Compositional Explanation of Types and Algorithmic Debugging of Type Errors

Olaf Chitil
University of York, UK
olaf@cs.york.ac.uk

ABSTRACT

The type systems of most typed functional programming languages are based on the Hindley-Milner type system. A practical problem with these type systems is that it is often hard to understand why a program is not type correct or a function does not have the intended type. We suggest that at the core of this problem is the difficulty of explaining why a given expression has a certain type. The type system is not defined compositionally. We propose to explain types using a variant of the Hindley-Milner type system that defines a compositional type explanation graph of principal typings. We describe how the programmer understands types by interactive navigation through the explanation graph. Furthermore, the explanation graph can be the foundation for algorithmic debugging of type errors, that is, semi-automatic localisation of the source of a type error without even having to understand the type inference steps. We implemented a prototype of a tool to explore the usefulness of the proposed methods.

1. INTRODUCTION

The type systems of most typed functional programming languages are based on the Hindley-Milner type system [11]. It combines the unobtrusiveness of not requiring any type annotations in the program with the flexibility of polymorphism. The basic ideas of the type system are intuitive: A function can have many types. The type of a polymorphic function represents all types that can be gained by instantiation of its type variables. Every function has a principal, that is, most general, type which represents all of its types. Practical experience shows that the type checker catches many errors, from trivial oversights to sometimes even deep logical errors. But experience also shows that from a type error message it is often hard to deduce the actual cause of the error and understand it [1, 2, 3, 4, 9, 10, 17, 19, 20, 21, 22, 23, 24, 25, 26].

Consider the following tiny Haskell program [16]:

```
f xs ys = ((map toUpper) . (++)) xs ys
```

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICFP'01, September 3-5, 2001, Florence, Italy.
Copyright 2001 ACM 1-58113-415-0/01/0009 ...$5.00.

The function `map toUpper` maps a list of characters to a list of (uppercase) characters, `(++)` concatenates two lists, and the infix operator `.` composes the two functions. The programmer thinks that the composition is a function mapping two lists of characters to a list of characters. However, the Haskell system Hugs[1] gives the following error message:

```
ERROR (line 1): Type error in application
*** Expression     : (map toUpper . (++)) xs ys
*** Term           : map toUpper
*** Type           : [Char] -> [Char]
*** Does not match : ([a] -> [a]) -> b -> c
```

So the program is not typable and hence the programmer's reasoning must be faulty. But what is wrong with the subexpression `map toUpper`? Why should its type match the type `([a] -> [a]) -> b -> c`? The Glasgow Haskell compiler[2] generally gives more detailed type error messages than Hugs, but here it is not better:

```
Couldn't match `[Char]' against `[a] -> [a]'
  Expected type: [a] -> [Char]
  Inferred type: [a] -> [a] -> [a]
In the second argument of `.', namely `++'
In the right-hand side of an equation for `f':
  ((map toUpper) . ++) xs ys
```

This message reports a type conflict for a different subexpression. The function `(++)` certainly has type `[a] -> [a] -> [a]`, but why should it also have type `[a] -> [Char]`?

We analyse why type error messages of current interpreters and compilers are unsatisfactory. The meaning of current type error messages, the meaning of the reported types and their relation to the program, is not well-defined. Furthermore, the program position given in an error message is often far from the source of the error. We argue that this lack of precise error location is unavoidable for a Hindley-Milner based type system. Because of this lack of precision, the programmer must be able to explore how an unexpected type was inferred to locate the source of the error. So the main component of a type error explanation system has to be an explanation system for types. We argue that neither a Hindley-Milner type inference tree nor Milner's type checking algorithm $\mathcal{W}$ [11] are suitable for explaining types.

We claim that a type explanation must be *compositional* to be comprehensible. That is, the whole explanation must have a tree structure where the types at each node are determined uniquely by the types of the node's children. A tree of principal types is not compositional, but a tree of *principal*

[1] http://haskell.org/hugs
[2] http://haskell.org/ghc

typings is. Unfortunately the Hindley-Milner type system only has principal types, but not principal typings. However, in [12] John Mitchell defines a type checking algorithm that implicitly defines a type system which is closely related to the Hindley-Milner type system and which has what we call a principal monomorphic typing property. The type system of principal monomorphic typings is our basis. The type checking algorithm can produce a compositional type explanation tree. In fact, to avoid duplication of subtrees, we actually construct an acyclic type explanation graph.

We show how the programmer can understand types by interactive navigation through the type explanation tree. Navigation at different levels enables the programmer to avoid useless information without losing important details. Just as type inconsistencies are reported far from the actual error, run-time errors are usually observed far from their source. For declarative languages algorithmic debugging has been used successfully to locate run-time errors. The type explanation graph is a good basis for algorithmic debugging of type errors. To algorithmically locate type errors, the programmer does not even have to follow type explanations but only needs to know which types he/she intends variables and expressions to have. We implemented a prototype type explanation and debugging tool to explore our ideas. The prototype produced the examples shown in this paper.

Although we use Haskell in examples, our analysis is valid and the proposed method is applicable to all Hindley-Milner based programming languages such as ML and Clean.

In Section 2 we analyse the problems with the type error messages of current systems and argue that neither a Hindley-Milner type derivation tree nor Milner's type checking algorithm $\mathcal{W}$ are a suitable basis for explaining types and type errors. In Section 3 we informally describe our idea of compositional type explanations. In Section 4 we formalise the type system of principal monomorphic typings and in Section 5 we define the construction of the type explanation graph. In Section 6 we discuss navigation through the explanation graph and in Section 7 we demonstrate how algorithmic debugging of type errors works. In Section 8 we briefly discuss the implementation of an explanation and debugging tool. In Section 9 we discuss related work. We conclude in Section 10.

2. THE PROBLEMS

To provide better support for a programmer with an untypable program, we first have to analyse why current type error messages are unsatisfactory and what makes giving more helpful information so hard.

The Meaning of Type Error Messages. The type checkers of most systems for Hindley-Milner based programming languages report a type error in the form of an expression and two contradictory types for it. Hugs reports a type and another type which 'does not match' the first. GHC reports an 'expected' type and an 'inferred' type. It is unclear what these descriptions mean and how the types relate to the expression and each other.

The types in the error messages contain type variables. They do not mean that the given expressions are or should be polymorphic but that their types are or should be instances of the given types. The occurrence of the same type variable in several types means that this type variable has to be replaced by the same type in all these types. These type variables introduced by the type checker are so called non-generic type variables that scope over the whole program in contrast to the generic type variables occurring in type annotations in a program. These two sorts of type variables are rather confusing.

The Error Location. The following program highlights part of the problem:

```
reverse [] = []
reverse (x:xs) = reverse xs ++ x

last xs = head (reverse xs)
init = reverse . tail . reverse

rotateR xs = last xs : init xs
```

Hugs gives the following error message:

```
ERROR (line 7): Type error in application
*** Expression     : last xs : init xs
*** Term           : last xs
*** Type           : [a]
*** Does not match : a
*** Because        : unification would give
                     infinite type
```

The Glasgow Haskell compiler says:

```
Occurs check: cannot construct the infinite type:
                a = [a]
  Expected type: [[[a]]]
  Inferred type: [[a]]
In the first argument of `init', namely `xs'
In the second argument of `:', namely `init xs'
```

The two Haskell systems give different expressions at which type checking fails. Unfortunately both expressions are far away from the source of the error: The right hand side of the second equation of `reverse` must be `reverse xs ++ [x]` to define the desired function that reverses the order of the elements of a list. The example highlights a problem of polymorphism: Because of the flexibility of polymorphism an erroneous expression is often still well-typed. Fortunately, the type of the erroneous expression is usually different from the type intended for this part by the programmer. Thus the usage of the expression usually leads to a type error in its context.[3]

The `reverse` example is contrived. If we had written the intended type of `reverse` in the program, then we would have obtained a more precise error message. However, too many type annotations are practically undesirable and the Hindley-Milner type system was designed to make them unnecessary. The right-hand side of a function definition usually does not contain type annotations and is often large enough to suffer from the polymorphism problem. So, because of the sparseness of type annotations, the type checker seldom knows the intended type for an expression which it could compare with the inferred type.

