Ordering type constraints: a structured approach

Jurriaan Hage

Bastiaan Heeren

institute of information and computing sciences, utrecht university

technical report UU-CS-2005-016

www.cs.uu.nl

Abstract. The separation between collecting, ordering, and solving constraints results in a flexible framework with fine-tuned control over the type inference process. We offer an abstraction which helps us deal with the process of the reordering of constraints in a generic way. As a result, our work generalizes various well-known algorithms such as $\mathcal{M}$, $\mathcal{W}$ and $\mathcal{G}$ and also various real-life implementations. This opens the way for comparing the behaviors of these algorithms to each other and to other methods of type inferencing, such as those that consider more than one constraint at the time. The framework has been implemented, and used to build a real-life compiler called Helium. This compiler allows the programmer to choose between various strategies for solving his constraints, and in this way can control what is reported to him in the case of a type error.

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

1 Introduction

Many algorithms exists in the literature and in real-life compilers that are based on the Hindley-Milner type system. All these algorithms rely on the unification of types. Nevertheless, the unifications they perform occur in different orders, with the result that inconsistencies, if they exist, are detected at a different location. It is however the location that strongly determines the error message that is reported to the programmer.

Consider for instance the following expression $\lambda f \rightarrow (f\ id, f\ True)$. Even for this simple expression, different algorithms put the 'blame' on different expressions. For instance the folklore algorithm $\mathcal{M}$ [7] 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 $\mathcal{U}_{\mathrm{AE}}$ stops a little earlier, i.e., when it has considered the tuple.

In the literature, many type systems have been formulated as a constraint based analysis. 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, the order in which constraints are solved is left unspecified. The order in which the constraints are solved is essential if constraints are considered sequentially, and we are interested in when we discover that the constraint set is inconsistent.

The importance of order has led us to include an intermediate phase in the constraint based analysis, in which we choose the order of the constraints. In the context of type inferencing, this choice is nothing else than choosing among various type inference algorithms, including those mentioned above.

In our approach we take the abstract syntax tree as a starting point, where each constraint is associated with a node in the tree. This makes sense, because type inference algorithms are usually syntax directed. To achieve this, the constraint generation phase constructs a constraint tree (which follows the shape of the abstract syntax tree), which is then converted to a list of constraints in the ordering phase. In fact, many existing type inference algorithms can be emulated by choosing the appropriate ordering. The relative ease with which we can capture so many existing algorithms is a further justification of this work.

We have developed a framework for constraint based type inferencing. Our abstraction for ordering constraints has led to a generic type inferencer, one in which the order is a parameter of the system. This means that we can exhibit behaviors similar to the mentioned inference algorithm, and compare them. Our framework already considers ways of solving constraints and as such our work here also forms a basis for comparing such a global constraint solver with the sequential solvers we consider in this paper.

We have implemented and tested our work. In fact, the Helium compiler [4] which has been developed especially for providing good feedback, is built on top of our implementation. Since Helium implements almost full Haskell 98 this shows that our approach scales well. It goes without saying that our methods are not limited to Haskell, and our library can just as easily be used for building compilers for other languages, such as ML. These may then have the same amount of flexibility, when it comes to type inferencing, that Helium has. In fact, we expect our ideas to translate just as easily to other forms of validating analyses in which control over the analysis process can used to yield more appropriate messages.

In Section 2 we describe the general characteristics of the framework. Although it is generic in the kinds of constraints it uses, we give the constraints we need for specifying the Hindley-Milner type system in Section 4. The major part of this paper is devoted to a discussion of the operators we use to specify orderings on constraints in type rules (Section 5). We illustrate the operators by giving our type rules for the lambda calculus with polymorphic let in Section 6, and show how our algorithm generalizes many of the existing implementations and algorithms for the Hindley-Milner type system in Section 7.

2 The basic framework

This section presents the philosophy behind and the general characteristics of the Top framework for constraint-based type inference.

A constraint-based analysis can be divided into constraint generation and constraint resolution. Such an approach has a number of advantages: a framework can have several constraint solvers available, and we can choose freely which solver to use. Because of the clear separation between constraint collecting and constraint solving, it becomes much easier to reuse constraint solvers. Collecting constraints must be programmed for each source language, but solving the constraints is implemented once and for all (in practice it turns out that most languages use a very limited set of kinds of constraints, such as equality constraints and subsumption constraints, so reuse is high).

The order in which the constraints are considered is of great importance in error messaging , especially if we consider solvers which handle constraints in a strictly sequential fashion. (A different, more sophisticated but also more complicated approach considers more than one constraint at the time, e.g., using type graphs [2].) Although the order of solving should not influence the outcome for a consistent constraint set, it strongly influences which constraint is blamed in case the set of constraints is inconsistent. Many algorithms (implicitly) have a fixed way of ordering the constraints, which shows up in a bias apparent in the reported error messages. As a way of improving this, we distinguish a intermediate phase.

