Ordering type constraints: a structured approach

Jurriaan Hage and Bastiaan Heeren

Department of Information and Computing Sciences, Universiteit Utrecht
P.O.Box 80.089, 3508 TB Utrecht, The Netherlands
{jur, bastiaan}@cs.uu.nl

Abstract

The order in which unfications take place during type inference algorithms strongly determines where an error is detected. We use a constraint based type inferencer which separates between collecting, ordering, and solving constraints. We use a small set of combinators as part of the type rules to specify a degree of non-determinism in the order in which constraints may be solved. In the ordering phase, the user can make the type inference process (more) deterministic by choosing an appropriate ordering, yielding a large degree of control on his part: the scientist can easily compare the behaviors of various existing type inferencers and algorithms by describing them in terms of these orderings; the programmer can experiment with various orderings, and choose the one which suits his style of programming best. Compilers based on this technology naturally support multiple solvers, multiple error message for a single compilation, and using various orderings "in parallel", so that the user can easily browse through different views on the type error. The framework has been implemented, and used to build the Helium compiler for the language Haskell.

Keywords: type inferencing, algorithms, constraints, order of solving

1 INTRODUCTION

The Hindley-Milner type system lies at the basis of many algorithms and implementations that exist for type inferencing polymorphic functional languages, such as Haskell and ML. These algorithms and implementations all rely on the unification of types. However, the unifications they perform occur in different orders, with the result that inconsistencies, if they exist, are detected at different locations. The crucial realization is that these locations strongly determine the error message that is reported to the programmer, each giving a different view on the type error. An alternative would be to offer a message that incorporates all possible views at once, for instance by listing all the sites contributing to the error as in [14], which is a good option if the compiler has no further information as to which view is the most likely one. Our technology offers ways for the programmer to select the view most suitable to his taste, experience or whatever other factors may play a role, thus making the message more specific and concise.

Consider, for instance, the expression $\lambda f \rightarrow (f\ id, f\ True)$. Even for this simple expression, different algorithms put the 'blame' on different parts of this expression. The folklore algorithm $\mathcal{M}$ [8] reports that the constant *True* does not fit the expected type for *f*, while algorithm $\mathcal{W}$ [1] considers the application of *f* to *True* to

be at fault. The Haskell interpreter Hugs infers tuples from right-to-left, putting the blame on the expression $f\ id$. A purely bottom-up algorithm finds an error when it binds the various occurrences of f, while an algorithm such as $\mathcal{U}_{\mathrm{AE}}$ [15] stops when it considers the tuple after considering the components of the tuple.

The use of constraints as a means to specify a type system, or program analysis in general, has been with us for quite some time. In his introductory book [11], Pierce uses constraints more than once in his logical deduction rules, and some of the chapters of the second book, e.g. Pottier and Remy's The Essence Of ML Type Inference, are in a similar vein [12]. The advantage of such a formulation is that it distinguishes between the declarative specification of the analysis, and solving the collected constraints. In most cases, however, the order in which constraints are solved is left unspecified, although, as our example shows, the exact order in which the constraints are solved strongly determines where an inconsistency is detected. Also, depending on the types of constraints used in the type rules, there may be orders that are invalid, i.e., certain constraints must be solved before others to guarantee soundness. On the other hand, we want to be as flexible as possible.

In our approach, we take the abstract syntax tree as a starting point, where each constraint is associated with a node in the tree. To achieve this, the constraint generation phase constructs a constraint tree, e.g., by decorating the abstract syntax tree with sets of constraints. Which constraint ends up where is specified by means of a small number of combinators in the type rules. Some of these can be used to forbid certain orders, e.g., the constraints of a let-definition should be solved before the body of that let, others allow us to treat the constraints generated in a single node of the abstract syntax tree differently, by associating them not with the node itself, but with one of its subtrees. We shall see examples of this later.

In the subsequent ordering phase, the constraint tree is converted to a list of constraints based on an ordering strategy. Many of the well-known type inference algorithms can be emulated simply by choosing the appropriate ordering. The ease with which we can capture so many existing algorithms is a further justification of this work. The list of constraints can then be fed into any of a whole range of solvers. This set-up opens up a host of opportunities: By compiling a program with different orderings we obtain multiple "views" on the same type error. A programmer can experiment with orderings to find out which fits his style of programming best. The framework also allows researchers in the area to compare these algorithms to each other and to alternative types of solvers, such as a type graph solver [3]. The system also lends itself naturally to generating multiple (independent) type error messages in a single compilation.

We used our library to implement the type inference process in the Helium compiler [6]. Since Helium implements almost the entire Haskell 98 standard [10], it shows that our approach scales well (see Chapter 6 of [4] for this much more extensive type system). Note, however, that nothing in our framework restricts its use to Haskell, nor to type inference, because new kinds of constraints can be added with ease.

The paper is structured as follows. In Section 3, we discuss the constraints that

we use. The major part of this paper is devoted to a description of operators that construct constraint trees (Section 4). We illustrate these operators by presenting type rules for the lambda calculus with polymorphic let in Section 5, and show how we can emulate the unification orders of some important algorithms in Section 6. The type system we discuss in this paper is large enough to compare it with existing algorithms.

2 PRELIMINARIES

We use a three layer type language: besides mono types (τ) we have type schemes (σ), and ρ's, which are either type scheme variables (σ_v) or type schemes.

$$\begin{array}{lll} \tau & ::= & a \mid \mathit{Int} \mid \mathit{Bool} \mid \tau_1 \rightarrow \tau_2 \\ \sigma & ::= & \tau \mid \forall a.\sigma \\ \rho & ::= & \sigma \mid \sigma_v \end{array}$$

The function $\mathit{ftv}(\sigma)$ returns the free type variable of its argument, and is defined in the usual way: bound variables in σ are omitted from the set of type variables in σ. For notational convenience, we represent $\forall a_1.\cdots\forall a_n.\tau$ by $\forall a_1 \ldots a_n.\tau$, and abbreviate $a_1 \ldots a_n$ by a vector of type variables $\overline{a}$; we insist that all a_i are different. We assume to have an unlimited supply of fresh type variables, denoted by β, β', β_1 etcetera. We use $v_0, v_1, \ldots$ for concrete type variables.

