Type Directives and Type Graphs in Elm

Falco Peijnenburg
Department of Information and Computing Sciences, Utrecht University
Utrecht, The Netherlands
fpeijnenburg@gmail.com

Jurriaan Hage
Department of Information and Computing Sciences, Utrecht University
Utrecht, The Netherlands
J.Hage@uu.nl

Alejandro Serrano
Department of Information and Computing Sciences, Utrecht University
Utrecht, The Netherlands
A.SerranoMena@uu.nl

ABSTRACT

We introduce type graphs into Elm in order to improve type error messages for infinite types, and integrate type qualifiers (for type classes a la Haskell) and Elm's row polymorphism into type graphs. We also discuss how specialized type rules and siblings can be used to achieve domain-specific type error diagnosis in the context of Elm.

CCS CONCEPTS

• **Software and its engineering** → **Functional languages**; **Compilers**; *Domain specific languages*; • **Theory of computation** → *Program analysis*;

KEYWORDS

type error diagnosis, Elm, embedded domain specific languages, type classes, type graphs

ACM Reference format:
Falco Peijnenburg, Jurriaan Hage, and Alejandro Serrano. 2017. Type Directives and Type Graphs in Elm. In *Proceedings of IFL Conference, Leuven, Belgium, August 31-September 2, 2016 (IFL 2016),* 12 pages.
DOI: http://dx.doi.org/10.1145/3064899.3064907

1 INTRODUCTION

Type error messages for strongly typed functional programming languages such as OCaml and Haskell can at times be daunting. Type errors displayed by compilers often reason about the types of various expressions of the program. These types can become rather involved, rendering the error messages harder to read. And type errors occur quite often: Hage and Keeken have observed a steady 30 percent of all compiles resulting in a type error for students learning Haskell [13]. Elm is, like OCaml and Haskell, a strongly typed functional programming language, though despite this, Elm [4] 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. This paper provides the basis for further improvements, working from two directions. First, we use type graphs to investigate the context of a type error more thoroughly before throwing an error. Type graphs can represent inconsistent sets of constraints, allowing for the definition of heuristics that inspect the type graph to look for well-known error patterns. Contributions of this paper include the extension of the type graphs of [10] to represent type class predicates within the type graph, provide support for Elm's row types, and improve error diagnosis for so-called "infinite type errors" in Elm. Elm itself does not support type classes, because of the adverse effect on error diagnosis. One goal of our work was to show the Elm designers that with our methods for error diagnosis, this need not be an issue. We have re-implemented the most important heuristics of the Helium compiler [12] to show that our type graphs work as expected. Evidence for the fact that type graphs and the associated heurstics can improve error diagnosis can be found in [2, 10]. In comparison to Helium we run less risk of reducing the quality of the provided error diagnosis, since our implementation in the Elm compiler typically still reports the same error location as before and we only add a hint to the message to help the programmer resolve the type error. For our improvements to the diagnosis of infinite type errors we may in fact change the original type error message, and the paper provides anecdotal evidence for the improved quality of our messages. Second, we have implemented mechanisms (inspired by [17]) to let experienced library programmers take control of type error messages to make the error messages domain-specific. This support is crucial for Embedded Domain Specific Language (EDSLs) [19]. 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 encoding of the DSL into the host language leaks in type errors which quickly become impossible to understand by the domain programmer. The mechanisms we provide help overcome this problem.

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 contributions of this paper. The 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.

IFL 2016, Leuven, Belgium

978-1-4503-4767-9/16/08…$15.00
DOI: http://dx.doi.org/10.1145/3064899.3064907

```
transparentPink =
    rgb 255 20 150 0.5
```

(a) An inconspicuous looking Elm statement

```
Function `rgb` is expecting 3 arguments,
  but was given 4.

67|       rgb 255 20 150 0.5
                         ^^^
Did you mean Color.rgba instead of Color.rgb?
```

(b) The resulting error message

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

```
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.

Error messages that suggest corrections based on conceptually similar functions can be very helpful to programmers. The improvements do not stop there, however. Type directives give authors of functions a high degree of control over the hints added to the error messages that are shown when their functions are not used correctly. This makes it possible for error messages to explain the problem as though they understand the meaning of the function. One example of such an explanation can be seen in figure 2. The expression that caused this error is