In our framework, we incorporate this new phase in the following way. Instead of collecting a constraint set, we construct a tree decorated with constraints. Typically, the shape of this constraint tree follows the shape of the abstract syntax tree on which we perform the analysis. In the second phase, we order the constraints by rearranging this tree, and then *choosing* a flattening of this tree, which gives us a list of constraints. After the constraints have been ordered, we solve the constraints one at the time (similar to most existing type inference algorithms).

The Top framework is both customizable and extensible, which are two essential properties to encourage reusability. We shall see examples of this later.

The framework enjoys the following characteristics.

1. *Extend the solution space.* We can add new (stateful) information that is available while we are solving the constraints.
2. *Abstract constraint information.* Each constraint carries information, but we make no assumption about the content of this information. Every user can decide which information is relevant to keep track of. Constraint information is also used to create error reports; thus, the formatting of the messages can be customized as well. Although the content is unspecified, there is some interaction between the constraint information and the process of constraint solving.
3. *New types of constraints can be added.* The framework can be extended to include new sorts of constraints. To solve a new type of constraint, we may have to extend the solution space (see item *1*).
4. *Multiple constraint solvers.* Our framework can support several constraint solvers, and new alternatives to solve the constraints can be included. In this paper we focus on the greedy solver, because we are interested in making comparisons to existing algorithms such as $\mathcal{M}$, which all behave greedily.
5. *Suitable for other analyses.* A part of the framework is not specific for type inference, and this part can be reused for other analyses that are formulated as a constraint problem.

3 Preliminaries

Consider an expression language which has variables, function application, lambda abstractions and a non-recursive let.

Expression:

$e ::= x$	*(identifier)*
$\mid\ e_1\ e_2$	*(application)*
$\mid\ \lambda x \to e$	*(abstraction)*
$\mid\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2$	*(local definition)*

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. These layers predict at which points we can expect a universal quantifier or a type scheme variable, and thus type our type language.

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

The function $\mathit{ftv}(\sigma)$ returns the free type variable of its argument, and is defined as expected: 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 additionally abbreviate $a_1 \ldots a_n$ by a vector of type variables $\overline{a}$. Here we insist that all a_i are different. We use β (and variants) to refer to a fresh type variable, of which we assume to have an unlimited supply. Type variables are usually $v_0, v_1, \ldots$.

A substitution S is a mapping from type variables to types. All our substitutions are idempotent and we use id to denote the empty substitution. Application of a substitution S to an expression e is simply denoted $S(e)$. 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 we may use vector notation and abbreviate this further 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:

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

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 $\mathit{domain}(S) \subseteq \overline{a}$.

4 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 (for the latter there may be many different ways of which we choose one). The semantics tells us whether a solution meets the requirements, while the 'how' tells us how to construct a solution that fulfills these requirementsWe show that our solver is sound with respect to the semantics.

In our framework, each constraint carries 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. We make no assumption about its content. This is a valuable abstraction within our framework: we can choose for ourselves which information we want to keep, and how the error messages will be presented. In the presentation, we omit the constraint information carried by a constraint whenever this is appropriate.

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

Basic constraints:

$c ::= (\tau_1 \equiv \tau_2)$	*(equality constraint)*
$\mid\ \sigma_v := \textsc{Gen}(\mathcal{M}, \tau)$	*(generalization)*
$\mid\ \tau \preceq \rho$	*((infix) instantiation)*

With a generalization constraint we can generalize a type with respect to a set of monomorphic type variables, and assign the resulting type scheme to a type scheme variable. Instantiation constraints $\tau \preceq \rho$ express that a type should be an instance of a type scheme, or the type scheme belonging to a type scheme variable.

The generalization and instance constraints are used to handle the polymorphic language constructs. We use special type scheme variables to function as place-holders for unknown type schemes. The reason is of course that we are only generating constraints at this point and we do not want to solve them yet. We shall see shortly that our method does induce a certain bias in the sense that if we generalize a type (belonging to a certain identifier), then we have to be sure that the type has been fully computed.

Both $\preceq$ and equality constraints are lifted to work on lists of pairs, where each pair consists of an identifier and a type. For instance,

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

Our solution space for solving constraints consists of a pair of mappings (S, Σ). Here S is a substitution on type variables, and Σ a substitution on type scheme variables. We proceed to define semantics for these constraints.

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

Note that we could have avoided the equality constraints, expressing these as instantiation constraints. The reason is that $\tau_1 \equiv \tau_2$ is equivalent to $\tau_1 \preceq \tau_2$ if τ_2 is a monomorphic type (which we know it is, because of the restrictions on $\equiv$).

Before we continue with how we may control the order in which constraints are solved, we explain how each of the constraints can be solved, as a rewriting 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.