A substitution S is a mapping from type variables to types. Application of a substitution S to a type τ is denoted $S(\tau)$. All our substitutions are idempotent, i.e. $S(S(\tau)) = S(\tau)$, and *id* denotes the empty substitution. We use the syntax $[a_1 := \tau_1, \ldots, a_n := \tau_n]$ to denote a substitution that maps a_i to τ_i (we insist that all a_i are different). Again, vector notation abbreviates this to $[\overline{a} := \overline{\tau}]$.

We can generalize a type to a type scheme while excluding the free type variables of some set $\mathcal{M}$, which are to remain monomorphic. Dually, we instantiate a type scheme by replacing the bound type variables with fresh type variables:

$$\begin{array}{lll} \mathit{gen}(\mathcal{M}, \tau) & =_{def} & \forall \overline{a}.\tau \quad \text{where } \overline{a} = \mathit{ftv}(\tau) - \mathit{ftv}(\mathcal{M}) \\ \mathit{inst}(\forall \overline{a}.\tau) & =_{def} & S(\tau) \quad \text{where } S = [\overline{a} := \overline{\beta}] \text{ and all in } \overline{\beta} \text{ are fresh} \end{array}$$

A type is an instance of a type scheme, written as $\tau_1 < \forall \overline{a}.\tau_2$, if there exists a substitution S such that $\tau_1 = S(\tau_2)$ and $\mathit{domain}(S) \subseteq \overline{a}$.

3 THE CONSTRAINTS

In this section, we describe a constraint language for type systems based on Hindley-Milner. For each kind of constraint, we define syntax, semantics and how they can be solved.

In our framework, a constraint may carry any kind of additional information, e.g., the reason why the constraint was generated. Typically, the amount of information carried around is enough to construct an error message if the constraint leads

to an inconsistency. Because of this genericity, we omit the constraint information carried by a constraint in this paper.

With the following constraints we can express type equivalence for monomorphic types, generalization, and instantiation.

$$c ::= (\tau_1 \equiv \tau_2) \mid \sigma_v := \textsc{Gen}(\mathcal{M}, \tau) \mid \tau \preceq \rho$$

With a generalization constraint we can generalize a type with respect to a set of monomorphic type variables $\mathcal{M}$, and assign the resulting type scheme to a type scheme variable. Instantiation constraints express that a type should be an instance of a type scheme, or the type scheme associated with a type scheme variable.

The generalization and instance constraints are used to handle the polymorphism introduced by let expressions. It is possible to use only equivalence constraints, but that comes at a price. Essentially, for each occurence of a let-defined variable, we must then duplicate the set of constraints associated with it, and thus much of our work. And if a such a set is itself inconsistent, the error is duplicated each time as well.

We use special type scheme variables to function as place-holders for unknown type schemes. The reason for this is that we do not want to solve constraints during the constraint generation phase. We shall see shortly that our method does induce a certain bias in the sense that we may only generalize types when we are certain that variables considered to be polymorphic will not become monomorphic due to future unifications.

Our generalization and instance constraints are the decomposition of what we called the implicit instance constraint in an earlier report [5]. The generalization and instantation constraints that we use are similar to the *let* constraint of Pottier and Rémy[12]. The main difference is that Pottier and Rémy convert an abstract syntax tree into a constraint that largely follows the structure of the program, e.g., their *let* constraint contains all the information imposed by both the definitions and the body of the let. Then they define a rather complicated rewrite system (Figure 10-11 of [12]) that traverses this constraint, to determine whether this constraint is satisfiable. Furthermore, they are not interested in finding out why it may not be satisfiable, and we see no way to make such provisions easily. We on the other hand have chosen our constraints to be as small and manipulable as possible, which makes building a solver a straightforward exercise [5], and allows for the reordering of constraints to influence the outcome of the type inference process.

Both instance and equality constraints are lifted to work on lists of pairs, where each pair consists of an identifier and a type (or type scheme). For instance,

$$A \equiv B \;\; =_{def} \;\; \{\tau_1 \equiv \tau_2 \mid (x:\tau_1) \in A, (x:\tau_2) \in B\} \; .$$

Our solution space for solving constraints consists of a pair of mappings (S, Σ), where S is a substitution on type variables, and Σ a substitution on type scheme variables. Next, we define semantics for these constraints: the judgement $(S, \Sigma) \vdash_s c$

expresses that the constraint c is satisfied by the substitutions (S,Σ).

$$\begin{array}{lll}
(S,\Sigma) \vdash_s \tau_1 \equiv \tau_2 & =_{def} & S(\tau_1) = S(\tau_2) \\
(S,\Sigma) \vdash_s \sigma_v := \textsc{Gen}(\mathcal{M},\tau) & =_{def} & S(\Sigma(\sigma_v)) = \mathit{gen}(S(\mathcal{M}),S(\tau)) \\
(S,\Sigma) \vdash_s \tau \preceq \rho & =_{def} & S(\tau) < S(\Sigma(\rho))
\end{array}$$

We explain how each of the constraints can be solved, formulated as a rewrite system. In addition to the solution itself, we add the set of constraints to be solved as the first element, and update it along the way.

