Heuristics for type error discovery and recovery (revised revised)

Jurriaan Hage

Bastiaan Heeren

Department of Information and Computing Sciences,
Utrecht University

Technical Report UU-CS-2006-054

www.cs.uu.nl

ISSN: 0924-3275

Abstract. Type error messages that are reported for incorrect functional programs can be difficult to understand. The reason for this is that most type inference algorithms proceed in a mechanical, syntax-directed way, and are unaware of inference techniques used by experts to explain type inconsistencies. We formulate type inference as a constraint problem, and analyze the collected constraints to improve the error messages (and, as a result, programming efficiency). A special data structure, the type graph, is used to detect global properties of a program, and furthermore enables us to uniformly describe a large collection of heuristics which embed expert knowledge in explaining type errors. Some of these also suggest corrections to the programmer. Our work has been fully implemented and is used in practical situations, showing that it scales up well. We include a number of statistics from actual use of the compiler showing us the frequency with which heuristics are used, and the kind and number of suggested corrections.

1 Introduction

Type inference algorithms for Hindley-Milner type systems typically proceed in a syntax-directed way. The main disadvantage of such a rigid and local approach is that the type error messages that are reported not always reflect the actual problem.

The need for precise type error messages is most apparent when teaching a course on functional programming to students. Over the last five years we have developed the TOP framework to support flexible and customizable type inference. This framework has been used to build the Helium compiler [7], which implements almost the entire Haskell 98 standard, and which is especially designed for learning the programming language. This compiler, and thus the type graph and heuristics that drive its type inference process, have been used with good results in an educational setting since 2002. It is freely available for download [7].

We follow a constraint-based approach: a set of constraints is collected by traversing the abstract syntax tree of a program, which is then passed to a constraint solver. This approach gives us the usual benefit of decoupling specification and computing a solution, which tends to simplify both. Because many program analyses share the same kinds of constraints, it also allows us to reuse our solvers.

A remaining issue is that the order in which the solver considers the constraints strongly influences at which point an inconsistency is detected. In existing compilers (which tend to solve constraints as they go), this has the disadvantage that a bias exists for finding errors towards the end of a program. Although our TOP framework provides various ways of ordering type constraints (see [3]), in this paper we discuss a constraint solver that uses type graphs, a data structure that allows a global analysis of the types in a program. More importantly, type graphs naturally support heuristics, which embed expert knowledge in explaining type errors.

Some of these heuristics correspond closely to earlier proposals for improving error messages, such as determining the most likely source of a type error by counting pieces of evidence [14]. In addition, we have defined a number of heuristics of our own. For example, there are heuristics which can discover commonly made mistakes (like confusing string and character literals, or confusing addition $+$ and append $+\!\!+$), and a sophisticated heuristic which considers function applications in detail to discover incorrectly ordered, missing, or superfluous arguments.

A number of these heuristics are tried in parallel, and a voting mechanism decides which constraints will be blamed for the inconsistency. These constraints are then removed from the type graph, and each of them results in a type error message reported back to the programmer. The use of type graphs thus leads naturally to reporting multiple, possibly independent type error messages.

The contributions we make in this paper are the following: we have integrated a large collection of heuristics into a comprehensive and extensible framework. Although some of these are known from the literature, this is the first time, to our knowledge, that they have been integrated into a full working system. In addition, we have defined a number of new heuristics based on our experiences as teachers of Haskell. Our work has been fully implemented into the Helium compiler which shows that it scales to a full programming language. Helium has been used in three complete courses of functional programming at Universiteit Utrecht comprising several hundreds of students. Furthermore, we have applied the compiler to a collection of 11,256 programs collected in 2005, which yields statistics on how often heuristics contribute to finding a mistake, and the number and kind of probable fixes suggested by the compiler. Many examples in this paper are taken from this collection of programs.

This paper is organized as follows. In the next section we set the scene and introduce the constraints we will use. Then we introduce each heuristic in turn in Section 3. In Section 5 we show how the heuristics are put together in the Helium compiler, and Section 6 gives statistical information about the usage of heuristics based on a large collection of programs compiled by first-year students. Section 4 considers the type graph data structure on which the heuristics are all defined. In Section 7 we consider related work, after which we conclude. The appendix includes a sample trace of the compiler, which has been included only to simplify the task of reviewers.

2 Constraints

In this paper we consider only sets of equality constraints. Naturally, polymorphism is part of the language, but it is used only between binding groups, to communicate the polymorphic type of a definition to its use sites. For every such use, the polymorphic type will be replaced by a fresh instance of that type. The major consequence of this approach is that definitions from previous binding groups are considered given and can not be blamed for a type error, only their use can. Due to space restrictions, we refer the reader to [5] for more details of this process.

For the purposes of this paper, we can thus simply assume that constraints are of the form $\tau_1 \equiv \tau_2$, in which τ_1 and τ_2 are monomorphic types, either type variables $v_1, v_2, \ldots$, type constants (such as Int and $\rightarrow$), or the application of a type to another. For example, the type of functions from integers to booleans is written $(((\rightarrow)\ \mathit{Int})\ \mathit{Bool})$. Type application is left-associative, and we omit parentheses where allowed. We often write the function constructor infix, resulting in $\mathit{Int} \rightarrow \mathit{Bool}$. We assume the types are well-kinded: types like $\mathit{Int}\ \mathit{Bool}$ do not occur.

3 Heuristics

In principle, all the constraints that contribute to an error are candidates for removal. However, some constraints are better candidates for removal than others. To select the "best" candidate for removal, we use a number of heuristics. These heuristics are usually based on common techniques used by experts to explain type errors. In addition to selecting what is reported, heuristics can specialize error messages, for instance by including hints and probable fixes. For each removed constraint, we create a single type error message using the constraint information stored with that constraint. The approach naturally leads to multiple, independent type error messages being reported.

Many of our heuristics are considered in parallel, so we need some facility to coordinate the interaction between them. The Helium compiler uses a voting mechanism based on weights attached to the heuristics, and the "confidence" that a heuristic has in its choice. Some heuristics, the tie-breakers, are only considered if none of the other heuristics came up with a suggestion.