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

where *mgu* is the usual definition of the most general unifier of types [9]. 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. The side condition for the instantiation constraint is rather obscure: we insist in the pattern match 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. The other condition is somewhat more complicated. 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 in a position in which it is considered to be polymorphic. The definition of activeness is straightforward, with only the case of the generalization constraint needing special attention:

$$\begin{array}{ll}
\mathit{actives}(\mathcal{C}) & = \{\mathit{active}(c) \mid c \in \mathcal{C}\}, \text{ where} \\
\mathit{active}(\tau_1 \equiv \tau_2) & = \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\tau_2) \\
\mathit{active}(\sigma_v := \textsc{Gen}(\mathcal{M}, \tau)) & = \mathit{ftv}(\mathcal{M}) \cap \mathit{ftv}(\tau) \\
\mathit{active}(\tau \preceq \rho) & = \mathit{ftv}(\tau) \cup \mathit{ftv}(\rho)
\end{array}$$

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 the semantics can be phrased 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 of [5]. Note that the implicit instance constraints in that proof can easily be mapped to a pair of generalization and instance constraints. □

5 Constraint ordering

If we solve one constraint at the time, then the order in which type constraints are solved determines at which point in the process of constraint solving we detect an inconsistency, since this determines in which order types are unified. Instead of limiting ourselves to one order in which the constraints can be solved, we consider a family of constraint orderings from which a user can select one. As we show in Section 7, we generalize various type inference algorithms.

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. Constraints are attached to the node which generates them, although we do have some flexibility here. Later we also introduce *spreading* which may move constraints down into the constraint tree. 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.

Constraint tree:

$$
\begin{array}{lr}
\mathcal{T}_{\mathcal{C}} ::= \{\!|\, \mathcal{T}_{\mathcal{C}1}, \ldots, \mathcal{T}_{\mathcal{C}n} \,|\!\} & \textit{(node)} \\
\quad \mid\ c \rhd \mathcal{T}_{\mathcal{C}} & \textit{(attach constraint)} \\
\quad \mid\ c \lhd \mathcal{T}_{\mathcal{C}} & \textit{(attach constraint from parent)} \\
\quad \mid\ \mathcal{T}_{\mathcal{C}1} \ll \mathcal{T}_{\mathcal{C}2} & \textit{(strict node)}
\end{array}
$$

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. The second and third alternatives add a single constraint to a tree. The difference between the two lies in where the constraint was created: in the case of $c \rhd \mathcal{T}_{\mathcal{C}}$, the constraint c was generated by the root of $\mathcal{T}_{\mathcal{C}}$. The case of $c \lhd \mathcal{T}_{\mathcal{C}}$ is conceptually harder to comprehend: in this case $\mathcal{T}_{\mathcal{C}}$ is a subtree of the node that generates c, but relates specifically to that subtree. The last case ($\mathcal{T}_{\mathcal{C}1} \ll \mathcal{T}_{\mathcal{C}2}$) combines two trees in a strict way: all the constraints in $\mathcal{T}_{\mathcal{C}1}$ should be considered before the constraints in $\mathcal{T}_{\mathcal{C}2}$. In the following example, we illustrate the two alternatives for attaching a constraint to a constraint tree.

Example 0.1. Consider the constraint tree for a conditional expression, say **if** e_1 **then** e_2 **else** e_3. Suppose we have constraint tree $\mathcal{T}_{\mathcal{C}i}$ for $e_i : \tau_i$ $(i = 1, 2, 3)$. We introduce the fresh type variable β to represent the type of the branches of the conditional, and generate three type constraints.

$$c_1 = (\tau_1 \equiv \mathit{Bool}) \qquad c_2 = (\tau_2 \equiv \beta) \qquad c_3 = (\tau_3 \equiv \beta)$$

Attaching all three to the if-then-else node results in the following constraint tree, $c_1 \rhd c_2 \rhd c_3 \rhd \{\!|\, \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2}, \mathcal{T}_{\mathcal{C}3} \,|\!\}$, displayed in Figure 1(a).

On the other hand, each of the three type constraints relates to one subexpression in particular. Attaching the constraints to their respective constraint tree, gives $\{\!|\, c_1 \lhd \mathcal{T}_{\mathcal{C}1}, c_2 \lhd \mathcal{T}_{\mathcal{C}2}, c_3 \lhd \mathcal{T}_{\mathcal{C}3} \,|\!\}$ displayed in Figure 1(b) (the constraints are written with an upward arrow to indicate that the constraint was created by the parent.)

