Improving type-error messages in functional languages

Bastiaan Heeren, Johan Jeuring, Doaitse Swierstra
Pablo Azero Alcocer

Utrecht University
bastiaan@cs.uu.nl

February 1, 2002

Abstract

Although type systems do detect type errors in programs, they often produce uninformative error messages, that hardly give information about how to repair a program. One important shortcoming is the inability to highlight the most likely cause for the detected inconsistency. This paper presents a type inferencer with improved error reporting facilities, based on the construction of type constraints. Unification of types is replaced by the construction of constraint graphs. This approach increases the chance to report the actual source of a type conflict, resulting in more useful error messages.

1 Introduction

Type systems are indispensable in modern higher-order, polymorphic languages. An important contribution to the popularity of Haskell and ML is their advanced type system, which enables detection of ill-typed expressions at compile-time. Modern language processors use type inference techniques that are derived from the algorithm proposed by Milner [12], and are based on the unification of types.

Since the error messages of most compilers and interpreters are often hard to interpret, programmer productivity is hampered. Also, programmers new to the language are likely to be discouraged from using it. Unfortunately it is not straightforward to change unification-based systems to produce clear type error messages. One serious problem is that type conflicts might be detected far away from the site of the error. Another problem is that the location where an inconsistency is detected is influenced by the order in which types are unified. Unification-based type systems have a bias to report type conflicts near the end of the program. This *left-to-right bias* is caused by the way unification and substitution are used. A type inference algorithm should be *symmetric*: subexpressions have to be handled identically without a tendency to report type conflicts more often in certain parts of the program. Other problems of type systems for polymorphic languages are the following.

- Type inference techniques are very local, assuming expressions to have correct types until an inconsistency is found. A global approach takes the complete program into consideration before it determines what is probably incorrect. The advantage of this approach is that extra information becomes available, resulting in better error messages.

- Error messages are a brief explanation of a conflict. Occasionally an extensive clarification of the conflict is required to identify errors. The desired level of detail in a message depends on the experience of a programmer.
- Only the first type conflict that is detected is reported. It is useful to report multiple (independent) type conflicts.

Example:
The following ill-typed function illustrates the problem with current error messages:

$$\begin{aligned} f = \lambda x \rightarrow \textbf{case } x \textbf{ of} \\ 0 &\rightarrow \mathit{False}; \\ 1 &\rightarrow \mathit{"one"}; \\ 2 &\rightarrow \mathit{"two"}; \\ 3 &\rightarrow \mathit{"three"}; \end{aligned}$$

The error message produced by Hugs, an interpreter for Haskell, for this definition is:

```
ERROR "example.hs" (line 1): Type error in case expression
*** Term           : "one"
*** Type           : String
*** Does not match : Bool
```

This message results from the different types of expressions on the right-hand side (three expressions of type *String* and one of type *Bool*) and points to term *"one"*. Considering the large proportion of *String* constants it is reasonable to assume that expression *False* has an incorrect type.

In this paper we present a new approach to type inference to remedy this shortcoming. A set of constraints on types is generated for an expression. The power of constraint-based program analysis is the separation of *constraint generation*, the specification of the analysis, and *constraint resolution*, which is the implementation [1]. Although constraints are typically generated locally, a set of constraints can describe global properties. Since we are no longer forced to solve the constraints while they are generated, the system does not have a left-to-right bias. Heuristics are used to remove inconsistencies in the complete set of constraints. Each resolved inconsistency results in a reported type conflict.

Several papers discuss type inference techniques related to our approach. Lee and Yi [9] formally define algorithm $\mathcal{W}$ (bottom-up), and a folklore algorithm $\mathcal{M}$ (top-down), that both work for the Hindley-Milner let-polymorphic type inference system. They proof that for each ill-typed expression, algorithm $\mathcal{M}$ detects an inconsistency earlier than $\mathcal{W}$. To find the source of a type error instead of the location where an inconsistency is detected, Wand [14] presents an algorithm that keeps track of *reasons* for deductions about the type of a variable. A similar approach is discussed by Beaven and Stansifer [3], where an interactive system traces all deductive steps to construct an explanation for a type conflict. Various techniques to improve type error messages use constraints on types. Walz and Johnson [13] collect a set of type equations that can be inconsistent. To resolve contradictions, variables are assigned a type such that most of the hypotheses are satisfied. Unfortunately, the order in which the equations are solved determines which conflict is reported. Aiken et al. [2] show how to perform a constraint-based program analysis in a type inference algorithm for the lambda-calculus without polymorphism. Gandhe et al. [7] discuss corrections of ill-typed expressions

using constraints on types; their system however cannot handle polymorphism. McAdam [10] discusses the use of unification and substitution in conventional inference algorithms which have a left-to-right bias. To remove this bias, a modification of the inference algorithm proposed by Hindley-Milner is suggested that unifies substitutions instead of types. In a different paper, McAdam presents a method to capture information about types in a graph [11]. This approach is a generalisation of several other techniques [4, 14, 6]. Jun [8] gives an inference algorithm that reports conflicting sites rather than the site where a conflict is detected. Only little knowledge about type checking is required to understand the reported conflicts.

