# Typeview: A Tool for Understanding Type Errors
## Work in Progress

Olaf Chitil[1], Frank Huch[2], and Axel Simon[2]

[1] University of York, UK
`olaf@cs.york.ac.uk`
[2] Lehrstuhl für Informatik II, RWTH Aachen, Germany
`{huch,simona}@i2.informatik.rwth-aachen.de`

**Abstract.** In modern statically typed functional languages, type inference is used to determine the type of each function automatically. Whenever this fails, the compiler emits an error message that is often very complex. Sometimes the expression mentioned in the type error message is not the one that is wrong. We therefore implement an interactive tool that allows programmers to browse through the source code of their program and query the types of each expression. If a variable cannot be typed, we would like to present a set of possible types from which the user can decide which is wrong. This should help finding the origin of type errors without detailed knowledge of type inference on the user side.

## 1 Introduction

Modern all purpose languages are mostly statically typed and come with a sophisticated type system. Although a type system always increases the length of the code through type annotations and casting, the advantages of finding errors at compile time prevail. In functional languages with a Hindley/Milner based type system [Mil78] (like Haskell [JH99] or ML [MTHM97]) the burden is further alleviated through type inference which computes the types of all functions automatically. The programmer therefore no longer needs to supply any types but relies on the compiler to do the annotation.

If a program is not typeable, the compiler will mark the expression where the type inference algorithm failed. The error is often hard to understand when non trivial programs are checked and the reported position may not agree with the part of the program that is erroneous.

```
elimDoubles [] = []
elimDoubles (x:xs) = x:eD x xs
  where
  eD c (x:xs) = if x==c then eD c x else x:eD x xs
  eD c [] = []
```

This function should iterate through a list and remove adjacent duplicate entries. On sorted lists it behaves like the `nub` function in the Haskell Prelude. The locally defined function `eD` discards list elements as long as they are equal to `c`. The example is not type correct because we wrote `x` in the `then`-clause which should be `eD c xs`. Due to the fact that `x` is defined in the pattern of the function, this typing mistake becomes a type error. `eD` expects a list (of type `[a]`) and is applied to an element (of type `a`) of the same list. Loading this function into the Haskell interpreter Hugs [JR] yields the following message:

```
ERROR "NubSort.hs" (line 3): Type error in application
*** Expression     : eD x xs
*** Term           : xs
*** Type           : [a]
*** Does not match : a
*** Because        : unification would give infinite type
```

Hugs generates this error message since it tries to unify the types of `xs` in the `else` branch with `x` in the `then` branch. The Glasgow Haskell Compiler [PHH$^+$93] gives a similar message. Since the error position is wrong the programmer has to find the real error source herself. The general approach looks like this:

The user starts by examining the types of all expressions in the neighborhood of the fragment the compiler reported as wrong. While some trivial errors can occur by using constructors only, more difficult errors involve variables that are used in several places. To find out why a variable has the reported type the user has to examine the surrounding expressions for every occurrence of this identifier. She will do this for all involved identifiers until the source of the error is found. This seems to be the "natural" way of finding the cause of a type error and it should be the goal to support the user in this task by letting her query the types of all expressions in the program.

There is already some support for querying expressions in Hugs: The interpreter can display the type of an arbitrary expression that is entered on the command line. If some non typeable function uses other top-level declarations it is useful to check that the types of these declarations are correct. This is only possible if the incorrect function is commented out because Hugs can only load modules that are error free. Local declarations (introduced by `let` or `where`) cannot be accessed at all due to the name scoping of Haskell. But the query function is very useful to explain curried function application and polymorphism to students, so it seems worthwhile adopting it.

We currently implement an interactive tool named "Typeview" that is able to show the type of any identifier. The complete source code of the user program is shown and any function and variable can be selected from it, even local ones. In case the program is not type correct, it is possible to query the types of all expressions that were inferred so far, giving the user some support to find the conflict. The tool does not try to explain how a type error was derived so it is not necessary that the user knows how type inference works. We think that this approach will support programmers in resolving type errors in the "natural" way described above.

## 2 Browsing a Program

Let us start with a small demonstration about how our tool works.

The "Typeview" application window is divided into two regions: The upper shows the full unaltered source code and the lower part holds a table for displaying the types of expressions.

When the user clicks somewhere in the source code area the program will underline smallest expression that includes this position. So selecting `x` in an expression `f x y` underlines `x`. Selecting the white space between `x` and `y` underlines the whole expression `f x y`. With this mechanism it is possible to easily mark all subexpressions in a program.

