Heuristics for type error discovery and recovery

Jurriaan Hage

Bastiaan Heeren

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-029

www.cs.uu.nl

Abstract. The problem of type correctness of a functional program can be formulated as a constraint satisfaction problem. Detecting that the set of constraints is inconsistent is not enough: to improve programming efficiency, finding the constraints that are most likely to be responsible for the inconsistency is crucial. Being able to indicate how the source program should be changed to resolve the inconsistency is yet another step forward.
First, a collection of constraints is represented as a type graph. Then we propose heuristics that work on the type graph data structure, each of which may suggest reasons for inconsistencies by selecting a constraint. Some of the more domain specific heuristics can even show how the program may be corrected, by considering local reformulation of the constraint satisfaction problem.

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

1 Introduction

Over the last few years we have developed the Top framework for constraint-based type inferencing [2], and applied it to the development of a compiler for a substantial part of the higher-order functional programming language Haskell, called Helium [6]. The main focus of the system is to improve the error messages that are returned by the compiler on detecting a type error, which is essential for teaching students to program Haskell. Our proposed solution consists of the following steps: step one is to formulate type correctness as the satisfiability of a set of constraints. These constraints are constructed during a simple traversal of the abstract syntax tree of the program. The problem that remains is that the order in which constraints are considered strongly influences the constraints that we find to be inconsistent with the others. 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 the program, even if there is more evidence to the contrary.

We deal with this problem by considering a collection of constraints at the time. We construct a type graph from (possibly inconsistent) sets of constraints. We have developed a collection of heuristics that work on this type graph to suggest which constraints are most likely to be responsible for the error. The more language specific heuristics of these, also suggest how to improve the program, e.g., by telling the programmer he should interchange certain function arguments. These heuristics consider a local reformulation of the original constraint set, to see whether certain slight changes to the source code would resolve the inconsistency.

Many of the 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.

Our choice for specifying the type inference problem using constraints has given us the usual benefit of decoupling specification and computing a solution, which tends to simplify both to a large extent. Furthermore, the fact that many program analyses share the same types of constraints allows us to reuse our solvers. Finally, the separation also allowed us to formulate user-defined specialized type rules, which are the subject of another paper [5].

In outline, our paper proceeds as follows. In the next section we set the scene and introduce the constraints we are going to use. Then we introduce type graphs, our main data structure for solving sets of constraints. Finally, we consider the heuristics we have developed in turn. The final section concludes.

2 Constraints

Our type language has monomorphic types (τ) and type schemes (σ). Type schemes are used to capture polymorphic types such as $\forall a.a \to a$, which is 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 $\to$ for function types), or the application of a type to another. For example, the type of functions from integers to booleans is written $((\to\ \mathit{Int})\mathit{Bool}$. Type application is left-associative, and we omit parentheses. We often write the function constructor $\to$ infix, resulting in $\mathit{Int} \to \mathit{Bool}$. We assume the types are well-kinded: types like *Int Bool* and $\to \mathit{Int}$ 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

let $f = \lambda x\ y \to$ **if** x **then** 0 **else** x
in (f `True` `False`, f 2 3)

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 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].

Although we do not to go into exact details, we would like to sketch how type inference on a program as a whole proceeds. Because we first want to generate constraints, we use 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 at this point to transform the implicit instance constraints into equality constraints, after which we can make these consistent as well. For more details on the special types of constraints we use, and the specific type rules necessary to inference Haskell, the reader is referred to [4].

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', which 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.

Of course, 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.

The type graphs presented in this paper resemble the path graphs that were proposed by Port [10], 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 [8]. In his case parts of the graph are duplicated to handle let-constructs.

Constructing a type graph Constructing a type graph for a given set of constraints is done by considering each constraint in the set in turn. We shall use the following constraint set as illustration.

