Heuristics for type error discovery and recovery (revised)

Jurriaan Hage

Bastiaan Heeren

institute of information and computing sciences, utrecht university

technical report UU-CS-2006-007

www.cs.uu.nl

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.

Keywords: *type inferencing, type graph, constraints, heuristics, error messages, error recovery*

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 years we have developed the TOP framework to support flexible and customizable type inference. This framework has been used to build the Helium compiler [5], 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 [5].

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 [2]), 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. Moreover, type graphs naturally support heuristics, which embed expert knowledge in explaining type errors. The resulting type graph solver is less sensitive to the order of the constraints.

A major advantage of type graphs is that it is relatively easy to define heuristics for guiding the construction of error messages. 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 [11]. In addition, we have defined a number of our own heuristics. 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.

Many 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 as follows: 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 full courses of functional programming at Universiteit Utrecht comprising several hundreds of students.

This paper is organized as follows. In the next section we set the scene and introduce the constraints we will use. Then we introduce type graphs in Section 3, after which we consider each heuristic in turn in Section 4. In Section 5 we consider related work and the validation and implementation of our work. The appendix includes a sample trace of the compiler.

2. Constraints

Our type language has monomorphic types (τ) and type schemes (σ). Type schemes are used to capture polymorphic types such as $\forall a.a \rightarrow a$, i.e., the type of the identity function. The monomorphic types are type variables ($v_1, v_2, \ldots$), type constants (the primitive types like Int, but also type constructors, like $\rightarrow$ for function types), or the application of a type to another. For example, the type of functions from integers to booleans is written $(((\rightarrow)Int)Bool)$. Type application is left-associative, and we omit parentheses where

allowed. We often write the function constructor infix, resulting in $Int \rightarrow Bool$. We assume the types are well-kinded: types like $Int\ Bool$ and $\rightarrow Int$ do not occur.

The Hindley-Milner type system is based on performing unification of types. These can be readily expressed using equality constraints on types: $\tau_1 \equiv \tau_2$. Although equality constraints suffice for dealing with polymorphism, there are good reasons to have special constraints for modeling it. For the expression

$$\begin{array}{l}\textbf{let}\ f\ x\ y = \textbf{if}\ x\ \textbf{then}\ 0\ \textbf{else}\ x \\ \textbf{in}\ (f\ \ True\ False, f\ 2\ 3)\end{array}$$

we generate a set of constraints C_f for the definition of f, which, for example, tells us that x should be of type $Bool$, because of its use in the condition, but also that x should be of type Int, because the types of the two branches of the conditional should be equal. The lack of constraints on y tells us that f is in fact polymorphic in y: an argument of any type will do.

A simple way to handle this is to duplicate the set of constraints C_f and to solve these separately for each use of f: the soundness of this method is a consequence of the semantics of the let, which is that each use of f may be replaced by a copy of its definition. This, however, has consequences for the type error messages: the set C_f is inconsistent, which means that the inconsistency is duplicated as well. And even if C_f is consistent, then we have just doubled the amount of work.

This led Damas and Milner to come up with algorithm $\mathcal{W}$ which essentially first computes the type of f (generalizing it appropriately to a type scheme), and only then continues with the body, giving each use of f its own instance [1]. Because of the decoupling of the generation of constraints from the solving of constraints, we need a special kind of implicit instance constraint which essentially administers the fact that the uses of f in the body of the let should be instances of the type which will, at some point, be found for f. To make this work, we insist that the constraints in C_f are solved before constraints arising from the body are considered: first C_f will be made consistent, resulting in a (polymorphic) type for f. This type will be propagated into the constraints generated for the body of the let, which allows us to transform the implicit instance constraints into equality constraints, after which we can make these consistent as well. Due to space restrictions, we refer the reader to [3] for more details of this process.

3. Type graphs

For constructing high quality type error messages, it is crucial to have as much information as possible available. Type graphs store information about each unification, and in our implementation the constraints themselves additionally keep track of a lot of information, e.g., location information that refers back to the location in the source code from which the constraint arose. Type graphs prevent the introduction of bias, because a set of equality constraints can be solved 'at once'. This is possible because type graphs can represent inconsistent sets of constraints. This gives us strictly more information in comparison to more traditional approaches, that compute types on the fly: after unification of two types, the fact that this unification has taken place and which types were unified is lost.