Each selection can be added to the "type view" window. This is a table with a colored square and a type annotation. The added expression will be permanently underlined with the same color as the square (until the entry in the type view table is removed). By using colored underlines it is possible to look at the types of several expressions simultaneously.

Consider the program `f x = (x,x)`. After adding `f` and `(x,x)` the source code will have one line under `f` and one under `(x,x)`. The type view table will contain the following entries:



**Fig. 1.** The table with type information.

Why are there underscores in the type of `(x,x)`? To explain this, we have to elaborate on how to display the types of local polymorphic identifiers.

## 3 Displaying Types of Local Expressions

Lowercase letters in type annotations denote a type variables. The function `f` from the example above is said to be polymorph in `a`. Each time this function is used, its type variables are instantiated to concrete types. All variables in a type annotation are implicitly all quantified, so writing

```
x :: a
y :: a
```

will be interpreted as `x :: ∀ a. a` and `y :: ∀ a. a` – they do not have the same type in this example. Consider the function

```
f :: a -> a
f y =  g
  where
    g = y
```

In Haskell 98 it is not possible to annotate `g`. Writing `g :: a` is incorrect due to the implicit universal quantification, it would mean that the function returns any type. The problem is that `g` makes use of a variable that is bound outside the definition of `g` and there is no way of saying that the type variable `a` is that of function `f`.[1] To be compatible with Haskell's type notation, we do not want to quantify explicitly. We decided to prepend an underscore whenever the variable is bound outside the current type expression. Thus the type of the local function would be displayed as `g :: _a`.

The implicit quantification allows the reuse of type variables and it is standard in Haskell to name type variables `a, b,...` in every new annotation. Our tool will always use new type variable names to display polymorphic annotations. Otherwise adding the type of the function `id` (which is defined as `id x = x`) in Fig. 1 as `a -> a` would make it impossible to tell to which `a` the `_a` variable is equal to.

With this technique it is possible to display polymorphic types of local variables as well.

## 4    Gathering Type Information

Given a type correct program it is now possible to view the type of each subexpression. The challenging task is to display types of expressions after the inference algorithm failed. Only some vague type information will be available, some of it obviously conflicting. In order to annotate the syntax tree with a maximum of information we need to check for inconsistent types as late as possible. In the following lambda abstraction the variable `x` is introduced and used three times. The resulting type of each usage – which is usually not visible to the user – is written as type annotation. The variables $\tau_i$ represent internally used variables that may be replaced by expressions over concrete types and other type variables.

```
\x -> ..... x::(τ₁,τ₁) ..... x::(Int,τ₂) ..... x::(τ₃, Float)
```

Instead of merging the type information while traversing through the body of the function, we gather these information in a list. All types in this list will be merged (unified, i.e. make them equal) when the algorithm has to annotate the type of `x` in `\x ->....`

In the example above our tool would stop the inference process and display the three types of `x` together with three colored squares that refer to the locations in the program. In order to find the error source the user can exclude some of the type information in the list. This is the main mechanism in our tool to find

---

[1] There is an extention to Hugs and GHC that allows it to annotate variables in patterns that have function wide scope and are excluded by the implicit quantification. We could add such a local type signature to the source code every time we have to display the type of a local expression but we do not want to change the user's source code.

type errors. Here the user asks the tool to remove the first usage of `x` so that the tool infers the type `(Int, Float)`. Now that the remaining type information is consistent, the excluded usage of type $(\tau_1, \tau_1)$ could be the cause of the type error. The user should investigate how this type was derived by looking at the types of the adjacent expressions.

A type inference algorithm that uses some kind of heuristic to determine where the error source is would have to choose what elements to ignore from the list. In our example any subset is consistent. This demonstrates how difficult it is to find the origin of a type error automatically.

The algorithm sketched so far is that of Bernstein and Stark [BS95] except that we handle `let` bindings differently. If the type of a function can be inferred without error it is most likely that it is correct. Beyond this, a function will most probably be used more often than it is defined and each wrong usage will make the error show up at the definition of that function. Consider the following example:

```
let head (x:_) = x in
  \l -> l:head l
```

The type of the constructor `:` is `a -> [a] -> [a]` and is instantiated where it is used (The term "instantiated" means that the type variable `a` is replaced by an internal variable $\tau$). At the point of definition, `l` will be of type $[\tau]$ and `head` of type $\tau$ `->` $[\tau]$ which will lead to an error when reaching the definition of `head` because the body of the function has the type `[a] -> a`. In order to avoid these awkward error positions we decided to instantiate `let` bound identifiers where they are used.