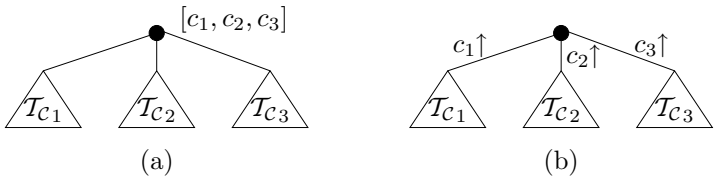


Fig. 1. Example constraint trees

Moreover, we will never generate constraint trees of the following form (where c corresponds to the current constraint tree node, and not by some node inside $\mathcal{T}_{\mathcal{C}1}$ or $\mathcal{T}_{\mathcal{C}2}$).

$$c \lhd \{\!|\, \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \,|\!\} \qquad\qquad \{\!|\, c \rhd \mathcal{T}_{\mathcal{C}1}, \mathcal{T}_{\mathcal{C}2} \,|\!\}$$

In both of these cases, the constraint c no longer belongs to the node at which it was created.

Before we continue, we introduce some abbreviations. In order of appearance, we have the empty constraint tree, the tree consisting of a single constraint set, and attaching a list of constraints (from the parent) to a tree.

$$\begin{array}{lll} \bullet & =_{def} & \{\!|\, |\!\} \\ \mathcal{C}^{\bullet} & =_{def} & \mathcal{C} \trianglerighteq \bullet \\ [c_1, \ldots, c_n] \trianglerighteq \mathcal{T}_{\mathcal{C}} & =_{def} & c_1 \rhd \ldots \rhd c_n \rhd \mathcal{T}_{\mathcal{C}} \\ [c_1, \ldots, c_n] \trianglelefteq \mathcal{T}_{\mathcal{C}} & =_{def} & c_1 \lhd \ldots \lhd c_n \lhd \mathcal{T}_{\mathcal{C}} \end{array}$$

In the remaining part of this section, we discuss various alternatives to flatten a constraint tree, which results in an ordered list of constraints. Furthermore, we present spreading which transforms a constraint tree.

5.1 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.

$$\textbf{newtype}\ \mathit{TreeWalk} = \mathcal{T}_{\mathcal{W}}\ (\forall a.[a] \to [([a],[a])] \to [a])$$

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 one 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 constraint 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. Recall that it is the *TreeWalk* that determines what happens locally in each node, and it is *flatten* which traverses the tree.

We define two mutually recursive helper-functions for *flatten*: *flattenTop* and *flattenRec*. In the definition of *flattenRec*, we maintain a list of constraints that are attached to the constraint tree ($c \rhd \mathcal{T}_{\mathcal{C}}$): this list, called *down*, is passed to all recursive calls. Furthermore, we use the list *up*

to collect the constraints that are attached to the tree by their parent node ($c \lhd \mathcal{T}_{\mathcal{C}}$). This list is computed in a bottom-up fashion.

```
flatten :: TreeWalk → ConstraintTree → [Constraint]
flatten (T_W f) = flattenTop
  where
    flattenTop :: ConstraintTree → [Constraint]
    flattenTop tree =
      let pair = flattenRec [] tree
      in f [] [pair]

    flattenRec :: [Constraint] → ConstraintTree
                  → ([Constraint], [Constraint])
    flattenRec down tree =
      case tree of
        ⦃ t_1, ..., t_n ⦄ → let pairs = map (flattenRec []) [t_1, ..., t_n]
                            in (f down pairs, [])
        c ▷ t             → flattenRec (down ++ [c]) t
        c ◁ t             → let (cset, up) = flattenRec down t
                            in (cset, up ++ [c])
        t_1 ≪ t_2         → let cs_1 = flattenTop t_1
                                cs_2 = flattenTop t_2
                            in (f down [(cs_1 ++ cs_2, [])], [])
```

Observe that for the case of a node, we let the tree walk decide how the constraint lists at that point are to be combined – i.e., the list of downward constraints, and for each subtree in $⦃ t_1, \ldots, t_n ⦄$, its flattened constraint set and the upward constraints. If the constraints of two constraint trees should be ordered in a strict way (the case $t_1 \ll t_2$), then we flatten the two constraint trees, which gives us the constraint sets cs_1 and cs_2. These sets are combined in a fixed way, regardless of the tree walk, namely $cs_1 +\!\!+ cs_2$. However, we let the tree walk decide how this combined list and the downward constraints are ordered.

The first *TreeWalk* we define is truly bottom-up.

```
bottomUp = T_W (λdown list → f (unzip list) ++ down)
  where f (csets, ups) = concat csets ++ concat ups
```

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 0.2. Assume that $\mathcal{T}_{\mathcal{C}} = down \trianglerighteq ⦃ up_1 \trianglelefteq \mathcal{C}_1^{\bullet}, \ldots, up_n \trianglelefteq \mathcal{C}_n^{\bullet} ⦄$. Flattening this constraint tree with the bottom-up tree walk gives us

$$flatten\ bottomUp\ \mathcal{T}_{\mathcal{C}} \quad = \quad \mathcal{C}_1 +\!\!+ \ldots +\!\!+ \mathcal{C}_n +\!\!+ up_1 +\!\!+ \ldots +\!\!+ up_n +\!\!+ down$$