$$\begin{array}{lll}
(\{\tau_1 \equiv \tau_2\} \cup \mathcal{C}, S, \Sigma) & \rightarrow & (S'(\mathcal{C}), S' \circ S, \Sigma) \text{ where } S' = \mathit{mgu}(\tau_1,\tau_2) \\
(\{\sigma_v := \textsc{Gen}(\mathcal{M},\tau)\} \cup \mathcal{C}, S, \Sigma) & \rightarrow & (\Sigma'(\mathcal{C}), S, \Sigma' \circ \Sigma) \text{ where } \Sigma' = [\sigma_v := \mathit{gen}(\mathcal{M},\tau)] \\
 & & \quad \text{if } \mathit{ftv}(\tau) \cap \mathit{actives}(\mathcal{C}) \subseteq \mathit{ftv}(\mathcal{M}) \\
(\{\tau \preceq \sigma\} \cup \mathcal{C}, S, \Sigma) & \rightarrow & (\{\tau \equiv \mathit{inst}(\sigma)\} \cup \mathcal{C}, S, \Sigma)
\end{array}$$

We use the standard algorithm *mgu* for finding a most general unifier of two types [13]. We already mentioned that our solving process imposes a certain order on when constraints can be solved. This fact is now apparent in the side conditions for the generalization and instantiation constraints. Observe the implicit side condition for solving an instantiation constraint: we insist that the right hand side is a type scheme *and not a type scheme variable*. This implies that the corresponding generalization constraint has been solved, and the type scheme variable was replaced by a type scheme. When we generalize a type τ due to a generalization constraint, the polymorphic type variables in that type are (conceptually) renamed so that their former identity is lost. This means that we must ensure that this identity plays no role in the future. This is the case if the type variable does not occur any longer in the constraint set, unless it only occurs in a position in which it is considered to be polymorphic. This notion is formalized in the definition of activeness, which is straightforward [4],

When none of the rules can be applied to a given non-empty constraint set, then the set is inconsistent, and the error solution, $(\emptyset,\top,\top)$ is returned. Such a solution trivially satisfies every constraint. That our non-deterministic rewriting system is sound and complete with respect to the semantics can be formulated as follows.

Theorem 1 *If $(\mathcal{C}, id, id) \rightarrow^* (\emptyset, S, \Sigma)$, then $(S,\Sigma) \vdash_s \mathcal{C}$. In fact, it is the most general solution that satisfies $\mathcal{C}$.*

Proof. The proof is similar to that of Theorem 4.9 in [4]. Note that the implicit instance constraints in that proof can easily be mapped to a pair of generalization and instance constraints. □

4 CONSTRAINT ORDERING

In the previous section we discussed how, non-deterministically, sets of constraints can be solved. In a practical setting such as a compiler, we want to make this

process more deterministic by linearizing the set of constraints, and then feeding these one by one into a solver. The idea is that, to an extent depending on the solver, this determines which constraint is responsible for any inconsistency, and thus what kind of error message will be given. A second advantage is that we can restrict ourselves to linearizations for which we do need to check the side conditions for the generalization and instantiation constraints: they will be guaranteed to hold when the constraints are solved.

To obtain more control over the order of the constraints, we collect the constraints in a tree. This tree has the same shape as the abstract syntax tree of the expression for which the constraints are generated. In fact, an implementation may simply associate constraints with nodes in the abstract syntax tree itself. A constraint is *attached* to the node N where it is generated. Furthermore, we may choose to *associate* it explicitly with one of the subtrees of N. Some language constructs demand that some constraints have to be solved before others, and we encode this in the constraint tree as well.

We consider four alternatives for constructing a constraint tree.

$$\mathcal{T}_C ::= \{\!|\, \mathcal{T}_{C1}, \ldots, \mathcal{T}_{Cn} \,|\!\} \mid c \Diamond \mathcal{T}_C \mid c \triangledown \mathcal{T}_C \mid \mathcal{T}_{C1} \ll \mathcal{T}_{C2}$$

To minimize the use of parentheses, all operators to build constraint trees are right associative. With the first alternative we combine a list of constraint trees into a single tree consisting of a root with the $\mathcal{T}_{Ci}$ as subtrees. The second and third alternatives add a single constraint to a tree. The case $c \Diamond \mathcal{T}_C$ is the most straightforward one: it makes the constraint part of the constraint set associated with the root of $\mathcal{T}_C$. A typical constraint to associate with a let-node is the constraint that the type of the body of the let equals the type of the let (see (LET) in Figure 3).

However, some of the constraints are more naturally associated with a subtree of a given node, such as the constraint that the condition of an if-then-else expression must have type *Bool*. In that case, we write $c \triangledown \mathcal{T}_C$, where $\mathcal{T}_C$ is the constraint tree obtained for the condition. To summarize, in the constraint tree $c \Diamond \{\!|\, \mathcal{T}_{C1}, \mathcal{T}_{C2} \,|\!\}$, c is a constraint generated by the root and associated with the root, while the constraint c in $\{\!|\, c \triangledown \mathcal{T}_{C1}, \mathcal{T}_{C2} \,|\!\}$ is generated by the root, but associated with its first subtree. The last case ($\mathcal{T}_{C1} \ll \mathcal{T}_{C2}$) combines two trees in a strict way: all the constraints in $\mathcal{T}_{C1}$ should be considered before the constraints in $\mathcal{T}_{C2}$. The typical example is that of the constraints for the definition in a let and those for the body.

Most of our operators add a single constraint to a constraint tree, e.g., $c \Diamond \mathcal{T}_C$. For brevity, we also use the underlined version of such an operator lifted to lists of constraints in a straightforward fashion. For example,

$$[c_1, \ldots, c_n] \mathbin{\underline{\Diamond}} \mathcal{T}_C \quad =_{def} \quad c_1 \Diamond \ldots \Diamond c_n \Diamond \mathcal{T}_C.$$

This also applies to similar operators to be defined later in this paper. We abbreviate the empty constraint tree $\{\!|\, |\!\}$ with $\bullet$, and $\mathcal{C}^\bullet$, the tree consisting of a single node with a constraint set attached to it, is defined as $\mathcal{C} \mathbin{\underline{\Diamond}} \bullet$.

In the remaining part of this section, we discuss various constraint ordering strategies: the flattening of constraint trees, specified by means of a tree walk, and spreading and phasing for transforming a constraint tree.

