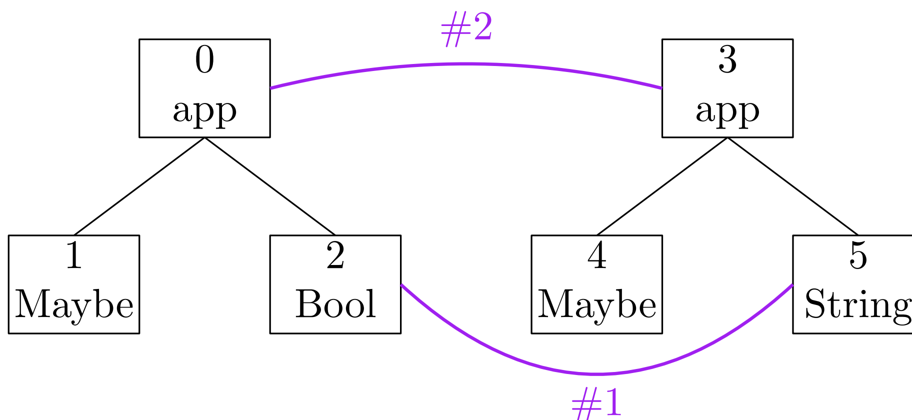


Figure 6.4: BFS limitation. Only constraint #1 would be identified.

pushed upon this stack. Walking down an edge is only possible when the stack is not empty. One can only walk down the edge direction stated by the top of the stack. Walking down an edge pops a value off the stack. The stack must be empty when the goal node has been reached.

6.2.1 Limitations

A very careful reader might notice that the BFS algorithm described in the previous section can only find shortest paths. In theory, the situation drawn in figure 6.4 could occur, where two paths describe the same conflict. In this example, only constraint #1 would be identified. One could perhaps change the algorithm to continue searching after a path has been found, making sure not to not get stuck in a cycle. This might cause trouble, though, when infinite types are involved. Currently, the path expansion algorithm always terminates when a finite path exists. Infinite types can present themselves as cycles in the type hierarchy. With such cycles, the BFS algorithm can walk up this type hierarchy indefinitely without ever going back down.

6.3 Reconstructing types

Building a type from a coherent type graph is relatively simple. Section 7.3.5 of Heeren's PhD thesis [16] describes how to find a substitution for a given type variable. The `substituteVariable` function described in that section takes a node's number and returns a `Maybe` type. This function fails by returning a `Nothing` when the type is involved in a type conflict or when the type is infinite. This is a shame, since those are precisely the types that are interesting to reconstruct to the best of the type graph's ability.

Reconstructing both conflicted and infinite types is useful for error messages. When a constraint has been decided to get the blame of a type conflict, the types on both ends of the constraint need to be displayed in the error shown to the end user. The node at one end of the constraint will hold the "expected" type, whereas the other will hold the "actual" type. Infinite types have their own error

messages, which show as much detail of the infinite type as possible, without printing incessantly.

Elm's built-in constraint solver already has a mechanism which attempts to reconstruct infinite types. The destructively updating nature of the built-in solver strongly hinders its ability to perform a proper reconstruction. Type graphs do not suffer from this hindrance, since they can represent infinite types in a finite amount of vertices and edges. The base technique for this reconstruction has however been the basis for reconstructing infinite types using type graphs.

In Elm's built-in solver, an infinite type is reconstructed by recursing over the type's structure, which is stored in the solving state. Whenever this recursive algorithm comes across a structure it has visited before, it concludes that has found an infinite type and replaces the structure with a type variable with the name "∞". Types involved in a conflict are replaced with a type variable named "?". The type displayed to the user will display the resulting type. Sadly, the errors are often just question marks, since infinite types are often also involved in type conflicts as well.

The type graph is much less restricted. After all, no information has been lost by destructive unifications of types. The algorithm for reconstructing infinite and conflicting types is the same. It is based on Heeren's `substituteVariable` function and includes Elm's technique of replacing seen type variables with "∞"

Figure 6.5 shows the algorithm that reconstructs a type in pseudo code. It keeps track of the visited vertices in a set. When a vertex is revisited, ∞ is filled in (line 3). Otherwise, the contents of the vertex is inspected. If the vertex holds a type application, the algorithm recurses on the left and right children and returns their combination. Type constructors are returned without modification.

Type variables require some more effort. The equivalence group containing the type variable vertex is inspected to see if there are any more specific types (line 27). This most specific type is returned if the type variable is part of a coherent equivalence group. That is, the variable is not involved in a type conflict. A uniquely named type variable is returned if it turns out that the most specific type of the group is again a type variable. The algorithm recurses when a type constructor or type application is found to be the most specific type of the group (line 32).

Type variables part of a type conflict also generates a uniquely named type var (line 35). One might wonder whether this does not defeat the purpose of reconstructing types involved in type conflicts. The reason why it does not lies with the type applications and constructors. In lines 14 to 23, type applications and constructors are treated as though they live in a coherent equivalence group. In reality, they could very well be. Reconstructing node 2 in figure 6.1, for example (also shown in 6.3), would give a `Bool`, even though this node is in conflict with the `String` in node 6.

In practice, reconstructing conflicted types works as well as Elm's built-in solver. Reconstructing infinite types works quite a bit better.

Table 6.1 shows the difference between the results of the reconstructions of infinite types. The first column contains expressions that have infinite types. The second column shows how Elm's built-in constraint solver reconstructs the infinite types. The last column shows the types reconstructed by the type graph. Sadly, built-in type reconstruction failed for all but the last expression.

```
reconstructType :: VertexId -> TypeGraph -> Set VertexId -> Type
reconstructType node graph seen
  | node `member` seen = TypeVar ∞
  | otherwise =
    let
      group = equivalenceGroupOf node
      seen' = insert node seen

      -- Some unique name
      uniqueName :: String
    in
      case vertexContents node group of
        -- Recurse on children
        App left right ->
          let
            ltp = reconstructType left graph seen'
            rtp = reconstructType right graph seen'
          in
            TypeApp ltp rtp

        -- Return constructor name
        Constructor name ->
          TypeConstructor name

        -- Try to find substitution
        Var ->
          case typeOfGroup group of
            Right (_, Var) ->
              TypeVar uniqueName

            Right (node', _) ->
              reconstructType node' graph seen'

            Left typeConflict ->
              TypeVar uniqueName
```

Figure 6.5: Reconstructing types

Code	Built-in	Type graph
let inf = inf inf in inf	?	∞ -> ∞
let inf = [inf] in inf	?	List ∞
let inf = ("str", inf) in inf	?	(String, ∞)
foo a = if True then a a else 0 >= a	∞ -> a	∞ -> a

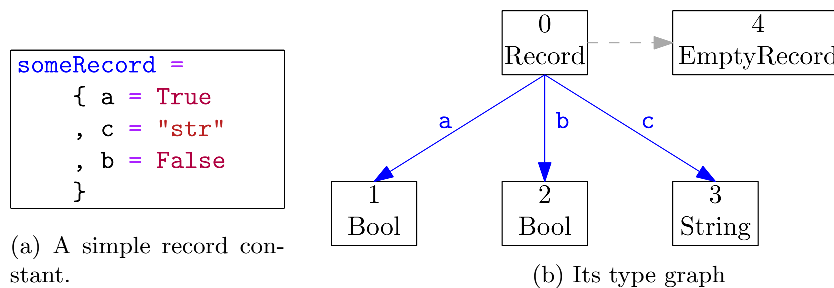
Table 6.1: Reconstructed types, the built-in constraint solver versus type graphs

The last expression is also the only one where the error was worded as a type inconsistency, rather than an infinite type.