We conclude that in general a type checker cannot pinpoint exactly the source of a type error but only report an inconsistency of types at some program location.

[3]The two occurrences of the word 'usually' in the two last sentences indicate that an erroneous program may be well-typed. This limitation of type systems is not the subject of this paper.

Explanation of Types. Of the types reported in an error message at least one does not agree with the type intended by the programmer. Hugs reports that in the definition of `rotateR` the expression `last xs` has type `[a]` which does not match the type `a`. The programmer intends `xs` to be a list, so the reported type `[a]` seems reasonable. But why should its type also be an instance of the type `a`? Similarly, the Glasgow Haskell compiler reports for the last occurrence of `xs` in the definition of `rotateR` the types `[[[a]]]` and `[[a]]`. The programmer actually intends `xs` to be a list of arbitrary elements. So how did the type checker obtain the types `[[[a]]]` and `[[a]]`? At the heart of a tool for explaining type errors must be a tool for explaining the types of program fragments. However, already for a well-typed expression it is rather difficult to explain its type.

The Type Inference Tree. A type system is formally defined through type rules. These define the valid type judgements. A type judgement $\Delta \vdash M :: \tau$ consists of an expression M, an environment Δ that associates each free variable of M with a type, and the type τ the expression M has in Δ. A type judgement is valid iff there exists a type inference tree for it. Figure 1 shows such a tree. (We assume the literal 3 not to be overloaded but to be just of type Int.)

The fact that we have to split the tree into subtrees to fit it onto the page underlines that it is impossible to understand the tree as a whole. We can only look at a small part at a time. We can verify the correctness of the tree by verifying that each inference step is an instance of a type rule. However, an inference step does not provide an explanation. For simplicity we assume that the types of `null` and `(:)` are globally known, but consider the tree leaves with the judgements for `xs` and `ys`. Why are both of type `[Int]`? Why not any other type and why the same?

Furthermore, the programmer may intend the expression $\lambda xs.\, \lambda ys.\, \ldots$ to have a more general type, for example the type `[a]->[Int]->([Int],[a])`. He wants to apply the expression to a list of `Char`s and a list of `Int`s. Only for the more general type this would be well-typed. However, a Hindley-Milner proof tree cannot prove that there exists no more general type.

Algorithm $\mathcal{W}$. Most type explanation systems that have been proposed are based on Milner's type checking algorithm $\mathcal{W}$. The algorithm recursively traverses an expression to determine its principal type. It implicitly constructs an inference tree. Figure 2 visualises some intermediate states of $\mathcal{W}$'s construction of an inference tree for our example. We do not discuss the details here but note that for each as yet unknown type the algorithm introduces a new type variable. For example, before state A it introduces the variables b and c. When subtrees are combined, type variables may have to be substituted. For example, to reach state F the type [Int] has to be substituted for d to make the type of the function and the type of the argument equal. So type variables scope over the whole tree that has yet been constructed and the algorithm may modify the tree that has already been constructed at any later time. Furthermore, at state C the algorithm uses the type that was already inferred for xs when traversing another subtree. These global modifications and flow of information between subtrees make it very hard to follow algorithm $\mathcal{W}$. $\mathcal{W}$ can be efficiently implemented but is not suitable for explaining types.

3. COMPOSITIONAL EXPLANATIONS

In the previous section we argued that a type explanation cannot be based on an inference tree with global dependencies or a type checking algorithm that modifies type variables with global scope. To be comprehensible, an explanation must consist of small manageable units, each of which is meaningful on its own. Hence we claim that a type explanation must be *compositional*. That is, the whole explanation must have a tree structure where the types at each node are determined uniquely by the types of the node's children. Such an inference step is a small explanation unit and only refers to the explanations of the child nodes.

Principal Typings. Let us consider the expression $f\ x\ y$. Without knowing anything else about the variables f, x and y we can infer that f must be a function which takes two arguments, the types of these arguments must equal the types of x and y, and the type of the whole expression is the result type of the function f. We can express this concisely as follows:

```
Expression: f x y
Type      : a
with   f :: b -> c -> a
       x :: b
       y :: c
```

Let us do the same for the expression null xs appearing in the example of the last section. We know that the predefined function null has type $[a] \to$ Bool. Hence we can infer

```
Expression: null xs
Type      : Bool
with  xs :: [a]
```

Similarly for the subexpression (xs, ys):

```
Expression: (xs,ys)
Type      : (a,b)
with  xs :: a
      ys :: b
```

Type variables express dependencies between types. The type of an expression and the types of its variables belong together, separately they are meaningless.

Definition 1. A type environment Δ plus a type τ is a *typing*, written $\Delta \vdash \tau$. A *type judgement* $\Delta \vdash M :: \tau$ states that M has type τ in Δ, that is, $\Delta \vdash \tau$ is a typing for the expression M.

We just inferred typings of expressions:

$f\ x\ y$	has typing	$\{f :: b \to c \to a, x :: b, y :: c\} \vdash a$
null xs	has typing	$\{xs :: [a]\} \vdash$ Bool
(xs, ys)	has typing	$\{xs :: a, ys :: b\} \vdash (a, b)$

Note, however, that the type inference tree of Figure 1 uses a different typing for the expression (xs, ys). But that typing is an instance of the typing which we inferred. We inferred the principal, that is, most general, typing for (xs, ys).

Definition 2. A typing $\Delta' \vdash \tau'$ is an *instance* of a typing $\Delta \vdash \tau$ iff there is a type substitution σ with $\Delta' = \Delta\sigma$ and $\tau' = \tau\sigma$. A *typing $\Delta \vdash \tau$ is principal for an expression M* iff it is a typing for M and all typings for M are instances of $\Delta \vdash \tau$. For comparison, a *type τ is principal for an expression M and a type environment* Δ iff M has type τ in Δ and all types τ' with $\Delta \vdash M :: \tau'$ are instances of τ.

(i)
$$\dfrac{\dfrac{\{\} \vdash \text{null} :: \forall a.\, [a] \to \text{Bool}}{\{\} \vdash \text{null} :: [\text{Int}] \to \text{Bool}} \quad \{xs :: [\text{Int}]\} \vdash xs :: [\text{Int}]}{\{xs :: [\text{Int}] \vdash \text{null}\ xs :: \text{Bool}}$$

(ii)
$$\dfrac{\{xs :: [\text{Int}]\} \vdash xs :: [\text{Int}] \quad \{ys :: [\text{Int}]\} \vdash ys :: [\text{Int}]}{\{xs :: [\text{Int}], ys :: [\text{Int}]\} \vdash (xs, ys) :: ([\text{Int}], [\text{Int}])}$$