The type graphs presented in this paper resemble the path graphs that were proposed by Port [9], 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). Besides, we have a special interest in type inference and type error messages, and formulate special-purpose heuristics. McAdam has also used graphs to represent type information [7]. 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.

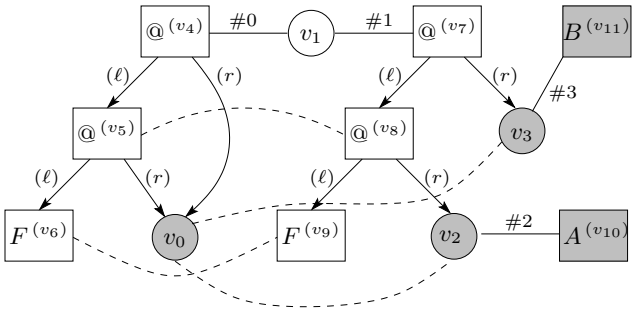

Figure 1. An inconsistent type graph

Constructing a type graph

A type graph for a given set of constraints is constructed by considering each constraint in the set in turn. We use the following constraint set as example.

$$\{v_1 \overset{\#0}{\equiv} F\ v_0\ v_0,\ v_1 \overset{\#1}{\equiv} F\ v_2\ v_3,\ v_2 \overset{\#2}{\equiv} A,\ v_3 \overset{\#3}{\equiv} B\}$$

Annotations like $\#0 \ldots$ are used for reference purpose only.

We use the constraint $v_1 \equiv F\ v_0\ v_0$ to illustrate how the type graph is constructed: term graphs are constructed for both the left-hand side type and right-hand side type of the equality constraint. The term graph for a type variable, like v_1, is a single vertex: this vertex is shared by all occurrences of this type variable in the constraint set. The term graph for a type constant, like F, is also a single vertex, which we annotate with the constant. For each occurrence of this constant in the constraint set, we introduce a new vertex. In case of a composite type ($(F\ v_0)\ v_0$), we first construct term graphs for the two subterms, $F\ v_0$ and v_0. Then, we introduce a new vertex for the composite type, and we add directed edges (*child edges*) to indicate the parent-child relation between the vertices. These edges are labeled with (ℓ) or (r) for the left and the right subterm respectively. See Figure 1 in which a term graph for the type $F\ v_0\ v_0$ is given to the left (and, similarly, one for $F\ v_2\ v_3$). Note the use of the label $\#0$ to show which constraints gives rise to a given edge.

Here, vertices labeled with $(@)$ correspond to type application in composite types. An edge is inserted between the two vertices that are the roots of the term graphs just constructed. We call such an edge an *initial edge*, since it represents type equality imposed directly by a single type constraint. Additional information that is supplied with a constraint is stored with the edge. *Equivalence groups* are the connected components of a type graph when we do not take the child edges into account. The vertices of an equivalence group are supposed to correspond to the same type.

The insertion of an initial edge in the previous step may cause two equivalence groups to be merged. For all pairs of composite types that are in the same equivalence group, we have to propagate equality to the children. In Figure 1, v_5 and v_8 are connected via an *implied* (dashed) edge, because the former is a left child of v_4, the latter is a left child of v_7, which reside in the same equivalence group. Insertion of implied equality edges may cause other equivalence groups to be merged. This, again, may result in the insertion of other implied edges, and so on. Note that for reasons of efficiency in dealing with the large clique-like subgraphs of implied edges, our actual implementation uses a special encoding [3].

3.1. Analyzing the type graph

Equality paths and error paths

The type graph of Figure 1 has four equivalence groups, including one that consists of all the shaded vertices. This group contains both type constants A and B, which indicates that the constraint set is inconsistent.