```
if const "str" True then "foo" else "bar".
```

This paper expands on earlier research done by Hage and Heeren [10, 16, 17], translating ideas implemented in the Helium compiler [12, 18] to the setting of Elm. The Helium compiler uses a graph-based technique in combination with heuristics to find the best explanation for error messages. This is part of a continuing effort to improve type error messages.

A manifesto for good type error diagnosis was written by Yang et al [35]. According to them, 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.

The contributions we provide in this paper are that we extend the type graphs of Heeren [10] with type class predicates and row polymorphism, and improve Elm's error diagnosis in the presence of infinite types. Our work confirms the usefulness of the type graph abstraction for controlling and improving type error diagnosis.

In addition, we offer Elm-inspired programmer friendly syntax for type classes, instances, siblings and type directives, in which we deviate in subtle ways from Helium/Haskell. In particular, our implementation of siblings decouples the decision of blaming a particular location from the hint that replacing a function by a sibling can solve the problem. Our implementation is available at https://github.com/FPtje/elm-compiler, on the master branch. We often omit code fragments due to space restrictions. These can be found in [26].

The usefulness of our techniques goes beyond Elm: some languages, e.g., OCaml and Purescript, provide records similar to Elm's row types and for which our type graphs can be employed. Other languages, e.g., Erlang and F#, support a simpler form of record type. In most, if not all, functional languages infinite type errors may occur, and our work can help to improve errors in those languages as well. Even though the Helium compiler uses type graphs to provide better error diagnosis, it has no special support for infinite type errors as the type graphs in this paper do.

2 A LITTLE ON ELM

Evan Czaplicki and Stephen Chong describe the essence of the Elm programming language [6]. Some design goals of Elm are to be simple, easy to learn, purely functional and to have a clean syntax. Elm is a purely functional language designed for Functional Reactive Programming. It is focussed on the design of user interfaces, strict in evaluation, and event driven.

To correctly position our work, and the contribution in terms of an implementation, we shortly discuss the Elm ecosystem. The Elm toolset consists of several packages, all of which are written in Haskell. The source is hosted on GitHub (https://github.com/elm-lang). The most noteworthy packages, for our purposes, are `elm-compiler`, `elm-repl` (Elm's GHCi), `core` (Elm's Prelude), and `error-message-catalog` is a regression test set of program designed to trigger exotic error messages in Elm.

The `elm-compiler` deals with the compilation of a single module. At the highest level, it provides two features: (1) compiling a module from source code to a type checked and optimized module and (2) take an optimized module and generates Javascript code.

Type inference is implemented in two phases: constraint generation, and constraint solving. The former works in a top-down fashion, collecting constraints much like those of [28]. The constraint solver takes these constraints and reconstructs the most general types. At this time, errors and warnings can be thrown. The compiling process halts immediately or at the end of the current phase when errors are discovered. 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. More about Elm can be read in Czaplicki's thesis [4].

3 ELM TYPE GRAPHS

Type graphs have been described in great detail in chapter 7 of Heeren's PhD thesis [14]. Like Helium, Elm is based on the Hindley-Milner type system, so we can largely follow the same principles.

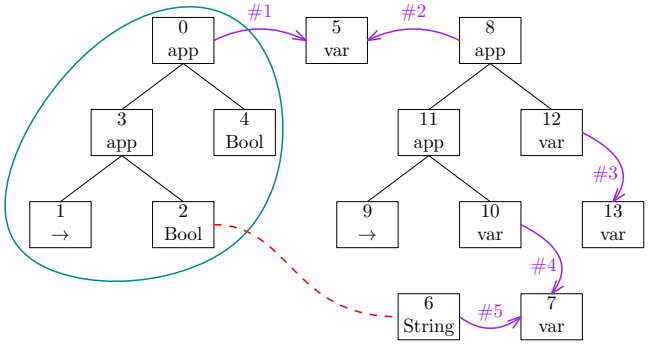


Figure 3: The type graph of `not "foo"`

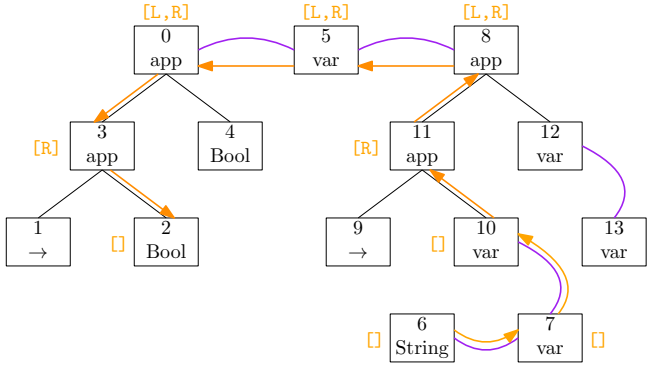


Figure 4: BFS in action

After a short summary of Helium's type graphs, we turn to our enhancements for Elm.

Consider figure 3. 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. In figure 3, constraint #1 says that node 5 must be an instance of the type represented by node 0. Similarly, #2 states that node 8 must represent a function with the same arity as node 5, #3 represents the return type of the function, #4 the first argument of the function, and #5 that a String literal was passed in.

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 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.

We compute the (shortest) error path between two vertices by a breadth-first search, keeping track of a list of successes (see [26], p.31 for the code). In every iteration, paths in the list of successes are expanded. If any of these reaches the destination, then the algorithm stops and returns the paths that have been found to take us from goal to destination. The algorithm is shown in action in figure 4. 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, e.g., 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 4, nodes 0 and 8 must be equal. This equality trickles down to the children: nodes 3 and 11 must be equal, etc.