(iii)
$$\dfrac{\dfrac{\dfrac{\{\} \vdash 3 :: \text{Int} \quad \dfrac{\{\} \vdash (:) :: \forall a.\, a \to [a] \to [a]}{\{\} \vdash (:) :: \text{Int} \to [\text{Int}] \to [\text{Int}]}}{\{\} \vdash (3 :) :: [\text{Int}] \to [\text{Int}]} \quad \{ys :: [\text{Int}]\} \vdash ys :: [\text{Int}]}{\{ys :: [\text{Int}]\} \vdash 3 : ys :: [\text{Int}]} \quad \{xs :: [\text{Int}]\} \vdash xs :: [\text{Int}]}{\{xs :: [\text{Int}], ys :: [\text{Int}]\} \vdash (3 : ys, xs) :: ([\text{Int}], [\text{Int}])}$$

$$\dfrac{\dfrac{\dfrac{\text{(i)}}{\vdots}}{\{xs :: [\text{Int}]\} \vdash \text{null}\ xs :: \text{Bool}} \quad \dfrac{\dfrac{\text{(ii)}}{\vdots}}{\{xs :: [\text{Int}], ys :: [\text{Int}]\} \vdash (xs, ys) :: ([\text{Int}], [\text{Int}])} \quad \dfrac{\dfrac{\text{(iii)}}{\vdots}}{\{xs :: [\text{Int}], ys :: [\text{Int}]\} \vdash (3 : ys, xs) :: ([\text{Int}], [\text{Int}])}}{\{xs :: [\text{Int}], ys :: [\text{Int}]\} \vdash \text{if null}\ xs\ \text{then}\ (xs, ys)\ \text{else}\ (3 : ys, xs) :: ([\text{Int}], [\text{Int}])}$$
$$\dfrac{\{xs :: [\text{Int}]\} \vdash \lambda ys.\, \text{if null}\ xs\ \text{then}\ (xs, ys)\ \text{else}\ (3 : ys, xs) :: [\text{Int}] \to ([\text{Int}], [\text{Int}])}{\{\} \vdash \lambda xs.\, \lambda ys.\, \text{if null}\ xs\ \text{then}\ (xs, ys)\ \text{else}\ (3 : ys, xs) :: [\text{Int}] \to [\text{Int}] \to ([\text{Int}], [\text{Int}])}$$

For space reasons the subtrees (i), (ii) and (iii) are displayed separately above the tree.

Figure 1: A Hindley-Milner Type Inference Tree

$$\dfrac{\{\} \vdash \text{null} :: \forall a.\, [a] \to \text{Bool}}{\{\} \vdash \text{null} :: [b] \to \text{Bool}} \quad \{xs :: c\} \vdash xs :: c$$

State A: Introduction of b and c

$$\dfrac{\dfrac{\{\} \vdash \text{null} :: \forall a.\, [a] \to \text{Bool}}{\{\} \vdash \text{null} :: [b] \to \text{Bool}} \quad \{xs :: [b]\} \vdash xs :: [b]}{\{xs :: [b]\} \vdash \text{null}\ xs :: \text{Bool}}$$

State B: Substitution $[[b]/c]$

$$\dfrac{\{xs :: [b]\} \vdash xs :: [b] \quad \{ys :: d\} \vdash ys :: d}{\{xs :: [b], ys :: d\} \vdash (xs, ys) :: ([b], d)}$$

State C: Introduction of d

$$\{\} \vdash 3 :: \text{Int} \quad \dfrac{\{\} \vdash (:) :: \forall a.\, a \to [a] \to [a]}{\{\} \vdash (:) :: e \to [e] \to [e]}$$

State D: Introduction of e

$$\dfrac{\dfrac{\{\} \vdash 3 :: \text{Int} \quad \dfrac{\{\} \vdash (:) :: \forall a.\, a \to [a] \to [a]}{\{\} \vdash (:) :: \text{Int} \to [\text{Int}] \to [\text{Int}]}}{\{\} \vdash (3 :) :: [\text{Int}] \to [\text{Int}]} \quad \{ys :: d\} \vdash ys :: d}$$

State E: Substitution $[[\text{Int}]/e]$

$$\dfrac{\{xs :: [b]\} \vdash xs :: [b] \quad \{ys :: [\text{Int}]\} \vdash ys :: [\text{Int}]}{\{xs :: [b], ys :: [\text{Int}]\} \vdash (xs, ys) :: ([b], [\text{Int}])} \qquad \dfrac{\dfrac{\{\} \vdash 3 :: \text{Int} \quad \dfrac{\{\} \vdash (:) :: \forall a.\, a \to [a] \to [a]}{\{\} \vdash (:) :: \text{Int} \to [\text{Int}] \to [\text{Int}]}}{\{\} \vdash (3 :) :: [\text{Int}] \to [\text{Int}]} \quad \{ys :: [\text{Int}]\} \vdash ys :: [\text{Int}]}{\{ys :: [\text{Int}]\} \vdash 3 : ys :: [\text{Int}]}$$

State F: Substitution $[[\text{Int}]/d]$

$$\dfrac{\dfrac{\vdots \qquad \vdots}{\{ys :: [\text{Int}]\} \vdash 3 : ys :: [\text{Int}]} \quad \{xs :: [b]\} \vdash xs :: [b]}{\{xs :: [b], ys :: [\text{Int}]\} \vdash (3 : ys, xs) :: ([\text{Int}], [b])}$$

State G

Figure 2: Some Intermediate Steps of Type Inference Tree Construction by Algorithm $\mathcal{W}$

A principal type is meaningless without a fixed type environment. In contrast, an expression determines its principal typing uniquely up to type variable renaming. Hence a principal typing is a meaningful unit of information about an expression.

An Inference Step. We have principal typings for null xs and (xs, ys) The following typing is also principal:

```
Expression: (3:ys,xs)
Type      : ([Int],a)
with  xs :: a
      ys :: [Int]
```

How do we determine from these principal typings the principal typing for if null xs then (xs, ys) else $(3 : ys, xs)$? First, we arrange the three typings in three columns side by side:

```
Expressions: null xs   (xs,ys)   (3:ys,xs)
Types      : Bool      (b,c)     ([Int],d)
with  xs     [a]       b         d
      ys               c         [Int]
```

We renamed the type variables of the second and third typing (the last two columns). The type variables express dependencies within a typing, but they are unrelated to type variables in other typings. For the if-then-else construct the type of the first argument must be Bool and the types of the second and third argument must be equal. We substitute [Int] for b, and c for d:

```
Expressions: null xs   (xs,ys)   (3:ys,xs)
Types      : Bool      ([Int],c) ([Int],c)
with  xs     [a]       [Int]     c
      ys               c         [Int]
```

Also the types of the variables xs and ys have to agree. Hence we substitute [Int] for c and Int for a:

```
Expressions: null xs  (xs,ys)       (3:ys,xs)
Types      : Bool     ([Int],[Int]) ([Int],[Int])
with  xs     [Int]    [Int]         [Int]
      ys               [Int]         [Int]
```

In short, we applied the most general substitution that gives the required type equalities. We obtain the principal typing:

```
Expression: if null xs then (xs,ys) else (3:ys,xs)
Type      : ([Int],[Int])
with  xs :: [Int]
      ys :: [Int]
```

Figure 3 shows the whole type inference tree of principal typings for our example. Type variables are local to a single principal typing. The typings at the leaves of the derivation tree are trivial and independent of the remaining tree. The conclusion of an inference step is uniquely determined by its premises. In a nutshell: the tree is compositionally defined.

A Type Error. For an untypable expression a type inference step will fail. In that case an error message can report the conflicting typings:

```
Type error in: (map toUpper) . (++)
  because
Expressions: (.) (map toUpper)        (++)
Types:       (a->[Char])->a->[Char]   [b]->[b]->[b]
```

This error message is surprisingly similar to the one given by the Glasgow Haskell compiler. However, all the information in our error message has a well-defined meaning. The function (.) (map toUpper) has the principal typing $\{\} \vdash (a \to [\text{Char}]) \to a \to [\text{Char}]$ and its argument (++) has the principal typing $\{\} \vdash [b] \to [b] \to [b]$ (here we assume the types of map, toUpper etc. as given). The underlining of types emphasises that the type of function and argument do not fit together. The remaining parts of the example, especially the arguments xs and ys, do not contribute to the error. If the programmer does not understand the principal typing for an expression, he/she can ask for more explanations as we will discuss in subsequent sections.

Polymorphic and Monomorphic Variables. Type inference for the Hindley-Milner system is in general not as easy as we have so far suggested. Consider the expression $x\ x$. According to the previous exposition it gives a type error:

```
Type error in: x x
  because
Expressions: x   x
Types      : a   b
with  x      a   b
```

The type of the first x needs to be a function, that is $c \to d$. The type of the second x must be equal to the argument type of the function. On the other hand both occurrences of x must have the same type.