This paper is organised as follows. In the next section, an expression and type language is presented, for which type inference rules are given in Section 3. In Section 4 we give a specification of the constraint solving process. We use this specification to discuss the correspondence between the presented type system and the Hindley-Milner type system in Section 5. Sections 6 and 7 present respectively an algorithm to solve constraints, and heuristics to remove inconsistencies in a set of constraints, whereas Section 8 gives conclusions and highlights topics for future research.

2 Expression and type language

We are interested in a higher-order functional language, suitable for type reconstruction. Our language is described by:

$$
\begin{array}{rcl}
\mathit{Expr} & ::= & \text{identifier} \\
& | & \text{constant} \\
& | & \mathit{Expr}\ \mathit{Expr} \\
& | & \lambda\text{identifier} \rightarrow \mathit{Expr} \\
& | & \textbf{case}\ \mathit{Expr}\ \textbf{of}\ (\mathit{Expr} \rightarrow \mathit{Expr};)^{+} \\
& | & \textbf{let}\ \text{identifier}\ (::\mathit{Type})?\ = \mathit{Expr}\ \textbf{in}\ \mathit{Expr}
\end{array}
$$

The set of constants contains literals and data constructors, all carrying their own constant type. Although expressions on the left-hand side of case-expressions are restricted to patterns, it is not necessary to distinguish patterns and expressions. A let expression contains a single declaration that can be assigned an explicit type, and recursive declarations are permitted. Although an explicit type is often just used to ensure that the inferred type is the intended type, it is necessary for polymorphic recursion.

The type language is given by:

$$\mathit{Type} \quad ::= \quad \text{variable} \mid \text{constant} \mid \mathit{Type}\ \mathit{Type}$$

The type system will benefit from the straightforwardness of this type language, especially since no quantified types are required. Type variables are written v, v_1, v_2 ..., type constants are written c, c_1, c_2 .... There is a special type constant $(\rightarrow)$ for representing function types. An example of a function type is $((\rightarrow)\ \mathit{Int})\ \mathit{Bool}$. In the rest of this paper we use the standard infix notation for function types. We assume that spurious types, like $((\rightarrow)(\rightarrow))$, do not occur.