There is one essential caveat: when going up the tree, one must remember to at one point go back down the tree, and in the "same way". In the example, when we go up the right edge of the tree from node 10 to node 11, this implies a promise that at some point we have take the corresponding step down; we fulfil this promise going from 3 to 2. Similarly, the edge from 11 to 8 is paired with the one from 0 to 3. The information about taking left and right child edges is kept track of by the algorithm (the lists displayed in orange at the nodes), taking care that we only look at pairs of equal elements that also have equal lists associated with them.

A limitation of the algorithm is that it only finds the shortest path. In theory, the situation drawn in figure 5 could occur, where two paths lead to the same conflict. In this example, only the edge for constraint #1 would be identified as an error path. One can

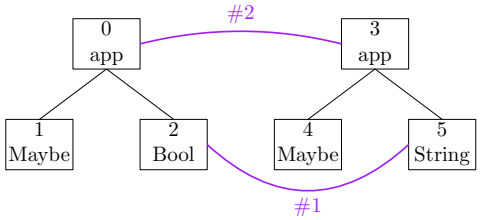


Figure 5: BFS limitation. Only constraint #1 would be identified.

change the algorithm to continue searching after a path has been found, making sure not to not get stuck in a cycle. This however becomes quite complicated in the presence of infinite types, so we have not explored this possibility.

To assure ourselves that our implementation of type graphs is sound, we implemented the heuristics "Share in error paths", "Constraint number" and "Trust factor", and verified that they behaved as expected. A description of these heuristics can be found in [10].

How type graphs are used inside the compiler. The type graphs in Elm form an alternative to the built-in solver. However, they are quite a bit more expensive to compute with. Fortunately, 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, and it will only do so for the binding group in which the inconsistency was detected. This keeps the performance overhead quite low.

In contrast to Helium's type inferencer, 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.

4 DEALING WITH INFINITE TYPES

Elm's built-in constraint solver already has a mechanism which attempts to reconstruct infinite types, but the destructive-updating nature of the built-in solver strongly hinders its ability to perform a proper reconstruction. Type graphs do not suffer from this hindrance.

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 it has found an infinite type and

Code with infinite type	Elm	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

Figure 6: Reconstructed types, Elm's built-in constraint solver versus type graphs

replaces the structure with a type variable named "∞". 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.

Type graphs do not suffer from destructive updates. Their ability to represent types and their relations in spite of both type conflicts and infinite types render them ideal for infinite type reconstruction. The native inferencer falls back on "?" only because the actual type has been lost in destructive unification. In type graphs this information is not lost. As such, they need never fall back on "?". The algorithm to reconstruct an infinite type using the type graph is based on Heeren's `substituteVariable` function [14], being inspired Elm's technique of replacing seen type variables with "∞".

During reconstruction (we omit the code for reasons of space, but see Figure 6.5 of [26]) we keep track of the visited vertices. When a vertex is revisited, ∞ is filled in. 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. 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.

Type variables that are part of a type conflict also generate a uniquely named type variable. One may wonder whether this does not defeat the purpose of reconstructing types involved in type conflicts. The reason why it does not lies in the fact that type applications and constructors are treated as though they live in a coherent equivalence group. Reconstructing node 2 in figure 3, for example (also shown in 4), 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 in Elm's built-in solver. Reconstructing infinite types works quite a bit better, as Table 6 shows.

A more elaborate example is the Y combinator that cannot be typed within Elm. In Figure 7 we have included one of the two type error messages from our compiler and the corresponding message

given by the standard Elm compiler. We omit the message for the other type incorrect application of `x` to `x` for reasons of space. Note that again our implementation reveals more information about the identifier `x`, *and* points to the application as the cause for the error. The standard compiler refers only to `x`'s binding site.