However, there exists infinitely many typings for $x\ x$, for example:

$$\{x :: \forall a.a\} \vdash x\ x :: \forall a.a$$
$$\{x :: \forall a.a \to a\} \vdash x\ x :: \forall a.a \to a$$

The point is that x can be a polymorphic variable. Then it can be used at different occurrences with different types. In the previous section we actually used the polymorphic variable null but we did not consider the possibility that any of the variables we listed in the type environments of the typings is polymorphic.

The expression $x\ x$ is given in [6] as an example for an expression for which no principal typing exists in the Hindley-Milner type system. The Hindley-Milner type system only has principal types, which are not sufficient for a compositional type explanation. So how can we get around this problem? The Hindley-Milner type system clearly distinguishes between polymorphic variables and monomorphic variables. Variables defined on the top-level of a program or within a `let` are polymorphic. The type of a polymorphic variable may contain $\forall$-quantifiers and then the variable may be used with different types. All other variables, basically those representing function arguments, are monomorphic. The type of a monomorphic variable may contain type variables, but the monomorphic variable may only be used with the same type at each occurrence.

Principal Monomorphic Typings. The type environments in all our previous examples contain only monomorphic variables. We assumed that the types of the polymorphic variables and data constructors such as `null` and `(:)` were implicitly globally known. Because new polymorphic variables can be defined in a program, possibly even within a `let` with only a limited scope, we need to make the types of polymorphic variables explicit in a formal type system. We

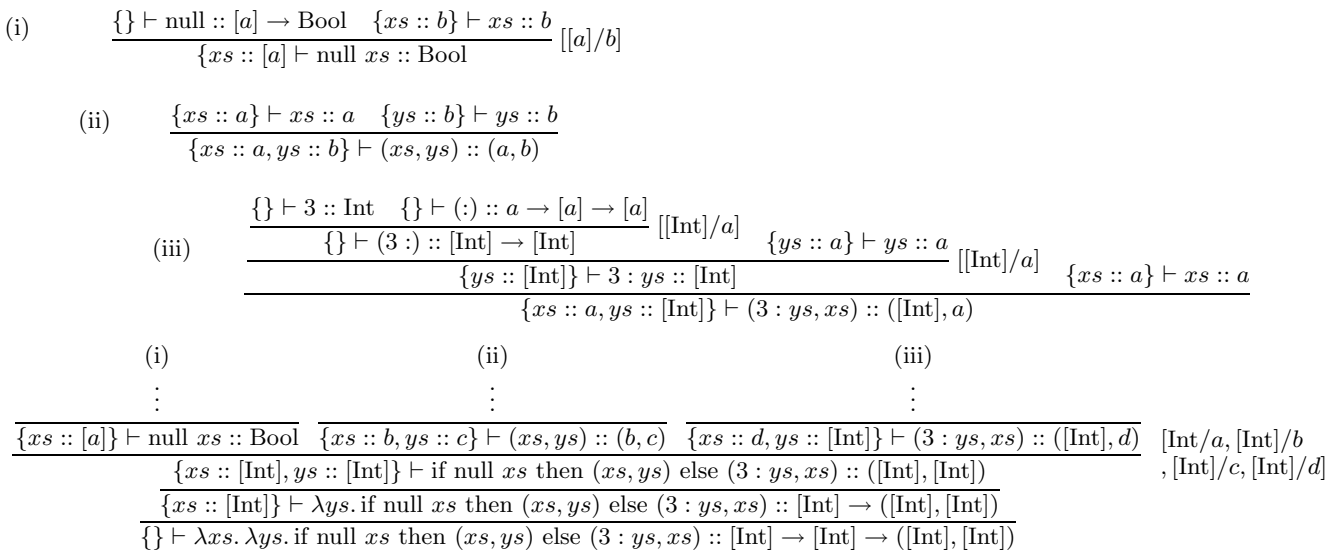


Figure 3: A Type Inference Tree of Principal Local Typings

introduce a second, separate environment for the types of polymorphic variables. The idea is that the type checker takes as input an expression together with an environment for the polymorphic variables. If the types are not inconsistent, then the type checker produces as output a type for the expression and a type environment for the monomorphic variables which occur freely in the expression. This type and type environment will be the most general of all types and type environments that give a valid type judgement, that is, it is a *principal monomorphic typing*.

4. THE TYPE SYSTEM

Following the preceding informal introduction we now define our type system of principal monomorphic typings.

The Language. Because we aim for a tool for real programs we do not use the classical λ-calculus plus `let`. Instead, our language uses the main features of functional languages. All subsequent examples are written in the language.

The syntax is given in Figure 4. For type checking purposes we can define patterns to be equal to expressions, but they shall not contain the `let` construct and in practice they will have to meet more restrictions. An equation consists of a left-hand side expression and a right-hand side expression, where the defined variable appears first on the left-hand side. A definition consists of one or more equations for the same variable. A program is a sequence of definitions. A definition may be recursive, but for simplicity we do not allow mutual recursion. Hence a polymorphic variable can only be used in its own and subsequent definitions.

$\forall$ is Superfluous. The Hindley-Milner type system uses the $\forall$-quantifier in types of polymorphic variables. For example, the variable null has type $\forall a.[a] \to \text{Bool}$. In our type system we always regard typings instead of types. We can express polymorphism in a typing, without using the $\forall$-quantifier. The polymorphic variable null has the principal typing $\{\} \vdash [a] \to \text{Bool}$. The fact that null does not occur in the domain of the type environment $\{\}$ indicates that null is polymorphic. In contrast, a principal typing $\{ys :: a\} \vdash a$ for a variable ys indicates that ys is monomorphic. We will see that ys cannot have several different types in an expression, because at each inference step the types of all variables in the type environments are unified.

So in our type system we do not use the $\forall$-quantifier at all. This keeps the type system simple. In particular, a type variable always scopes over the typing in which it appears. It is furthermore useful in practice, because no functional languages based on the Hindley-Milner type system uses the $\forall$-quantifier for types of polymorphic variables. A type signature

```
null :: [a] -> Bool
```

in a functional language can simply be interpreted as specifying for `null` the typing

$$\{\} \vdash [a] \to \text{Bool}$$

For every polymorphic variables *with a type signature* the type environment of the typing is empty. However, not for every polymorphic variables is this type environment empty, as we will demonstrate later in a discussion of restricted polymorphism.

Type Judgements. The environments for monomorphic and polymorphic variables are separate. Whereas the former associate monomorphic variables with types, the latter associate polymorphic variables with typings(!). A monomorphic (type) environment Δ is a mapping from variables to types. A (monomorphic) typing $\Delta \vdash \tau$ is a pair of a monomorphic type environment and a type. A polymorphic (typing) environment Γ is a mapping from variables to typings. We often write an environment as a set of pairs. $\text{dom}(\Gamma)$ denotes the domain of the environment Γ. The $+$ operator combines two environments such that the right supersedes the left. The $\cup$ operator combines two environments under the assumption that they agree on the common domain. Let V be a set of variables. $\Gamma \setminus V$ denotes the environment that

$$
\begin{array}{lll}
\text{variable} & & x, y, f \\
\text{Int literal} & & n := \ldots \mid -2 \mid -1 \mid 0 \mid 1 \mid 2 \mid \ldots \\
\text{data constructor} & & c := (:) \mid [] \mid \text{True} \mid \text{False} \mid \ldots \\
\text{expression} & & M, N := n \mid c \mid x \mid M\ N \mid \text{let } \mathit{bind} \text{ in } M \\
\text{pattern} & & P := M \\
\text{equation for variable } f & & \mathit{eq}(f) := f\ P_1\ \ldots\ P_i = M; \\
\text{definition of variable } f & & \mathit{def}(f) := \mathit{eq}_1(f) \ldots \mathit{eq}_i(f); \\
\text{program} & & \mathit{prog} := \mathit{def}_1(f_1) \ldots \mathit{def}_i(f_i); \\
 & & \\
\text{type variable} & & \alpha \\
\text{type constructor} & & T := \text{Int} \mid \text{Bool} \mid [] \mid \ldots \\
\text{type} & & \tau := \alpha \mid \tau_1 \to \tau_2 \mid T\ \tau_1 \ldots \tau_i
\end{array}
$$