6.4 Records

Records in Elm are quite flexible. One can create a record with any number of fields, access them and update the values of individual fields in a record. One can even write functions that take any record as argument as long as they have certain members with certain types. These flexible records are based on a paper written by Leijen [26], differing only in the ability to add or remove members to and from records dynamically, which Elm has disabled for performance and program readability reasons [7].

Unlike functions and algebraic data types, records cannot be described in terms of type constants, type applications and type variables. The biggest reason for this is the polymorphism that allows functions to merely demand the existence of specific members in the records provided as arguments. This means that the type graphs described thus far are ill equipped for reasoning about records. Some kind of extension is needed to be able to cope with records.



(a) A simple record constant.

(b) Its type graph

Figure 6.6: A simple record constant and its graph representation.

Let us first take a look at an ordinary record. Figure 6.6 shows a simple record and the type graph representing its type. In the type graph, node 0 contains a simple type constant called `Record`. Nodes 1, 2 and 3 contain the types of the members of the record. Note that the order of members differs between the code and the type graph. The type graph adds the types of the members arbitrarily in alphabetical order since order is irrelevant.

The blue edges going from node 0 to nodes 1, 2 and three are neither type hierarchy nor constraint edges. The records keep track of where the types of its members are located in the graph in a special kind of edge called “member” edges. These member edges simply store the name of the member and a reference to the node representing its type.

Finally, node 4 contains an `EmptyRecord` constructor. This is to account for the ability to extend records. Even though adding or removing members to and from records is disallowed, one can still update values in records. One could, for example, write `otherRecord = { someRecord | a = False }` or even `otherRecord = { someRecord | a = "str" }`, although the ability to change types of record members may be unintentional. Since the record in figure 6.6 is not based on any other record, the type graph just holds an empty record as placeholder.

Records must always be an extension of either an empty record, another record or a type variable. Records that are not based on other records, like the one in figure 6.6, extend the empty record. A Record that extends another

record forms a record containing the union of the members of both records. The polymorphism of records becomes apparent when a record extends a type variable, which means that the record can have any number of other members besides the ones defined in the record itself. This can be seen in figure 6.7, where the argument to `foo` can take any record as long as it contains a member `baz` of type `Bool`. Record extension is a transitive relationship between records. A record can extend another record which in turn extends a type variable. This would make the first record as polymorphic as the record it extends.

```
foo : { bar | baz : Bool } -> Bool
foo x = x.baz
```

Figure 6.7: An instance of row polymorphism.

6.4.1 Finding type conflicts

Type conflicts are found by finding two nodes in the same equality group that cannot be unified. The algorithm that does this needs not be modified to reason about the individual members of records. After all, unless those members are records themselves, they act like any other normal type. The records themselves, however, require some special attention. Fortunately, the rules of type constants still apply to records. `Bool` types, for example, cannot be unified with records and will thus be marked as incompatible.

```
1  members :: Record -> Set Member
2  members EmptyRecord = empty
3  members TypeVariable = empty
4  members (Record mems extends) =
5    mems ∪ members extends
6
7  membersMatch :: Record -> Record -> Bool
8  membersMatch left right =
9    case (monomorphic left, monomorphic right) of
10     (True, True) -> members left == members right
11     (False, True) -> members left ⊆ members right
12     (True, False) -> members left ⊇ members right
13     (True, True) -> True
```

Figure 6.8: Deciding whether two record types conflict

Additional logic, however, is needed to compare a pair of records. This logic is shown in 6.8. The comparison depends on what both records extend. If both records are monomorphic, meaning that their set of members is fixed, then neither record is allowed to contain members that the other record does not have. In other words, the sets of members must be equal. This is not the case when either or both of the records are polymorphic. When only one of the records is polymorphic, then its set of members must be a subset of that of the monomorphic record. The members may be completely disjoint when both of the records are polymorphic.

When checking compatibility between two records, not only must the two records have comparable sets of members, the members the two records have in common must have the same type. This is a bit trickier and cannot easily be done in the function that checks whether an equivalence group is consistent. After all, the types of the members exist in different equivalence groups which themselves may be inconsistent for other reasons. Figure 6.9 demonstrates a situation where two records are linked by a constraint.

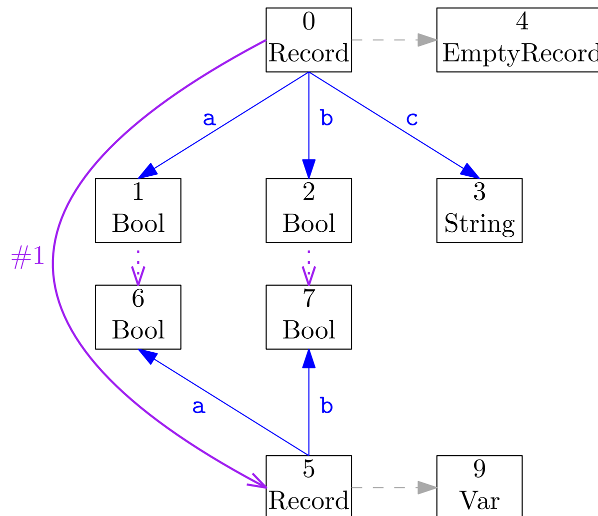


Figure 6.9: A monomorphic record constrained to a polymorphic one.

The record at node 0 is the same as the record in figure 6.6. Since the record at node 5 extends a type variable, it is polymorphic. Its members must be a subset of the members of the record at node 0. Nevertheless, the members both records have in common must have the same type. Consequently, those common members must end up in the same equivalence group. This is made sure of during the type graph building phase.

Whenever a constraint is introduced that causes two records to be put in the same equivalence group, this constraint is duplicated among the common members of the two records. In the example, constraint #1 forces the two records directly into the same equivalence group. This means that the common members of the two records, namely `a` and `b` must share equivalence groups. After all, member `a` of one record must have the same type as the `a` member of the other record. The two dotted purple arrows show how constraint #1 is duplicated amongst those members. Since the record at node 5 lacks a member `c`, the `c` member of the record at node 0 is left alone.

This method properly observes the rules of when two records can be unified. The duplication of a constraint among the members of two records does, however, have a bad effect on error messages. After all, when two records have a common member but with conflicting types, e.g. `{a = True}` and `{a = "str"}`, the type error will not state the type conflict not as one between two records, but one between a `Bool` and a `String`. This is highly undesirable. This can be resolved by holding a reference back to the original records in the duplicated constraints and making sure to display the original records rather their members

in the type error.

Chapter 7

Siblings

Sibling functions are functions that are conceptually similar, but in reality have different types. Often, one function is confused for the other. Hints that say "Did you mean function <foo> instead?" can be very powerful when shown at the right time. This is exactly what siblings do.

For this to work, the compiler requires some way of knowing which functions are conceptually similar. When are two functions easily confused? That question is nontrivial, especially when looking for an algorithm to have the compiler find similar functions on its own. After all, several factors contribute to whether functions are easily confused. Sometimes functions look similar (e.g. `(<|)` and `(|>)`). Sometimes two functions have very similar functionalities, but take a different approach (e.g. `foldl` and `foldr`). Other times two functions are confused because people are used to a different behavior from another programming language. The + operator, for instance, performs concatenation in JavaScript. In Elm, it merely adds numbers. The `(++)` operator denotes concatenation.

Rather than trying to find some algorithm to judge these cases, it is better to use human insight to judge when two functions are similar. To allow humans to tell the compiler about similar functions, some (provisional) syntax has been introduced:

```
sibling foo resembles bar
```

This statement states that some function called `foo` can be confused for another function called `bar`. In the current implementation, this is a one-way relationship, meaning that when `foo` is written down while `bar` is meant, a hint will be shown suggesting to use `bar` instead. Confusing `bar` for `foo` will not throw such a hint. This is a design decision that leaves open the possibility of adding a custom message to this hint. It is trivial to make sibling definitions go both ways.