```
I am inferring a weird self-referential type for `x`

7| fix f g = (\x -> \a -> f (x x) a) (\x -> \a -> f (x x) a) g
                                                  ^
Here is my best effort at writing down the type. You will see
? and ∞ for parts of the type that repeat something already
printed out infinitely.

    ∞ -> a -> b
...
```

(a) Provided by our compiler.

```
I am inferring a weird self-referential type for `x`

7| fix f g = (\x -> \a -> f (x x) a) (\x -> \a -> f (x x) a) g
                                    ^
Here is my best effort at writing down the type. You will see
? and ∞ for parts of the type that repeat something already
printed out infinitely.

    ?
...
```

(b) Provided by the standard Elm compiler.

Figure 7: Messages for the Y combinator for our own compiler and the standard Elm compiler.

5 ADDING ROW TYPES

Elm features a simple form of the extensible records described by Leijen [22]. Extensible records contain values that can be indexed by a member name. New members and values can be added to the records, values of existing members 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 members anymore. These were supported until version 0.16 [5]. The features were removed to allow for more program optimisations and because it encouraged overly complex code.

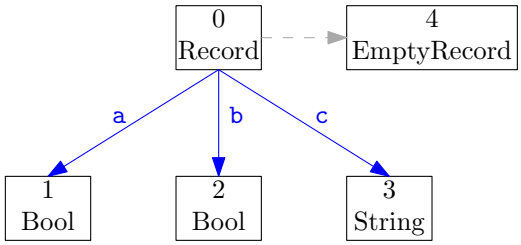
Unlike functions and algebraic data types, records cannot be described in terms of type constants, type applications and type variables. The biggest reason for this is the polymorphism that allows functions to merely demand the existence of specific members in the records provided as arguments. This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

Figure 8 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

```
someRecord =
    { a = True
    , c = "str"
    , b = False
    }
```

(a) A simple record constant.



(b) Its type graph

Figure 8: A simple record constant and its graph representation.

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 8 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 8, 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 9, 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 former as polymorphic as the record it extends.

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

Figure 9: An instance of row polymorphism.

5.1 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 ordinary type. The rules of type constants apply as usual to records. `Bool` types, for example, cannot be unified with records and should therefore still be marked as incompatible.

Additional logic, however, is needed to compare a pair of records. 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. 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, which means that polymorphic records always match.

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 10 demonstrates a situation where two records are linked by a constraint.

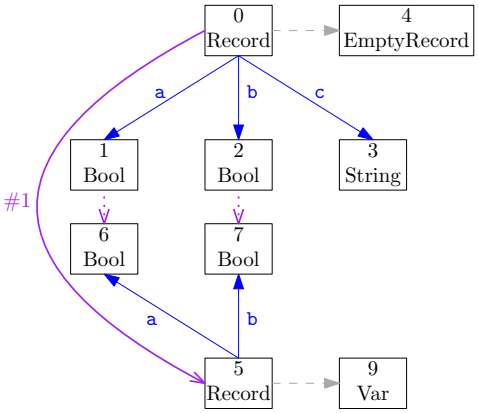


Figure 10: A monomorphic record constrained to a polymorphic one.

The record at node 0 is the same as the record in figure 8. 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 into 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 detrimental 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. We believe this effect can be countered by holding a reference back to the original records in the duplicated constraints and making sure to display the original records, rather than their members, in the type error. At this time our implementation does not do so.

6 SIBLINGS

Siblings were introduced in [17] to provide additional hints in type error messages that involve identifiers, e.g., `foldr`, that are easily confused with its "sibling", e.g., `foldl`. If substituting the type of the sibling for the identifier resolves a type error, the hint is attached to the default message.

Siblings can be introduced by the programmer, typically the implementor of the library that defines the siblings. In Elm we employ the following syntax:

```
sibling foldr resembles foldl
```

This means that `foldl` will be tried whenever an expression containing `foldr` is type incorrect, but not the other way aroundl; for that another sibling directive will be needed. The reason we have chosen for the non-symmetric interpretation is to allow a different error messages to be attached for each case.

Once the compiler is informed of all siblings, its task becomes deciding when it is useful to show a hint. There are three important factors here: (1) the function is involved in a type error, (2) its sibling function solves this type error, and (3) the sibling does not cause new type errors.

The implementation of siblings is easily done directly on the type graphs. In Figure 11 we provide a piece of Elm code, and in Figure 12 we depict the associated type graph. The type graph suggests how the type of `foo` is replaced by `bar` to see if that resolves the type error. This is done by redirecting the edge labelled with #1 to the type graph representation of the type of `bar`. 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.

Contrary to Heeren's implementation of siblings [14], siblings do *not* decide which constraint gets the blame. In Heeren's implementation, the instance constraint (constraint #1) would have been blamed, since redirecting it led to a consistent type graph. This constraint holds an error message that says the function `foo` is

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

sibling foo resembles bar

baz = foo "15"
```