An *equality path* between two constants is a path consisting of initial and implied edges, that witnesses the supposed equality. We want to put the blame only on equality constraints from which the type graph was constructed. Each initial edge corresponds directly to such an equality constraint, but for implied edges we have to trace why such an edge was inserted in the type graph. For this reason, we expand equality paths to paths that contain only initial constraints. Expanding a path entails replacing its implied edges by the equality paths between the two parent vertices that were responsible for adding the implied edge in the first place. Repeatedly replacing implied edges yields a path without implied edges.

To denote an expanded equality path, we use the annotations $\mathit{Up}_i^{(\delta)}$ and $\mathit{Down}_i^{(\delta)}$, where δ is either ℓ (left child) or r (right child). The annotation *Up* corresponds to moving upwards in the term graph by following a child edge in the opposite direction, whereas *Down* corresponds to moving downwards (from parent to child). Each *Up* annotation in an equality path should have a *Down* annotation at a later point (otherwise this implies the existence of an infinite path, see later in this section), which we make explicit by assigning unique *Up*-*Down* pair numbers, written as subscript. These emphasize the stack-like behavior of *Up*-*Down* pairs, and serve no other purpose.

Consider Figure 1 again, and in particular the error path π from the type constant $A^{(v_{10})}$ to the type constant $B^{(v_{11})}$ (via the type variable v_0). Expanding the implied edge between v_2 and v_0 yields a path that contains the implied edge between $@^{(v_8)}$ and $@^{(v_5)}$. Expansion of this implied edge gives the path between $@^{(v_7)}$ and $@^{(v_4)}$, which consists of two initial edges. Hence, we get the path $[\#2, \mathit{Up}_0^{(r)}, \mathit{Up}_1^{(\ell)}, \#1, \#0, \mathit{Down}_1^{(\ell)}, \mathit{Down}_0^{(r)}]$ after expanding the path of initial and implied edges between $A^{(v_{10})}$ and v_0. Similarly, we expand the path between v_0 and $B^{(v_{11})}$. The expanded error path π is now:

$$[\ \ \#2, \mathit{Up}_0^{(r)}, \mathit{Up}_1^{(\ell)}, \#1, \#0, \mathit{Down}_1^{(\ell)}, \mathit{Down}_0^{(r)}, \\ \mathit{Up}_2^{(r)}, \#0, \#1, \mathit{Down}_2^{(r)}, \#3\ \ ].$$

Both $\#0$ and $\#1$ appear twice in π. Note that by following $\mathit{Up}_2^{(r)}$ from v_0, we can arrive at either v_4 or v_5. In general, *Up* annotations do not uniquely determine a target vertex. This ambiguity can be circumvented straightforwardly by including a target vertex in each *Up* annotation.

Note that we may ignore all error paths that are implied by a smaller error path: removing a constraint on the smaller path removes both error paths at the same time. Similarly, we avoid analyzing detour equality paths: such a path contains two consecutive implied edges from the same clique, i.e., a subgraph in which all vertices are connected to each other.

Infinite types and paths

There is a another category of error paths, which are closely related to the *occurs check* found in unification algorithms. Such a path starts and ends in the same vertex v, and may contain any number of equality edges (both initial and implied), and at least one edge from parent to child, without a matching edge in the opposite direction. Such a path, an *infinite* path, is a proof that v represents an infinite type (such types are forbidden in Haskell).

Consider the following set of type constraints (see Figure 3 for the type graph).

$$\{v_0 \overset{\#0}{\equiv} G\ v_1, v_0 \overset{\#1}{\equiv} G\ v_2, v_1 \overset{\#2}{\equiv} G\ v_2\}$$

From the first two constraints ($\#0$ and $\#1$) we conclude that v_1 and v_2 should be the same type, but the third constraint ($\#2$) contradicts this conclusion. Starting in v_1, we follow edge $\#2$ and arrive at v_2 by taking the right-child edge of the application vertex. The implied equality edge brings us back to our starting point v_1. After expansion of this implied edge, we get the following error path π.

$$\pi = [\#2, \mathit{Down}_\infty^{(r)}, \mathit{Up}_0^{(r)}, \#1, \#0, \mathit{Down}_0^{(r)}]$$

The one child edge followed downwards with no matching upward child edge is annotated with ∞.