A consideration is how to present the errors to a user, taking into consideration the limitations imposed by the used output format. In this paper we restrict ourselves to simple textual error messages.

In the following we shall consider a number of heuristics, a subset of what is currently available in Helium. Heuristics available in Helium have been omitted for various reasons: some of the heuristics are still in their experimental stages (e.g., the repair heuristics developed as part of a Master Thesis project by Langebaerd [8]), some have been considered elsewhere (e.g., the type inference directives [6]), and some deal with overloading, an issue we do not address in this paper.

We have grouped the heuristics into three major groups: the general heuristics that apply to constraint solving in general, the language dependent heuristics that are specific for functional programming languages and Haskell in particular, and finally we consider a number of program correcting heuristics that include a probable fix as part of the type error message.

We illustrate the heuristics by means of examples. Many of these are taken from a collection of $11,256$ actual compiles made by students in the course year 2004/2005. They can be recognized by the fact that they are followed by an error message. For reasons of brevity we only include the parts of the program that are involved in the error, and in some cases have translated identifier names to English and removed some unimportant aspects of the code, for reasons of clarity.

3.1 General heuristics

The heuristics in this section are not restricted to type inference, but they can be used for other constraint satisfaction problems as well.

Participation ratio heuristic Our first heuristic applies some common sense reasoning: if a constraint is involved in more than one conflict, then it is a better candidate for removal. The set of candidates is thus reduced to the constraints that occur most often in conflicts. This heuristic is driven by a ratio r (typically at least 95%): only constraints that occur in at least r percent of the conflicts are retained as candidates. This percentage is computed relative to the maximum number of conflicts any of the constraints in the set was involved in.

Note that this heuristic also helps to decrease the number of reported error messages, as multiple conflicts are resolved by removing a single constraint. However, it does not guarantee that the compiler returns the minimum number of error messages.

The participation-ratio heuristic implements the approach suggested by Johnson and Walz [14]: if we have three pieces of evidence that a value should have type *Int*, and only one for type *Bool*, then we should focus on the latter.

First come, first blamed heuristic The next heuristic we present is used as a final tie-breaker since it always reduces the number of candidates to one. This is an important task: without such a selection criteria, it would be unclear (even worse: arbitrary) what is reported. We propose a tie-breaker heuristic which considers the position of a constraint in the constraint list.

In [3] we address how to flatten an abstract syntax tree decorated with constraints into a constraint list L. Although the order of the constraints is irrelevant while constructing the type graph, we store it in the constraint information, and use it for this particular heuristic: for each error path, we take the constraint which completes the path – i.e., which comes *latest* in L. This results in a list of constraints that complete an error path, and out of these constraints we pick the one that came *first* in L.

3.2 Language dependent heuristics

The second class of heuristics involves those that are driven by domain knowledge. Although the instances we give depend to some extent on the language under consideration, it is likely that other programming languages allow similarly styled heuristics.

Trust factor heuristic The trust factor heuristic computes a trust factor for each constraint, which reflects the level of trust we have in the validity of a constraint. Obviously, we prefer to report constraints with a low trust factor. We discuss four cases that we found to be useful.

(1) Some constraints are introduced *pro forma*: they trivially hold. An example is the constraint expressing that the type of a let-expression equals the type of its body. Reporting such a constraint as incorrect would be highly inappropriate. Thus, we make this constraint highly trusted. The following definition is ill-typed because the type signature declared for $squares$ does not match with the type of the body of the let-expression.

```
squares :: Int
squares = let f i = i * i
          in map f [1 .. 10]
```

Dropping the constraint that the type of the let-expression equals the type of the body would remove the type inconsistency. However, the high trust factor of this constraint prevents us from doing so. In this case, we select a different constraint, and report, for instance, the incompatibility between the type of $squares$ and its right-hand side.

(2) The type of a function imported from the standard Prelude, that comes with the compiler, should not be questioned. Ordinarily such a function can only be *used* incorrectly.

(3) Although not mandatory, type annotations provided by a programmer can guide the type inference process. In particular, they can play an important role in the reporting of error messages. These type annotations reflect the types expected by a programmer, and are a significant clue where the actual types of a program differ from his perception. We can decide to trust the types that are provided by a user. In this way, we can mimic a type inference algorithm that pushes a type signature into its definition. Practice shows, however, that one should not rely too much on type information supplied by a novice programmer: these annotations are frequently in error themselves.

(4) A final consideration for the trust factor of a constraint is in which part of the program the error is reported. Not only types of expressions are constrained, but errors

can also occur in patterns, declarations, and so on. Hence, patterns and declarations can be reported as the source of a type conflict. Whenever possible, we report an error for an expression. In the definition of $increment$, the pattern $(\_ : x)$ (x must be a list) contradicts with the expression $x + 1$ (x must be of type *Int*).

$$increment\ (\_ : x) = x + 1$$

We prefer to report the expression, and not the pattern. If a type signature supports the assumption that x must be of type *Int*, then the pattern can still be reported as being erroneous.

Avoid folklore constraints heuristic Some of the constraints restrict the type of a subterm (e.g., the condition of a conditional expression must be of type *Bool*), whereas others constrain the type of the complete expression at hand (e.g., the type of a pair is a tuple type). These two classes of constraints correspond very neatly to the unifications that are performed by algorithm $\mathcal{W}$ and algorithm $\mathcal{M}$ [9] respectively. We refer to constraints corresponding to $\mathcal{M}$ as *folklore* constraints. Often, we can choose between two constraints – one which is folklore, and one which is not. In the following definition, the condition should be of type $Bool$, but is of type $String$.

$$test :: Bool \rightarrow String$$
$$test\ b = \mathbf{if}\ \texttt{"b"}\ \mathbf{then}\ \texttt{"yes!"}\ \mathbf{else}\ \texttt{"no!"}$$

Algorithm $\mathcal{W}$ detects the inconsistency at the conditional, when the type inferred for `"b"` is unified with $Bool$. As a consequence it mentions the entire conditional and complains that the type of the condition is $String$ instead of $Bool$. Algorithm $\mathcal{M}$, on the other hand, pushes down the expected type $Bool$ to the literal `"b"`, which leads to a similar error report, but now only the literal `"b"` will be mentioned. The former gives more context information, and is thus easier to understand for novice programmers. For this reason we prefer not to blame folklore constraints for an inconsistency.