Figure 4: Syntax of the Language

$$
\begin{array}{lll}
\text{monomorphic (type) environment} & & \Delta := \{x_1 :: \tau_1, \ldots, x_i :: \tau_i\} \\
\text{(monomorphic) typing} & & \Delta \vdash \tau \\
\text{polymorphic (type) environment} & & \Gamma := \{x_1 \mapsto (\Delta_1 \vdash \tau_1), \ldots, x_i \mapsto (\Delta_i \vdash \tau_i)\} \\
\text{type judgements} & & \Gamma; \Delta \vdash M :: \tau, \quad \Gamma; \Delta \vdash \mathit{eq}(f), \quad \Gamma_1; \Gamma_2; \Delta \vdash \mathit{def}(f), \quad \Gamma; \Delta \vdash \mathit{prog}
\end{array}
$$

Figure 5: Type Judgements and their Components

$$\textsc{Int}\ \frac{}{\Gamma; \{\} \vdash n :: \text{Int}} \qquad \textsc{Constructor}\ \frac{\Gamma(c) = \{\} \vdash \tau}{\Gamma; \{\} \vdash c :: \tau} \qquad \textsc{PolyVar}\ \frac{\Gamma(x) = \Delta \vdash \tau}{\Gamma; \Delta \vdash x :: \tau} \qquad \textsc{MonoVar}\ \frac{x \notin \text{dom}(\Gamma) \qquad \alpha \text{ new}}{\Gamma; \{x :: \alpha\} \vdash x :: \alpha}$$

$$\textsc{Application}\ \frac{\Gamma; \Delta_1 \vdash M :: \tau_1 \qquad \Gamma; \Delta_2 \vdash N :: \tau_2}{\Gamma; \Delta \vdash M\ N :: \tau_4}\ (\Delta, \tau_3 \to \tau_4) = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1, \tau_2 \to \alpha\}) \quad \alpha \text{ new}$$

$$\textsc{Let}\ \frac{\Gamma; \Gamma'; \Delta_1 \vdash \mathit{def}(f) \qquad \Gamma + \Gamma'; \Delta_2 \vdash M :: \tau'}{\Gamma; \Delta \vdash \text{let } \mathit{def}(f) \text{ in } M :: \tau}\ (\Delta, \tau) = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau'\})$$

$$\textsc{Equation}\ \frac{\Gamma \setminus \text{vars}(\{P_1, \ldots, P_i\}); \Delta_1 \vdash f\ P_1 \ldots P_i :: \tau_1 \qquad \Gamma \setminus \text{vars}(\{P_1, \ldots, P_i\}); \Delta_2 \vdash M :: \tau_2}{\Gamma; \Delta \setminus \text{vars}(\{P_1, \ldots, P_i\}) \vdash f\ P_1 \ldots P_i = M}\ (\Delta, \tau) = \mathcal{U}(\{\Delta_1, \Delta_2\}, \{\tau_1, \tau_2\})$$

$$\textsc{Definition}\ \frac{\Gamma \setminus \{f\}; \Delta_1 \vdash \mathit{eq}_1(f) \qquad \ldots \qquad \Gamma \setminus \{f\}; \Delta_i \vdash \mathit{eq}_i(f)}{\Gamma; \{f \mapsto \text{reduce}(\Delta' \vdash \Delta(f))\}; \Delta' \vdash \mathit{eq}_1(f) \ldots \mathit{eq}_i(f);}\ \begin{array}{l} \Delta = \mathcal{U}(\{\Delta_1, \ldots, \Delta_i\}) \\ \Delta' = \Delta \setminus \{f\} \end{array}$$

$$\textsc{Program}\ \frac{\Gamma; \Gamma_1; \Delta_1 \vdash \mathit{def}_1(f_1) \qquad \Gamma \cup \Gamma_1; \Gamma_2; \Delta_2 \vdash \mathit{def}_2(f_2) \qquad \ldots \qquad \Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_{i-1}; \Gamma_i; \Delta_i \vdash \mathit{def}_i(f_i)}{\Gamma \cup \Gamma_1 \cup \ldots \cup \Gamma_i; \Delta \vdash \mathit{def}_1(f_1) \ldots \mathit{def}_i(f_i);}\ \Delta = \mathcal{U}(\{\Delta_1, \ldots, \Delta_i\})$$

Figure 6: The Type System of Principal Monomorphic Typings

$$
\begin{array}{rll}
\mathcal{U}(\{\Delta_1, \ldots, \Delta_i\}) = \text{let} & \alpha_x \text{ new with } x \in \text{dom}(\Delta_1) \cup \ldots \cup \text{dom}(\Delta_i) & \\
 & \sigma = \text{mgu}(\{\alpha_x = \Delta(x) \mid \Delta \in \{\Delta_1, \ldots, \Delta_i\}, x \in \text{dom}(\Delta)\}) & \\
\text{in} & \Delta_1\sigma \cup \ldots \cup \Delta_i\sigma & \\
\mathcal{U}(\{\Delta_1, \ldots, \Delta_i\}, \{\tau_1, \ldots, \tau_j\}) = \text{let} & \alpha \text{ new and } \alpha_x \text{ new with } x \in \text{dom}(\Delta_1) \cup \ldots \cup \text{dom}(\Delta_i) & \\
 & \sigma = \text{mgu}(\{\alpha_x = \Delta(x) \mid \Delta \in \{\Delta_1, \ldots, \Delta_i\}, x \in \text{dom}(\Delta)\} \cup \{\alpha = \tau \mid \tau \in \{\tau_1, \ldots, \tau_j\}\}) & \\
\text{in} & (\Delta_1\sigma \cup \ldots \cup \Delta_i\sigma, \tau_1\sigma) &
\end{array}
$$

Figure 7: Unification of Type Environments and Types

$$\text{reduce}(\Delta \vdash \tau) = (\Delta \setminus \{x \in \text{dom}(\Delta) \mid \text{tyVars}(\Delta(x)) \cap \text{tyVars}(\tau) = \emptyset\} \vdash \tau$$

Figure 8: Reduction of a Typing

agrees with Γ except that it is not defined for the variables in V.

Figure 5 shows the syntax of typings, type environments and the four sorts of type judgements for the various program fragments. The type judgements for equations, definitions and programs do not have a type, only environments and a program fragment. The type judgement for definitions has two polymorphic environments. The first is for the polymorphic variables that can be used within the group and the second is for the polymorphic variable defined by the group.

Type Rules. The type rules of our type system of principal monomorphic typings are given in Figure 6. Variables in the domain of the polymorphic environment are polymorphic, all other variables are monomorphic. Note that within its own definition a variable is monomorphic. In rule EQUATION the variables occurring in the patterns have to be removed from environments to achieve correct variable scoping, similarly the defined variable has to be removed in rule DEFINITION. In rules LET and PROGRAM the combination of environments with + assures correct variable scoping. In rule PROGRAM Γ assigns typings to data constructors.

Unification of Environments and Types. The unification of monomorphic environments and types is defined in Figure 7. The function $\mathcal{U}$ uses a function mgu which determines the most general unifier of a set of type equations. Such a unification function is defined in Section 11.2.2 of [12]. Environments are unified by unifying the types for each variable. The unified environment has types for all variables that occur in any of the input environments.