Infinite paths can be found by analyzing the parent-child dependencies between the equivalence groups of a type graph: map all vertices belonging to the same equivalence group to a single vertex, while retaining the adjacencies of the child edges. In the example, we obtain a vertex that represents the group $\{v_1, v_2, v_8\}$ which has a loop, because of the child edge from v_8 to v_2. The existence of this cycle implies that an infinite path is present in the type graph.

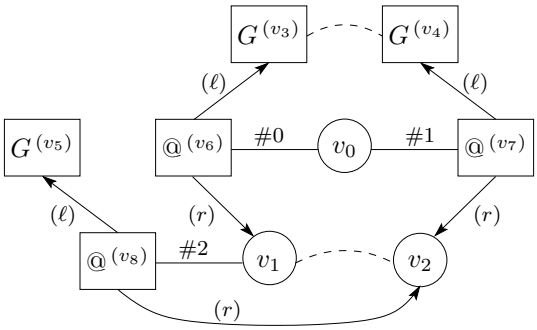

Figure 2. A type graph with an infinite path

A type graph as a substitution

To make a type graph consistent, we first determine all (expanded) error paths, and then remove at least one initial edge from each path. When we remove an initial edge from the type graph, all implied edges that rely on this initial edge disappear. Type graphs naturally support multiple type error messages in a single compilation and may resolve any number of independent error paths. However, since the number of error paths is potentially huge, we imposed a fixed limit on the amount of error paths it considers. This avoids long compile times, and we do not think a programmer is really interested in obtaining more than a dozen type error messages.

A consistent type graph represents a substitution. Given a vertex v in an equivalence group E, the type (or type variable) associated with v can be determined as follows.

- If E has exactly one type constant and no application vertices, then this type constant is the type we assign to v.
- If E has no type constants, but there is at least one application vertex, then the type associated with v is a composite type. Choose one left child of one of the application vertices in E (say v_0) and one right child (say v_1). Now, assign to v the application of the type associated with v_0 to the type associated with v_1. The absence of infinite paths ensures that this process terminates.
- If E has no type constants and there is no application vertex in E, then a type variable is chosen to represent all vertices of E.

We revisit the example of Figure 1 which has a single error path:

$$\begin{aligned}\pi = \quad & [\#2, \mathit{Up}_0^{(r)}, \mathit{Up}_1^{(\ell)}, \#1, \#0, \mathit{Down}_1^{(\ell)}, \mathit{Down}_0^{(r)}, \\ & \mathit{Up}_2^{(r)}, \#0, \#1, \mathit{Down}_2^{(r)}, \#3].\end{aligned}$$

We can choose to remove any of the four constraints on π 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 $F\ A\ B$, v_2 to A, and v_3 to B. If we choose to remove $\#3$ instead, then the substitution maps v_0, v_2 and v_3 to A, and v_1 to $F\ A\ A$.

Every constraint that is removed results in a single error message to be reported to the user (the part on heuristics will show some typical examples). The content of the message is determined by which constraint we choose to remove.

4. Heuristics

In principle, all the constraints that are on an error path are candidates for removal. However, some constraints are better candidates for removal than others. To select the "best" candidate for removal, we consult a number of type graph heuristics. These heuristics are mostly 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 initial edge removed from the type graph, we create one type error message using the constraint information stored with that edge. 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 can override all others (for example, the user-defined specialized type rules [4]), while a collection of others, the tie-breakers, are only considered if none of the other heuristics came up with a suggestion.

A final 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.

4.1. 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 error path, then it is a better candidate for removal than a constraint appearing in just one error path. The set of candidates is thus reduced to the constraints that occur most often in the error paths. This heuristic is driven by a ratio r (typically at least 95%): only constraints that occur in r percent of the error paths are retained as candidates.

Note that this heuristic also helps to decrease the number of reported error messages, as multiple error paths disappear 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 [11]: 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. By itself, this heuristic usually does not reduce the set of candidates to a singleton.

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 [2] 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.