Once the compiler knows which functions can be considered easily confused, its task becomes deciding when it is useful to show such a hint. There are three important factors in deciding when to throw a hint:

1. The function is involved in a type error.
2. Its sibling function solves this type error.
3. This sibling does not cause new type errors.

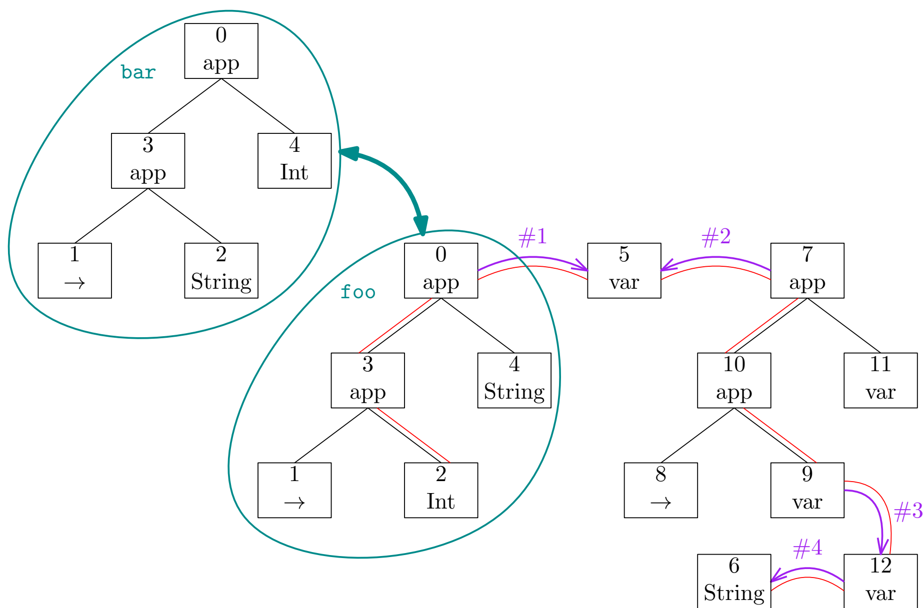
When a certain function is not involved in any type error, directly or indirectly, one can be sure that the function has nothing to do with the error. Suggesting alternatives for functions that have nothing to do with an error would help no one. It would at best be useless and at worst be very confusing. Secondly, the compiler must be sure that a function's sibling actually solves the type error. Otherwise the hints can be deceptive, leading one to think the wrong function is used even though the problem lies elsewhere. Thirdly, the use of a sibling should not cause new type errors. As it turns out, type graphs are ideal for deciding when a sibling solves the type error some function is involved in.

```
foo : Int -> String
bar : String -> Int

sibling foo resembles bar

baz = foo "15"
```

(a) A simple record constant.



(b) Its type graph

Figure 7.1: Siblings in action. `foo` here is confused for `bar` in the definition `baz` The type graph shows how the type of `foo` is replaced by `bar` to see if that resolves the type error.

Figure 7.1 shows siblings in action. Figure 7.1a shows the types of the functions `foo` and `bar`. The implementations are not given. The sibling states that `foo` resembles `bar`. In the definition of `baz`, `foo` is called with a String, even though the function expects an Int. Figure 7.1b shows the type graph of the expression `foo "15"`.

The type represented by the type hierarchy starting at node 0 is the type

of `foo`. Constraint #1 states that node 5 must be an instance of the type of `foo`. Constraint #2 states that the type represented by note 7 must have the same arity as the function represented by node #5. Constraint #3 states that node 12 is the first argument of the function. Constraint #4 states that a literal `String` was filled in as the first argument of the function call. This means that nodes 2 and 6 are in conflict: node 2 holds an `Int` and node 6 holds a `String`. The red path connecting the two nodes shows which constraints are involved in this type error. In this case, all three constraints are involved.

The most interesting constraint here is #1. This constraint specifically states that the type of the function `foo` is inserted there. Since this constraint appears in the error path, it is now certain that the `foo` is involved in the type conflict. It is then useful to see if any of its siblings, which in this case is just the function `bar`, would resolve this type error. To try this, the type of `foo` is taken out of the type graph and replaced by the type of `bar`, reconnecting constraint #1 to the new node 0.

One useful property of an instance constraint, such as constraint #1, is that an instance constraint is necessarily the *only* constraint that connects the the type of the function to anything else. This is because the type of a function is freshly created when an instance constraint is encountered. This property is necessary for verifying that a sibling resolves a type conflict and does not cause any new type conflicts.

When the type graph holds the type of the sibling `bar`, it can be verified that using `bar` instead of `foo` would fix the type error and not cause any new type errors. When, after the replacement, the instance constraint is still part of an error path, then either the original type conflict was not resolved or a new type conflict was created. Since the instance constraint is the only constraint connecting the type of the sibling to the rest of the graph, it being a part of an error path must necessarily mean that the type of the sibling is in conflict with some other type in the graph.

In figure 7.1, the type of `bar` luckily resolves the error and does not cause any new ones. Its first parameter is a String, which is exactly the same as the argument given. Whereas nodes 2 and 6 were in conflict before, they are not anymore when `foo` is replaced by `bar`. A hint is generated that suggests to use `bar` instead of `foo`.

Unlike in Heeren's implementation of siblings [16], siblings do not decide which constraint gets the blame. In Heeren's implementation, the instance constraint (constraint #1) would have gotten the blame, since it is the deciding factor in deciding whether the sibling works. This constraint holds an error message that says the function `foo` is somehow used incorrectly. The above described implementation, however, allows the type graph heuristics to blame *any* of the constraints in the error path. This includes the instance constraint, #1, but also constraints #2, #3 and #4. In this particular case, constraint #3 is chosen, since it states specifically that the first argument of the function has the wrong type. Figure 7.2 shows the resulting error.

```
The argument to function `foo` is causing a mismatch.

39|        foo "15"
               ^^^^
Function `foo` is expecting the argument to be:

    Int

But it is:

    String

Hint: Did you mean bar instead of foo?
```

Figure 7.2: The error shown for the code shown in figure 7.1

Chapter 8

Interfaces

The interfaces implemented in Elm are very similar to Haskell 98 type classes. As with any implementation of type classes, some design decisions were made. Most design decisions were made to make type classes as simple as possible. This is because it saves time and keeps things relatively easy. Following Simon P. Jones' description of the design space of type classes [24], the design decisions are as follows:

- Multi-parameter type classes are not supported.
- All of the constraints in the context of an interface declaration must be of the form $C\ \alpha$, where C is another interface and α is a type variable.
- No context reduction is performed.
- Overlapping instances are not forbidden.
- Instance declarations, called "implementations", demand that the types are simple, meaning that the types for which an interface is implemented must be of the form $T\,\alpha_1\,\ldots\,\alpha_n$, with T a type constructor and α_1 to α_n distinct type variables.
- The constraints in the context of an implementation must be simple.
- The set of type variables occurring in the context of an interface declaration must be a subset of the type variables in the type of the interface.
- The members of an interface are allowed to have a context with more constraints than the interface header describes.
- The check for missing implementations happens during type inferencing. This is as opposed to resolving the implementations at the end of the compilation process. Implementations of concrete types must be available when they are demanded.

Since the research focused on implementing support for type classes/interfaces in type classes, the support of them ends right after constraint solving. This means that interfaces and implementations generate no code. This justifies particularly the lack of context reduction, since that would happen after

the constraint solving. This chapter will not focus on how interfaces are implemented. There is plenty of literature that describes this process [24], [31]. Instead, the focus lies on their specific interaction with type graphs.