Avoid application constraints heuristic This heuristic is surprising in the sense that we only found out that we needed it after using our compiler, and discovering that some programs gave counterintuitive error messages. Consider the following fragment

$$\mathbf{if}\ plus\ 1\ 2\ \mathbf{then}\ ...\ \mathbf{else}\ ...$$

in which $plus$ has type $Int \rightarrow Int \rightarrow Int$.

The application heuristic (a program correcting heuristic discussed in Section 3.3) finds that the arguments to $plus$ indeed fit the type of the function. However, the result of the application does not match the expected $Bool$ for the condition. In this situation, algorithm $\mathcal{W}$ would put the blame on the condition, while $\mathcal{M}$ would blame the use of $plus$. There is (unfortunately) another possibility: the application itself is blamed. However, given that the arguments do fit, it is quite unlikely that the application as a whole is at fault, and such an error message becomes unnatural. The task of this heuristic is to remove these constraints from the candidate set. There is a similar heuristic for negations, which is necessary in Haskell, because negation is part of the language and not just another function.

Unifier vertex heuristic At this point, the reader may have the impression that heuristics always put the blame on a single location. If we have only two locations that contradict, however, then preferring one over another introduces a bias. Our last heuristic illustrates that we can also design heuristics to restore balance and symmetry in error messages, by reporting multiple program locations with contradicting types. This technique is comparable to the approach suggested by Yang [15].

The design of our type rules (Chapter 6 of [5]) accommodates such a heuristic: at several locations, a fresh type variable is introduced to unify two or more types, e.g., the types of the elements in a list. We call such a type variable a *unifier*. In our heuristic, we use unifiers in the following way: we remove the edges from and to a unifier type variable. Then, we try to determine the types of the program fragments that were equated via this unifier. With these types we create a specialized error message. In the following example, the type of the context is also a determining factor.

All the elements of a list should be of the same type, which is not the case in f's definition.

$f\ x\ y = [x, y, id, \texttt{"\textbackslash n"}]$

In the absence of a type signature for f, we choose to ignore the elements x and y in the error message, because their types are unconstrained. We report that id, which has a function type, cannot appear in the same list as the string `"\n"`. By considering how f is applied in the program, we could obtain information about the types of x and y. In our system, however, we never let the type of a function depend on the way it is used.

In the following definition, the branches have different types.

$test\ c = \textbf{if}\ c\ \textbf{then}\ [1\,..\,10]\ \textbf{else}\ \texttt{"abc"}$

Neither of the two branches is more likely to contain the error. Therefore, we report both branches without indicating which should be changed.

```
(1,10): Type error in branches of conditional
expression  : if c then [1..10] else "abc"
then branch : [1..10]
  type      : [Int]
else branch : "abc"
  type      : String
```

An example from the collection of logged programs is the following.

$simplify :: Prop \to Prop$
$simplify = (...)$
$simplifyAnd :: [Prop] \to [Prop]$
$simplifyAnd\ (p : ps) = [simplify\ p, simplifyAnd\ ps]$

yields the error message

```
(5,22): Type error in list (elements have different types)
  expression  : [simplify p, simplifyAnd ps]
  1st element : simplify p
  type       : Prop
 2nd element : simplifyAnd ps
  type       : [Prop]
```

which simply lists all the participating uses and the types inferred for these uses and leaves putting the blame in the hands of the programmer.
Without the unifier heuristic, Helium returns the following message

```
(5,22): Type error in element of list
 expression       : [simplify p, simplifyAnd ps]
 term             : simplifyAnd ps
   type           : [Prop]
   does not match : Prop
```

which puts the blame squarely on the second element in the list.

Our last example shows that even if we want to put blame on one of the cases, we can still use the other cases for justification.

The following definition contains a type error.

```
maxOfList :: [Int] → Int
maxOfList []      = error "empty list"
maxOfList [x]     = x
maxOfList (x, xs) = x ‘max‘ maxOfList xs
```

A considerable amount of evidence supports the assumption that the pattern (x, xs) in $maxOfList$'s third function binding is in error: the first two bindings both have a list as their first argument, and the explicit type expresses that the first argument of $maxOfList$ should be of type $[Int]$. In a special hint we enumerate the locations (1,14), (2,11), (3,11), that support this assumption. Each location consists of a line number, followed by the position on that line.

Lastly, observe that the type variables used to instantiate a type scheme serve the same purpose as a unifier. Hence, we could apply the same techniques to improve error reporting for a polymorphic function. For instance, consider the operator $+\!\!+$, which has type $[a] \rightarrow [a] \rightarrow [a]$. If two operands of $+\!\!+$ cannot agree upon the type of the elements of the list, we could report $+\!\!+$ and its type, together with the two candidate types for the type variable a.

3.3 Program correcting heuristics

A different direction in error reporting is trying to discover what a user was trying to express, and how the program could be corrected accordingly. Given a number of possible edit actions, we can start searching for the closest well-typed program. An advantage of this approach is that we can report locations with more confidence. Additionally, we can equip our error messages with hints how the program might be corrected. However, this approach has a disadvantage too: suggesting program fixes is potentially harmful since there is no guarantee that the proposed correction is the semantically intended one (although we can guarantee that the correction will result in a well-typed program).

Furthermore, it is not immediately clear when to stop searching for a correction, nor how we could present a complicated correction to a programmer.

An approach to automatically correcting ill-typed programs is that of type isomorphisms [11]. Two types are considered isomorphic if they are equivalent under (un)currying and permutation of arguments. Such an isomorphism is witnessed by two morphisms: expressions that transform a function of one type to a function of the other type, in both directions. For each ill-typed application, one may search for an isomorphism between the type of the function and the type expected by the arguments and the context of that function. The heuristics described in this section elaborate on this idea.