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

We illustrate the construction using the constraint $v_1 \equiv F\ v_0\ v_0$. Term graphs are constructed for both the left-hand side type and right-hand side type of the equality constraint as follows. 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 a single vertex, which we annotate with the constant. For each occurrence of this constant in the constraint

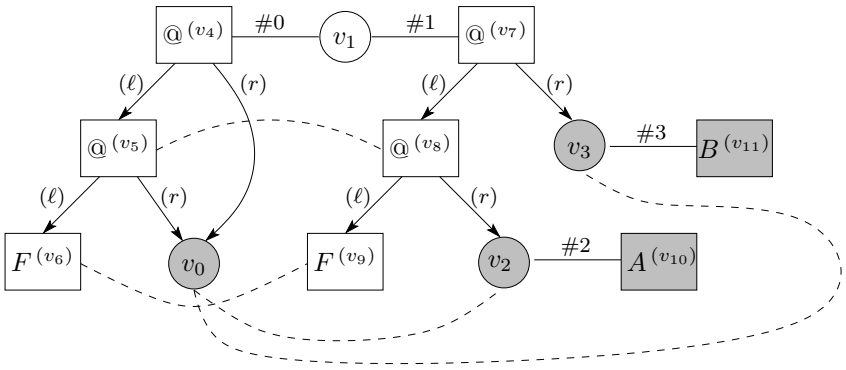

Fig. 1. An inconsistent type graph

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 Fig. 1 in which a term graph for the type $F\ v_0\ v_0$ is given to the left (and, similarly, one for $F\ v_2\ v_3$). Here, vertices labeled with (@) correspond to type application in composite types. Note that our term graphs are directed acyclic graphs. 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 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 Fig. 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 [4].

3.1 Analyzing the type graph

Equality paths and error paths The type graph of Fig.1 contains four equivalence groups, including one that contains all the shaded vertices. This group contains both type constants A and B. Since the two are different, this implies an inconsistency in the constraint set. The first question is how to determine this, and we consider it next. The question of finding the constraint to blame for the inconsistency is addressed in detail in the next section on heuristics.

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 of the inconsistency 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 $Up_i^{(\delta)}$ and $Down_i^{(\delta)}$, where δ is either ℓ (left child) or r (right child). The annotation Up corresponds to moving upwards in

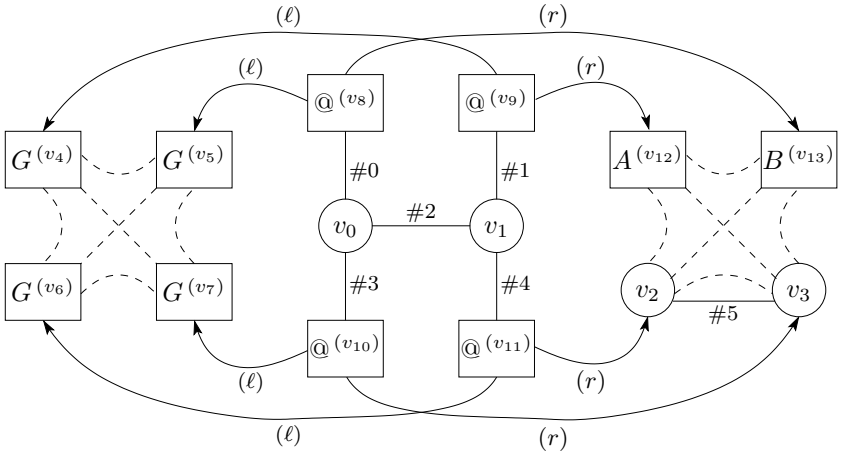

Fig. 2. A type graph with two implied equality cliques

the term graph by following a child edge, whereas *Down* corresponds to moving downwards (from parent to child). Each *Up* annotation in an equality path comes with a *Down* annotation at a later point, 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 Fig. 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 $[\#2, Up_0^{(r)}, Up_1^{(\ell)}, \#1, \#0, Down_1^{(\ell)}, Down_0^{(r)}]$ 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:

$$[\#2, Up_0^{(r)}, Up_1^{(\ell)}, \#1, \#0, Down_1^{(\ell)}, Down_0^{(r)}, Up_2^{(r)}, \#0, \#1, Down_2^{(r)}, \#3].$$

Both #0 and #1 appear twice in π. Note that by following $Up_2^{(r)}$ 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.

Avoiding detours Note that we can restrict ourselves to error paths that do non-trivially contain an induced error path: removing a constraint on the smaller path removes both error paths at the same time. More savings can be made by avoiding detours: a *detour* is a path that consists of two (consecutive) implied edges from the same clique, a subgraph in which all vertices are connected to each other. A *detour equality path* is an equality path that contains at least one detour. A *shortcut* of a detour path is the path in which we remove the two implied edges that form a detour, say from v_0 to v_1 and from v_1 to v_2, and replace it by the implied edge from v_0 to v_2.