4.2. 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 five 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.

$$\begin{aligned} &squares :: Int \\ &squares = \mathbf{let}\ f\ i = i * i \\ &\qquad\qquad \mathbf{in}\ map\ f\ [1\,..\,10] \end{aligned}$$

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 $\mathcal{M}$ [6] 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 $String$.

$$\begin{aligned} &test :: Bool \to String \\ &test\ b = \mathbf{if}\ \texttt{"b"}\ \mathbf{then}\ \texttt{"yes!"}\ \mathbf{else}\ \texttt{"no!"} \end{aligned}$$

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 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

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

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

The application heuristic (a program correcting heuristic discussed in Section 4.3) 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 context, 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. It is important to realize that this heuristic may only be applied after the application heuristic to be described later on.

Unifier 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 [12].

The design of our type rules (Chapter 6 of [3]) 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.

For example, 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 example above, the type of the context is also a determining factor. 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.

$$\begin{aligned} &maxOfList :: [Int] \to Int \\ &maxOfList\ [\,] \quad\ = error\ \texttt{"empty list"} \\ &maxOfList\ [x] \quad\ = x \\ &maxOfList\ (x, xs) = x\ `max`\ maxOfList\ xs \end{aligned}$$

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.

4.3. 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.

One approach to automatically correcting ill-typed programs is based on a theory of type isomorphisms [8]. 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, we search for an isomorphism between the type of the function and the type expected by the arguments and the context of that function. We illustrate this idea with a simple example.

The definition of $squareList$ is not well-typed, although we supply a list to map, and a function that works on the elements of this list.

$$
\begin{aligned}
&square :: Int \to Int \\
&square\ i = i * i \\
\\
&squareList :: Int \to [Int] \\
&squareList\ n = map\ ([1\,..\,n], square)
\end{aligned}
$$

To correct the application $map\ ([1\,..\,n], square)$, we search for an adapted version of map which expects its arguments paired and in reversed order. Consider the following two morphisms between the real type of map (on the left), and its expected type (on the right).

$$(a \to b) \to [a] \to [b] \underset{\mu_2}{\overset{\mu_1}{\rightleftarrows}} ([a], a \to b) \to [b]$$

where $\mu_1 = \lambda f\ (xs, g) \to f\ g\ xs$ and $\mu_2 = \lambda f\ g\ xs \to f\ (xs, g)$. In this case, the type variables a and b are both *Int*. Note that applying μ_1 to map corrects the type error, and by partial evaluation of μ_1 we can obtain the corrected expression $map\ square\ [1\,..\,n]$.

A second possibility is to change the structure of an abstract syntax tree by inserting or removing parentheses. Beginning Haskell programmers have a hard time understanding the exact priority and associativity rules for operators and applications. As a result, every now and then, a pair of parentheses is missing, which often results in a type error. This led us to search for slightly modified well-typed abstract syntax trees. The following definition is not type correct.

$$
\begin{aligned}
&isZero :: Int \to Bool \\
&isZero\ i = not\ i == 0
\end{aligned}
$$

In this definition, not of type $Bool \to Bool$ is applied to i, which is (probably) of type Int, because of the declared type. By looking at the constituents of $not\ i == 0$, we learn that $not\ (i == 0)$ is the only well-typed arrangement. Hence, we can suggest the programmer to insert parentheses at these locations. Note that the type signature supports this rearrangement, which increases the confidence that this is the correction we want.

Clearly, a combination of the two methods just described allow us to suggest complex sequences of edit operations. However, the more complicated our suggestions become, the less likely it is that it makes sense to the programmer.

configuration 1 :

$function$	$(a \to b)$	$\to$	$[a]$	$\to$	$[b]$
$arguments + context$	$\bullet$	$\to$	$(Int \to Int)$	$\to$	$[Int]$

configuration 2 :

$function$	$(a \to b)$	$\to$	$[a]$	$\to$	$[b]$
$arguments + context$	$(Int \to Int)$	$\to$	$\bullet$	$\to$	$[Int]$

Figure 3. Two configurations for column-wise unification

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 $map :: (a \to b) \to [a] \to [b]$ has two, and the function $foldr :: (a \to b \to b) \to b \to [a] \to b$ 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).