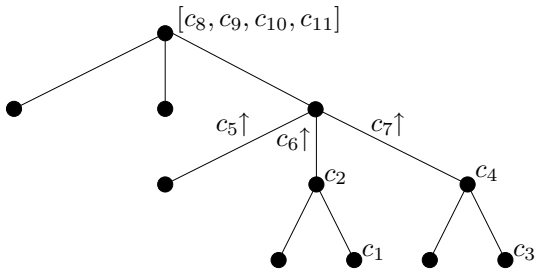
Similarly we can define the dual *TreeWalk*, which is a top-down approach.

```
topDown = T_W (λdown list → down ++ f (unzip list))
  where f (csets, ups) = concat ups ++ concat csets
```

Example 0.2 (continued). If we use this tree walk to flatten $\mathcal{T}_{\mathcal{C}}$, then we obtain

$$flatten\ topDown\ \mathcal{T}_{\mathcal{C}} \quad = \quad down +\!\!+ up_1 +\!\!+ \ldots +\!\!+ up_n +\!\!+ \mathcal{C}_1 +\!\!+ \ldots +\!\!+ \mathcal{C}_n$$

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



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

Fig. 2. A constraint tree

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.

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

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, results in the following behavior:

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

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.

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

We conclude our discussion on flattening a constraint tree with an example, which illustrates the impact the order of constraints has on which constraint is reported. We use the type rules from Figure 5 to generate the constraints.

Example 0.4. Let us consider the following ill-typed expression. 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.

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

The constraint tree $\mathcal{T}_{\mathcal{C}}$ constructed for this expression is shown in Figure 2. 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 marked constraints in the list as incorrect. We consider three flattening strategies. The underlined constraints are the locations where the inconsistency is detected.

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

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

The definition of *flatten* can easily be generalized to treat different language constructs differently, as follows:

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

This extension enables us to model inference processes such as the one of Hugs which infers tuples from right-to-left, while most other constructs are inferred from left-to-right. It also allows us to emulate all instances of $\mathcal{G}$, 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 treewalk that visits the subtree with the most type constraints first, or an ordering which is specialized for the constraints of a particular set of expressions, like "all applications of the function *map* to two arguments".

5.2 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, instead of changing constraint generation itself. More specifically, by spreading constraints we can emulate algorithms that use a top-down type environment, while using a bottom-up assumption set ourselves to collect the constraints.

The grammar for constraint trees is extended with three cases.

Constraint tree:

$\mathcal{T}_{\mathcal{C}} ::= (\ldots)$	*(alternatives on page 7)*
$\mid\ (\ell, c) \bowtie \mathcal{T}_{\mathcal{C}}$	*(spread constraint)*
$\mid\ (\ell, c) \ll\!\!\circ\ \mathcal{T}_{\mathcal{C}}$	*(spread constraint strict)*
$\mid\ \ell^{\circ}$	*(receiver)*

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 $\bowtie$ (and $\ll\!\!\circ$). Hence spreading can only occur 'downwards'.

The function *spread* is responsible for passing down constraints deeper into the tree, until they (hopefully) end up at their destination label. For reasons of brevity we only give a type signature. We pass a list of labeled type constraints that are spread as an inherited attribute.

```
spread :: ConstraintTree → ConstraintTree
spread = spreadRec [ ]
  where
    spreadRec :: [(Label, Constraint)] → ConstraintTree → ConstraintTree
    spreadRec list tree =
      case tree of
        ⦃ t1, ..., tn ⦄ → ⦃ map (spreadRec list) [t1, ..., tn] ⦄
        c ▷ t           → c ▷ spreadRec list t
        c ◁ t           → c ◁ spreadRec list t
        t1 ≪ t2         → spreadRec list t1 ≪ spreadRec list t2
        (ℓ, c) ⋈ t      → spreadRec ((ℓ, c) : list) t
        (ℓ, c) ≪∘ t     → spreadRec ((ℓ, c) : list) t
```

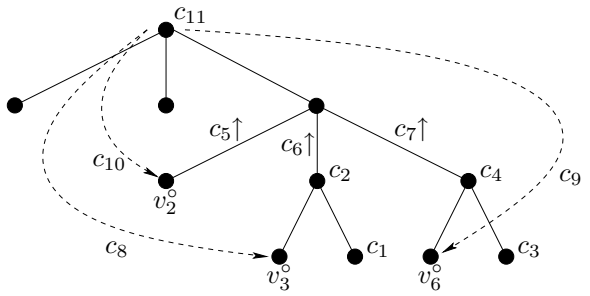


Fig. 3. A constraint tree with type constraints that have been spread

$$\ell^\circ \qquad \rightarrow [\, c \mid (\ell', c) \leftarrow \mathit{list}, \ell == \ell' \,]^\bullet$$