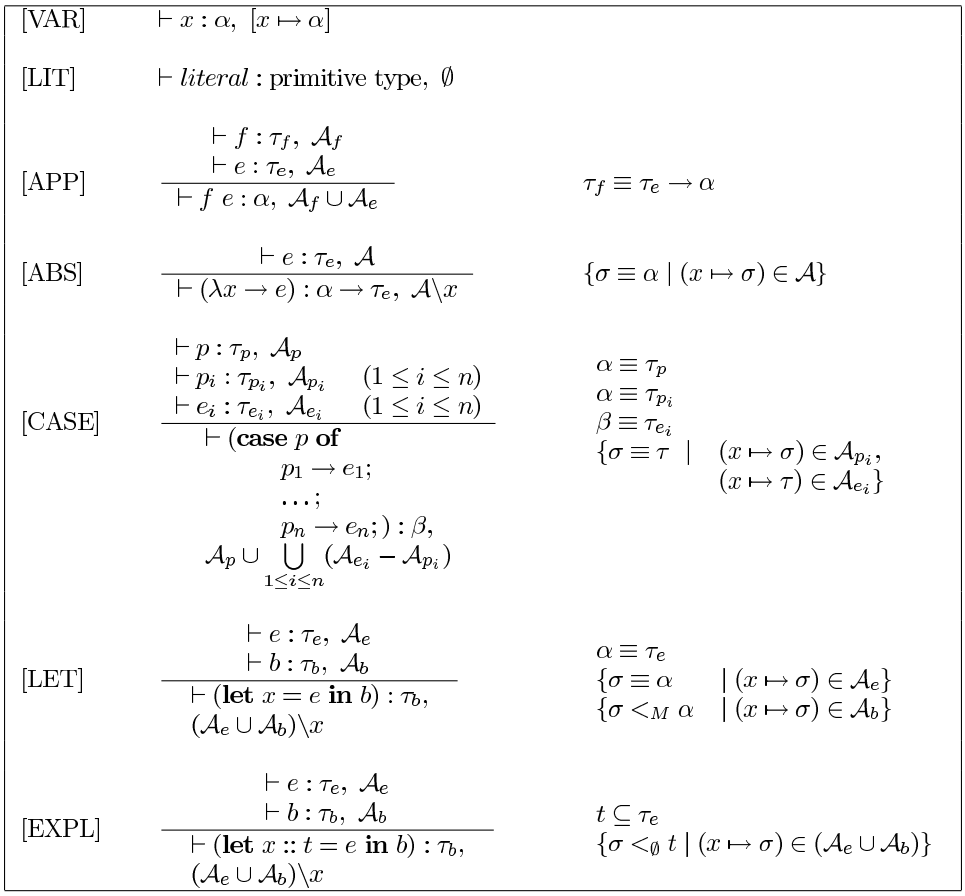

Figure 1: Type inference rules

3 Type inference rules

We present a set of type inference rules to assign a type to an expression. We use judgements of the form $(\vdash e : \tau,\ \mathcal{A},\ C)$ to express that expression e has type τ in environment $\mathcal{A}$, provided that the constraints on types in C are satisfied. The environment contains type variables assigned to the variables that are free in e. Figure 1 gives a typing schema that provides a rule for each language construct. In this figure, the set of constraints is not included in the judgements; the constraints that are constructed for the language construct at hand are given in the right-hand column, next to the inference rule. We implicitly assume that the set of constraints for expression e contains the union of all constraints that are generated in subexpressions of e.

The inference rule for a variable is straightforward: the variable is assigned a new type variable α and this is recorded in an environment. A literal is associated with its constant type, together with an empty environment. In the case of a constructor with a polymorphic type a specialisation of the type is required, for instance, the constructor `Cons` with type $(\forall a\ .\ a \to \mathit{List}\ a \to \mathit{List}\ a)$ is specialised to $(\alpha \to \mathit{List}\ \alpha \to \mathit{List}\ \alpha)$, with α as a new type variable. For each application a new type variable α is introduced, which represents the type of the application. The relation between the type of the function, the argument, and the result, is expressed through an equality constraint. An equality constraint, written

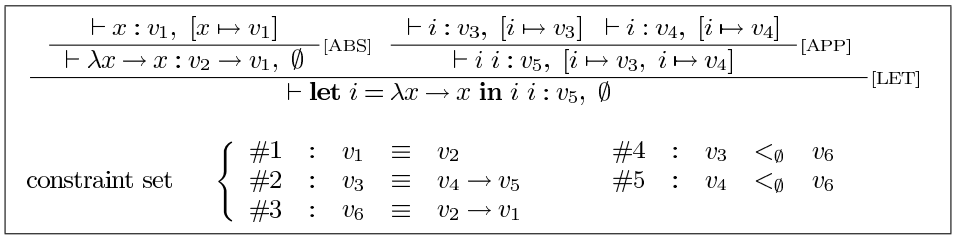

Figure 2: Example of an inference tree

($t_1 \equiv t_2$), represents a *delayed* unification of two types. Because the environments of the function and its argument are combined after inferencing them separately, a variable can be bound to different type variables in an environment. A lambda expression removes all variables from the environment that become bound, and constructs an equality between the associated type variables and a fresh type variable α. Two new type variables are introduced for a case expression, representing the types of the left-hand sides and right-hand sides of the alternatives respectively. Variables in a pattern bind free variables that are used on the right-hand side of the same alternative. An equality constraint is constructed for each variable that is bound by a pattern after this variable is removed from the environment.

Let expressions introduce polymorphism into a language. For each occurrence of a declared variable in the body of a let expression, a specialisation of the inferred type of the declaration is created. Equality constraints on types, however, are not sufficient to express specialisation of polymorphic types. We introduce instance constraints of the form ($t_1 <_M t_2$) to express that t_1 is a specialisation of t_2. A set of monomorphic type variables M is stored for each instance constraint. This set contains all type variable that were introduced by a lambda expression (type variable α in [ABS]) outside the current let expression. The values of M at the various sites can be computed by a simple traversal of the syntax tree. We have left this implicit. Monomorphic type variables (or type variables equal to a monomorphic type variable) are not generalised when a new instance is created.

When a let declaration is annotated with a type, each applied occurrence of the declared variable is made an instance of the annotated type. There are no monomorphic type variables for this instantiation. The declared type cannot be more general than the inferred type of the declaration, which is expressed by a specialisation constraint ($t_1 \subseteq t_2$).

Example:
Figure 2 depicts an inference tree for (**let** $i = \lambda x \to x$ **in** $i\ i$), together with the collected set of constraints. The set of monomorphic type variables is empty for both instance constraints because there are no free variables in the declaration. In section 4 we present the function $\mathcal{S}$, which solves such a set of constraints. Applying $\mathcal{S}$ to the set of constraints results in the polymorphic type ($a \to a$) for v_5.