4.1 Flattening a constraint tree

Our first concern is how to flatten a constraint tree to a list: for this, we use the function *flatten*. How a tree is flattened depends on the tree walk of our choice, which is a parameter of *flatten*. A tree walk specifies the order of the constraints for a single node in the constraint tree. We use the following Haskell datatype to represent a tree walk.

$$\textbf{data}\ \mathit{TreeWalk} = \mathit{TW}\ (\forall a.[a] \to [([a],[a])] \to [a])$$

The first argument of the tree walk function specifies the constraints belonging to the node itself, the second one contains pairs of lists of constraints, one for each child of the node. The first element of such a pair contains the constraints for the subtree, the second element those constraints associated by the node with the subtree.

The function *flatten* has the following type signature:

$$\mathit{flatten} :: \mathit{TreeWalk} \to \mathit{ConstraintTree} \to [\mathit{Constraint}]$$

The *flatten* function simply traverses the constraint tree, and for most nodes lets the *TreeWalk* determine how the constraints attached to the node itself, the constraints attached to the various subtrees and the lists of constraints from the subtrees themselves, should be turned into a single list. Only if the node is a strict node, then the order in which the constraints are put together is fixed.

The first tree walk we define is truly bottom-up.

$$\begin{array}{l}\mathit{bottomUp} = \mathit{TW}\ (\lambda \mathit{down}\ \mathit{list} \to f\ (\mathit{unzip}\ \mathit{list}) +\!\!+ \mathit{down}) \\ \quad \textbf{where}\ f\ (\mathit{csets}, \mathit{ups}) = \mathit{concat}\ \mathit{csets} +\!\!+ \mathit{concat}\ \mathit{ups}\end{array}$$

This tree walk puts the recursively flattened constraint subtrees up front, while preserving the order of the trees. These are followed by the constraints associated with each subtree in turn. Finally, we append the constraints attached to the node itself.

Example 1. Assume that $\mathcal{T}_C = \mathit{down} \mathbin{\underline{\Diamond}} \{\!|\, \mathit{up}_1 \underline{\triangledown} C_1^\bullet, \ldots, \mathit{up}_n \underline{\triangledown} C_n^\bullet \,|\!\}$. Flattening this constraint tree with the bottom-up tree walk gives us

$$\mathit{flatten}\ \mathit{bottomUp}\ \mathcal{T}_C \quad = \quad C_1 +\!\!+ \ldots +\!\!+ C_n +\!\!+ \mathit{up}_1 +\!\!+ \ldots +\!\!+ \mathit{up}_n +\!\!+ \mathit{down}$$

Similarly, we define the dual tree walk, which is a top-down approach.

$$\begin{array}{l}\mathit{topDown} = \mathit{TW}\ (\lambda \mathit{down}\ \mathit{list} \to \mathit{down} +\!\!+ f\ (\mathit{unzip}\ \mathit{list})) \\ \quad \textbf{where}\ f\ (\mathit{csets}, \mathit{ups}) = \mathit{concat}\ \mathit{ups} +\!\!+ \mathit{concat}\ \mathit{csets}\end{array}$$

Example 1 (continued). If we use this tree walk to flatten $\mathcal{T}_C$, then we obtain

$$\mathit{flatten}\ \mathit{topDown}\ \mathcal{T}_C \quad = \quad \mathit{down} +\!\!+ \mathit{up}_1 +\!\!+ \ldots +\!\!+ \mathit{up}_n +\!\!+ C_1 +\!\!+ \ldots +\!\!+ C_n$$

Other useful treewalks are those that interleave the upward constraints and the flattened constraint trees at each node. Here, we have two choices to make: do the the upward constraints precede or follow the constraints from the corresponding child, and do we put the downward constraints in front or at the end of the list? These two options lead to the following helper-function.

$$\begin{aligned}&\mathit{variation} :: (\forall a.[a] \to [a] \to [a]) \to (\forall a.[a] \to [a] \to [a]) \to \mathit{TreeWalk}\\ &\mathit{variation}\ f_1\ f_2 = \mathit{TW}\ (\lambda \mathit{down}\ \mathit{list} \to f_1\ \mathit{down}\ (\mathit{concatMap}\ (\mathit{uncurry}\ f_2)\ \mathit{list}))\end{aligned}$$

For both arguments of *variation*, we consider two alternatives: combine the lists in the order given ($+\!\!+$), or flip the order of the lists (*flip* ($+\!\!+$)). For instance, this results in the following behavior:

$$\mathit{flatten}\ (\mathit{variation}\ (+\!\!+)(+\!\!+))\ \mathcal{T}_C \quad = \quad \mathit{down} +\!\!+ C_1 +\!\!+ \mathit{up}_1 +\!\!+ \ldots +\!\!+ C_n +\!\!+ \mathit{up}_n$$

Our next, and final, example is a tree walk transformer: at each node in the constraint tree, the children are inspected in reversed order. Of course, this reversal is not applied to nodes with a strict ordering. With this transformer, we can inspect a program from right-to-left, instead of the standard left-to-right order.

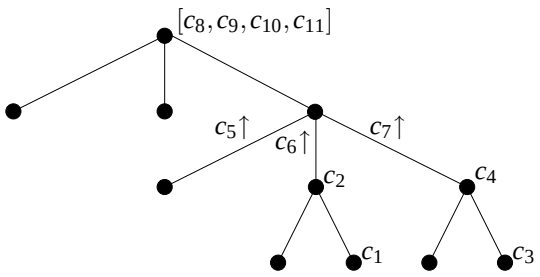
$$\begin{aligned}&\mathit{reversed} :: \mathit{TreeWalk} \to \mathit{TreeWalk}\\ &\mathit{reversed}\ (\mathit{TW}\ f) = \mathit{TW}\ (\lambda \mathit{down}\ \mathit{list} \to f\ \mathit{down}\ (\mathit{reverse}\ \mathit{list}))\end{aligned}$$

We conclude our discussion on flattening a constraint tree with an example, which illustrates the impact of the order of constraints.