The type rules specify whether a certain constraint may be spread or not. To actually perform spreading is a choice that is made afterwards. This implies that we have to specify how *flatten* handles both $\rhd\!\!\circ$ and $\lll\!\!\circ$. Only the flatten function actually distinguishes between the non-strict $\rhd\!\!\circ$ and the strict version $\lll\!\!\circ$, essentially by forgetting the $\circ$. The first attaches the constraint to the tree, thus obeying the chosen flattening strategy, whereas the second demands that the constraint is considered before any of the constraints in the tree.

$$\begin{array}{l}
\mathit{flattenRec}\ \mathit{down}\ \mathit{tree} = \\
\quad \textbf{case}\ \mathit{tree}\ \textbf{of} \\
\qquad \begin{array}{ll}
\ldots & \rightarrow \ldots \\
(\ell, c) \rhd\!\!\circ\ t & \rightarrow \mathit{flattenRec}\ \mathit{down}\ (c \rhd t) \\
(\ell, c) \lll\!\!\circ\ t & \rightarrow \mathit{flattenRec}\ \mathit{down}\ ([\, c \,]^\bullet \lll t) \\
\ell^\circ & \rightarrow \mathit{flattenRec}\ \mathit{down}\ \bullet
\end{array}
\end{array}$$

Spreading comes with a warning: improper use may lead to disappearance or duplication of type constraints. For instance, we expect for every constraint that is spread to have exactly one receiver in its scope. The definition of *spread* can be extended straightforwardly to test this property for a given constraint tree.

Example 0.4 (continued). We spread the type constraints introduced for the monomorphic pattern variables f and b to their use sites in Example 0.4. Hence, the constraints c_8, c_9, and c_{10} are moved to a different location in the constraint tree. At the three nodes of the variables (two for f, one for b), we put a receiver. 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 3.

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

The *bottomUp* treewalk 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 process of Hugs and GHC. 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 treewalk. The behavior of GHC can be mimicked by an inorder top-down treewalk.

5.3 Phasing constraint trees

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 directly to implement parts of the specialized type rules [3], but also help take advantage of type signatures, by assigning a lower phase number to constraints which originate from a type signature. This amounts to 'pushing down' constraints in a the definition.

When we push down monomorphic types, then we can handle this using the constraints defined earlier: if a function f has an explicit type $Int \to Int$, then the constraint that its first argument has type Int is an equality constraint that can be solved before considering the function definition. This is in fact what the GHC compiler does. Pushing down polymorphic types is a little bit more complicated, but can be handled by skolemization constraints, which are rather similar to the instantiation constraints discussed in this paper. 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.

The idea of phasing is very simple: we assign phase numbers to parts of the constraint tree. Constraints with a low phase number should be considered before constraints with a high phase number, although we respect the restrictions on the constraint order imposed by strict nodes.

We extend our definition of constraint trees with an extra case to assign a phase number to a constraint tree.

Constraint tree:

$\mathcal{T}_{\mathcal{C}} ::= (\ldots)$ *(alternatives on page 11)*

$\quad \mid$ *Phase* i $\mathcal{T}_{\mathcal{C}}$ *(phasing)*

In the new case, i is the phase number. We assume 5 to be the default phase number. The definitions for flattening and spreading are extended to handle phased constraint trees, simply by ignoring the phase number when encountered.

```
flattenRec down tree =
  case tree of
    ...              -> ...
    Phase i t -> flattenRec down t

spreadRec list tree =
  case tree of
    ...              -> ...
    Phase i t -> Phase i (spreadRec list t)
```

We now define a function *phase*, which transforms a tree with phase numbers into a tree where the phases are encoded by strict nodes. To perform this transformation, we use *phase maps*. A phase map is a list of pairs of a phase number and a constraint tree.

```
newtype PhaseMap = PM [(Int, ConstraintTree)]
```

We make sure that the phase numbers in a phase map are always strictly increasing. First, we define a number of helper-functions to construct, combine, and use phase maps.

```
pmEmpty :: PhaseMap
pmEmpty = PM []

pmSingleton :: Int -> ConstraintTree -> PhaseMap
pmSingleton i tree = PM [(i, tree)]
```

Our next function creates a constraint tree from a phase map by ordering the constraint trees of the phases in a strict way.

```
pmToTree :: PhaseMap -> ConstraintTree
```

$$pmToTree\ (PM\ xs) = foldr\ (\lambda(\_, t_1)\ t_2 \to t_1 \ll t_2)\ \bullet\ xs$$

The next functions combine phase maps.

```
pmPlus :: PhaseMap → PhaseMap → PhaseMap
pmPlus (PM xs) (PM ys) = PM (f xs ys)
  where
    f [] ys = ys
    f xs [] = xs
    f xs@((i, tx) : restx) ys@((j, ty) : resty)
      | i == j = (i, {| tx, ty |}) : f restx resty
      | i < j = (i, tx) : f restx ys
      | i > j = (j, ty) : f xs resty

pmConcat :: [PhaseMap] → PhaseMap
pmConcat = foldr pmPlus pmEmpty

pmAddDefault :: ConstraintTree → PhaseMap → PhaseMap
pmAddDefault tree = pmPlus (pmSingleton 5 tree)
```