We can safely ignore detour equality paths (see [4] for the proof):

Lemma 1. *Let π be a detour equality path, and let π' be a shortcut for π. If removal of a constraint removes π' from the graph, then π is removed as well.*

Consider the following set of type constraints (see Fig. 2 for the type graph).

$$\mathcal{C} = \{\ v_0 \overset{\#0}{\equiv} G\ B\ ,\ v_1 \overset{\#1}{\equiv} G\ A\ ,\ v_0 \overset{\#2}{\equiv} v_1\ ,\ v_0 \overset{\#3}{\equiv} G\ v_3\ ,\ v_1 \overset{\#4}{\equiv} G\ v_2\ ,\ v_2 \overset{\#5}{\equiv} v_3\ \}$$

The only irregularity in the type graph is that the constants A and B are part of the same equivalence group. The most obvious error path is the implied edge from the vertex $A^{(v_{12})}$ to

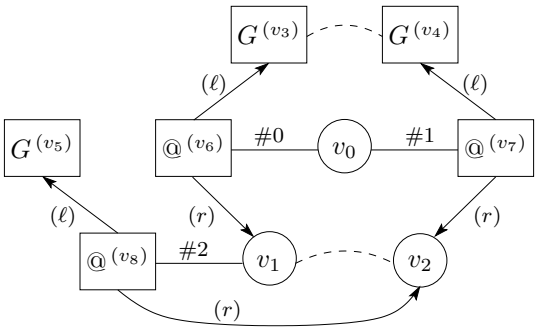

Fig. 3. A type graph with an infinite path

$B^{(v_{13})}$. Expansion of this edge gives us a path between the application vertices v_9 and v_8. Hence, our first error path is

$$\pi_0 = [Up_0^{(r)}, \#1, \#2, \#0, Down_0^{(r)}].$$

But this is not the only error path – if that were the case, then removing #0, #1, or #2 would make the constraint set consistent. For the first two constraints this is true, but removing #2 from $\mathcal{C}$ leaves the constraint set inconsistent.

A second possibility is the error path from $A^{(v_{12})}$ to v_3 to $B^{(v_{13})}$. However, Lemma 1 tells us that this equality path is a detour. And indeed, expanding this path gives us

$$[Up_0^{(r)}, \#1, \#2, \#3, Down_0^{(r)}, Up_1^{(r)}, \#3, \#0, Down_1^{(r)}],$$

which is a detour of π_0.

The additional error path π_1 is from $A^{(v_{12})}$ to v_2 by following the implied edge, then taking the initial equality edge #5 to vertex v_3, and from there to $B^{(v_{13})}$ with an implied edge. This path visits the implied equality clique twice, but there are no two consecutive steps within the same clique. A third path π_2 exists, going first to v_3, then to v_2, before ending in $B^{(v_{13})}$.

$$\pi_1 = [Up_0^{(r)}, \#1, \#4, Down_0^{(r)}, \#5, Up_1^{(r)}, \#3, \#0, Down_1^{(r)}]$$

$$\pi_2 = [Up_0^{(r)}, \#1, \#2, \#3, Down_0^{(r)}, \#5, Up_1^{(r)}, \#4, \#2, \#0, Down_1^{(r)}]$$

The error path π_1 remains after the removal of #2, which confirms that only removing #2 does not make the type graph consistent.

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.

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

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

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.

$$\pi = [\#2, Down_\infty^{(r)}, Up_0^{(r)}, \#1, \#0, Down_0^{(r)}]$$

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.

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.

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 continue the example of Fig. 1 which had a single error path,

$$\pi = [\#2, Up_0^{(r)}, Up_1^{(\ell)}, \#1, \#0, Down_1^{(\ell)}, Down_0^{(r)}, Up_2^{(r)}, \#0, \#1, Down_2^{(r)}, \#3].$$

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 $F\ A\ A$.

Whichever constraints we choose to remove: each will result 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.

Dealing with type synonyms Type synonyms introduce new type constants as abbreviations for existing types. This serves two purposes: complex types become easier to write and read, and one can introduce intuitive names for a type. Simple examples of type synonyms in Haskell include:

$$\begin{aligned}
&\textbf{type } \mathit{Number} &&= \mathit{Int}\\
&\textbf{type } \mathit{Telephone} &&= \mathit{Number}\\
&\textbf{type } \mathit{String} &&= [\mathit{Char}]\\
&\textbf{type } \mathit{Parser}\ s\ a &&= [s] \to [(a, [s])]
\end{aligned}$$

Many implementation completely unfold type synonyms. This is not satisfactory if we want to report type error messages (or present inferred types) in terms of the original program. Therefore, a conservative unfolding policy is to be preferred. In a type graph, we associate type synonym information with each vertex. Let τ be a type for which we want to construct a term graph. We unfold τ until the top-level type constructor is not a type synonym. For this unfolded type we build a term graph, and we remember the original type τ in the top-level vertex. Likewise, type synonyms "inside" τ are handled.