Example 3. We use the type rules in Figure 3 to generate the constraints for the expression given below. We take the liberty of including a conditional in the example although we do not give the type rule; it simply enforces that the type of the condition is *Bool*, and that the type of the conditional as a whole equals the type of the then and the else branches.) Various parts of the expression are annotated with their assigned type variable. Furthermore, v_9 is assigned to the if-then-else expression, and v_{10} to the complete expression.

$$\begin{array}{ccccccccccccccc} & & & & & & & & \overbrace{}^{v_5} & & & & \overbrace{}^{v_8} & & \\ \lambda & f & b & \to & \textbf{if} & b & \textbf{then} & f & 1 & \textbf{else} & f & \mathit{True} & & & \\ & v_0 & v_1 & & & v_2 & & v_3 & v_4 & & v_6 & v_7 & & & \end{array}$$

We continue with the constraint tree $\mathcal{T}_C$ depicted in Figure 1. The constraints in this tree are inconsistent: the constraints in the only minimal inconsistent subset are marked with a star. Hence, a sequential constraint solver will report the last of the



$$
\begin{array}{lcl}
{c_1}^* & = & v_4 \equiv \mathit{Int} \\
{c_2}^* & = & v_3 \equiv v_4 \to v_5 \\
{c_3}^* & = & v_7 \equiv \mathit{Bool} \\
{c_4}^* & = & v_6 \equiv v_7 \to v_8 \\
c_5 & = & v_2 \equiv \mathit{Bool} \\
c_6 & = & v_5 \equiv v_9 \\
c_7 & = & v_8 \equiv v_9 \\
{c_8}^* & = & v_0 \equiv v_3 \\
{c_9}^* & = & v_0 \equiv v_6 \\
c_{10} & = & v_1 \equiv v_2 \\
c_{11} & = & v_{10} \equiv v_0 \to v_1 \to v_9
\end{array}
$$

FIGURE 1. A constraint tree

marked constraints in the list as incorrect. We consider three flattening strategies. The underlined constraints are the locations where the inconsistency is detected.

$$
\begin{array}{lcl}
\mathit{flatten\ bottomUp}\ \mathcal{T}_C & = & [c_1, c_2, c_3, c_4, c_5, c_6, c_7, c_8, \underline{c_9}, c_{10}, c_{11}] \\
\mathit{flatten\ topDown}\ \mathcal{T}_C & = & [c_8, c_9, c_{10}, c_{11}, c_5, c_6, c_7, c_2, c_1, c_4, \underline{c_3}] \\
\mathit{flatten}\ (\mathit{reversed\ topDown})\ \mathcal{T}_C & = & [c_8, c_9, c_{10}, c_{11}, c_7, c_6, c_5, c_4, c_3, c_2, \underline{c_1}]
\end{array}
$$

Observe that for each of the tree walks, the inconsistency shows up while solving a different constraint. These constraints originated from the root of the expression, the subexpression *True*, and the subexpression 1, respectively.

If a constraint tree retains information about the names of the constructors of the abstract tree, then the definition of *flatten* can easily be generalized to treat certain language constructs differently:

$$\mathit{flatten} :: (\mathit{String} \to \mathit{TreeWalk}) \to \mathit{ConstraintTree} \to [\mathit{Constraint}]$$

This extension enables us to model inference processes such as the one of Hugs [7] which infers tuples from right to left, while most other constructs are inferred left-to-right. It also allows us to emulate all instances of $\mathcal{G}$ [9], such as exhibiting $\mathcal{M}$-like behavior for one construct and $\mathcal{W}$-like behavior for another.

Of course, *flatten* could be generalized further to include other orderings. For example, a tree walk that visits the subtree with the most type constraints first.

4.2 Spreading type constraints

We present a technique to move type constraints from one place in the constraint tree to a different location. This can be useful if constraints generated at a certain place in the abstract syntax tree are also related to a second location. In particular,

we will consider constraints that relate the definition site and the use site of an identifier. The advantage is that we get more ways to reorganize the type constraints after constraint generation, without changing the type rules themselves (which can be dangerous since we may inadvertently change the type system). More specifically, by spreading constraints we can also emulate algorithms that use a top-down type environment, even though our rules use a bottom-up assumption set to collect the constraints.

The grammar for constraint trees is extended with three cases.

$$\mathcal{T}_C ::= (\ldots) \mid (\ell, c) \mathbin{\triangledown^\circ} \mathcal{T}_C \mid (\ell, c) \mathbin{\lll^\circ} \mathcal{T}_C \mid \ell^\circ$$

The first two cases serve to spread a constraint, whereas the third marks a position in the tree to receive such a constraint. Labels ℓ are used only to find matching spread-receive pairs. The scope of spreading a constraint is limited to the right argument of $\triangledown^\circ$ (and $\lll^\circ$). We expect for every constraint that is spread to have exactly one receiver in its scope.

The function *spread* is responsible for passing down constraints deeper into the tree, until they end up at their destination label. For reasons of brevity we only give a type signature.

$$\mathit{spread} :: \mathit{ConstraintTree} \to \mathit{ConstraintTree}$$

The type rules specify whether a certain constraint can potentially be spread. To actually perform spreading is a choice that is made by the programmer. This implies that we have to specify how *flatten* handles both $\triangledown^\circ$ and $\lll^\circ$. Only the flatten function actually distinguishes between the non-strict $\triangledown^\circ$ and the strict version $\lll^\circ$, essentially by forgetting the $\circ$.