At first sight one might wonder why we generate trivial constraints such as ($v_1 \equiv v_2$), instead of substituting this immediately as is done in more conventional approaches. The answer to this question touches the essence of our approach: since we may have to resolve inconsistent constraints when dealing with inconsistencies, it is nice to have as much information as possible available.

$$\mathcal{S} :: \mathit{ConstraintSet} \to \mathit{Type} \to \mathit{Type}$$

$$\begin{array}{lrcl}
(1) & \mathcal{S}(\emptyset)\ \tau & = & \tau \\
(2) & \mathcal{S}(\{c \equiv c\} \cup C)\ \tau & = & \mathcal{S}(C)\ \tau \\
(3) & \mathcal{S}(\{t_1\ t_2 \equiv t_3\ t_4\} \cup C)\ \tau & = & \mathcal{S}(\{t_1 \equiv t_3, t_2 \equiv t_4\} \cup C)\ \tau \\
(4) & \mathcal{S}(\{v \equiv t\} \cup C)\ \tau & = & \mathcal{S}([v := t]C)\ ([v := t]\tau) \\
 & \text{if type variable } v \text{ does not occur in } t & & \\
(5) & \mathcal{S}(\{t_1 <_M t_2\} \cup C)\ \tau & = & \mathcal{S}(\{t_1 \equiv \overline{M}(t_2)\} \cup C)\ \tau \\
 & \text{if no type variable in } t_2 \text{ occurs in } C & & \\
(6) & \mathcal{S}(\{t_1 \subseteq t_2\} \cup C)\ \tau & = & \mathcal{S}(C)\ \tau \\
 & \text{if no type variable in } t_2 \text{ occurs in } C & & \\
 & \text{and there exists a } \sigma \text{ such that } t_1 = \sigma(t_2) & &
\end{array}$$

Figure 3: Definition of function $\mathcal{S}$

4 Constraint solving: specification

We give a specification of how to solve a set of constraints. We define the function $\mathcal{S}$, that solves a consistent set of constraints. If $\mathcal{S}$ succeeds, a substitution of types for type variables is returned. Figure 3 shows an inductive definition of $\mathcal{S}$ that solves one constraint at a time. The order in which the constraints are solved is irrelevant. Definition (1) expresses that if the set of constraints is empty, no substitution is required. According to definition (2), (3), and (4), solving equality constraint $(t_1 \equiv t_2)$ corresponds to the unification of t_1 and t_2. Note that function $\mathcal{S}$ fails if the two types cannot be unified. Definition (5) and (6) solve instance constraints and specification constraints respectively. To make an instantiation of type t, we define $\overline{M}(t)$ as the closure of type t with respect to the set of monomorphic type variables M. This definition is slightly different from the definition given by Damas and Milner [5], since type variables are not quantified but replaced by fresh type variables.

$$\overline{M}(t) = [\alpha_1 := \phi_1,\ \ldots,\ \alpha_n := \phi_n]t$$

where $\alpha_1 \ldots \alpha_n$ are all the type variables in t but not in M,
and $\phi_1 \ldots \phi_n$ are fresh type variables

5 Correctness

We briefly explain the correspondence of our type system with the type system of Hindley-Milner. According to our type inference rules, there is exactly one assumption for each variable that is free in an expression. Furthermore, the inference rules of Hindley-Milner require an assumption for each free variable in an expression. The function $(\triangle)$ constructs a set of equality constraints from two assumption sets:

$$(\triangle) :: \mathit{AssumptionSet} \to \mathit{AssumptionSet} \to \mathit{ConstraintSet}$$
$$\mathcal{A} \triangle \mathcal{A}' = \{\tau \equiv \tau' \mid (x \mapsto \tau) \in \mathcal{A},\ (x \mapsto \tau') \in \mathcal{A}'\}$$

This function merges the assumptions of two sets. Since in general a set of assumptions can contain type-schemes, it follows that quantified types can occur in equality constraints. Type-schemes are transformed into types by instantiating the quantified type variables; each quantified type variables is replaced by a fresh type variable. We extend the definition of $\mathcal{S}$ with the following rule.

$$(7) \qquad \mathcal{S}(\{\forall x \,.\, \sigma \equiv \tau\} \cup C)\ \tau \quad = \quad \mathcal{S}(\{[x := y]\sigma \equiv \tau\} \cup C)\ \tau$$
where y is a fresh type variable

We claim that our type system is correct with respect to the type system of Hindley-Milner. Our type system has the following property:

Property:

$$\vdash e : \tau,\ \mathcal{A},\ C \qquad \Leftrightarrow \qquad \mathcal{A}' \vdash_{\text{HM}} e : \tau'$$
$$\text{such that} \quad \sigma\tau' \ < \ \overline{\sigma\mathcal{A}'}(\sigma\tau)$$
$$\text{with} \quad \sigma = \mathcal{S}(C \cup (\mathcal{A} \triangle \mathcal{A}'))$$

Given an expression e, we use our typing rules to derive a type τ, a set of assumptions $\mathcal{A}$, and a constraint set C. Given a set of assumptions $\mathcal{A}'$, the function $(\triangle)$ constructs a set of equality constraints for corresponding assumptions in $\mathcal{A}$ and $\mathcal{A}'$. Applying $\mathcal{S}$ to C and the set obtained by merging $\mathcal{A}$ and $\mathcal{A}'$ results in a substitution σ. Because substitution σ is relevant for type variables that are free in $\mathcal{A}'$, we apply σ not only to τ, but also to $\mathcal{A}'$ and τ'. The closure of $(\sigma\tau)$ is the principle type-scheme of e under assumptions $(\sigma\mathcal{A}')$; in other words, each τ', such that $(\mathcal{A}' \vdash_{\text{HM}} e : \tau')$ holds, must be a generic instance of this type.