The application heuristic Function applications are often involved in type inconsistencies. Hence, we introduce a special heuristic to improve error messages involving applications. It is advantageous to have *all* the arguments of a function available when analyzing such a type inconsistency. Although mapping n-ary applications to a number of binary ones simplifies type inference, it does not correspond to the way most programmers view their programs.

The heuristic behaves as follows. First, we try to determine the type of the function. We can do this by inspecting the type graph after having removed the constraint created for the application. In some cases, we can determine the maximum number of arguments that a function can consume. However, if the function is polymorphic in its result, then it can receive infinitely many arguments (since a type variable can always be instantiated to a function type). For instance, every constant has zero arguments, the function $map :: (a \rightarrow b) \rightarrow [a] \rightarrow [b]$ has two, and the function $foldr :: (a \rightarrow b \rightarrow b) \rightarrow b \rightarrow [a] \rightarrow b$ a possibly infinite number.

If the number of arguments passed to a function exceeds the maximum, then we can report that too many arguments are given – without considering the types of the arguments. In the special case that the maximum number of arguments is zero, we report that *it is not a function*.

To conclude the opposite, namely that not enough arguments have been supplied, we do not only need the type of the function, but also the type that the context of the application is expecting. An example follows.

The following definition is ill-typed: map should be given more arguments (or xs should be removed from the left-hand side).

$$\begin{aligned}&doubleList :: [Int] \rightarrow [Int]\\&doubleList\ xs = map\ (*2)\end{aligned}$$

At most two arguments can be given to *map*: only one is supplied. The type signature for *doubleList* provides an expected type for the result of the application, which is $[Int]$. Note that the first $[Int]$ from the type signature belongs to the left-hand side pattern *xs*. We may report that not enough arguments are supplied to *map*, but we can do even better. If we are able to determine the types inferred for the arguments (this is not always the case), then we can determine at which position we have to insert an argument, or which argument should be removed. We achieve this by unification with *holes*. First, we have to establish the type of *map*'s only argument: $(*2)$ has type $Int \rightarrow Int$. Because we are one argument short, we insert one hole (•) to indicate a forgotten argument.

(Similarly, for each superfluous argument, we would insert one hole in the function type.) This gives us the two configurations depicted in Figure 1.

$$
\begin{array}{llcccc}
\multicolumn{2}{l}{\textbf{configuration 1}:} & & & & \\
 & \mathit{function} & (a \to b) & \to & [a] & \to\ [b] \\
 & \mathit{arguments} + \mathit{context} & \bullet & \to & (\mathit{Int} \to \mathit{Int}) & \to [\mathit{Int}] \\
 & & & & & \\
\multicolumn{2}{l}{\textbf{configuration 2}:} & & & & \\
 & \mathit{function} & (a \to b) & \to & [a] & \to\ [b] \\
 & \mathit{arguments} + \mathit{context} & (\mathit{Int} \to \mathit{Int}) & \to & \bullet & \to [\mathit{Int}]
\end{array}
$$

Fig. 1. Two configurations for column-wise unification

Configuration 1 does not work out, since column-wise unification fails. The second configuration, on the other hand, gives us the substitution $S = [a := \mathit{Int}, b := \mathit{Int}]$. This informs us that our function (*map*) requires a second argument, and that this argument should be of type $S([a]) = [\mathit{Int}]$ (see also Appendix A).

The final technique we discuss attempts to blame one argument of a function application in particular, because there is reason to believe that the other arguments are all right. If such an argument exists, then we put extra emphasis on this argument in the reported error message.

$$
\begin{array}{l}
\mathit{evaluate} :: \mathit{Prop} \to [\mathit{String}] \to \mathit{Bool} \\
\mathit{evaluate}\ (\mathit{And}\ [p : q])\ \mathit{xs} = \mathit{all}\ [p \mid p \leftarrow \mathit{xs}]
\end{array}
$$

```
(2,27): Type error in application
  expression       : all [p | p ← xs]
  term             : all
    type           : (a → Bool) → [a] → Bool
    does not match : [String] → Bool
  probable fix     : insert a first argument
```

The tuple heuristic Many of the considerations for the application heuristic also apply to tuples. As a result, this heuristic can suggest that elements of a tuple should be permuted, or that some component(s) should be inserted or removed.

The permutation heuristic A mistake that is often made is the simple exchange of one or more arguments to a function. The permutation heuristics considers applications which are type incorrect, and tries to determine whether there is a *single* permutation that makes the application correct. For this to work, we need the type of the application expected by the context, and the types of the arguments (if any of these cannot be typed, then it makes no sense to apply this heuristic). By local changes to the type graph, the compiler then determines how many permutations result in a correctly typed application. If there is only one, then a fix to the program is suggested (in addition to the usual error message). If there are more, then we deem it impossible to suggest a probable fix, and no additional hint is given.

$$\begin{aligned}&\mathit{zero} :: (\mathit{Float} \rightarrow \mathit{Float}) \rightarrow \mathit{Float} \rightarrow \mathit{Float}\\ &\mathit{zero}\ f\ \mathit{y0} = \mathit{until}\ (\lambda b \rightarrow b -.\ f\ b\ /.\ \mathit{diff}\ f\ b)\\ &\qquad\qquad\qquad (\lambda b \rightarrow f\ b <. 0.000001)\ \mathit{y0}\end{aligned}$$

with the following error message as a result

```
(2,13): Type error in application
  expression       : until (λb → b −. f b /. ...) (λb → ...) y0
  term             : until
    type           : (a → Bool) → (a → a) → a → a
    does not match : (Float → Float) → (Float → Bool) → Float → Float
  probable fix     : re-order arguments
```

The sibling function heuristic Novice students often have problems distinguishing between specific functions, e.g., concatenate two lists $(+\!\!+)$ and insert an item at the front of a list $(:)$. We call such functions *siblings*. If we encounter an error in an application in which the function that is applied has a sibling, then we can try to replace it by its sibling to see if this solves the problem (naturally only at the type level). This can be done quite easily and efficiently on type graphs by a local modification of the type graph. The main benefit is that the error message may include a hint suggesting to replace the function with its sibling. (Helium allows programmers to add new pairs of siblings, which the compiler then takes into account [6].)