8.1 Qualifiers in type graphs

The main purpose of type graphs is to investigate type conflicts and to reason about the types of an expression. The introduction of interfaces demands the reasoning of types to include interfaces and the existence of a certain implementation for a given type. It is not strictly necessary to let type graphs do this heavy lifting. In Helium, for example, the type graphs do not reason about type class predicates, although its implementation of type classes is limited in the first place: all instances are predefined, no new instances can be created. Even with a full fledged class/interface system, though, a type graph could be left to reason about type conflicts unrelated to them. The resolving of type classes can after all be done with the built-in solver, albeit with some modifications.

Despite being able to reason about interfaces without invoking type graphs, it is useful to investigate whether type graphs can improve error messages. As has been said before, the built-in constraint solver is biased in blaming the first constraint that directly causes some unification to fail. Type graphs can eliminate this bias completely. Perhaps this can have some effect on the quality of type error messages related to interfaces too.

The most important aspect of interfaces is that they allow ad-hoc polymorphism [38]. One can make a function polymorphic in some type variable, but demand that an implementation of some interface exists for whichever type is filled in for this type variable in a function call. The reasoning about these predicates and how they flow from type variables to specific types is something that can be built into type graphs. The way of doing this is labeling vertices with their interface predicates.

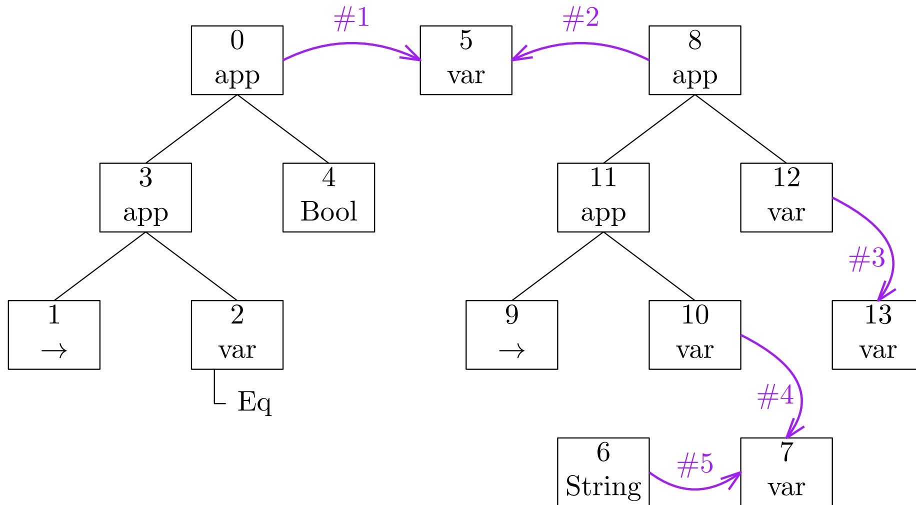


Figure 8.1: The type graph of `foo "str"`, with `foo : a -> Bool | Eq a`

Let us first take a look at how this labeling affects the type graph. Figure

8.1 shows the type graph of the expression `foo "str"`. The type of `foo` is the type hierarchy starting at node 0. Note that the first parameter of `foo` is a type variable, but has a label `Eq` hanging under it. Type variables know their own qualifiers. If type variable `a` were to be repeated in the type of `foo`, all nodes representing `a` would hold the `Eq` label.

With nodes holding labels with predicates, algorithms on the graph can reason about these predicates in relation to the other types of the graph. Currently, there is a separate algorithm specifically designed to find missing implementations. It investigates the equivalence groups as described in the following pseudo code algorithm:

```
groupPredicates :: EquivalenceGroup -> [Predicate]
groupPredicates grp =
  concatMap predicateLabels (typeVars grp)

checkGroup :: EquivalenceGroup -> Bool
checkGroup grp =
  and
    [ implementationExists pred  (reconstructType vertex)
    | pred <- groupPredicates grp
    , vertex <- vertices grp
    , not (isTypeVar vertex)
    ]

checkGraph :: TypeGraph -> Bool
checkGraph grph =
  and checkGroup (equivalenceGroups graph)
```

This rather primitive algorithm searches the type graph for missing implementations. The `checkGroup` function returns false if a type is missing information. The code to generate an error for the specific missing implementation has been left out for simplicity. The `checkGroup` function checks for missing implementations in a single group. First, through `groupPredicates`, it collects all the predicates held by the type variables. Then for all vertices that do not hold type variables, it checks whether there exists an implementation of the type represented by that vertex.

The `findImplementation` function is independent from the type graph. It returns whether an implementation can be found that matches the given type. This algorithm merely replicates the behavior of the built-in constraint solver. One benefit of this algorithm is that it can reason about missing implementations even when type conflicts are present. This means that one can decide to check for missing implementations after it has been made sure that there are not any other type errors. The compiler can be made to prefer throwing normal errors before errors about missing implementations, or the other way around. The built-in solver resolves implementations as soon as a type variable is unified with a type constant (or type application). This means that it could throw a missing implementation error when a normal type error would explain the situation more appropriately.

This situation is shown in figure 8.2. In Figure 8.2a, function `foo` is defined with type `foo : a -> a | Eq a`. Under that, `Foo` is defined as an Algebraic Data Structure with just one constructor. The `bar` function is where it goes

wrong. With Foo being its argument, there are two problems: There is no implementation of Eq for Foo and even if there was, the return type (also Foo) would not match the Bool expected by it being the condition of the if-statement. The (modified) built-in inferencer throws two errors: the ones seen in figure 8.2b and 8.2d. The type graph only throws one error, which is the one shown in 8.2a. Since both problems have the same cause, namely Foo being given as an argument to foo, it can be argued that the error thrown by the type graph is more appropriate.

```
foo : a -> a | Eq a
foo a =
  if isis a a
  then a
  else a

type Foo = Foo

bar =
  if foo Foo
  then "baz"
  else "qux"
```

(a) The code.

```
`foo` is being used in an unexpected way.

320| bar = if foo Foo then "baz" else ""
                  ^^^
Based on its definition, `foo` has this type:

    a -> a | Main.Eq a

But you are trying to use it as:

    Foo -> Bool
```

(b) Error thrown by type graph.

```
This condition does not evaluate to a boolean value, True or False.

320| bar = if foo Foo then "baz" else ""
              ^^^^^^^
You have given me a condition with this type:

    Foo

But I need it to be:

    Bool
```

(c) First error thrown by built-in inferencer.

```
Missing a specific implementation of an interface.

320| bar = if foo Foo then "baz" else ""
              ^^^^^^^
In order for this code to work, there needs to be an implementation of `Eq` for

    Foo

This implementation should either be in this file or in one of your imports.
Perhaps you forgot to import a module that provides this implementation?
```

(d) Second error thrown by built-in inferencer.

Figure 8.2: Code with both a type conflict and a missing implementation.

Chapter 9

Specialized type rules

Specialized type rules allow the authors of functions to create custom hints for error messages shown when their functions are misused. Those custom hints can be used to explain common pitfalls or otherwise explain an otherwise technical error message in terms of the functionality that a function provides. Specialized type rules give developers the feeling that they are in control of type inferencing and the messages shown when two types cannot be unified. They cannot, however, write type rules that would fail when the original type inferencer would succeed, or succeed when the original type inferencer would fail. They also do not control the error messages themselves, only the hints that are added to them.

Specialized type rules essentially entail the overriding of the type constraints generated when the function is applied to some arguments. The syntax of type rules thus reads as a description of which constraints are to be generated and in what order. This requires some insight about the constraint generation process, though luckily there are very clear error messages when the type rules of a function are inconsistent with its type annotation.