Type Variables. Type variables always scope over a typing, no matter if the typing is within a polymorphic environment or part of a type judgement. For unification typings are split apart into monomorphic environments and types. Because type variables from different typings are unrelated, the type variables of typings that are unified have to be disjoint. To ensure disjointness we demand by the common informal statement 'α new' in the type rules that every such type variable α is distinct from all other type variables introduced in the type inference tree. This requirement of globally unique type variables is easy to implement efficiently.

Alternatively, we could require disjointness in the premises of rules and add a rule for renaming type variables in typings. This approach makes the scope of type variables explicit in the type system, but it also leads to a larger type system that is further away from an efficient implementation.

Restricted Polymorphism. Consider the program:

```
f xs = let g y = y : xs
       in g 1 ++ g True
```

For the local definition we obtain the type judgement:

$$\{(++) \mapsto \ldots\}; \{\mathrm{g} \mapsto (\{xs :: [a]\} \vdash a \to [a])\} \vdash \mathrm{g}\ y = y : xs$$

Although g is a polymorphic variable after the in, it cannot be instantiated to different types there. We can infer

$$\{(++) \mapsto \ldots, \mathrm{g} \mapsto (\{xs :: [a]\} \vdash a \to [a])\}; \{xs :: [\mathrm{Int}]\} \vdash g\ 1 :: [\mathrm{Int}]$$

and

$$\{(++) \mapsto \ldots, \mathrm{g} \mapsto (\{xs :: [a]\} \vdash a \to [a])\}; \{xs :: [\mathrm{Bool}]\} \vdash g\ \mathrm{True} :: [\mathrm{Bool}]$$

but the type inference step for g 1 ++ g True requires unification of the types of xs and hence fails. The program is rightly not typable.

The monomorphic variable xs in the monomorphic environment of g expresses that the type of the variable g is not polymorphic in a. For programs such as this example a polymorphic environment must associate polymorphic variables with typings, not just types. There is however no point for such a typing to contain a monomorphic variable whose type does not contain any type variable of the type of the typing. Such superfluous monomorphic variables are removed by the function reduce defined in Figure 8 and used in rule DEFINITION.

Type Inference and the Hindley-Milner system. We obtained our type system of principal monomorphic typings by rewriting John Mitchell's type checking algorithm PTL ([12], Section 11.3.3) as a type inference system and extending it to our language. Mitchell uses PTL to expose the strong relationship between type checking for the simply typed λ-calculus and for the Hindley-Milner type system.

Our type rules can be read as a type checking algorithm. The (first) polymorphic environment and the program fragment are the input. The remaining components of a type judgement, which may be a second polymorphic environment, the monomorphic environment and a type, are the output.

For example, for a polymorphic environment Γ and an application $M\ N$ the algorithm recursively calls itself twice. One call with Γ and M as input determines the principal monomorphic typing $\Delta_1 \vdash \tau_1$, and the other call with Γ and N as input determines the principal monomorphic typing $\Delta_2 \vdash \tau_2$. The two calls are independent. Finally, the unification function $\mathcal{U}$ combines the two typings to obtain the principal monomorphic typing $\Delta \vdash \tau_4$ for $M\ N$ and Γ.

Definition 3. A (monomorphic) *typing* $\Delta \vdash \tau$ *is principal* for an expression M and a polymorphic environment Γ iff $\Gamma; \Delta \vdash M :: \tau$ and all (monomorphic) typings $\Delta' \vdash \tau'$ with $(\Gamma')^\forall \cup \Delta' \vdash_{\mathrm{HM}} M :: \tau'$ are instances of $\Delta \vdash \tau$. The second type judgement is a judgement of the Hindley-Milner type system and $(\cdot)^\forall$ translates a polymorphic environment into an environment with $\forall$-quantified types.

We claim that if $(\Gamma)^\forall \cup \Delta' \vdash_{\mathrm{HM}} M :: \tau'$, then a type judgement $\Gamma; \Delta \vdash M :: \tau$ is provable in our type system and $\Delta \vdash \tau$ is principal for M and Γ. Theorems 11.3.5, 11.3.9, 11.3.10 and 11.3.13 of [12] prove similar properties for algorithm PTL. An adaption of Mitchell's proofs should be routine but is outside the scope of this paper.

5. THE EXPLANATION GRAPH

Our type system has principal monomorphic typings, but the polymorphic environment still creates global dependencies in a type inference tree. The typing for the use of a polymorphic variable is a leaf in the inference tree. To understand why the polymorphic variable has the given typing we have to search the inference tree for the place where the

typing is added to the polymorphic typing environment. So the inference tree is not compositional.

The solution is simple: We copy the whole inference tree of the definition of a polymorphic variable to every use occurrence of this variable in the tree, so that the typing for a used polymorphic variable is no longer a leaf in the tree. Consider the expression `id 3` where

```
id x = x
```

From the non-compositional derivation tree

$$\frac{\Gamma;\{\} \vdash \mathrm{id} :: a \to a \qquad \Gamma;\{\} \vdash 3 :: \mathrm{Int}}{\Gamma;\{\} \vdash \mathrm{id}\ 3 :: \mathrm{Int}}$$

we construct the tree

$$\dfrac{\dfrac{\dfrac{\dfrac{\{\mathrm{id} :: b\} \vdash \mathrm{id} :: b \quad \{x :: a\} \vdash x :: a}{\{\mathrm{id} :: a \to a, x :: a\} \vdash \mathrm{id}\ x :: a} \qquad \{x :: a\} \vdash x :: a}{\{\mathrm{id} :: a \to a\} \vdash \mathrm{id}\ x = x} \qquad \dfrac{\{\} \vdash \mathrm{id} :: a \to a \qquad \{\} \vdash 3 :: \mathrm{Int}}{\{\} \vdash \mathrm{id}\ 3 :: \mathrm{Int}}}$$

This explanation tree is not completely syntax directed as the type inference tree, but it is compositional. The polymorphic environment is no longer needed. We also collapse trivial inference steps for definitions with a single equation. The typing for a data constructor is still a leaf of the tree, but it may be useful to put the definition of the data type above the typing in the tree, because it implicitly declares the type of the data constructor.

Copying the tree for every definition of a polymorphic variable to every use of it is a waste of space. So we share such subtrees and construct an acyclic explanation graph instead of an explanation tree. Sharing is also useful to tell a programmer who is navigating through the explanation graph that he/she already visited a certain subgraph, even if he/she did so by coming from a different use point of a polymorphic variable.

6. NAVIGATION THROUGH THE GRAPH

We defined the graph to be compositional so that each inference step is meaningful on its own. In practice the programmer will only be interested in a fraction of the inference steps of the explanation graph. The programmer understands typings best by interactively navigating through the graph.

At the Level of Program Fragments. Type checking our example program from Section 2 gives the following error message:

```
Type error in: (last xs) : (init xs)
  because
Expressions: (:) (last xs)   init xs
Types      : [a]->[a]        [b]
with  xs     [[a]]           [[[b]]]
```

The typings of both subexpressions are surprising. Why is `xs` a list of lists in the left subexpression and even a three times nested list in the right one? It should just be a list with arbitrary elements.

The central point in locating errors is that the type of `xs` in the typings of the subexpressions `(:) (last xs)` and `init xs` may well be more general than the type we intend `xs` to have. However, our intended types should be an instance of the types given in the typings. The fact that this is not the case is a clear indication of an error.

Hence we ask for an explanation of the first typing:

```
Expression: (:) (last xs)
Type      : [a]->[a]
with  xs    [[a]]
  because
Expressions: (:)           last xs
Types:       a->[a]->[a]   b
with  xs                   [[b]]
```

Here the typing for `(:)` is as intended, but not that of `xs` in the typing for `last xs`. So we demand an explanation:

```
Expression: last xs
Type      : b
with xs     [[b]]
  because
Expressions: last       xs
Types      : [[b]]->b   a
with xs                 a
```

We intend the function `last` to have type `[b]->b`, which is not an instance of `[[b]]->b`. We enquire further:

```
Function   : last
Type       : [[a]]->a
  because of its definition
Lhs/Rhs    : last xs   head (reverse xs)
Types      : c         a
with last    b->c
     xs      b         [[a]]
```

The left-hand side of the equation of `last` is correct, but the type of `xs` in the typing for the right-hand side contradicts our intentions.