6 Constraint solving: implementation

In this section we present an algorithm to solve the generated set of constraints. If the algorithm is unable to find a solution for a set of constraints, it gives a good indication of why it cannot be solved. This algorithm is an implementation of the function $\mathcal{S}$. It provides a way to pin-point the most likely location of a type error in an ill-typed expression, and enables us to use heuristics to resolve inconsistencies. As a result, we are able to produce a better explanation of the error by giving a better error message.

An *equality graph* is an undirected graph that is used as intermediate data structure to store equalities between types. A vertex in an equality graph corresponds to a type variable or a type constant, an edge represents an equality constraint between two types, and is labelled with the constraint number. We start with a graph containing one vertex for each type variable but without edges. Each constraint is translated into a transformation of the graph. When all the constraints are resolved, a substitution can be obtained from the graph. The constraint solving process maintains two invariants: a vertex containing a type constant has exactly one edge to a vertex containing a type variable, and each type variable should occur exactly once in a vertex, whereas a type constant can occur in several vertices.

The remaining part of this section explains how the three different types of constraints ($\equiv$, $<$ and $\subseteq$) are solved using the equality graph. Finally, we present an algorithm to solve a set of constraints.

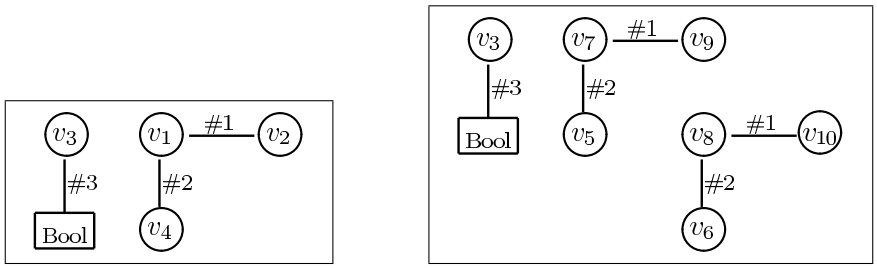

Figure 4: Equality graph before and after decomposition

Solving equality constraints

Each equality constraint results in a transformation of the equality graph.

- $(c_1 \equiv c_2)$: this constraint is removed from the set without modifying the graph. However, if the constants c_1 and c_2 are different, it should be marked as erroneous.
- $(c \equiv v)$: a vertex containing constant c is added to the graph, together with an edge between this new vertex and the vertex containing type variable v.
- $(c \equiv t_1\ t_2)$: because this constraint cannot be solved, it can be removed from the set and marked as erroneous.
- $(v_1 \equiv v_2)$: an edge is added between the vertices of v_1 and v_2.
- $(v \equiv t_1\ t_2)$: to satisfy this constraint, v is *decomposed* in an application of two types.
- $(t_1\ t_2 \equiv t_3\ t_4)$: this constraint is replaced by $(t_1 \equiv t_3)$ and $(t_2 \equiv t_4)$.

The remaining three cases are obtained by swapping the types in an equality constraint.

Decomposition of a type variable means substituting the type variable by an application of two fresh type variables. Decomposition of a type variable can only occur when its connected component has no type constants, which would indicate that there is an inconsistency, and also requires decomposition of the other type variables in its connected component. The substitution is applied to each type in the constraint set, so the equality constraint that requested the decomposition is also substituted into an application. The connected component with the decomposed variables is duplicated, where the fresh type variables on the left-hand side of the application replace the original type variables in one copy and the type variables on the right-hand side alter the other copy. The structure of the duplicated component remains the same.

Example:
Consider the equality graph on the left in Figure 4 in combination with the constraint $(v_4 \equiv \mathit{List}\ v_3)$. To satisfy this constraint, type variable v_4 requires a decomposition. The substitution for the type variables in the component of v_4 is $[v_4 := v_5\ v_6,\ v_1 := v_7\ v_8,\ v_2 := v_9\ v_{10}]$. Consequently, the equality constraint is changed to $(v_5\ v_6 \equiv \mathit{List}\ v_3)$. The equality graph after decomposition is shown on the right in Figure 4.