Let us take a look at the syntax of type rules. Figure 9.1 shows an example function with specialized type rules, it is the same as figure 5.4. As explained in chapter 5.4, `constrain` rules recursively constrain either one of the arguments or the return type. When left out, they are inserted automatically during the validation phase (the validation phase is described in section 4.2). The `unify` rules create equality constraints. Finally, `check` rules enforce interface predicates. Specialized type rules, when defined, must reside between the type annotation and the definition of a function. This conveniently forces the type annotation to exist, which is strictly necessary. Specialized type rules can be created for all curried versions of a function, in the example, errors for `checkMaybe` and `checkMaybe maybe` could have optionally been defined besides `checkmaybe maybe val`, but this was left out for brevity. Rules pertaining unification and predicate checking can optionally be given a reason. This reason will be shown in a hint to the user when the constraint cannot be satisfied.

```
checkMaybe : Maybe a -> a -> Bool | Eq a
errors for checkMaybe maybe val where
  constrain maybe
  constrain val

  unify maybe with Maybe a_1
    because The first argument has to be a Maybe.

  unify val with a_2

  unify a_2 with a_1
    because The second argument must match the thing in the Maybe.

  check Eq a_1
    because Eq is needed to test equality.

  unify return with Bool

  constrain return

checkMaybe maybe val =
    case maybe of
        Nothing -> False
        Just x -> isis x val
```

Figure 9.1: A function definition with type rules.

9.1 Replacing constraints

The magic of specialized type rules happens in the constraint generation phase of the compiler. As mentioned in chapter 4.2, constraint generation works in a top-down fashion. When the constraint generator comes across a function application, the state is searched to see if there are any type rules that match the function application with the amount of arguments it has been given. If such type rules exist, then the constraints defined by those type rules are generated, otherwise the normal constraints are generated. The function deciding this is described with pseudo code in figure 9.2.

The `customConstraints` function generates the constraints as defined by the specialized type rules. It iterates over the constraints defined in the type rules from top to bottom and builds the constraint tree from there. The function for generating the constraint(s) of a single type rule is shown in figure 9.3.

The `customTypeRule` takes the expression of the function application, a single type rule and it returns a tree of constraints. The constraints generated depends on the type rule. With `constrain` type rules, shown on lines 4 and 5, the existing `constrain` function is called to generate the constraints of an argument or the return type of the function.

The `unify` rule generates an equality constraint, shown on lines 7 to 13. First it instantiates the types of both the left hand side (lhs) and right hand side (rhs). This instantiation uses the state monad (which in reality is IO) to make sure

```haskell
constrainTypeApplication :: Expression -> ConstraintTree
constrainTypeApplication (FunctionApplication funcName args) =
  let
    typeRules = findTypeRules funcName (length args)
  in
    case typeRules of
      Nothing -> builtInConstraints funcName args
      Just rules -> customConstraints rules funcName args
```

Figure 9.2: The function deciding whether normal or custom constraints are to be generated.

```haskell
customTypeRule :: Expression -> TypeRule -> State ConstraintTree
customTypeRule expr rule =
  case rule of
    ConstrainRule arg ->
      constrain (getArg arg expr)

    UnifyRule lhs rhs hint ->
      do
        lhsTp <- instantiateType lhs
        rhsTp <- instantiateType rhs
        errorMessage <- decideErrorMessage lhs rhs hint

        return (EqualityConstraint lhsTp rhsTp errorMessage)

    CheckRule iface var hint ->
      do
        varTp <- intantiateType var
        fresh <- freshTypeVariable
        fresh' <- addQualifier iface fresh
        freshTp <- instantiateType fresh'

        errorMessage <- decideErrorMessage var freshTp hint

        return (EqualityConstraint varTp freshTp errorMessage)
```

Figure 9.3: The function for generating the constraint(s) of one single type rule.

that already instantiated types are re-used. The function `decideErrorMessage` generates an appropriate error message to show to the user when the constraint is broken. The instantiated types and the error message are stored in an equality constraint, which is returned.

The `check` rule generates a constraint very similar to that of the `unify` rule. Conceptually, the rule `check Eq a_1` can be seen as an equality constraint between the type variable `a_1` and a fresh type variable that holds the `SomeClass` qualifier. This means that `check Eq a_1` is merely syntactic sugar for `unify a_1 with b | Eq b`.

The `decideErrorMessage` requires some attention, as it decides the error message based on a `unify` or `check` type rule. The error generated by this function can tell the user whether the error lies with a single argument that has an unexpected type, the return type being different than expected, a conflict between the types suggested by the arguments and the return type or a conflict between two or more arguments. These errors differ from the error messages added to constraints when no type rules are involved, yet they are still generated automatically. The big question here is: how does this function know which parameters the type rules reason about?

Part of the answer to this question is numbered type variables. In figure 9.1, the type of the function is `Maybe a -> a -> Bool | Eq a`. In the type rules of that function, there are mentions of `a_1` and `a_2`. These numbered type variables actually refer to the type variables in the type annotation. The numbers referring to the occurrence of the type variable. In this example, `a_1` refers to the `a` in `Maybe a`. The variable `a_2` refers to the `a` of the second parameter. Type variables in qualifiers never need to be referred to. As such, the `a` of the `Eq` qualifier has no associated numbered type variable.

It is easy to calculate to which parameters (or return value) numbered type vars refer to. After all, this information is stored in the type annotation. Besides numbered type variables, type rules can also contain references to the arguments and return value themselves. In figure 9.1, these are `maybe`, `val` and `return`. These already refer to parameters and the return value. Finally, one can use fresh type variables. It is difficult to reason about the parameters those refer to, since they have no inherent link to the type annotation or variables. Luckily, the numbered type vars and arguments can provide enough information to judge what fresh type variables refer to: all fresh type variables must somehow be connected to either numbered type variables or arguments through type rules. Fresh type variables that appear in the same rules as numbered type vars or arguments are judged to refer to the same parameters as those other variables. These references can be passed on from one fresh variable to another. With a fixed point algorithm no fresh variable will be left without at least one reference to a parameter or return value, unless the fresh type variable describes a dangling type. Dangling types are, however, not possible, since it would require a type variable on the left hand side that has not yet been introduced.

Once all type variables are linked to the parameters and/or return value they refer to, the type rules containing these type variables can use that information to figure out what they refer to. The parameters and/or return type that a type rule refers to is defined as the union of the parameters/return types all type variables mentioned in the type rules refer to.

Figure 9.4 shows the results of this effort. The `checkMaybe` function in figure 9.4a is defined in figure 9.1. The algorithm finds out that the conflict is between the types of the first and second arguments of the function. It also shows the hint provided by the author of `checkMaybe` at the bottom of the error message.

9.2 Validating type rules

Specialized type rules can be very powerful in defining custom hints to error messages and defining an order in which the type of a function application is to be checked. When left unchecked, though, a programmer could write type

```
foo =
  checkMaybe
    (Just True)
    "bar"
```

(a) An error in the use of `checkMaybe`

```
The 1st and 2nd arguments of function `checkMaybe` conflict with one another.

273|    checkMaybe (Just True) "bar"
        ^^^^^^^^^^^^^^^^^^^^^^^^^^^^
Function `checkMaybe` is expecting the 2nd argument to be:

    Bool

But it is:

    String

Hint:
The author of function `checkMaybe` gives the following explanation:
  The second argument must match the thing in the Maybe
```

(b) Resulting error

Figure 9.4: A type error in the use of a function with specialized type rules.

rules that make no sense or describe a type that is not in line with the type of the function annotation. One could, for example write type rules that demand implementations of interfaces that the type annotation does not demand. Programmers ashamed of how complicated their types have become could hide complicated details away from the type annotation. In another sense, a programmer could make a mistake and forget a rule or have the type rules describe the wrong type. This would have a disastrous effect, as it would mean the type annotation of a function is not necessarily the real type of the function anymore. This is highly undesirable. Fortunately, there is a solution for this: validating the type rules.