At the Level of Polymorphic Variables. Asking for more explanations will finally lead us to the source of the error. Unfortunately this search process is long. We can speed up the search. We only ask for type explanations in terms of used polymorphic variables, that is, when traversing the explanation graph we skip the inference steps for all program fragments but polymorphic variables. Thus we can quickly locate the erroneous definition. We start again at the type error:

```
Type error in: (last xs) : (init xs)
  because
last :: [[a]]->a
init :: [[[a]]]->[a]
```

We intend the types of both polymorphic functions to be more general. We ask for an explanation of the first one:

```
last :: [[a]]->a
  because
head :: [a]->a
reverse :: [[a]]->[a]
```

Here the type of `head` is as intended, but not the type of `reverse`. So we ask for an explanation of its type:

```
reverse :: [[a]]->[a]
  because
(++) :: [a] -> [a] -> [a]
```

The type of the only polymorphic variable that is used is correct. Hence the error must be in the definition of `reverse`.

To determine the exact location of the error we now switch to explanations at the level of program fragments:

```
reverse :: [[a]]->[a]
  because
Equation:      .. [] = []    .. (x:xs) = ..
with reverse   [b]->[c]      [[a]]->[a]
```

The expected type of `reverse`, `[a]->[a]` is an instance of the type given in the typing for the first equation, but not an instance of the type given in the typing for the second equation. Hence we ask about the second typing:

```
Equation    : .. (x:xs) = ..
with reverse  [[a]]->[a]
  because
Lhs/Rhs       : reverse (x:xs)  (reverse xs) ++ x
Types         : b               [a]
with  reverse [c]->b            d->[a]
      x       c                 [a]
      xs      [c]               d
```

In the second typing `x` is a list. Because that is not our intention, we ask further:

```
Expression   : (reverse xs) ++ x
Type         : [a]
with  reverse  d->[a]
      x        [a]
      xs       d
  because
Expressions : (++) (reverse xs)  x
Types       : [b]->[b]           c
with  reverse a->[b]
      x                          c
      xs      a
```

Here both typings are reasonable. Hence we have located the error: The expression `(reverse xs) ++ x` is wrong. By comparing our intentions with the definition of `reverse` and the given typings we realise that the correct expression is `(reverse xs) ++ [x]`.

Navigation at different levels enables us to avoid unnecessary detail and to quickly reach the source of a type error. We usually start at the high level, regarding only polymorphic functions, and later move to individual inference steps at program fragment level. It is also conceivable to have an even finer level, which shows the unification process of a type inference step in several stages.

7. ALGORITHMIC DEBUGGING

The problem that a type checker notices type inconsistencies often far from the sources of the errors reminds of the similar problem for run-time errors, which also usually are observed far from the source. Algorithmic debugging was introduced by Shapiro to diagnose wrong and missing answers in Prolog [18]. Later algorithmic debugging was successfully applied to locate the sources of run-time errors in functional and other languages [15, 5]. The principle of algorithmic debugging is not linked to run-time errors. It is quite clear from [13] that algorithmic debugging can be applied to any propositions such as evaluation judgements or type judgements which are defined by a compositional tree (or acyclic graph). At every tree node is a proposition which can be correct or erroneous. A node is the source of an error, if its proposition is erroneous but the propositions of all its children are correct. Algorithmic debugging consists of constructing a tree with an erroneous root when we observe erroneous behaviour and then locating in the tree a source of this error. To determine if a node proposition is erroneous, an oracle is used. Usually the oracle is the programmer, who is asked questions about the validity of propositions.

Here is an example session with user input (y/n) in italics:

```
Type error in: (last xs) : (init xs)

last :: [[a]]->a
Is intended type an instance? (y/n)   n

head :: [a]->a
Is intended type an instance? (y/n)   y

reverse :: [[a]]->[a]
Is intended type an instance? (y/n)   n

(++) :: [a] -> [a] -> [a]
Is intended type an instance? (y/n)   y
```

At this point the system knows that the source of the error is in the definition of `reverse` and starts asking about typings of fragments of the definition.

```
reverse :: [b]->[c]
Is intended type an instance? (y/n)   y

reverse :: [[a]]->[a]
Is intended type an instance? (y/n)   n
```

The system could actually know the answer to this question from the third question.

```
reverse (x:xs) :: b
reverse        :: [c]->b
x              :: c
xs             :: [c]
Are intended types an instance? (y/n)   y
```

Note that equal type variables of separate types must be instantiated equally to obtain the intended types.

```
(reverse xs) ++ x :: [a]
reverse           :: d->[a]
x                 :: [a]
xs                :: d
Are intended types an instance? (y/n)   n

(++) (reverse xs) :: [b]->[b]
reverse           :: a->[b]
xs                :: a
Are intended types an instance? (y/n)   y

Error located. Wrong expression:
(reverse xs) ++ x
```

The system assumes that the typing for a single variable such as `x` is correct. It probably should also never ask about the types of data constructors such as `(:)`, assuming that type definitions are correct. It is useful and common practice in algorithmic debugging that the programmer can declare a set of variables as correct, as *trusted*; for example all variables defined in some standard libraries. This reduces the number of questions.

To reduce the number of questions further, it is feasible that the programmer, instead of just answering *no*, also indicates which part of which type does not meet his/her intentions. As answer to the question

```
reverse :: [[a]]->[a]
Are intended types an instance?
```

the programmer may indicate, that the inner list of the argument type is erroneous. Hence the second equation of the definition of `reverse` must be erroneous and the system can skip the question

```
reverse :: [b]->[c]
Is intended type an instance? (y/n)
```

Similarly, the programmer could indicate that two occurrences of the same type variable conflict with his/her intentions of instantiating these occurrences differently.

The questions of algorithmic debugging are shorter than explanations of typings. Also algorithmic debugging leads to the source of the error without the programmer having to understand how typings were inferred. On the other hand the programmer might want to understand typings. Furthermore, it is in practice much easier to locate an erroneous typing in a set of typings than to state whether a typing is an instance of the intended one.

We believe that a combination of algorithmic debugging together with free navigation through explanations of type inference steps is desirable. Practical experience is needed to determine how exactly the programmer can use the explanation tree most effectively.

8. IMPLEMENTATION

We built a prototype type explanation and debugging tool. For a program in the language defined in Section 6 it constructs the type explanation graph; in case of an untypable program the root of the graph is a type error message. The tool only has a simple textual user interface but enables navigation through the explanation graph in various ways. With the prototype we tested many examples and refined our ideas. All the examples in this paper were obtained from the output of the prototype.

The prototype is written in Haskell, based on Mark Jones' type checker for core Haskell [7]. Although we had to replace the actual type checking algorithm by our own type checking algorithm, Jones' type checker provides a framework and will be even more useful when we extend the prototype to handle the Haskell class system.

In the development of our prototype we concentrated on quick development and ease of modification to explore our ideas. In return it is not efficient at all. The main efficiency issue for a practical tool is the space required for the explanation graph. The graph is huge. However, it does not need to be constructed in full but can be constructed in small pieces as needed. The type checker may first only store the typings of polymorphic variables. Then, when the programmer requires an explanation of some program fragment, this fragment is type checked again and its part of the explanation graph constructed. Note that the type checking algorithm of principal monomorphic typings only requires the typings of all polymorphic variables in scope to type check a program fragment.