Solving instance constraints

An instance constraint ($t_1 <_M t_2$) can be transformed into an equality constraint as soon as t_2 is *fixed*. A type t_2 is fixed if the interpretation of each connected component that contains a type variable of t_2 will remain unchanged while the remaining constraints are solved. The interpretation of a connected component can change as long as one of its type variables occurs in an equality constraint that has not been dealt with: a constant can be inserted into the connected component or an edge can combine two connected components. A type variable occurring in an instance constraint can also result in a modification of the associated connected component, because eventually this constraint will be transformed into an equality. In other words, instantiation of a type is not possible as long as it can change when other constraints in the set are taken into account. Postponing instance constraints corresponds to the order in which let expressions are typed in algorithm $\mathcal{W}$ of Damas and Milner [5]: a type for the declaration must be inferred before the type of the body can be dealt with.

When an instance constraint ($t_1 <_M t_2$) is solved, a unique instance of t_2 is created that must be equal to t_1. The type variables in M are monomorphic; they are introduced at a lambda expression containing the let expression that generated the instance constraint. Monomorphic type variables (or type variables that are equal to a monomorphic type variable) are not instantiated. To create an instance of t_2, a substitution is constructed. Type variables that are in the same component as one of the type variables in M remain unchanged, while type variables in a connected component containing a constant are mapped to this constant. All type variables in a connected component without a type constant are replaced by the same fresh type variable.

Solving specialisation constraints

Specialisation constraints are only dealt with after the equality constraints and instance constraints. An explicit type cannot be more general than the inferred type of the declaration. For each constraint ($t_1 \subseteq t_2$) we try to find a substitution σ such that $\sigma(t_2) = t_1$. If such a substitution does not exist we report that the inferred type is not general enough.

Algorithm

INPUT : a set of constraints
OUTPUT : a substitution

Create an initial equality graph, and apply the following rules as long as possible, with earlier rules having a higher priority. Return the substitution obtained from the equality graph.

(1)	If there is an equality constraint that does not require a decomposition, solve this constraint.
(2)	If the equality graph is inconsistent, resolve this inconsistency.
(3)	If there is an instance constraint ($t_1 <_M t_2$) and t_2 is fixed, create an equality constraint between t_1 and an instantiation of t_2.
(4)	If there is an equality constraint ($v \equiv t_1\ t_2$), decompose v and the type variables in the same connected component as v.

When this algorithm terminates, the set of constraints only contains specialisation constraints

and the equality graph is consistent. For each specialisation constraint ($t_1 \subseteq t_2$), check if t_1 is an instance of t_2. The order in which rules are applied for a consistent set of constraints does not influence the outcome of this algorithm. However, the order is important for solving inconsistent sets. Resolving an inconsistency in the equality graph is postponed until there is no more relevant information available in the set of constraints. Note that for decomposition, instantiation, and specialisation, a consistent equality graph is required.

7 Solving inconsistencies

From an ill-typed expression we obtain an inconsistent set of constraints. When solving this set, a conflict appears in the equality graph. We distinguish two types of inconsistencies.

- Two different type constants are in the same connected component. Because the graph is undirected, there is at least one path connecting these constants. This path, referred to as an *error path*, serves as evidence for the inconsistency and is used to construct an appropriate error message. To avoid infinite paths only paths containing different vertices are considered.

- A type variable of a connected component containing a type constant requires a decomposition to satisfy an equality constraint. A *decompose path* is a path from a type constant to a type variable requiring a decomposition, together with a constraint causing the decomposition.

To remove an inconsistency at least one edge for each error path and decompose path has to be removed, which results in splitting connected components in smaller parts. At this point several heuristics can be used to determine which constraints to throw away. We discuss one approach in which we select a set of edges with the lowest total removal cost.

We calculate a removal cost for each constraint. The removal cost represents the cost to remove all edges in the graph produced by this constraint. This cost depends on the *trust value* of a constraint, a measure of confidence in this constraint which is determined by the origin of the constraint. For instance, a constraint originating from a reference to a prelude function has a high trust value, and so do constraints for user-defined expressions with an explicit type signature. Other constraints have an average trust value, for instance constraints resulting from an application or a lambda abstraction.

Example:
Assuming that *even* has type $Int \to Bool$, the expression (*even True*) has type v_2 with the inconsistent set of constraints $[\#1 : v_1 \equiv Int \to Bool,\ \#2 : v_1 \equiv Bool \to v_2]$. The first constraint has a high trust value; the second constraint has an average trust value because it was constructed for an application. Constraint #2 is removed to restore consistency.

Another heuristic is to consider the number of occurrences of a type constant in a connected component. Multiple occurrences of a constant increase the probability that this is the intended type constant. Reconsider the case expression introduced in Section 1. In Figure 5 the equality graph is presented for the constraints of this expression. The graph consists of two connected components. The component on the right is inconsistent. The three occurrences of *String* versus one occurrence of *Bool* suggest the removal of #6. To achieve this we collect all *correct paths*, a path between two vertices containing the same type constant. To

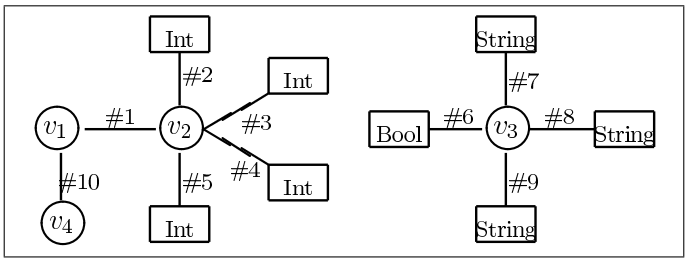

Figure 5: Inconsistent equality graph

prevent breaking a correct path, a constraint corresponding to an edge of a correct path is assigned a higher removal cost.