Before validation, a little check runs to see if there are any missing `constrain` rules. Any missing parameter `constrain` rules are added before all other rules. If the `return constrain` rule is missing, it is added at the end. This way, `constrain` rules can be left out.

The first real form of validation requires that all parameters of the function (in figure 9.1, `maybe` and `val`) and `return` appear on the left hand side of at least one `unify` type rule. Secondly, all numbered type variables must appear at least once on the right hand side of a `unify` type rule. The first check enforces that rules exist that reason about every part of the function. The second check is needed for the parameter matching described in the previous section. It would be difficult to find out which parameters type rules reason about if the numbered type variables are not used.

Once these basic checks have been performed, the type inferencer makes sure that the type annotation agrees fully with the type described by the type rules. On a high level, this is done by generating constraints between the type of the type annotation and the type described by the type rules. This happens in the constraint generation phase. Once these constraints exist, the constraint solver will make sure that the right error message is thrown when the type described by the type rule does not match the type annotation.

Two sets of constraints are generated. The first set, generated for the type rules in figure 9.1, can be seen in figure 9.5. In the box on the left we can see the type of the type annotation, split up by parameters and return value. The pairs of boxes on the right match the type rules in order. An observant reader might notice that `constrain` rules are missing in this set of constraints. This is because `constrain` rules do not affect the type represented by the type rules. Finally, type variables with the same name are linked with gray lines. This shows that variables with the same name represent the same entity. In a type graph, all variables with the same name would share the same vertex.

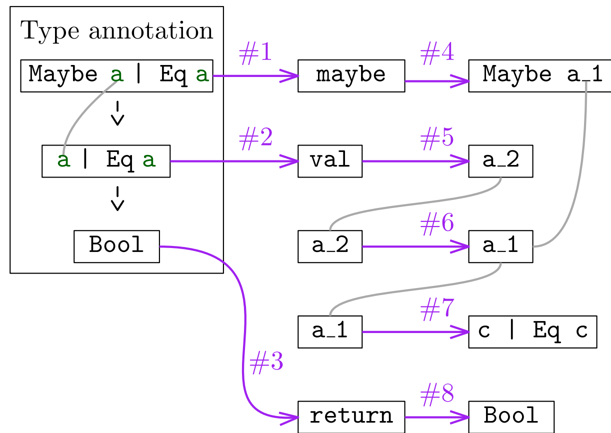


Figure 9.5: The first set of constraints used to validate type rules.

The first three constraints link the type annotation to the type rules. The type of the first parameter is linked to the type variable `maybe` in constraint #1. The type of the second parameter to `val` in constraint #2 and the return type `Bool` to `return` in constraint #3. Constraints such as #1 and #2 carry an error message that states that the argument in the type rule does not match the argument in the type annotation. Constraint #3 carries the message that the return type of the type rules does not match that of the type annotation.

Constraints #4, #5, #6 and #8 link the left hand sides of the `unify` rules to their right hand sides. Constraint #7 conceptually does the same for the `check` rule, with the variable being checked being the left hand side and a fresh variable (arbitrarily called `c` in this example) as the right hand side. Constraints #4, #5, #6, #7 and #8 carry an error message that states that the left hand side of the type rule does not match the right hand side.

The type variables in the type annotation, in the example just `a`, are made rigid (a.k.a. skolem variables). Rigid type variables, unlike flexible variables, cannot be unified with concrete types. They must remain polymorphic. This prevents type rules such as `unify a_1 with Bool` from passing the validation, which would have made the type rules describe a more monomorphic type than the type annotation. Rigid type variables also cannot be unified with other type variables that have *more* interface qualifiers than themselves. This excludes the possibility of writing `check` rules with qualifiers that the type annotation knows nothing about.

To demonstrate this, let us add the following rule to the type rules in figure 9.1:

```
check Ord a_1
```

The type of `checkMaybe` does not include `Ord a`. As such the addition of this `check` rule results in an error. This error is shown in figure 9.6. It shows the qualifiers that the type annotation expects for that specific variable, and the qualifier(s) described by the type rules.

One error that has not been accounted for yet is forgetting to check for a qualifier. While rigid type variables cannot be unified with type variables with more qualifiers, they *can* be unified with type variables with fewer. The above

```
The qualifier in this constraint does not exist in the type annotation.

261|    check Ord a_1

The type annotation describes this type:

    a | Main.Eq a

But the type rule describes this type:

    a | Main.Ord a

Hint:
Note that the previous rules and the type annotation decide the types of the variables
```

Figure 9.6: The error shown when the type rules check for qualifiers that are not mentioned in the type annotation.

set of constraints would not account for this. To account for this situation, a second set of constraints is built. This is shown in figure 9.7.

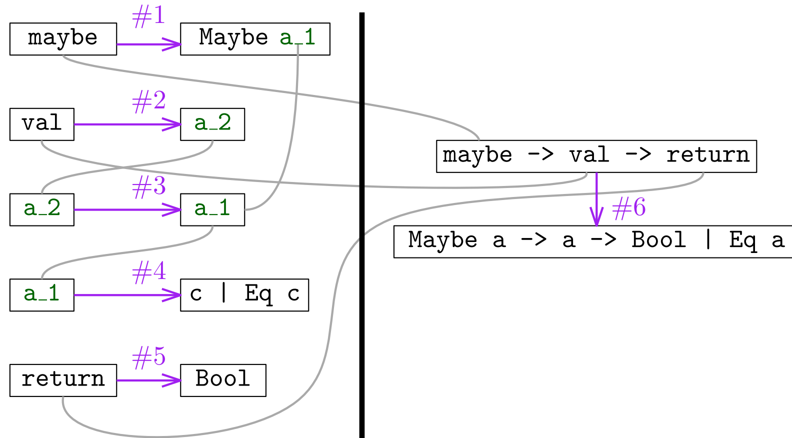


Figure 9.7: The second set of constraints used to validate type rules.

On the left side of the black bar the same constraints are seen as in the first set of type rules. Again, those constraints build the type represented by the type rules. The constraints have been renumbered to start at #1. Again, the gray lines indicate how variables with the same name represent the same entity. The black bar shows the structure of a let-constraint. A let-constraint is named as such because it models the types of let-expressions: the types of the expressions in the `let` of a `let ... in ...` statement must be generalized when the type of the body is inferred. This means that specific variables in the `let` can be made rigid when the inferencing of the body begins.

This property is precisely what is needed to find out whether any predicates are missing in the type rules. The constraints on the left side of the black bar are conceptually put into the `let`, the constraint on the right side of the bar is conceptually put as the body of the `let`. After the constraints in the `let` (on the left side of the bar) have been resolved, the constraint solver moves on to the body of the `let` (the right side of the bar), but not before making some type variables rigid. The numbered type variables, in the example `a_1` and `a_2`, are

made rigid when the constraint solver moves from the left side of the bar to the right side. When constraint #6 is then resolved, those numbered type variables cannot be unified with type variables that have more qualifiers. If the type variables in the type annotation have more qualifiers than the type represented by the type rules, the numbered variables in the type rules will refuse to be unified, causing an error.

This can be demonstrated by removing the `check` rule on lines 14 and 15 in figure 9.1. This would remove constraint #4 in figure 9.7. Without that rule, the type described by the type rules would be `Maybe a -> a -> Bool`, which is missing the `Eq a` qualifier. The error resulting from this mistake can be seen in figure 9.8. Like Elm's traditional error messages, the error describes the situation in human terms and shows the conflicting types.

```
The type generated by the type rules does not match the type annotation.

245| checkMaybe : Maybe a -> a -> Bool | Eq a
                  ^^^^^^^^^^^^^^^^^^^^^
The type rules generate this type:

    Maybe a -> a -> Bool

But the type annotation has this type:

    Maybe a -> a -> Bool | Main.Eq a
```

Figure 9.8: An error shown when the type rules and the type annotation describe different types.