$$\begin{aligned}&\mathit{concat} :: [a] \rightarrow [a]\\ &\mathit{concat}\ [\,]\ \ \ = [\,]\\ &\mathit{concat}\ [a] = \mathit{head}\ [a] +\!\!+\ \mathit{concat}\ (\mathit{tail}\ [a])\end{aligned}$$

with the following error message as a result

```
(3,22): Type error in variable
  expression      : ++
    type          : [a] → [a] → [a]
    expected type : b → [b] → [b]
  because         : unification would give infinite type
  probable fix    : use : instead
```

The sibling literal heuristic A similar kind of confusion that students have is that they mix floating points numbers with integers (in Helium we distinguish the two), and characters with strings. This gives rise to a heuristic that may replace a string literal `"c"` with a character literal `'c'` if that resolves the inconsistency.

$$\begin{aligned}&\mathit{writeRow} :: [[\mathit{String}]] \rightarrow \mathit{Int} \rightarrow \mathit{String}\\ &\mathit{writeRow}\ \mathit{tab}\ n = \textbf{if}\ n == (\mathit{length}\ \mathit{tab} + 3)\ \textbf{then}\ \texttt{""}\\ &\qquad\qquad\qquad\qquad \textbf{else}\ \mathit{replicate}\ (\mathit{columnWidth}\ \mathit{tab}\ n)\ \texttt{"\ \ "} +\!\!+ \texttt{"\ \ "} +\!\!+ \ldots.\end{aligned}$$

results in

```
(3,61): Type error in literal
  expression      : " "
    type          : String
    expected type : Char
  probable fix    : use a char literal instead
```

4 Type graphs

The heuristics of the previous sections share the characteristic that they have all been implemented in Helium as functions that work on type graphs. Essentially, a type graph represents a set of constraints, and as such is similar to a substitution. The main difference is that type graphs can also represent inconsistent sets of constraints.

The type graphs resemble the path graphs that were proposed by Port [12], and which can be used to find the cause of non-unifiability for a set of equations. However, we follow a more effective approach in dealing with derived equalities (i.e., equalities obtained by decomposing terms, and by taking the transitive closure).

McAdam has also used graphs to represent type information [10]. In his case parts of the graph are duplicated to handle let-constructs, which implies a lot of duplication of effort, and, worse, it can give rise to duplication of errors if the duplicated parts themselves are inconsistent. We avoid this complication by first handling the definitions of a let (which gives us the complete types of those definitions), before continuing with the let body. This implies that in case of a mismatch between the definition and the use of an identifier, the blame is always on the latter.

Due to lack of space we only try to convey the essential ideas, intuitions, and features of type graphs and how they may be used. For a complete description we refer the reader to Chapter 7 of the PhD thesis of the second author [5].

In this section we consider a set of equality constraints as a running example and show how type graphs may be used to determine which constraints should be removed to make the set of constraints consistent, resulting in a consistent set of constraints that can then be converted into a substitution (the usual outcome of the type inference process). As explained in Section 2, we may assume that we deal with equality constraints exclusively: polymorphism is handled at a different level.

Consider the following set of equality constraints.

$$\{v_1 \overset{\#0}{\equiv} v_0 \to v_0,\ v_1 \overset{\#1}{\equiv} v_2 \to v_3,\ v_2 \overset{\#2}{\equiv} \mathit{Int},\ v_3 \overset{\#3}{\equiv} \mathit{Bool}\}$$

Annotations like $\#0\ldots$ are used for reference purpose only. For each left and right hand side of a constraint we construct a term graph, which reflects the hierarchical structure of type terms. These term graphs consist of vertices and directed edges, as shown on the left and right hand side of Figure 2. Recall that the type $v_0 \to v_0$ is represented by a binary type application $((\to)\ v_0)\ v_0$, and it is this type that is used in type graph construction. For readability, we continue to refer to $v_0 \to v_0$ in the text.

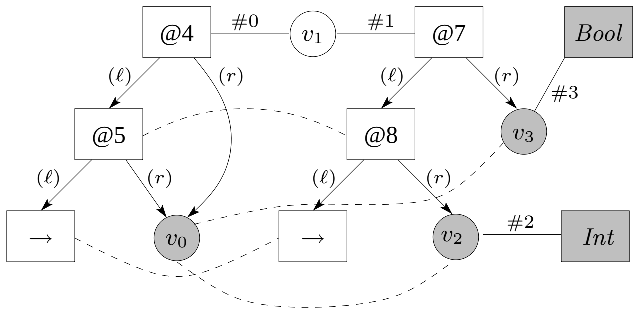

Fig. 2. An inconsistent type graph

The equality constraints between terms is reflected in the introduction of undirected edges in the type graph. Thus each constraint results in a single undirected edge (with its number as a label), called an initial edge. When we equate two structured types, we implicitly equate the subtypes of these types. In the example, v_0 and v_2 become equated, because through v_1, $v_0 \to v_0$ and $v_2 \to v_3$ become equated. This gives rise to the derived edges, occurring as dashed edges in Figure 2. The connected components that arise when considering all vertices that are connected via an initial or derived edge, are called equivalence groups. Clearly, each vertex in an equivalence groups should represent the same type. This is not the case in Figure 2, because Int and Bool end up in the same equivalence group. The paths between such clashing constants are called error paths, which may contain both initial and derived edges. When we encounter such an error path, we unfold the derived edges until we end up with a path that consists solely of initial edges (remember that these relate directly to the constraints from which the type graph was built).

The example type graph has only a single error path, but can in principle contain many. The task of the type graph solver is to dissolve all error paths and it may do so by selecting a constraint from each error path. This is exactly where the heuristics discussed earlier in this paper come in: they operationalize what are the best places to cut. After a set of constraints is selected the removal of which dissolves all error paths, then we can use the resulting type graph to construct a substitution as the end result of the solving process.