Even an improved implementation of our type checking algorithm is probably less efficient than a good implementation of Milner's $\mathcal{W}$ algorithm. Our algorithm introduces more type variables and performs more unifications. Furthermore, during the construction of the explanation tree type variables cannot be implemented as mutable variables for efficient substitution, because all type variables appear in the explanation graph. However, a combination of our algorithm with algorithm $\mathcal{W}$ is possible. Both stop at a type conflict in the same top-level definition. So $\mathcal{W}$ may be used first and only the erroneous definition has to be type checked again with our algorithm.

9. RELATED WORK

Many people have investigated methods for improving the understanding of type errors. Several of these also saw the need for a type checking algorithm different from $\mathcal{W}$. Bernstein and Stark use a type checking algorithm similar to ours that defines type inference trees that are compositional without any post-processing [2]. Basically, the algorithm determines a type for each occurrence of a variable. These types are unified (or matched in the case of a polymorphic variable) at the binding occurrence of the variable. The system enables the programmer to obtain the types of subexpressions; the authors do not take advantage of the compositionality. The large number of types for a single variable make types and especially typings hard to understand. Together with Simon and Huch we developed a variation of this type checking algorithm that reduces the problem [19]: the algorithm collects several types only for monomorphic variables. However, we had not yet realised the importance of compositionality and typings. Yang also outlines a similar algorithm [24]. He suggests combining it with algorithm $\mathcal{M}$. $\mathcal{M}$ passes more type information downward than $\mathcal{W}$ when traversing an expression. Lee and Yi show that $\mathcal{M}$ finds type conflicts earlier than $\mathcal{W}$ [8]. McAdam defines unification of substitutions to avoid the left-to-right bias of $\mathcal{W}$ [9].

Walz and Johnson apply a maximum flow technique to the set of type equations to determine the most likely source of an error [21]. Wand [22] modifies the unification algorithm used by $\mathcal{W}$ to keep track for every type variable which program fragment forces its instantiation. Beaven and Stansifer [1] and later Duggan and Bent [4] improve Wand's method. Choppella and Haynes present a related method [3]. It might be possible to transfer some of these approaches to our type system to guide and reduce the number of questions in algorithmic debugging.

Several people note the importance of an interactive tool. Soosaipillai developed a tool for a small functional language that interactively explains each step of algorithm $\mathcal{W}$ [20]. Rittri outlines the design of an interactive type error explanation system based on Wand's method [17]. Together with Simon and Huch we developed a tool for interactively viewing the types of subexpressions in their context [19].

McAdam defines a graph with type information to generalises the approaches of Wand and Bernstein and Stark [10] Yang and Michaelson investigate psychological aspects of explaining type errors [25, 26]. Yang, Michaelson, Trinder and Wells present a manifesto of properties a good type error reporting system should have [23]. We think our system has all seven properties.

10. SUMMARY AND FUTURE WORK

We analysed the problem of understanding types and type errors and identified compositionality as a key to generating

good explanations. A tree of principal types is not compositional. A tree of principal typings is compositional. In the Hindley-Milner type system not every expression has a principal typing, but we noticed that Mitchell's type algorithm PTL implicitly defines a type system of principal monomorphic typings. From the type inference tree of this type system we construct a compositional acyclic type explanation graph. Each inference step of the graph is uniquely determined by the premises and can hence be understood on its own. An explicit ∀-quantifier or generic and non-generic type variables are unnecessary. We demonstrated how interactive navigation of the explanation graph assists understanding types and type errors and how algorithmic debugging based on the explanation graph can semi-automatically locate the source of type errors.

Experiments with our prototype tool are encouraging. The tool needs a better user interface. To improve orientation in the explanation graph, we envision the tool to show explanations of typings in one window and the source program with the relevant program fragments highlighted in a second window. The programmer should also be free to mark any program fragment and ask for its typing. A mouse pointer would ease marking erroneous parts of types.

The polymorphism of the Haskell class system makes type errors even worse. Just view Hugs' error message for the tiny expression `(print . div) 42`:

```
ERROR: Illegal Haskell 98 class constraint in
       inferred type
* Expression : (print . div) 42
* Type       : (Show (a -> a), Integral a) => IO ()
```

We are currently working on extending our prototype to handle the Haskell class system. The extension of the explanation graph by classes appears to be straightforward.

Type systems for various kinds of program analysis have been developed [14]. We speculate that explanation graphs similar to ours can be constructed for many of these systems. Such a graph may be a good basis for showing the inferred information to the developer of the analysis or even the programmer.

A navigation and algorithmic debugging tool based on the type explanation graph is no magic wand which turns all problems with types into wisps of white smoke. But we claim that it substantially helps to understand types and to find the cause of most type errors.

Acknowledgements

I thank Simon Thompson, Axel Simon, Frank Huch, Colin Runciman and the anonymous referees.

11. REFERENCES

[1] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM LOPLAS*, 2(4):17–30, 1993.

[2] K. Bernstein and E. Stark. Debugging type errors. Technical report, Stony Brook, 1995.

[3] V. Choppella and C. T. Haynes. Diagnosis of ill-typed programs. TR426, Indiana University, 1995.

[4] D. Duggan and F. Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, 1996.

[5] P. Fritzson, N. Shahmehri, M. Kamkar, and T. Gyimothy. Generalized algorithmic debugging and testing. *ACM LOPLAS*, 1(4):303–322, Dec. 1992.

[6] T. Jim. What are principal typings and what are they good for? In *POPL '96*, pages 42–53. ACM, 1996.

[7] M. P. Jones. Typing Haskell in Haskell. In *Proceedings of the 1999 Haskell Workshop*, pages 1–14. Universiteit Utrecht, UU-CS-1999-28, 1999.

[8] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM TOPLAS*, 20(4):707–723, July 1998.

[9] B. J. McAdam. On the unification of substitutions in type inference. In *IFL'98*, LNCS 1595, pages 137–152, 1999.

[10] B. J. McAdam. Generalising techniques for type debugging. In *Trends in Functional Programming*, chapter 6. Intellect, 2000.

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

[12] J. C. Mitchell. *Foundations for Programming Languages*. MIT Press, 1996.

[13] L. Naish. A declarative debugging scheme. *Journal of Functional and Logic Programming*, 1997(3), 1997.

[14] F. Nielson, H. R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer, 1999.

[15] H. Nilsson. *Declarative Debugging for Lazy Functional Languages*. PhD thesis, Linköping, Sweden, May 1998.

[16] S. L. Peyton Jones, J. Hughes, et al. Haskell 98: A non-strict, purely functional language. http://www.haskell.org, Feb. 1999.

[17] M. Rittri. Finding the source of type errors interactively. In *Proceedings of El Wintermöte*, pages 273–276. University of Göteborg and Chalmers University of Technology, 1993. PMG report 73.

[18] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, 1983.

[19] A. Simon, O. Chitil, and F. Huch. Typeview: A tool for understanding type errors. In *Draft Proc. of IFL 2000*, pages 63–69. RWTH Aachen, 2000. AIB 00-7.

[20] H. Soosaipillai. An explanation based polymorphic type checker for Standard ML. Master's thesis, Heriot Watt University, Edinburgh, Scotland, 1990.

[21] J. A. Walz and G. F. Johnson. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *POPL'86*, pages 44–57. ACM, 1986.

[22] M. Wand. Finding the source of type errors. In *POPL'86*, pages 38–43. ACM, 1986.

[23] J. Yan, G. Michaelson, P. Trinder, and J. B. Wells. Improved type error reporting. In *Draf Proc. of IFL 2000*, pages 71–86. RWTH Aachen, 2000. AIB 00-7.

[24] J. Yang. Explaining type errors by finding the source of a type conflict. In *Trends in Functional Programming*, chapter 7. Intellect, 2000.

[25] J. Yang and G. Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, 2000.

[26] J. Yang, G. Michaelson, and P. Trinder. How do people check polymorphic types? In *Proceedings of 12th Workshop on Psychology of Programming*, pages 67–77. Edizioni Memoria, 2000.