The next program fragment calculates the removal cost of a constraint in a given equality graph:

```
trust        :: Constraint -> Int
correctpaths :: Graph -> [[Constraint]]

removalcost :: Constraint -> Graph -> Int
removalcost con graph = f graph * trust con
   where f = (+1) . length . filter (con `elem`) . correctpaths
```

The function `trust` returns the trust value of a constraint, `correctpaths` returns the set of correct paths containing constraints. The *minimal* set of constraints with the lowest total removal cost is removed from the equality graph. C is a minimal set of constraints if removal of edges that correspond to a constraint in C results in a consistent equality graph, and if no subset of C is minimal.

Example:
The following expression illustrates the working of the algorithm. We assume that the function *plus* has type $Int \rightarrow Int \rightarrow Int$.

$$
\begin{array}{lll}
1. & \lambda a \rightarrow plus\ ((\lambda b \rightarrow \textbf{case}\ b\ \textbf{of} & \\
2. & \qquad\qquad\qquad\qquad\qquad\qquad True & \rightarrow\ b \\
3. & \qquad\qquad\qquad\qquad\qquad\qquad False & \rightarrow\ a\ )\ True\ )\ 3
\end{array}
$$

This expression is ill-typed because different type constants are assigned to variable b. There are three indications that b has type *Bool*: b has to match the two expressions on the left-hand side of the alternatives, and the term $(\lambda b \rightarrow \ldots)$ is applied to *True*. However, the first argument of *plus*, which is the type of the expressions on the right-hand side of the case expression, including expression b, must have type *Int*. Haskell produces the following error message for this type conflict:

```
ERROR "example.hs" (line 1): Type error in application
*** Expression     : plus ((\b -> case b of {...}) True) 3
*** Term           : (\b -> case b of {...}) True
*** Type           : Bool
*** Does not match : Int
```

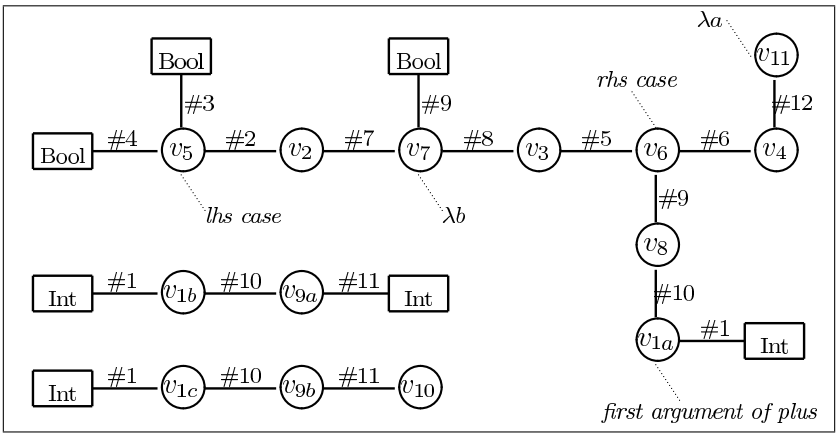

Figure 6: Inconsistent equality graph

The constraint-based approach leads to a better understanding of the type conflict. The set of constraints for this expression is generated:

$$
\begin{array}{lllll}
\#1 & : & v_1 & \equiv & \mathit{Int} \to \mathit{Int} \to \mathit{Int} \\
\#2 & : & v_5 & \equiv & v_2 \\
\#3 & : & v_5 & \equiv & \mathit{Bool} \\
\#4 & : & v_5 & \equiv & \mathit{Bool} \\
\#5 & : & v_6 & \equiv & v_3 \\
\#6 & : & v_6 & \equiv & v_4
\end{array}
\qquad
\begin{array}{lrcl}
\#7 & : & v_7 & \equiv & v_2 \\
\#8 & : & v_7 & \equiv & v_3 \\
\#9 & : & v_7 \to v_6 & \equiv & \mathit{Bool} \to v_8 \\
\#10 & : & v_1 & \equiv & v_8 \to v_9 \\
\#11 & : & v_9 & \equiv & \mathit{Int} \to v_{10} \\
\#12 & : & v_{11} & \equiv & v_4
\end{array}
$$

Constraints #1, #10 and #11 are the only constraints that demand a decomposition of a type variable. Decomposition of type variables results in the substitution:

$$
\begin{array}{lcl}
v_1 & \rightsquigarrow & v_{1a} \to v_{1b} \to v_{1c} \\
v_9 & \rightsquigarrow & v_{9a} \to v_{9b}
\end{array}
$$