In the example, there are a number of possibilities to dissolve the error path. This is generally the case, and the place where the heuristics play a role in selecting the most likely candidate for removal. We can choose to remove any of the four constraints to make the type graph consistent, each choice leading to a substitution obtained from the remaining type graph. For example, if we remove $\#0$, then the resulting substitution maps v_1 to $\mathit{Int} \to \mathit{Bool}$, v_2 to Int, and v_3 to Bool. If we choose to remove $\#3$ instead, then the substitution maps v_0, v_2 and v_3 to Int, and v_1 to $\mathit{Int} \to \mathit{Int}$. In our implementation, the constraint is provided with enough information to be able to generate a precise error message that tells the user why it was removed, in terms of types computed from the remainder type graph. For example, in the latter case it will contrast the type it expected for v_3 which is Int with the type it found for v_3, which is $Bool$.

Thus far, we have explained rather informally how type graphs are built and handled, but in practice there are a number of complications: The number of vertices in a type graph grows quickly, as does the number of derived edges. The number of error paths in any given type graph can be very large, even when one disregards error paths that may be considered superfluous. Furthermore, how can one effectively deal with infinite types, which occur as a result of constraints such as $v_1 \equiv v_1 \rightarrow Int$? How does one deal with type synonyms, that introduce new type constants as abbreviations for existing types? Detailed descriptions of solutions to these complications can be found in [5].

5 Putting it all together

The Helium compiler includes all the heuristics we have discussed (and more), and has been used for a number of years to teach students to program in Haskell. Reactions in the first year were very promising (some of these students had used Hugs before and indicated that the quality of error messages was much improved). Since then we have improved the compiler in many ways, adding new language features and new heuristics. Unfortunately, the students who currently do the course have never encountered any other system for programming in Haskell and thus cannot compare their experiences. For completeness, we have included a sample trace of the execution of the Helium compiler in Appendix A (with highly verbose output concerning the type inference process). It shows in detail what the effect is of applying the various heuristics. The Helium compiler itself is available for download to anyone interested in further experimentation [7].

Another issue we would like to address here is that of efficiency of the compiler. We have constructed a special kind of solver that partitions the program into a number of relatively independent chunks (in a first approximation every top level definition is a chunk), applies a fast greedy solver to each, and only when it finds a type error in one of the chunks, does it apply the slower but more sophisticated type graph solver to this erroneous chunk (but *not* to the foregoing chunks). This means that the type graph solver is only used when a type error is encountered, and only on a small part of the program. Additionally, there is a maximum to the number of error paths that the type graph solver will consider in a single compile. Still, constructing and inspecting a type graph involves additional overhead, which slows down the inference process. In a practical setting (teaching Haskell to students), we have experienced that the extra time spent on type inference does not hinder programming productivity.

To give the reader some idea how the ideas of the previous section take form in an actual compiler, we have included the function $listOfHeuristics$ in Figure 3. It takes a (partially user specified) list of siblings [6] to generate the list of available heuristics for this compilation.

Each heuristic can be categorized as either a filtering heuristic or a selector heuristic. The $avoidTrustedConstraints$ is an example of the former: it filters out all the constraints from the candidate set that have a high trust value, thus making sure that these are never reported. Note that $avoidForbiddenConstraints$ avoids constraints of

```
listOfHeuristics siblings path =
  [avoidForbiddenConstraints
  , highParticipation 0.95 path
  , Heuristic (Voting
    [siblingFunctions siblings
    , similarNegation
    , siblingLiterals
    , applicationHeuristic
    , variableFunction
    , tupleHeuristic
    , fbHasTooManyArguments])
  , avoidApplicationConstraints
  , avoidNegationConstraints
  , avoidTrustedConstraints
  , avoidFolkloreConstraints
  , firstComeFirstBlamed]
```

Fig. 3. The list of heuristics taken from the Helium compiler

the sort described under (1) of the trust factor heuristic, only (3) and (4) are part of *avoidTrustedConstraints* (case (2) is already taken care of by our choice that the use of an identifier can never influence its type). It is easy to make the distinction between selectors and filters in *listOfHeuristics*: all the heuristics that are part of the *Voting* construct in the middle are selectors, the others are filters.

A voting heuristic is built out of a number of subsidiary heuristics, each of which looks to see whether it can suggest a constraint likely to be responsible for the type inconsistency. Each voting heuristic also returns a value that gives a measure of trust the heuristic has in its suggestion. Based on these measures the combined voting heuristic will decide which constraint to select, if any.

Most of the heuristics in Figure 3 are connected directly with heuristics discussed in the paper. There are a few special cases, however: *variableFunction* has largely the same functionality as the *applicationHeuristic*, but the latter is only triggered on applications (a function followed by at least one argument). Instead, *variableFunction* is triggered on identifiers that have a function type, but that do not have arguments at all. It may for instance suggest to insert certain arguments to make the program type correct. Another thing to remark is that the permutation of arguments in applications is implemented as part of the *applicationHeuristic* as well.

The heuristic *similarNegation* provides the same functionality as *siblingFunctions*, but specifically for the negation function, which is a syntactic construct in Haskell and must be treated somewhat differently. The heuristic *fbHasTooManyArguments* tries to discover whether the type inconsistency can be explained by a discrepancy between the number of formal arguments, and the expected number of arguments derived from the function's explicit type signature.

The heuristics in the final block, starting with *avoidApplicationConstraints* are low priority heuristics that are used as tie-breakers. For these, it should be noted that *avoidNegationConstraints* provides the same functionality as *avoidApplicationConstraints*, but specifically for negation (which, as explained before, is not a function in Haskell, but a syntactic construct).

The function that applies the list of heuristics starts with a set of constraints that lie on an error path. It considers the heuristics in $listOfHeuristics$ in sequence. A filtering heuristic may remove any number of candidates from the set, but never all. If a constraint is selected by a selector heuristic, all other constraints will be removed from the set of candidates leaving only the selected constraint.

6 Validation and statistics

The existence of an actual implementation of our work immediately raises another issue: by means of this implementation it should be possible to establish whether the implemented heuristics are effective. Indeed, the "quality" of a type error message is not likely to get a precise definition any time soon, which means that the usability of Helium can only be verified empirically. However, to perform such experiments is a difficult problem in itself and beyond the scope of this paper. Note though that we have initiated this particular path of research, of which the first results can be found in a technical report [4].