Conclusion

This thesis has described several methods of improving error messages in Elm. It applies techniques developed by Heeren and Hage [18] and extends the techniques. The first technique is an implementation of type graphs, which can remove bias in a type inferencer and improve error messages by laying the types and constraints out in a graph like structure. Type graphs are shown to be applicable in Elm, meaning they capable of reasoning about record types. The ability to have records build upon other records is captured accurately in type graphs. Besides the records, type graphs have proven themselves to be more capable of reconstructing infinite types, which is beneficial for the explanation shown in the error to the end user.

Siblings are two functions that are conceptually similar, but differ in types. A syntax has been defined with which a developer can mark pairs of functions as siblings. The compiler can use this information to add hints to error messages which note that perhaps a sibling of some function was supposed to be used instead. To prevent these siblings from causing confusion, there are three demands: a hint must not be shown when the function for which a sibling is suggested is not part of the type conflict, when the sibling does not solve the type conflict or when the sibling causes a new type conflict. Through type graphs, sibling hints can be made sure to meet all three demands.

Interfaces were implemented as a simple form of type classes. Rather than using keywords `class` and `instance`, the words `interface` and `implementation` are used instead. Their features are very similar to those of type classes in Haskell 98, though the processing of interfaces stops after type inferencing. Type graphs are shown to be able to reason about interface qualifiers in types, although this brings no particular benefits, since missing implementations are sought for in a separate pass over the type graph.

Specialized type rules allow the authors of functions to create custom hints for error messages shown when their functions are misused. A syntax is created to influence the constraints generated by the type inferencer when a specific function is called. With this syntax, the constraints can be re-ordered and custom hints can be added for when a specific constraint is broken. When such constraint does indeed get broken, the end user will see an error message with a hint in which that the author of the function explains the type conflict and possibly how it can be resolved.

Type rules seemingly give the ability to redefine the type constraints associated with the usage of a function. It should not, however, be possible to have the type rules describe a type that is not in conflict with the type of the function. An algorithm is described to verify that the type rules of a function accurately and fully describe the type of said function. This verification can throw de-

scriptive error messages in the same style as any other type error message in Elm.

Future work

This thesis investigates the effects of type classes on the type graph. While the type graph is shown to be able to reason about the qualifiers in type variables, doing so seems to have no benefit above doing it the classical way. Perhaps the algorithm can be modified to integrate the check for missing implementations with the normal type substitution algorithm. Would this have any benefits? Could the check for the existence of an implementation be useful for heuristics on the type graph?

It has not been decided whether type classes are the best solution for ad-hoc polymorphism in Elm [8]. Some alternatives have been mentioned: Rank-N polymorphism, higher kinded polymorphism and implicit arguments. It might be interesting to see whether type graphs are powerful enough to reason about these concepts. This information might prove vital if the wish exists to indeed implement type graphs while leaving the option open to implement some other kind of ad-hoc polymorphism than type classes.

Siblings currently have no added hints. This may or may not be useful, implementing them should be trivial.

Specialized type rules currently require numbered type variables that refer to the type annotation. These are required for two reasons, the first is automatically generating the type error messages associated to the constraints, using the numbered type variables to know which parameters (and return type) type rules refer to. The second is validating the type rules, where the numbered type variables are made rigid (skolem) to make the constraint solver check for missing qualifiers. These numbered type variables can be cumbersome and counterintuitive. Were they not strictly needed, it would have been better to remove them and allow developers to solely use concrete types and fresh type variables in their type rules. Perhaps the algorithms that need numbered variables can be adapted such that these numbered variables are no longer needed.

Reflection

Elm is an interesting language. Its syntax looks similar to that of Haskell, but it is much simpler and the error messages are very much different. Elm is already known for understandable error messages, yet this thesis is about finding ways of improving these same error messages. "Why fix things that are not broken?" is a question that often gets asked in such cases. I would like to turn this question around: why should one dismiss an opportunity to make something good even better? With this attitude, Elm is chosen precisely *because* it already has nice error messages, and there are opportunities to make them even better. Type graphs and their heuristics alone are much better at reconstructing infinite types than the original implementation is, for example. This shows that by introducing this concept, an effort to improve error messages (specifically the ones pertaining infinite types) is taken one step further.

This thesis brings forth some nice ideas, none of which I expect to be merged into the official Elm compiler's repository. While they do indeed improve error messages and show what is possible with the techniques described, the implementation is rushed, incomplete. Rushed because of the limited time to investigate the concepts and incomplete because minor details (such as simple validity checks for interfaces) have been left out. This is not to say that the findings are not genuine. They work and do indeed improve error messages, but their rushed implementations simply make them not ready for production.

The second reason it should not be merged into the official repository involves the design decisions. It can be very meaningful to have a discussion about whether these ideas should be implemented in the official Elm compiler, and if they do, in what form. Siblings, for example, currently have a syntax that describes which function resembles which. In an early discussion about this, people have already voiced their opinions, stating that it might be better to put those siblings in special comments above the function definitions. The reason why they have been implemented as statements is simply because parsing those comments would have been more work.

Type rules also come with a range of design decisions. Should they really live between the type annotation and definition of a function? Should developers be allowed to write their own error messages or just add hints? Would it be better to put those in the special comments too? What should the syntax look like? All of these questions are answered in the specific implementation described in this thesis, though it would be unreasonable to think that these decisions are to be taken at face value. There might be very good reasons to take the idea and give it a completely different syntax, give developers more (or less) freedom to alter error messages and/or allow them to be written anywhere.

Interfaces

There have been numerous discussions on whether and how Elm should tackle ad-hoc polymorphism. In one GitHub issue [8], Czaplicki lays out several options:

1. Type classes
2. Higher kinded polymorphism in combination with Rank-N polymorphism
3. Higher kinded polymorphism in combination with Rank-N polymorphism and implicit arguments
4. Module functors

Czaplicki describes his approach as one of "wait and see". Despite this attitude, this thesis describes some implementation of type classes. This is not because I personally think type classes are the preferred solution to this problem. They are implemented purely to investigate their effects on type graphs. It simply seemed interesting to see whether type graphs can also reason about type classes. In the end it turned out that this is the case, although not much benefit has been gained from doing so. After all, errors about missing implementations are not any different than those generated by the primitive solver.

Closing Remarks

In the end I hope that the ideas inspire people to use them to solve real problems with type error messages. Elm has many good error messages where similar languages do not. Despite this I feel that the techniques described in this thesis attack precisely the points where Elm falls a little short. Whether by type graphs improving the choice of error messages and a better ability to reconstruct infinite types or by giving advanced developers some control the error messages of their functions, helping end users understand type errors in the context of the library they are working with. These techniques solve specific problems, and that is why I believe they deserve at least some serious discussion.

The implementation of the techniques described in this thesis can be found at https://github.com/FPtje/elm-compiler, on the `master` branch.

Bibliography

[1] Heinrich Apfelmus. Reactive-banana, 2015. `https://wiki.haskell.org/Reactive-banana` Accessed: 2015-12-04.

[2] Lennart Augustsson. Custom type errors, 2015. `https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors` Accessed: 2015-11-25.

[3] Sheng Chen and Martin Erwig. Counter-factual typing for debugging type errors. In Jagannathan and Sewell [22], pages 583–594.

[4] Sheng Chen and Martin Erwig. Guided type debugging. In Michael Codish and Eijiro Sumii, editors, *Functional and Logic Programming - 12th International Symposium, FLOPS 2014, Kanazawa, Japan, June 4-6, 2014. Proceedings*, volume 8475 of *Lecture Notes in Computer Science*, pages 35–51. Springer, 2014.