With these helper-functions, we define how to phase a constraint tree. The local function *phaseRec* returns a constraint tree and a phase map. The tree that is returned by this function is the current constraint tree, for which we do not have a phase number.

```
phase :: ConstraintTree → ConstraintTree
phase = phaseTop
  where
    phaseTop :: ConstraintTree → ConstraintTree
    phaseTop tree =
      let (t, pm) = phaseRec tree
      in pmToTree (pmAddDefault t pm)

    phaseRec :: ConstraintTree → (ConstraintTree, PhaseMap)
    phaseRec tree =
      case tree of
        {| t_1, ..., t_n |} → let pairs = map phaseRec [t_1, ..., t_n]
                                  ([t'_1, ..., t'_n], pms) = unzip pairs
                              in ({| t'_1, ..., t'_n |}, pmConcat pms)
        c ▷ t               → let (t', pm) = phaseRec t
                              in (c ▷ t', pm)
        c ◁ t               → let (t', pm) = phaseRec t
                              in (c ◁ t', pm)
        t_1 ≪ t_2           → (phaseTop t_1 ≪ phaseTop t_2, pmEmpty)
        (ℓ, c) ⋈▷ t         → let (t', pm) = phaseRec t
                              in ((ℓ, c) ⋈▷ t', pm)
        (ℓ, c) ≪◦ t         → ((ℓ, c) ≪◦ phaseTop t, pmEmpty)
        ℓ°                  → (ℓ°, pmEmpty)
        Phase i t           → let (t', pm) = phaseRec t
                              in (•, pmPlus (pmSingleton i t') pm)
```

Most definitions are relatively straightforward: we discuss those that are not. First, we explain the case for *Phase i ctree*. From the recursive call, we get the constraint tree t' which is assigned phase number i. This tree is added with the appropriate phase number to the phase map. We return the empty constraint tree as the first component of the pair since there are no more constraints for which we do not know their corresponding phase. Secondly, the two cases that impose a strict ordering on the constraints ($t_1 \ll t_2$ and $(\ell, c) \ll\!\circ\ t$) should be handled with care. In both cases,

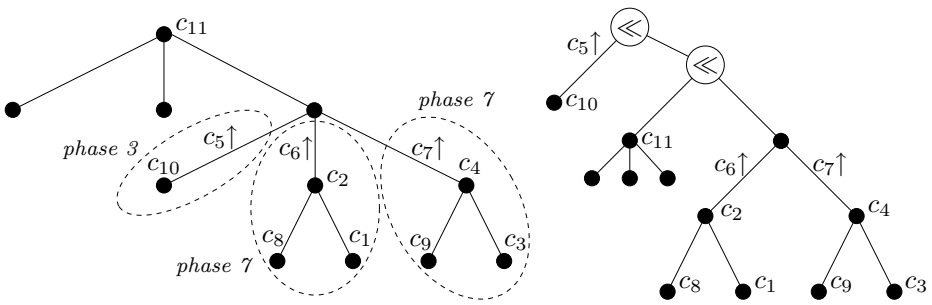


Fig. 4. A constraint tree before and after phasing

we use the function *phaseTop*, which uses *pmToTree* for converting the phase map to a constraint tree, and we return an empty phase map.

Example 0.4 (continued). Consider once more the expression

$$\lambda f\ b \to \textbf{if}\ b\ \textbf{then}\ f\ 1\ \textbf{else}\ f\ \mathit{True},$$

and its constraint tree after spreading type constraints. This tree is shown in Figure 4 on the left. Suppose that we want to treat the subexpressions of conditionals in a special way. For example, we consider the constraints of the condition (including the constraint that this expression should have type *Bool*) before all the other type constraints, so we assign phase 3 to this part of the constraint tree. In a similar way, we postpone the constraints for the two branches, and use phase number 7 for these parts. The remaining type constraints are assigned to the default phase (which is 5). The right part of Figure 4 shows the constraint tree after phasing. The two strict nodes combine the three constraint trees of phase 3, 5, and 7 (from left to right). Note that a number of empty constraint trees have been omitted to simplify the presentation of the tree.

In conclusion: building a constraint tree for an expression that follows the shape of the abstract syntax tree provides the flexibility to come up with different orderings of the type constraints. Several tree walks have been proposed, including a bottom-up and a top-down approach. Furthermore, we have presented two techniques to rearrange the constraint tree, namely the spreading of type constraints, and assigning phase numbers to constraint trees.