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

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 $Int \to Int$. Because we are one argument short, we insert one hole ($\bullet$) 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 4.

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

The expression (-1) is of type Int, and can thus not be used as the first argument of map.

$$decrementList :: [Int] \rightarrow [Int]$$
$$decrementList\ xs = map\ (-1)\ xs$$

The following error message therefore focuses on the first argument of map.

```
(2,25):  Type error in application
expression      : map (-1) xs
function        : map
  type          : (a -> b) -> [a] -> [b]
1st argument    : -1
  type          : Int
  does not match : a -> b
```

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 siblings 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 [4].)

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.

5. Related work, validation, implementation

There is quite a large body of work on improving type error messages for polymorphic, higher-order functional programming languages such as Haskell, cf. [11, 9, 7, 8, 12, 13]. The drawback of these papers is that they have not led to full scale implementations, although in many cases they do suggest one.

In recent years, there is a trend towards implementation. One of these systems is Chameleon [10] 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.

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.

The Helium compiler which includes all the heuristics we have discussed (and more), 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 [5].

The existence of implementations immediately raises another issue: by means of this implementation it should be possible to establish whether the implemented methods really help. 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 these systems can only be verified empirically. However, to perform such experiments is a problem in itself and beyond the scope of this paper.

A final issue we would like to address is that of efficiency of the compiler. We have introduced 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 in fact 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. Besides, accurate error messages reduce the time programmers have to spend correcting their mistakes.

6. 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 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). A second project continues the work on rearranging abstract syntax trees so that they become type correct. Finally, the use of generics in Java and the subsequent unintelligible type error messages offer a fresh new challenge to our library.

References

[1] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[2] J. Hage and B. Heeren. Ordering type constraints: A structured approach. Technical Report UU-CS-2005-016, Institute of Information and Computing Science, Universiteit Utrecht, Netherlands, April 2005.

[3] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/phdthesis`.

[4] 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.

[5] 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. `http://www.cs.uu.nl/helium`.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] J. Yang. Explaining type errors by finding the sources of type conflicts. In P. Trinder, G. Michaelson, and H-W. Loidl, editors, *Trends in Functional Programming*, volume 1, pages 58–66. Intellect, Bristol, UK, 2000.

[13] 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

Before we give a sample trace, we first consider a small part of the code of the compiler: the part that governs which heuristics are present and how they will be applied.

Note that we have taken the liberty to remove a few heuristics that are either experimental, or which have to do with overloading and Haskell's class system, or which are not for normal usage. The names of the heuristics generally correspond to headings in the paper.

```
rate = 0.95

listOfHeuristics siblings path =
  [forbiddenFilter
  ,highParticipation rate path
  ] ++
  [Heuristic (Voting
    [siblingFunctions siblings
    ,siblingLiterals
    ,applicationHeuristic
    ,tupleHeuristic
    ,fbHasTooManyArguments
    ,variableFunction
    ,unifierVertex])
  ] ++
  [avoidApplicationConstraints
  ,avoidNegationConstraints
  ,avoidTrustedConstraints
  ,avoidFolkloreConstraints
  ,firstComeFirstBlamed
```

$]$

Note first that this piece of code gives a very compact and comprehensive view on what heuristics are available. We explain the code in a bit more detail below.

The function $listOfHeuristics$ uses a (partially user specified) list of siblings [4] to generate the list of available heuristics for this compilation. Heuristic at the front of the list have a higher priority than one towards the back: they are executed first.

Each heuristic is either a filtering heuristic or a voting 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 the forbidden filter removes those constraints of the sort described under (1) of the trust factor heuristic, the other three cases are part of $avoidTrustedConstraints$.

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

Most of these heuristics are connected directly with heuristics discussed in the paper. There are two special cases that may need some more explanation: $variableFunction$ 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. 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 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. If a constraint is selected by a voting heuristic, all other constraints will be removed from the set of candidates leaving only the selected constraint. The implementation guarantees that the filtering heuristics never make the set of candidates empty.

The heurstics in the final block, starting with $avoidApplicationConstraints$ are low priority heuristics that are used as tie-breakers.

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 4.2 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.