After application of this substitution, the set of constraints is solved straightforwardly. Figure 6 shows the equality graph when the set of constraints is empty. The three error paths (from *Int* to *Bool*) in the graph represent the type conflict. To restore consistency (at least) one edge of each error path is removed. First, the minimal sets are computed: {#1}, {#5}, {#8}, {#9} and {#10}. In this example, the edges in the minimal sets are exactly those edges of the overlapping part of the error paths. Then the removal cost for each constraint is calculated. The total number of occurrences in either one of the four correct paths, combined with the trust value, result in a removal cost. Table 1 presents the removal costs. For the calculation, the high trust value assigned to constraint #1 (a reference to *plus*) is replaced by the value 10, with a default value of 1 for the other trust values. Finally, we compare the total removal cost of each minimal set. The lowest total removal cost is 1 for the minimal sets {#5} and {#8} is 1. The constraints in one of the two sets are removed to resolve the inconsistency.

	good	*trust*	*cost*
#1	1	*high*	20
#2	2	–	3
#3	2	–	3
#4	2	–	3
#5	–	–	1
#6	–	–	1

	good	*trust*	*cost*
#7	2	–	3
#8	–	–	1
#9	2	–	3
#10	1	–	2
#11	1	–	2
#12	–	–	1

Table 1: Removal costs

In Figure 6 extra information is provided about the origin of type variables and constraints. With this information we can produce a good explanation of the type conflict. The following error message is produced for the removal of constraint #8:

```
TYPE ERROR: conflicting types for variable b because:

=>  (\b -> {...}) is applied to True                              (line 1)
    results in b :: Bool

=>  b is in the rhs of case expression                            (line 2)
    which is used as the first argument of plus                   (line 1)
    results in b :: Int
```

8 Conclusion and future work

This paper presents a different approach to inferring the type of an expression. The approach tries to improve the quality and exactness of a reported error message for ill-typed expressions. The underlying concept is the construction of constraints on types, representing unification, instantiation and specialisation of types. Inconsistencies that are detected while solving these constraints are resolved using heuristics. Advantages of this approach are the following.

- The left-to-right bias is completely removed because the unification of types is delayed. The order in which constraints are solved does not influence the outcome.
- Heuristics help to point to the most likely error in the source. It is possible to add more heuristics.
- It is possible to produce multiple, and more useful error messages.

More heuristics have to be added to the system to construct better error messages for common mistakes. Several features and extensions of the expression language must be included in the type system before it can be used in a practical setting. One important extension for the type system is the introduction of *type* and *constructor* classes, which provide a way to overload functions. As a result of this extension, kind inferencing is necessary to determine appropriate kinds. Using type synonyms in reported error messages will increase understanding, but also introduces new problems for the type system.

Good error reporting facilities in a type assignment algorithm require additional overhead, for instance using a graph as an intermediate data structure, and maintaining histories about the deduction of type variables. Consequently, it is inevitable that our algorithm is less time-efficient. We want to investigate if it is possible to combine a time-efficient unification-based system with a constraint-based system with excellent error reporting, such that the two desired properties are maintained. Currently we are working on a unification-based type checker, that switches to the approach that is presented in this paper when a type conflict is encountered.

To proof the practicality of the approach, real data should be collected from the intended group of users, that is, programmers that are new to functional programming. These empirical measurements should be obtained in a practical setting, for instance in a *functional programming* course for first year students.

In Section 5 we briefly discussed the correctness of our algorithm with respect to the Hindley-Milner type inference rules. A formal proof of this property will appear in a forthcoming technical report.

References

[1] A. Aiken. Introduction to set constraint-based program analysis. In *Science of Computer Programming, 35(1)*, pages 79–111, 1999.

[2] A. Aiken, M. Fahndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proceedings of the second International Workshop on Types in Compilation*, Kyoto, Japan, March 1998.

[3] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, volume 2, pages 17–30, December 1993.

[4] Karen Bernstein. *Debugging Type Errors (Full version).* State University of New York at Stony Brook, November 1995. Technical Report.

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

[6] D. Duggan. Correct type explanation. In *Workshop on ML: ACM SIGPLAN, 1998*, pages 49–58, 1998.

[7] M. Gandhe, G. Venkatesh, and A. Sanyal. Correcting errors in the curry system. In *Chandrum V. and Vinay, V. (Eds.): Proc. of 16th Conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS vol. 1180, Springer-Verlag*, pages 347–358, 1996.

[8] Yang Jun. 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.

[9] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[10] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.

[11] Bruce J. McAdam. Generalising techniques for type explanation. In *Scottish Functional Programming Workshop*, pages 243–252, 1999. Heriot-Watt Department of Computing and Electrical Engineering Technical Report RM/99/9.

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

[13] 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.

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