In this paper we take a different approach and present a number of statistics computed from programs collected by logging Helium compilations in a first year programming course. Each logging corresponds to a unique compile performed by a student in the student network: this allows us to reconstruct the compile exactly. More information about the logging process can be found in a technical report [2]. We use the data sets collected for the course year 2004-2005, which include a total of $11,256$ loggings of which $3,448$ resulted in one or more type errors. In total, the type incorrect programs produced $5,890$ type error messages.

For the heuristics described in the previous sections, Figure 4 shows how often each contributed to eliminating candidate constraints, and in how many cases it not only contributed, but was decisive in bringing the number of candidates down to one. In other words, it was responsible for selecting the constraint to be removed and as such strongly influences the error message reported to the programmer. Note that the contributing count includes the deciding count. One thing that can be noted from the results is that the tuple heuristic and the special heuristics for negation are hardly used. The reason for this is that the programming assignments in 2004/2005 did not call for heavy use of tuples and negation.

Figure 5 focuses on the type of probable fixes given to the programmer. Of the 5,890 error messages, a total of 1,116 actually gave such a probable fix (in addition to the standard error message). The table is more detailed in the sense that for example the application heuristic in Figure 4 may result in a variety of probable fixes: re-order arguments, insert missing argument and so on. On the other hand, some of the fixes suggested by the variable function heuristic are the same as those of the application heuristic. As explained before, the variable function heuristic is conceptually the same as the application heuristic. For reasons of brevity, we have kept the table compact, lumping a number of similar probable fixes of lesser frequency together, for example "insert a first and second argument" falls into the category of "insert a first/second/...

heuristic	type	contributing	deciding
Avoid forbidden constraints	filter	3756	22
Participation ratio (ratio=0.95)	filter	3791	202
Function siblings	selector	479	433
Similar negation	selector	0	0
Literal siblings	selector	196	145
Application heuristic	selector	2229	1891
Variable function	selector	123	111
Tuple heuristic	selector	5	5
Function binding has too many arguments	selector	35	35
Avoid application constraints	filter	726	15
Avoid negation constraints	filter	0	0
Avoid trusted constraints	filter	2371	1146
Avoid folklore constraints	filter	1298	922
First come, first blamed	filter	963	963

Fig. 4. The frequency of heuristics

argument". We do make the distinction between "insert a first argument" and "insert one argument". In the former case, the compiler was able to conclude unambiguously that the first argument was missing.

After running the experiments with the above results, we included the $unifierHeuristic$, which has been an experimental option in Helium for some time. The implemented instance of the $unifierHeuristic$ considers only the situation in which a unifier was used to unify two different types, with the result that the error message obtains a neutral, unbiased error message.

7 Related work

There is quite a large body of work on improving type error messages for polymorphic, higher-order functional programming languages such as Haskell, cf. [14, 12, 10, 11, 15]. The drawback of these papers is that they have not led to full scale implementations and in many cases disregard issues such as efficiency and scalability. Since we refer to the articles who have influenced our choice of heuristic where we discuss the heuristic, we shall consider only some of the more current approaches in the remainder of this section.

In recent years, there is a trend towards implementation. One of these systems is Chameleon [13] which is an interactive system for type-debugging Haskell. The viewpoint here is that no static type inference process will come up with a good message in every possible situation. For this reason, they prefer to support an interactive dialogue to find the source of the error. A disadvantage of such a system is that is not very easy to use by novice programmers, and more time consuming as well. An advantage is that the process itself may give the programmer insight into the process of type inferencing, helping him to avoid repeating the mistake. As far as we know, Chameleon has not been used on groups of (non-expert) programmers. The approach taken in Chameleon,

probable fix	generated by	frequency
insert a first/second/... argument	application/variableFunction	142
insert one/two/three/... argument(s)	application/variableFunction	107
remove a first/second/... argument	application	139
swap the two arguments	application	57
re-order arguments	application	56
re-order elements of tuple	tuple	3
use a char/int/float/string literal instead	sibling literals	154
use ++ instead	sibling functions	100
use : instead instead	sibling functions	142
use concatMap instead	sibling functions	62
use eqString instead	sibling functions	45
other sibling fixes	sibling functions	109

Fig. 5. Probable fix frequency for the loggings of 2004/2005

to consider sets of minimal corresponds closely to the error slicing approach of Haack and Wells [1].

Ideally, a compiler provides a combination of feedback and interaction: if the provided heuristics are reasonably sure that they have located the source of error, then a type error message may suffice, otherwise an interactive session can be used to examine the situation in detail. Our unifier heuristic occupies a middle point: it makes no judgment on who is to blame, but only describes which types clash and where they arise from. It only applies if there is no overwhelming amount of evidence against one of the candidates for removal (for a particular choice of "overwhelming").

Finally, our focus on expert knowledge was inspired by work of Jun, Michaelson, and Trinder [16]. Their idea of interviewing experts has appeal, but a drawback of their work is that the resulting algorithm $\mathcal{H}$ is very incomplete (only 10 out of 40 rules are given), and we have not been able to find an implementation.

8 Conclusion and future work

We have discussed heuristics for the discovery of and the recovery from type errors in Haskell. Knowledge of our problem domain allows us to define special purpose heuristics that can suggest how to change parts of the source program so that they become type correct. Although there is no guarantee that the hints always reflect what the programmer intended, we do think that they help in many cases. Moreover, we have shown that it is possible to integrate various heuristics known from the literature with our own resulting in a full scale, practical system that can be easily extended with new heuristics as the need arises. We have applied our compiler to a large body of programs that have been compiled by students during a first year functional programming course, resulting in information about the frequency of hints and particular heuristics. Many of the examples in the paper are taken from this body of programs, lending additional strength to our work.

We are currently proceeding along several lines: the first is doing a quantitative analysis of the effect of hints on program productivity (based on programming sessions logged by the compiler) [4]. A second project continues the work on rearranging abstract syntax trees so that they become type correct [8].