Example 3 (continued). We spread the type constraints introduced for the monomorphic pattern variables f and b to their use sites. Hence, the constraints c_8, c_9, and c_{10} are moved to a different location in the constraint tree. We put a receiver at the three nodes of the variables (two for f, one for b). The type variable that is assigned to an occurrence of a variable (which is unique) is also used as the label for the receiver. Hence, we get the receivers v_2°, v_3°, and v_6°. The constraint tree after spreading ($\mathcal{T}_C{}'$) is displayed in Figure 2.

$$\begin{array}{lcl}
\mathit{flatten\ bottomUp}\ \mathcal{T}_C{}' & = & [c_{10}, c_8, c_1, c_2, c_9, c_3, \underline{c_4}, c_5, c_6, c_7, c_{11}] \\
\mathit{flatten\ topDown}\ \mathcal{T}_C{}' & = & [c_{11}, c_5, c_6, c_7, c_{10}, c_2, c_8, c_1, c_4, c_9, \underline{c_3}] \\
\mathit{flatten}\ (\mathit{reversed\ bottomUp})\ \mathcal{T}_C{}' & = & [c_3, c_9, c_4, c_1, c_8, \underline{c_2}, c_{10}, c_7, c_6, c_5, c_{11}]
\end{array}$$

The *bottomUp* tree walk after spreading leads to reporting the constraint c_4: without spreading type constraints, c_9 is reported.

One could say that spreading undoes the bottom-up construction of assumption sets for the free identifiers, and instead applies the more standard approach to pass down a type environment (usually denoted by Γ). Therefore, spreading type constraints gives a constraint tree that corresponds more closely to the type inference

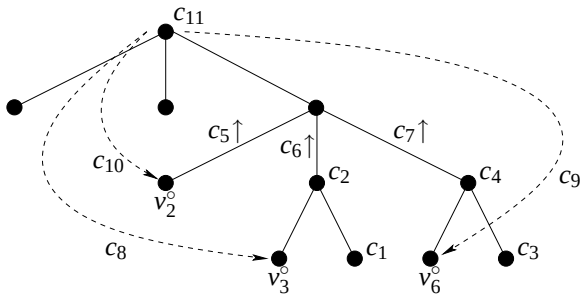


FIGURE 2. A constraint tree with type constraints that have been spread

process of Hugs [7] and GHC [2]. Regarding the inference process for a conditional expression, both compilers constrain the type of the condition to be of type *Bool* before continuing with the then and else branches. GHC constrains the type of the condition even before its type is inferred: Hugs constrains this type afterwards. Therefore, the inference process of Hugs for a conditional expression corresponds to an inorder bottom-up tree walk. The behavior of GHC can be mimicked by an inorder top-down tree walk.

4.3 Phasing constraint trees

The phasing operator for constraint trees is not treated in detail in this paper (see [4]). We restrict ourselves to a short high-level description. Phasing can be used to model non-local influences on the order of constraints by assigning a phase number to each constraint and solving constraints with phase number i before those with phase number $i+1$. It can be used to take advantage of type signatures by assigning a low phase number to constraints originating from a type signature. This amounts to 'pushing down' an expected type.

For example, consider a function f with explicit type $Int \to Int$. Then f's formal parameter has type *Int*, which is expressed by an equality constraint that can be solved before considering the function definition. This is in fact what GHC does. If the type of f contains type variables, then the situation is somewhat more complicated: for this, we need skolemization constraints, which are rather similar to the instantiation constraints.

Another application of phasing is to assign early phase numbers to constraints that were generated (and satisfied) during an earlier compilation. This has the effect of putting the blame on more recently developed pieces of code.

5 AN EXAMPLE TYPE SYSTEM

The type rules for the following simple language (variables, application, lambda abstraction and a non-recursive let) are presented in Figure 3.

$$e ::= x \mid e_1\ e_2 \mid \lambda x \to e \mid \textbf{let}\ x = e_1\ \textbf{in}\ e_2$$

These rules specify how to construct a constraint tree for a given expression (see [4] for a much more extensive example). The type rules are formulated in terms of judgements of the form $\mathcal{M}, \mathcal{A}, \mathcal{T}_C \vdash e : \tau$. Such a judgement should be read as: "given a set of types $\mathcal{M}$ that are to remain monomorphic, we can assign type τ to expression e if the type constraints in $\mathcal{T}_C$ are satisfied, and if $\mathcal{A}$ enumerates all the types that have been assigned to the identifiers that are free in e". The set $\mathcal{M}$ of monomorphic types is provided by the context: it is passed top-down. The assumption set $\mathcal{A}$ contains an assumption $(x : \beta)$ for each unbound *occurrence* of x (here β is a fresh type variable). Hence, $\mathcal{A}$ can have multiple assertions for the same identifier. For now, the operator $+\!\!+$ should be read as concatenation, and $\mathcal{A}\backslash x$ denotes the removal of all assumptions about x from $\mathcal{A}$.

All our type rules maintain the invariant that each subexpression is assigned a fresh type variable. For example, consider the type rule APPLY. Here, τ_1 is a placeholder for the type of e_1, and is used in the constraint $\tau_1 \equiv \beta_1 \to \beta_2$. Because of the invariant, we know that τ_1 is actually a type variable, and we have no clue about the type it will become. During the solving process, type variables will be replaced by more complicated types. Remember that the result of every preceding solving step (which is a substitution) is immediately applied to the remaining constraints.

We could have replaced c_i $(i = 1,2,3)$ in the type rule APPLY with a single constraint $\tau_1 \equiv \tau_2 \to \beta_3$. Decomposing this constraint, however, opens the way for fine-grained control over when a certain fact is checked. In the type rule (LET), we use $\ll^\circ$ for spreading the instantiation constraints, and not $\triangledown^\circ$. As a result, we have made the rule for the LET a bit more restrictive than necessary: it is perfectly possible that some constraints coming from the body can be solved before constraints arising from the definitions. This decision is practically motivated: by choosing this strict ordering, we can be sure that the side condition for a generalization constraint is fulfilled when we encounter it in the constraint list, and thus the solver does not need to verify the condition, or look for constraints for which the condition is satisfied.