Figure 11: Elm code with siblings directive

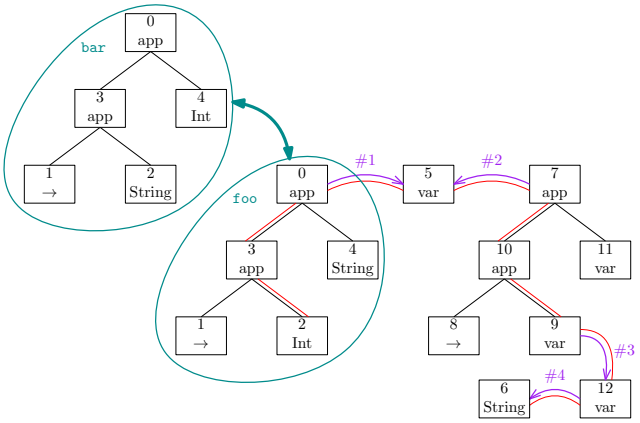


Figure 12: Siblings in action.

```
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 13: The error shown for the code shown in figure 11

somehow used incorrectly. Our 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. The error in Figure 13 is displayed.

7 TYPE CLASSES IN ELM

We have added Haskell 98 type classes, using an alternative, novice friendly syntax, to the Elm compiler, in order to see whether they would affect type error diagnosis. We have extended type graphs to include type class predicates. In Heeren's work these are dealt with outside of type graphs, but this prevents the introduction of heuristics that employ knowledge of instances to decide where to lay the blame.

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

type Foo = Foo

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

Figure 14: Sample code

Technically, extending type graphs to deal with type class predicates is rather simple: we label vertices in the type graph with any predicates that should hold for the content of the vertex. So, `Eq a` is represented by a vertex that contains the type variable `a`, to which we then attach a label that says that `Eq` should hold for that vertex. Clearly, a vertex should be able to have multiple labels.

We have implemented an algorithm to look for missing implementations. For type variables, it simply collects all the predicates that should hold, while for non-variables, types like `Int`, it will check whether matching implementations can be found. This check is applied to all binding groups in the graph. The advantage of such an approach is that looking for implementations and solving unification constraints become relatively independent: we can look for missing implementations even if the unification constraints turn out to be inconsistent, and can potentially use this information to drive heuristics to use this information when choosing a constraint to blame for the inconsistency. The set-up allows us to generate unification errors instead, instead of missing instances errors, but also vice versa.

Elm's built-in solver, on the other hand, 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.

Assuming a definition of an `Eq` type class that supports a member function `equals` to test for equality, Figure 14 defines first a function `foo` that has the type `foo : a -> a | Eq a` (equivalent to Haskell's `Eq a => a -> a`). Underneath, `Foo` is defined as an algebraic data structure with just one nullary constructor. The (modified) built-in inferencer throws two errors: the ones seen in figure 15b and 15c. The type graph only throws one error, which is the one shown in 15a. Since the two errors computed without type graphs have the same root cause, `Foo` is given as an argument to `foo`, it can be argued that the error thrown by the type graph is more appropriate.

8 SPECIALIZED TYPE RULES

Our implementation also supports the specialized type rules of Helium [17]. *Specialized type rules* allow the creation of custom (=domain-specific) hints for error messages shown when their

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

320| bar = if foo Foo then "bas" else ""

Based on its definition, `foo` has this type:
    a -> a | Main.Eq a
But you are trying to use it as:
    Foo -> Bool
```

(a) Error thrown based on 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
```

(b) 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?
```

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

Figure 15: Code with both a type conflict and a missing implementation.

functions are misused. Like in Helium the rules are automatically checked for soundness and completeness with respect to the built-in type system. Besides a more friendly syntax, in the spirit of Elm, they are very much like those in Helium.

Figure 16 shows a specialized type rule for the function `checkMaybe` when applied to two arguments. Here, `constrain` rules recursively constrain either one of the arguments or the return type. When left out, they are inserted automatically during the validation phase in the compiler pipeline. The `unify` rules create equality constraints. Note the use of subscripts here to refer to what are at first different "uses" of `a`. Essentially, the type of `checkMaybe` should be read as `Maybe a_1 -> a_2 -> Bool | Eq a` (type variables in class predicates need not be numbered). The numbering is used by the compiler for reasons discussed below. Finally, `check` rules enforce type class 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. For brevity, we have kept these messages simple.

```
1  checkMaybe : Maybe a -> a -> Bool | Eq a
2  errors for checkMaybe maybe val where
3    constrain maybe
4    constrain val
5
6    unify maybe with Maybe a_1
7      because The first argument has to be a Maybe.
8
9    unify val with a_2
10
11   unify a_2 with a_1
12     because The second argument must match the
13             thing in the Maybe.
14
15   check Eq a_1
16     because Eq is needed to test equality.
17
18   unify return with Bool
19
20   constrain return
21
22 checkMaybe maybe val =
23    case maybe of
24      Nothing -> False
25      Just x  -> isis x val
```

Figure 16: A function definition with type rules.

We note that writing specialized type rules may not be as simple as writing code, but remember that these are not written by the end users of libraries but by those who either develop the library, or they may be added to the libraries by some third party. By writing these rules once, many end users of the library may profit.

8.1 Implementing specialized type rules

The magic of specialized type rules happens in the constraint generation phase of the compiler. Constraint generation works in a top-down fashion. When the constraint generator comes across a function application, the compiler checks if there are any type rules that match the function application (with the right number of arguments). If such a type rule exist, then the constraints defined by that type rule replace the constraints that would otherwise be generated by the compiler (the necessary code is omitted for reasons of space, but see p. 50 of [26]).

A `unify` rule, e.g., `unify val with a_2`, leads to an equality constraint. First it instantiates the types of both the left hand side (lhs) and right hand side (rhs). This instantiation uses a state monad to make sure that already instantiated types are re-used. A function `decideErrorMessage` generates an appropriate error message to show to the user when the constraint is broken. Similarly, a `check`

rule generates a constraint. A rule like `check Eq a_1` can be viewed as syntactic sugar for `unify a_1 with b | Eq b`, where b is fresh.