[5] David Raymond Christiansen. Reflect on your mistakes! lightweight domain-specific error messages. In *Preproceedings of the 15th Symposium on Trends in Functional Programming*, 2014.

[6] Evan Czaplicki. Elm: Concurrent FRP for functional GUIs. *Senior thesis, Harvard University*, 2012.

[7] Evan Czaplicki. Compilers as assistants, nov 2015. `http://elm-lang.org/blog/compilers-as-assistants` Accessed: 2015-11-25.

[8] Evan Czaplicki. type system extensions, aug 2015. `https://github.com/elm-lang/elm-compiler/issues/1039` Accessed: 2015-12-18.

[9] Evan Czaplicki and Stephen Chong. Asynchronous functional reactive programming for guis. In Hans-Juergen Boehm and Cormac Flanagan, editors, *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 411–422. ACM, 2013.

[10] Conal Elliott. Push-pull functional reactive programming. In Stephanie Weirich, editor, *Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell, Haskell 2009, Edinburgh, Scotland, UK, 3 September 2009*, pages 25–36. ACM, 2009.

[11] Conal Elliott and Paul Hudak. Functional reactive animation. In *ACM SIGPLAN Notices*, volume 32, pages 263–273. ACM, 1997.

[12] Christian Haack and Joe Wells. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.*, 50(1-3):189–224, 2004.

[13] Jurriaan Hage et al. Domain specific type error diagnosis (domsted). *Technical Report Series*, (UU-CS-2014-019), 2014.

[14] Jurriaan Hage and Bastiaan Heeren. Heuristics for type error discovery and recovery. In Zoltán Horváth, Viktória Zsók, and Andrew Butterfield, editors, *Implementation and Application of Functional Languages, 18th International Symp osium, IFL 2006, Budapest, Hungary, September 4-6, 2006, Revised Selected Papers*, volume 4449 of *Lecture Notes in Computer Science*, pages 199–216. Springer, 2006.

[15] Jurriaan Hage and Bastiaan Heeren. Strategies for solving constraints in type and effect systems. *Electr. Notes Theor. Comput. Sci.*, 236:163–183, 2009.

[16] Bastiaan Heeren. *Top Quality Type Error Messages.* PhD thesis, Universiteit Utrecht, The Netherlands, September 2005.

[17] Bastiaan Heeren and Jurriaan Hage. Type class directives. In Manuel Hermenegildo and Daniel Cabeza, editors, *Practical Aspects of Declarative Languages, 7th International Symposium, PADL 2005, Long Beach, CA, USA, January 10-11, 2005, Proceedings*, volume 3350 of *Lecture Notes in Computer Science*, pages 253–267. Springer, 2005.

[18] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Constraint based type inferencing in Helium. *Immediate Applications of Constraint Programming (ACP)*, page 57, 2003.

[19] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Scripting the type inference process. *SIGPLAN Notices*, 38(9):3–13, 2003.

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

[21] Paul Hudak. Building domain-specific embedded languages. *ACM Computing Surveys (CSUR)*, 28(4es):196, 1996.

[22] Suresh Jagannathan and Peter Sewell, editors. *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*. ACM, 2014.

[23] Mark P. Jones. A theory of qualified types. *Sci. Comput. Program.*, 22(3):231–256, 1994.

[24] Simon Peyton Jones, Mark Jones, and Erik Meijer. Type classes: exploring the design space. In *Haskell workshop*, volume 1997, 1997.

[25] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Trans. Program. Lang. Syst.*, 20(4):707–723, 1998.

[26] Daan Leijen. Extensible records with scoped labels. In Marko van Eekelen, editor, *Revised Selected Papers from the Sixth Symposium on Trends in Functional Programming, TFP 2005, Tallinn, Estonia, 23-24 September 2005.*, volume 6 of *Trends in Functional Programming*, pages 179–194. Intellect, 2005.

[27] Bruce McAdam. On the unification of substitutions in type interfaces. In Kevin Hammond, Antony Davie, and Chris Clack, editors, *Implementation of Functional Languages, 10th International Workshop, IFL'98, London, UK, September 9-11, Selected Papers*, volume 1595 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 1998.

[28] Robin Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17(3):348–375, 1978.

[29] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *TAPOS*, 5(1):35–55, 1999.

[30] Zvonimir Pavlinovic, Tim King, and Thomas Wies. Finding minimum type error sources. In Andrew Black and Todd Millstein, editors, *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages & Applications, OOPSLA 2014, part of SPLASH 2014, Portland, OR, USA, October 20-24, 2014*, pages 525–542. ACM, 2014.

[31] John Peterson and Mark P. Jones. Implementing type classes. In Robert Cartwright, editor, *Proceedings of the ACM SIGPLAN'93 Conference on Programming Language Design and Implementation (PLDI), Albuquerque, New Mexico, USA, June 23-25, 1993*, pages 227–236. ACM, 1993.

[32] François Pottier and Didier Rémy. The essence of ML type inference. In Benjamin Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.

[33] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. Skalpel: A type error slicer for standard ML. *Electr. Notes Theor. Comput. Sci.*, 312:197–213, 2015.

[34] Alejandro Serrano and Jurriaan Hage. Type error diagnosis for embedded dsls by two-stage specialized type rules. In Peter Thiemann, editor, *Programming Languages and Systems - 25th European Symposium on Programming, ESOP 2016, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2016, Eindhoven, The Netherlands, April 2-8, 2016, Proceedings*, volume 9632 of *Lecture Notes in Computer Science*, pages 672–698. Springer, 2016.

[35] Kanae Tsushima and Kenichi Asai. An embedded type debugger. In Ralf Hinze, editor, *Implementation and Application of Functional Languages - 24th International Symposium, IFL 2012, Oxford, UK, August 30 - September 1, 2012, Revised Selected Papers*, volume 8241 of *Lecture Notes in Computer Science*, pages 190–206. Springer, 2012.

[36] Kanae Tsushima and Olaf Chitil. Enumerating counter-factual type error messages with an existing type checker. *12th Asian Symposium on Programming Languages and Systems, APLAS, Singapore, November 17-19*, 2014.

[37] Atze van der Ploeg and Koen Claessen. Practical principled FRP: forget the past, change the future, FRPNow! In Kathleen Fisher and John Reppy, editors, *Proceedings of the 20th ACM SIGPLAN International Conference on Functional Programming, ICFP 2015, Vancouver, BC, Canada, September 1-3, 2015*, pages 302–314. ACM, 2015.

[38] Philip Wadler and Stephen Blott. How to make ad-hoc polymorphism less ad-hoc. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages, Austin, Texas, USA, January 11-13, 1989*, pages 60–76. ACM Press, 1989.

[39] Mitchell Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages, St. Petersburg Beach, Florida, USA, January 1986*, pages 38–43. ACM Press, 1986.

[40] Jun Yang. Explaining type errors by finding the source of a type conflict. In Philip Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Selected papers from the 1st Scottish Functional Programming Workshop (SFP99), University of Stirling, Bridge of Allan, Scotland, August 29th to September 1st, 1999*, volume 1 of *Trends in Functional Programming*, pages 59–67. Intellect, 1999.

[41] Jun Yang, Greg Michaelson, Phil Trinder, and Joe Wells. Improved type error reporting. In *In Proceedings of 12th International Workshop on Implementation of Functional Languages*, pages 71–86, 2000.

[42] Danfeng Zhang and Andrew Myers. Toward general diagnosis of static errors. In Jagannathan and Sewell [22], pages 569–582.

[43] Danfeng Zhang, Andrew Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. Diagnosing type errors with class. In David Grove and Steve Blackburn, editors, *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, OR, USA, June 15-17, 2015*, pages 12–21. ACM, 2015.