When we determine the type associated with some equivalence group E, we take into account the original types stored in the vertices of E. For instance, if E has two type constants *Int*, and both are annotated with the type synonym *Telephone*, then the latter constant will be used it is

the more informative. If the constants *Int*, *Number*, and *Telephone* are the original types in an equivalence group, then it is unclear which of the three should be appointed as representative. The type constant *Int* is the safest alternative, since this is the common unfolded type, although one could also follow the reasoning that *Number* and *Telephone* are user-defined types, and opt for these instead.

4 Heuristics

In principle, all the constraints that are on an error path are candidates. 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 probably 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 type error messages being reported.

Many of our heuristics are independent, so we need some facility to coordinate the interaction between them. The 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 (the user-defined specialized type rules which are the subject of another paper [5]), 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 constraint satisfaction problems for type inference.

Share in error paths 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. Note that this heuristic helps to decrease the number of reported error messages, as multiple error paths disappear by removing one constraint. However, repeatedly removing the constraint on the largest number of error paths does not necessarily lead to the minimum number of error messages. We show this by example. Consider the set P consisting of six error paths.

$$\{\{\#1, \#2, \#5\}, \{\#1, \#3, \#6\}, \{\#1, \#4, \#7\}, \{\#2, \#8\}, \{\#3, \#9\}, \{\#4, \#10\}\}$$

For each set in P, we have to remove at least one of its constraint from the type graph. At first sight, the constraint #1 is the best candidate for removal because it is present in three error paths. The smallest set is, however, $\{\#2, \#3, \#4\}$.

The share-in-error-path 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. In general, this heuristic does not yet reduce the set of candidates to a singleton.

Constraint number heuristic The next heuristic we present is used as a final tie-breaker since it 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.

Another report [3] suggests how to flatten an abstract syntax tree decorated with constraints into a list. Although the order of the constraints is irrelevant in 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 has the highest constraint number. This results in a list of constraints that complete an error path, and out of these constraints we pick the one with the lowest constraint number.

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 constraints satisfaction problems 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.

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

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) 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 $\mathcal{M}$ [7] 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*.

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

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.

(3) The type of a function imported from the Prelude should not be questioned. Ordinarily such a function can only be *used* incorrectly.

(4) 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.

(5) 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.

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 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 two cases to consider.

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	$(Int \to Int)$	$\to$	$\bullet$	$\to$	$[Int]$	

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 $S([a]) = [Int]$.

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 alright. 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.

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

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
```

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 of (Chapter 6 of [4]) 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 use the other cases for justification.

The following definition contains a type error.

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

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.

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.

An approach to automatically correcting ill-typed programs is based on a theory of type isomorphisms [9]. 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.

$$
\begin{aligned}
&\mathit{square} :: \mathit{Int} \to \mathit{Int}\\
&\mathit{square}\ i = i * i
\end{aligned}
$$

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

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).

$$
\begin{array}{ccc}
 & \mu_1 = \lambda f\ (xs, g) \to f\ g\ xs & \\
(a \to b) \to [a] \to [b] & \rightleftarrows & ([a], a \to b) \to [b]\\
 & \mu_2 = \lambda f\ g\ xs \to f\ (xs, g) &
\end{array}
$$

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.

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

In this definition, *not* of type $Bool \rightarrow Bool$ 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.

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. The Helium compiler allows programmers to add new pairs of siblings, which the compiler then takes into account. More details can be found in an earlier paper [5].

5 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. Discussions with students who have used our compiler are encouraging (especially with those that worked with other Haskell compilers). We are currently proceeding along two 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.

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. Top project website. `http://www.cs.uu.nl/ST/Top`.
3. J. Hage and B. Heeren. Ordering type constraints: A structured approach. Technical Report UU-CS-2005-016, Institute of Information and Computing Science, University Utrecht, Netherlands, April 2005. Technical Report.
4. B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/TopQuality.pdf`.
5. 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.
6. B. Heeren, D. Leijen, and A. IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press.
7. 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.
8. 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.
9. 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.
10. 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.

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 Greg Michaelson, Phil Trindler, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.