The function `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 16, `Maybe a -> a -> Bool | Eq a` of the type of the function `checkMaybe`. 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. 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; this is clear from the type annotation. Type rules can also contain references to the arguments to the function, and its return value. In figure 16, these are `maybe`, `val` and the special identifier `return`. Finally, fresh type variables can be used at will. The role these fresh variables play is implicit in how the rules relate them to the numbered variables, the named parameters and `return`. A simple fixed point algorithm verifies that all mentioned fresh variable will related in such a way, and to which variables they are related.

Figure 17 shows the results of this effort for `checkMaybe` function defined in figure 16, albeit in a somewhat trivial setting. When the solving fails when looking at `unify a_2 with a_1`, the algorithm can discover that the types we are comparing originate from the first and second parameter. This information is used in the error message to give precise location information. It also shows the hint provided by the author of `checkMaybe` at the bottom of the error message.

```
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 17: A type error in the use of a function based on a specialized type rules

8.2 Validating type rules

Specialized type rules can be very powerful for adding 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 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. Fortunately, we can automatically validate the type rules.

Before validation, we first check there are no 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 actual part of validation requires that all parameters of the function (in figure 16, `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 were 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 16, can be seen in figure 18. In the box on the left we can read the type of the type annotation, split up into the types of parameters and the 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.

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

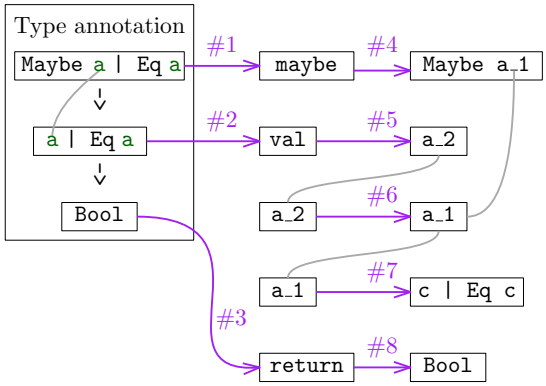


Figure 18: The first set of constraints used to validate type rules.

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

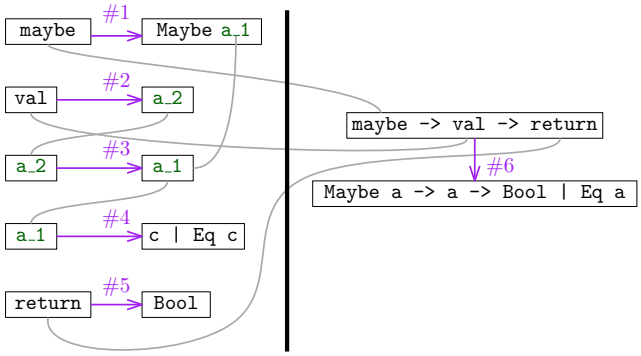


Figure 20: The second set of constraints used to validate type rules.

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 16:

```
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 19. 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 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 20.

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 16. This would remove constraint #4 in figure 20.