In Section 3 we formulated the solving process based on sets of constraints. From now on, we consider the set of constraints as a list (one which has been obtained via a flattening of the constraint tree for the expression), and solve the constraints in this order. In other words, the operator $\cup$ in $\{c\} \cup \mathcal{C}$ should be read as 'insert at or take from the front'. The soundness of our algorithm with respect to the type rules is formulated as follows.

Theorem 2 *Let $\mathcal{M}, \mathcal{A}, \mathcal{T}_C \vdash e : \tau$ for a closed expression e, and let $\mathcal{C}$ be a list of constraints obtained by flattening $\mathcal{T}_C$. If $(\mathcal{C}, id, id) \to^* (\emptyset, S, \Sigma)$, then $\vdash_{\text{HM}} e : S\tau$, where $\vdash_{\text{HM}}$ denotes the Hindley-Milner type system.*

Proof. The proof is similar to that of Theorem 4.16 in [4], again with the note that implicit instance constraints can be replaced by separate generalization and instantiation constraints. A notable difference with this proof is that we have to

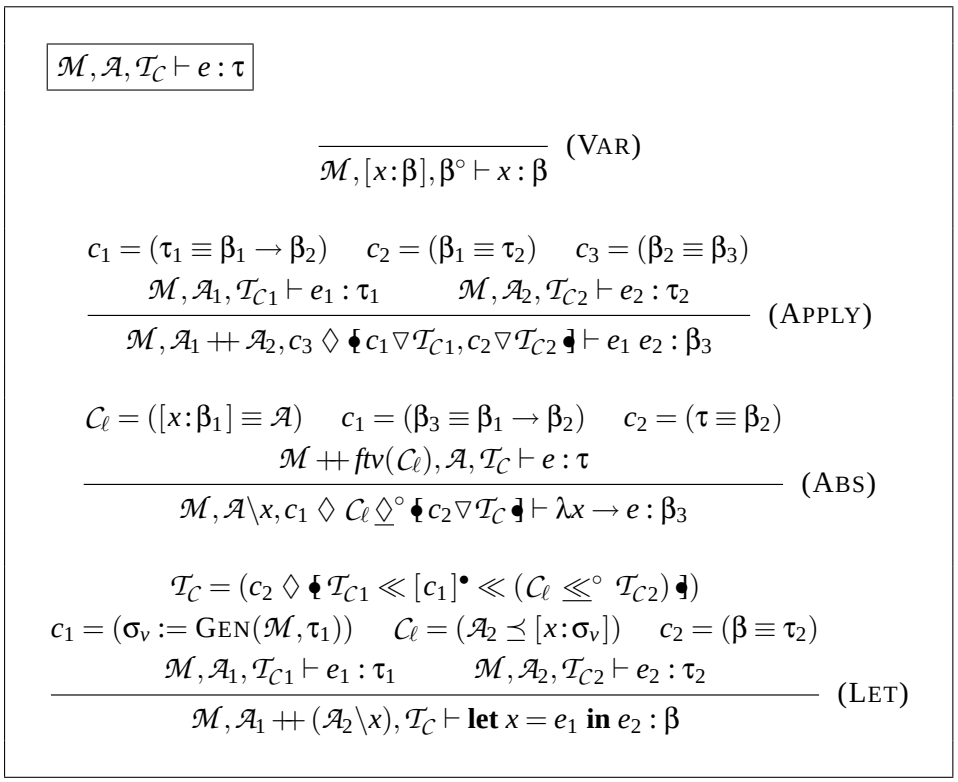


FIGURE 3. Type rules for a simple expression language

take the flattening into account. The use of the $\ll$ operator in the rule for the let, and the fact that *flatten* obeys this strict order imply that the constraints of the let definitions are solved before those of the body. Thus, solving $\mathcal{C}$ does not block on the side conditions of the generalization or the instantiation constraint, and the result follows from Theorem 1. □

A corollary is that if any of the algorithms for implementing the Hindley-Milner type system fails, then ours fails as well. The only observable difference is that the algorithms might fail 'at different points'. In the next section, we present constraint ordering strategies for emulating some classic algorithms, such that a simple sequential solver stops at the same point.

6 COMPARISONS TO OTHER TYPE INFERENCING ALGORITHMS

We now have everything set-up for a comparison of our algorithm with existing algorithms in the literature. We consider the classic algorithms, $\mathcal{W}$ [1] and $\mathcal{M}$ [8], but also $\mathcal{G}$ [9] and $\mathcal{U}_{\text{AE}}$ [15].

Consider first the standard algorithm $\mathcal{W}$. The algorithm proceeds in a bottom-up fashion, and considers the children from left-to-right. Second, $\mathcal{W}$ treats the let-expression in exactly the same way as we do: first the definition, followed by generalization, and finally the body. Finally, we see that a type environment is passed down. Together this implies that the combination of the *bottomUp* tree walk with spreading corresponds to algorithm $\mathcal{W}$. Note that because spreading is used, the *bottomUp* tree walk ensures that the algorithm only fails at applications.

Similarly, the folklore algorithm $\mathcal{M}$ is a top-down inference algorithm which also uses a type environment. Unification now takes place in the identifier and lambda nodes, and not in the application node. These properties determine that spreading in combination with the *topDown* tree walk emulates $\mathcal{M}$.

Algorithm $\mathcal{G}$ defined by Lee and Yi [9] is a combination of $\mathcal{M}$ and $\mathcal{W}$ and generalizes both. It actually defines a set of algorithms. The essential idea is that the unifications of $\mathcal{W}$ and $\mathcal{M}$ are broken into pieces so that some of these can be performed when arriving at a certain node, some between two subtree visits, and some before going back up. This decomposition can be chosen independently for each non-terminal.