References

1. Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Symposium on Programming*, pages 284–301, April 2003.
2. J. Hage. The Helium logging facility. Technical Report UU-CS-2005-055, Institute of Information and Computing Sciences, Utrecht University, 2005.
3. J. Hage and B. Heeren. Ordering type constraints: A structured approach. Technical Report UU-CS-2005-016, Institute of Information and Computing Science, Utrecht University, Netherlands, April 2005. Technical Report.
4. J. Hage and P. van Keeken. Mining for Helium. Technical Report UU-CS-2006-047, Institute of Information and Computing Sciences, Utrecht University, 2006.
5. B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/phdthesis`.
6. B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
7. B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.
8. A. Langebaerd. Repair systems, automatic correction of type errors in functional programs. `http://www.cs.uu.nl/wiki/Top/Publications`.
9. O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, July 1998.
10. B. J. McAdam. Generalising techniques for type debugging. In P. Trinder, G. Michaelson, and H-W. Loidl, editors, *Trends in Functional Programming*, volume 1, pages 50–59, Bristol, UK, 2000. Intellect.
11. B. J. McAdam. How to repair type errors automatically. In Kevin Hammond and Sharon Curtis, editors, *Trends in Functional Programming*, volume 3, pages 87–98, Bristol, UK, 2002. Intellect.
12. G. S. Port. A simple approach to finding the cause of non-unifiability. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 651–665, Seatle, 1988. The MIT Press.
13. P. J. Stuckey, M. Sulzmann, and J. Wazny. Interactive type debugging in Haskell. In *Haskell'03: Proceedings of the ACM SIGPLAN Workshop on Haskell*, pages 72–83, New York, 2003. ACM Press.
14. J. A. Walz and G. F. Johnson. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 44–57, St. Petersburg, FL, January 1986.
15. J. Yang. Explaining type errors by finding the sources of type conflicts. In Greg Michaelson, Phil Trindler, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.
16. J. Yang, G. Michaelson, and P. Trinder. Explaining polymorphic types. *The Computer Journal*, 45(4):436–452, 2002.

A A sample trace of the compiler

The following information can be easily obtained from a simple run of the Helium compiler downloadable from the website. We give it here to simplify the task of the reviewers. It is not intended for publication.

We are now ready to give a sample run of our compiler on the program

$$doubleList :: [Int] \to [Int]$$
$$doubleList\ xs = map\ (*2)$$

The result of running *helium -d DoubleList.hs* (the *-d* flag is responsible for the very verbose output which shows what is happening under the hood of the type inference process) is:

```
Compiling DoubleList.hs
(2,12): Warning: Variable "xs" is not used
---------------------------------------------------
Constraints
-----------
v0 == v2 -> v1 : {function bindings, #0}
MakeConsistent
v0 := Skolemize([], [Int] -> [Int]) :
     {explicitly typed binding, #1}
v3 == v2   : {pattern of function binding, #2}
v5 := Inst(forall a b . (a -> b) -> [a] -> [b]) :
     {variable, #3}
v9 := Inst(Int -> Int -> Int) : {variable, #4}
Int == v10   : {literal, #5}
v9 == v8 -> v10 -> v7 : {infix application, #6}
v8 -> v7 == v6 : {left section, #7}
v5 == v6 -> v4 : {application, #8}
v4 == v1 : {right-hand side, #9}
(11 constraints, 0 errors, 0 checks)

CombinationSolver:
  GreedySolver: found 1 errors
  Switching to second solver
---------------------------------------------------

Error path found with constraints:
   (#1, #0, #9, #8, #3)

After filtering out the forbidden constraints:
   {"#1","#9","#3","#8"}

cnr  edge       ratio   info
--------------------------------------------------
#1*  (0-22)     100%    {explicitly typed binding}
#9*  (1-4)      100%    {right-hand side}
#3*  (5-37)     100%    {variable}
#8*  (5-59)     100%    {application}

Participation ratio [ratio=0.95] (filter)
   {"#1","#9","#3","#8"}
Highest phase number (filter)
   {"#1","#9","#3","#8"}
Voting with 7 heuristics
- Sibling functions (selector)
- Sibling literals (selector)
- Application heuristics (selector)
     not enough arguments are given.
     Two were expected, one was given.
     Selected #8, {"(5-59)"} with priority 4.
```

```
- Tuple heuristics (selector)
- Function binding heuristics (selector)
- Variable function (selector)
- Unification vertex (selector)

*** Selected with priority 4:
  constraint #8 / edge {"(5-59)"}

Avoid application constraints (filter)
   {"#8"}
Avoid negation edge (filter)
   {"#8"}
Avoid trusted constraints (filter)
   {"#8"}
Avoid folklore constraints (filter)
   {"#8"}
First come, first blamed (filter)
   {"#8"}

*** The selected constraint: #8 ***

---------------------------------------------------

(2,17): Type error in application
expression       : map (* 2)
term             : map
   type           : (a   -> b  ) -> [a] -> [b]
   does not match : (Int -> Int) -> [Int]
probable fix     : insert a second argument

Compilation failed with 1 error
```

The first thing to notice is the effect of using a combined solver: first a greedy solver is tried. This results in the discovery of a type inconsistency, after which the same set of constraints is submitted to the type graph solver. It builds the type graph, discovers the error path, sets the candidate set equal to the constraints on the error path, after which it applies the heuristics.

In the example, $\#0$ is the only forbidden constraint, and therefore it is removed from the candidate set.

Then the partipication ratio filter leaves only those constraints that occur in a large enough percentage of error paths (usually we take this value equal or very close to 100%). In this case, all constraints participate in every error path, so all are kept.

Subsequently, the voting process is initiated, in which seven heuristics are used. Only the application heuristic of Section 3.2 leads to a constraint being selected, $\#8$ which labels the edge from node 5 to 59 in the type graph. The reason is that it could determine that not enough arguments were given to map. Because a single constraint could be selected, the other candidates are removed from the candidate set. The subsequent filtering heuristics are considered, but they do not change the candidate set. Finally, an error message is displayed that indeed reflects the fact that map expects a second argument. This fact is stressed by the 'probable fix' appended at the end of the message.