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 21.

```
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 21: An error shown when the type rules and the type annotation describe different types.

9 RELATED WORK

Type classes were introduced by Philip Wadler and Stephen Blott [32]. Wadler and Blott argue for type classes to be the standard solution for ad-hoc polymorphism. Type classes, and extensible records as well, are instances of the general theory of qualified types [20].

The type systems of functional languages are typically based on the Hindley-Milner type system for the polymorphic λ-calculus, implemented as a variant of algorithm W [7]. It unifies types on the go to find the most general types of expressions. Lee and Yi [21] proved that the folklore algorithm M generally finds type errors before algorithm W does, and that combining the two algorithms can give a strictly more precise location of the error.

An issue with traditional inference algorithms is the inherent left-to-right bias. McAdam [23] removes bias by treating subexpressions symmetrically. Yang [34] describes two algorithms U_{AE} and IEI. The first removes left-to-right bias by typing subexpressions independently. Like W, U_{AE} finds errors only in applications. When this happens, a switch is made to algorithm M to narrow down the error locations. This combination is called algorithm IEI.

A type inferencing algorithm (1) finds out how subexpressions are related, and (2) finds a way to assign types such that all these "relations" are respected. Part (1) corresponds to the generation of constraints to be satisfied for the program to be type correct, while solving implements part (2). The separation of the parts allows us to manipulate the order of solving constraints, or to consider multiple constraints at once, reducing the bias that any variant of W will suffer from. Various implementations of this idea exist [11, 24, 28].

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 is described in the TOP framework [16]. 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 [10]. A downside is that generating type graphs can be quite expensive.

Another use for type graphs is described by Zhang and Myers [36], who outline a method of analysing both satisfiable and unsatisfiable constraints in a type graph using context-free grammars for a large subset of ML. It uses a Bayesian algorithm to rank the most likely explanation of an error. Sadly, the graph structure is quite expensive to calculate. A continuation of this work [37] supports some of the more complicated structures of Haskell (like GADTs and type families) and is capable of counterfactual reasoning. The performance does seem to go down quickly as the lines of code increase.

Type directives in the form of specialized type rules and siblings were introduced in [17] and implemented into the Helium compiler [12, 18]. A later extension addressed type classes [15]. A coherent story is provide in [14].

Recently, work on type directives has begun to address Haskell's many type level extensions [9]. Using GHC's `OutsideIn(X)` framework as the basis, [30] provides the ability to to define advanced conditions on when a specialised type rule should apply by means of a two-stage type checker.

In a completely different approach, Christiansen [3] defined post-processing directives for the dependently typed Idris by means of reflection. It allows library writers to rewrite errors before they are shown to the end user. Domain specific type error diagnosis is also achieved in the work of Hubert Plocziniczak [27] for the Scala language. The idea is to post-process by programmatically exploring the typing derivation constructed by the Scala compiler. In that way, common error patterns can be detected.

GHC version 8 introduces a completely different kind of type directives [1]. Specifically designed for type level programming, a special TypeError 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.

Wand [33] devised a method to explain *why* the compiler comes to the conclusion that there is a type error by keeping track of the reasons for unifying type variables. Once a type conflict arises, both incompatible sides of the conflict will have a list of reasons as to why it was inferred.

Chen and Erwig [2] coined the term counterfactual typing to describe variational types for a simple lambda calculus. Variational types represent the type of an expression not as a single decision, but as a choice between types. A very different approach for counterfactual typing is outlined by [31], which uses the type checking algorithm as a black box. When an error occurs, subexpressions are replaced by 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 displayed as "expected type".

The focus of error slicing lies in reporting all parts in a program that may be responsible for a type conflict (called a "slice"). Such a slice then excludes all parts for which no change can fix the type error. Haack and Wells [8] introduced this concept for the Hindley-Milner type system. A continuation of this work, (Rahli et al. [29]) 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 generally there tend to be many locations at which a type error can be fixed.

Pavlinovic et. al. [25] 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. The algorithm is expensive, but accuracy can be traded for speed.

10 CONCLUSION AND FUTURE WORK

We have shown how to add specialized type rules and siblings to Elm, adding support for type classes, row types, and improving type error diagnosis for infinite types. As future work, we can extend our specialized type rules to two phase type rules, following [30]. The designers of Elm see alternative solutions to type classes, most notable some support for rank-N polymorphism, higher kinded polymorphism or implicit arguments. It would be good to give these the same treatment, before deciding which approach to take for Elm. Hopefully, the choice for which approach to take is partly informed by the quality of type error message the choice enables.

11 ACKNOWLEDGMENTS

This work was supported by the Netherlands Organisation for Scientific Research (NWO) project on "DOMain Specific Type Error Diagnosis (DOMSTED)" (612.001.213). We thank Bastiaan Heeren and anonymous reviewers for providing feedback.