To see how we can model the choices to be made in $\mathcal{G}$, we consider the case for applications $e_1\ e_2$. The formulation of Lee and Yi allows us to start with choosing a θ_1 equal to either a fresh type variable, a function type in which the argument and result types are fresh, or a function type in which the argument is fresh and the result type is equal to the expected type ρ (see constraint *(2)* in Fig. 3 of [9]). Next, we can decide to strengthen our demands on the type of e_1, or postpone this to after considering e_2 *(3)*. For the argument position we decide what to pass down: the type β (which is partly known from inferring e_1's type), or a fresh type variable *(4)*. After visiting e_2, all constraints are considered again to make sure that which was not checked before, is assuredly taken care of.

Instead of exhaustively listing all the possibilities and showing how these can be specified in our system by means of some tree walk, we illustrate by considering algorithm $\mathcal{H}$ defined in [9]. The corresponding tree walk should give $[c_1] +\!\!+ \mathcal{C}_1 +\!\!+ [c_3, c_2] +\!\!+ \mathcal{C}_2$, where $\mathcal{C}_i$ is the flattened list of constraints for subtree e_i, and the c_i are the constraints mentioned in the type rule for applications.

With the realization that our algorithm can handle a great variety of constraint orderings (including reversed orders), the above can be summarized as follows:

Theorem 3 *The algorithm described in this paper strictly generalizes algorithms $\mathcal{W}$, $\mathcal{M}$ and $\mathcal{G}$ (and thus also the inference algorithms of the OCaml and SML/NJ compilers).*

6.1 Algorithm $\mathcal{U}_{\mathrm{AE}}$

Yang describes a type inference algorithm which proceeds by unifying assumption environments [15]. The main idea is to handle the two subexpressions of an application independently in order to get rid of the left-to-right bias. The basic characteristic of algorithm $\mathcal{U}_{\mathrm{AE}}$ is that for a compound expression such as the application

$e_1\ e_2$, information obtained from analysis of e_1 cannot influence the analysis of e_2. Their solution is to first recursively analyze the subexpressions, and only afterwards unify the resulting assumption environments, which record the types of identifiers used in these subexpressions.

Take a look at the following erroneous expression:

$$f = \lambda\ \underset{v_1}{x}\ \rightarrow (\textbf{if}\ \underset{v_2}{x}\ \textbf{then}\ \underset{v_3}{x}\ +\ \underset{v_4}{x}\ \textbf{else}\ 2) * \underset{v_5}{x}$$

$\mathcal{U}_{\mathrm{AE}}$ determines that the x in the condition is of type *Bool*, and that the type of the x's in the expression $x + x$ both have type *Int*. The conditional expression obtains three assumption sets from its subexpressions: $\{(x : \mathit{Bool})\}$ from the condition, $\{(x : \mathit{Int})\}$ from the then branch, and an empty assumption set from the else branch. The inconsistency is detected when it unifies these assumption sets, because given that x is monomorphic, it cannot be of type *Bool* and of type *Int*.

Using the type rules of Fig. 3, the rule (ABS) generates $\{v_1 \equiv v_2, v_1 \equiv v_3, v_1 \equiv v_4, v_1 \equiv v_5\}$ for equating the uses of x. Note that there is no ordering strategy in our system that can detect at the conditional that the type of x in the condition is not consistent with the types of the x's in the then branch. Either the inconsistency will be found at the binding site for x (when spreading is not used), or it will be found at one of the uses of x (when spreading is used).

The above behavior can be simulated by making a straightforward extension to our framework. We change the operator $+\!\!+$ for combining assumption environments: instead of concatenating assumption sets, it returns a new set of assumptions together with a constraint set. If $(x : v_1) \in \mathcal{A}_1$ and $(x : v_2) \in \mathcal{A}_2$, then we replace these assumptions by $(x : \beta)$, where β is a fresh type variable. Furthermore, we generate the additional constraints $v_1 \equiv \beta$ and $v_2 \equiv \beta$. If we now apply the tree walk *bottomUp*, then the behavior of our algorithm mimics that of $\mathcal{U}_{\mathrm{AE}}$. The same result can be obtained by generalizing the notion of spreading constraints (without the need to change the process of collecting constraints).

7 CONCLUSION

In this paper we have shown how a special constraint ordering phase in the type inferencing process can be formulated, and how it can be used to specify the various orders in which the constraints may be solved. Many of the existing algorithms for type inferencing can be emulated by choosing the appropriate ordering. In practice this gives many benefits in terms of having alternative configurations for the type inference process, either as alternatives, or to compare them. It also paves the way for global heuristics which combine and compete to determine the most likely sources of an inconsistency, because the type rules that use the combinators immediately rule out invalid solving orders.

We have shown how constraint trees can be built, and that they can be converted into lists of constraints by choosing an appropriate tree walk. A library

that implements our ideas has been used in developing the Helium compiler for Haskell, showing that it scales up well. Due to the gained flexibility, the programmer can experiment with various ordering strategies, to see which fits his way of programming. Extending a compiler based on our work to use various strategies in parallel and thus offers multiple "views" on the same type error is straightforward.

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] GHC Team. *The Glasgow Haskell Compiler*. http://www.haskell.org/ghc.

[3] J. Hage and B. Heeren. Heuristics for type error discovery and recovery (revised). Technical Report UU-CS-2006-007, Institute of Information and Computing Science, Universiteit Utrecht, Netherlands, January 2006. Technical Report.

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

[5] B. Heeren and J. Hage. Parametric type inferencing for Helium. Technical Report UU-CS-2002-035, Institute of Information and Computing Science, University Utrecht, Netherlands, August 2002. Technical Report.

[6] 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.

[7] M.P. Jones et al. *The Hugs 98 system*. OGI and Yale, http://www.haskell.org/hugs.

[8] 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.

[9] O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, March 2000.

[10] S. Peyton Jones, editor. *Haskell 98 Language and Libraries: The Revised Report*. Cambridge University Press, 2003.

[11] B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[12] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, pages 389 – 489. MIT Press, 2005.

[13] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.

[14] Peter J. Stuckey, Martin Sulzmann, and Jeremy 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.

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