## 5   Future Work

Up to now the tool can display all types of a program that is type correct. We are working on dealing with non typeable programs and how to specify a subset of types for a given identifier.

When a list of types for one identifier is not compatible, the whole list is shown which may be quite long. Finding the candidate which has the wrong type might be difficult due to the amount of information given. Perhaps it is sensible to automatically calculate a biggest subset of the list that has no conflicts. This is similar to the idea of Johnson and Walz [JW86] where a majority decision is taken.

Up to now we have no experiences how useful the set of type information is to find an error. In case it turns out that our approach speeds up the finding of type errors, full Haskell 98 support would be valuable. Currently our language is a simple functional language with lambda, `let` and `case` constructs.

## 6   Related Work

Much work has gone into improving type error handling. As standard inference algorithms like $\mathcal{W}$ in [LY98] use a left to right, bottom-up traversal trough the

syntax tree, they produce errors that depend on the structure of the program. They report the first inconsistent application which is often not the source of the error. To alleviate the influence of traversal order, McAdams [McA98] and Wand [Wan86] infer all branches of a node separately and unify the substitutions (the result of type inference). Other approaches collect all type information for each identifier and unify these at the point where the variable is defined [BS95, Jun99]. This is close to how our tool works. In the event of an error, this technique allows to search for a biggest unifiable subset and report all other occurrences as errors [JW86]. This improved algorithm may be implemented in a future version of Hugs [JR, Nor].

Besides generating better diagnostic messages, a couple of interactive tools have been developed to explain how type inference works and how the compiler derives the type error [BS94, Soo90, DB94]. Most of these systems suffer from the tremendous amount of information they produce. Even for simple erroneous expressions, the information generated is too overwhelming to be usable in practice. We further think that such tools are unsuitable for students who do not know how type inference works.

## 7   Conclusion

We discussed the difficulty of presenting type errors at the right location and pointed out that even sophisticated algorithms might fail and deliver incomprehensible error messages. The natural way to find the source of a type error is to take a look at all the subexpressions in the neighborhood of the error position. Our tool supports the user by letting her browse through the program, querying the type of every subexpression. We solved the problem of showing local polymorphic identifiers through prepending an underscore to the type variable name. In case of incompatible types the tool collects as much type information about an identifier as possible and presents this list to the user. We are about to implement the functionality that the user may remove the information that some usages of the identifier induced. If this leads to a type without conflicts, the excluded usages are a possible error source. Since the latter is not implemented yet, we have no experimental results. But we think our interactive approach could give better support in finding type errors than complicated heuristic algorithms.

## Acknowledgments

# References

[BS94]      M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, West Lafayette, Indiana 47907-7821, 1994. Department of Computer Sciences Purdue University.

[BS95]      Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). electronically available, November 1995.

[DB94]      Dominic Duggan and Frederick Bent. Explaining type inference. Technical Report CS-94-14, University of Waterloo, Department of Computer Science, 1994.

[JH99]      Simon Peyton Jones and John Huges. Haskell 98: A non-strict, purely functional language. Technical report, Microsoft Research Cambridge, Chalmers University of Technology, 1999.

[JR]        Mark P. Jones and Alastair Reid. Hugs - The Haskell User's Gofer System, www.haskell.org/hugs.

[Jun99]     Yang Jun. Explaining type errors by finding the sources of type conflicts. electronically available, August 1999.

[JW86]      G.F. Johnson and J.A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type-inference. In *Proc. 13th ACM Symposium on Principles of Programming Languages (POPL '86)*, pages 44–57, 1986.

[LY98]      O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. In *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 1998.

[McA98]     B. McAdam. On the unification of substitutions in type inference. In *LNCS 1595, IFL '98*, James Clerk Maxwell Building, The Kings Buildings, Mayfield Road, Edinburgh, UK, March 1998. The University of Edinburgh.

[Mil78]     Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, December 1978.

[MTHM97]    Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The Definition of Standard ML (Revised)*. MIT Press, 1997.

[Nor]       Thomas Nordin. Personal communication.

[PHH+93]    Simon L. Peyton Jones, Cordelia V. Hall, Kevin Hammond, Will Partain, and Philip Wadler. The glasgow haskell compiler: a technical overview. In *Proc. UK Joint Framework for Information Technology (JFIT) Technical Conference*, 93.

[Soo90]     H. Soosaipillai. An explanation based polymorphic type checker for standard ml. Master's thesis, HeriotWatt University, 1990.

[Wan86]     M. Wand. Finding the source of type errors. In *13th Annual ACM Symp. on Principles of Prog. Languages (POPL)*, pages 38–43, January 1986.