REFERENCES

[1] Lennart Augustsson. 2015. Custom Type Errors. (2015). https://ghc.haskell.org/trac/ghc/wiki/Proposal/CustomTypeErrors Accessed: 2015-11-25.

[2] Sheng Chen and Martin Erwig. 2014. Counter-factual typing for debugging type errors. In *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, USA*. 583–594. DOI: https://doi.org/10.1145/2535838.2535863

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

[4] Evan Czaplicki. 2012. Elm: Concurrent FRP for Functional GUIs. *Senior thesis, Harvard University* (2012).

[5] Evan Czaplicki. 2015. Compilers as Assistants. (nov 2015). http://elm-lang.org/blog/compilers-as-assistants http://elm-lang.org/blog/compilers-as-assistants Accessed: 2015-11-25.

[6] Evan Czaplicki and Stephen Chong. 2013. Asynchronous functional reactive programming for GUIs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, Hans-Juergen Boehm and Cormac Flanagan (Eds.). ACM, 411–422. DOI: https://doi.org/10.1145/2462156.2462161

[7] L. Damas and R. Milner. 1982. Principal type-schemes for functional programs. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages, Albuquerque, New Mexico, January 1982*. ACM Press, 207–212.

[8] Christian Haack and Joe Wells. 2004. Type error slicing in implicitly typed higher-order languages. *Sci. Comput. Program.* 50, 1-3 (2004), 189–224. DOI: https://doi.org/10.1016/j.scico.2004.01.004

[9] Jurriaan Hage. 2014. Domain specific type error diagnosis (DOMSTED). *Technical Report Series* UU-CS-2014-019 (2014).

[10] Jurriaan Hage and Bastiaan Heeren. 2006. Heuristics for Type Error Discovery and Recovery. In *Implementation and Application of Functional Languages, IFL 2006, Budapest, Hungary, Revised Selected Papers*. 199–216. DOI: https://doi.org/10.1007/978-3-540-74130-5_12

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

[12] J. Hage, B. Heeren, A. Middelkoop, and others. The Helium Compiler. (????). http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome.

[13] J. Hage and P. van Keeken. 2009. Neon: A Library for Language Usage Analysis. In *Proceedings of the 1st Conference on Software Language Engineering (SLE '08) (Lecture Notes in Computer Science)*, D. Gasevic, R. Lämmel, and E. Van Wyk (Eds.), Vol. 5452. Springer, 35 – 53. Revised selected papers.

[14] Bastiaan Heeren. 2005. *Top Quality Type Error Messages*. Ph.D. Dissertation. Universiteit Utrecht, The Netherlands. http://www.cs.uu.nl/people/bastiaan/phdthesis

[15] Bastiaan Heeren and Jurriaan Hage. 2005. Type Class Directives. In *PADL 2005, Long Beach, CA, USA, January 10-11, 2005 (LNCS)*, Manuel Hermenegildo and Daniel Cabeza (Eds.), Vol. 3350. Springer, 253–267. http://dx.doi.org/10.1007/978-3-540-30557-6_19

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

[17] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. 2003. Scripting the type inference process. *SIGPLAN Notices* 38, 9 (2003), 3–13. DOI: https://doi.org/10.1145/944746.944707

[18] B. Heeren, D. Leijen, and A. van IJzendoorn. 2003. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*. ACM Press, New York, 62 – 71.

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

[20] Mark P. Jones. 1994. A Theory of Qualified Types. *Sci. Comput. Program.* 22, 3 (1994), 231–256. DOI: https://doi.org/10.1016/0167-6423(94)00005-0

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

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

[23] Bruce McAdam. 1998. On the Unification of Substitutions in Type Interfaces. In *Implementation of Functional Languages, 10th International Workshop, IFL'98, London, UK, September 9-11, Selected Papers (LNCS)*, Kevin Hammond, Antony Davie, and Chris Clack (Eds.), Vol. 1595. Springer, 137–152. DOI: https://doi.org/10.1007/3-540-48515-5_9

[24] Martin Odersky, Martin Sulzmann, and Martin Wehr. 1999. Type Inference with Constrained Types. *TAPOS* 5, 1 (1999), 35–55.

[25] Zvonimir Pavlinovic, Tim King, and Thomas Wies. 2014. Finding minimum type error sources. In *Proceedings of OOPSLA 2014, Portland, USA*, Andrew Black and Todd Millstein (Eds.). ACM, 525–542. DOI: https://doi.org/10.1145/2660193.2660230

[26] F. Peijnenburg. Type Directives in Elm (MSc thesis). (????). http://foswiki.cs.uu.nl/foswiki/Hage/TypeDirectivesForElm.

[27] Hubert Plociniczak, Heather Miller, and Martin Odersky. 2014. Improving Human-Compiler Interaction Through Customizable Type Feedback.

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

[29] Vincent Rahli, Joe Wells, John Pirie, and Fairouz Kamareddine. 2015. Skalpel: A Type Error Slicer for Standard ML. *Electr. Notes Theor. Comput. Sci.* 312 (2015), 197–213. DOI: https://doi.org/10.1016/j.entcs.2015.04.012

[30] Alejandro Serrano and Jurriaan Hage. 2016. Type Error Diagnosis for Embedded DSLs by Two-Stage Specialized Type Rules. In *25th European Symposium on Programming, ESOP 2016, Eindhoven, the Netherlands (LNCS)*, Peter Thiemann (Ed.), Vol. 9632. Springer, 672–698. DOI: https://doi.org/10.1007/978-3-662-49498-1_26

[31] Kanae Tsushima and Olaf Chitil. 2014. 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).

[32] Philip Wadler and Stephen Blott. 1989. 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*. ACM Press, 60–76. DOI: https://doi.org/10.1145/75277.75283

[33] Mitchell Wand. 1986. 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*. ACM Press, 38–43. DOI: https://doi.org/10.1145/512644.512648

[34] Jun Yang. 1999. Explaining Type Errors by Finding the Source of a Type Conflict. In *Selected papers from the 1st Scottish Functional Programming Workshop (SFP99) (Trends in Functional Programming)*, Philip Trinder, Greg Michaelson, and Hans-Wolfgang Loidl (Eds.), Vol. 1. Intellect, 59–67.

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

[36] Danfeng Zhang and Andrew Myers. 2014. Toward general diagnosis of static errors. In *The 41st Symposium on Principles of Programming Languages, POPL '14, San Diego, USA*. 569–582. DOI: https://doi.org/10.1145/2535838.2535870

[37] Danfeng Zhang, Andrew Myers, Dimitrios Vytiniotis, and Simon Peyton Jones. 2015. Diagnosing type errors with class. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, Portland, USA*, David Grove and Steve Blackburn (Eds.). ACM, 12–21. DOI: https://doi.org/10.1145/2737924.2738009