6 The type system

The type rules for our language can be found in Figure 5, specifying constraint trees using the operators just defined. The type rules are formulated in terms of judgements of the form $\mathcal{M}, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash e : \tau$. 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 $\mathcal{T}_\mathcal{C}$ are satisfied, and if $\mathcal{A}$ enumerates all the types that have been assigned to the identifiers that are free in e". The set of monomorphic types ($\mathcal{M}$) is provided by the context: it is passed top-down. This is only to simplify constraint collection: alternatively, we could change the monomorphic sets of the implicit instance constraints when they are collected upwards. The assumption set ($\mathcal{A}$) contains an assumption $(x : \beta)$, for each *occurrence* of an unbound identifier (here β is a unique 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 from $\mathcal{A}$ of all assumptions for x.

The reason for using $\ll_\circ$ for the instantiation constraints in the let while the lambda uses $\rhd\!\!\lhd$ is that we cannot allow that a treewalk decides to instantiate polymorphic variables in a let-body after inferring the let-body.

It is important to realize that a constraint such as $\tau_1 \equiv \tau_2 \to \beta$ is not about the eventual types of e_1 and e_2, but a relation between the fresh type variables assigned to them, and the

$\boxed{\mathcal{M}, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash e : \tau}$ *Expression*

$$\frac{\mathcal{T}_\mathcal{C} = [\beta \equiv \mathit{Int}]^\bullet}{\mathcal{M}, \emptyset, \mathcal{T}_\mathcal{C} \vdash i : \beta}\ (\textsc{Int}) \qquad \frac{\mathcal{T}_\mathcal{C} = [\beta \equiv \mathit{Bool}]^\bullet}{\mathcal{M}, \emptyset, \mathcal{T}_\mathcal{C} \vdash b : \beta}\ (\textsc{Bool})$$

$$\frac{}{\mathcal{M}, [x{:}\beta], \beta^\circ \vdash x : \beta}\ (\textsc{Var})$$

$$\frac{\begin{array}{c} c_1 = (\tau_1 \equiv \beta_1 \to \beta_2) \quad c_2 = (\beta_1 \equiv \tau_2) \quad c_3 = (\beta_2 \equiv \beta_3) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash e_1 : \tau_1 \qquad \mathcal{M}, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash e_2 : \tau_2 \end{array}}{\mathcal{M}, \mathcal{A}_1 +\!\!+ \mathcal{A}_2, c_3 \rhd \lfloor\!\lfloor\, c_1 \lhd \mathcal{T}_{\mathcal{C}1}, c_2 \lhd \mathcal{T}_{\mathcal{C}2} \,\rfloor\!\rfloor \vdash e_1\ e_2 : \beta_3}\ (\textsc{Apply})$$

$$\frac{\begin{array}{c} \mathcal{T}_\mathcal{C} = \lfloor\!\lfloor\, c_1 \lhd \mathcal{T}_{\mathcal{C}1}, c_2 \lhd \mathcal{T}_{\mathcal{C}2}, c_3 \lhd \mathcal{T}_{\mathcal{C}3} \,\rfloor\!\rfloor \\ c_1 = (\tau_1 \equiv \mathit{Bool}) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash e_1 : \tau_1 \\ \mathcal{M}, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash e_2 : \tau_2 \qquad \mathcal{M}, \mathcal{A}_3, \mathcal{T}_{\mathcal{C}3} \vdash e_3 : \tau_3 \end{array}}{\mathcal{M}, \mathcal{A}_1 +\!\!+ \mathcal{A}_2 +\!\!+ \mathcal{A}_3, \mathcal{T}_\mathcal{C} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \beta}\ (\textsc{Cond})$$

$$\frac{\begin{array}{c} \mathcal{C}_\ell = ([x{:}\beta_1] \equiv \mathcal{A}) \quad c_1 = (\beta_3 \equiv \beta_1 \to \beta_2) \quad c_2 = (\tau \equiv \beta_2) \\ \mathcal{M} +\!\!+ [\beta_1], \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash e : \tau \end{array}}{\mathcal{M}, \mathcal{A}\backslash x, c_1 \rhd \mathcal{C}_\ell \mathbin{\underline{\bowtie}} \lfloor\!\lfloor\, c_2 \lhd \mathcal{T}_\mathcal{C} \,\rfloor\!\rfloor \vdash \lambda x \to e : \beta_3}\ (\textsc{Abs})$$

$$\frac{\begin{array}{c} \mathcal{T}_\mathcal{C} = (c_2 \rhd \lfloor\!\lfloor\, \mathcal{T}_{\mathcal{C}1} \ll [c_1]^\bullet \ll (\mathcal{C}_\ell \mathbin{\underline{\ll\!\circ}} \mathcal{T}_{\mathcal{C}2}) \,\rfloor\!\rfloor) \\ c_1 = (\sigma_v := \textsc{Gen}(\mathcal{M}, \tau_1)) \quad \mathcal{C}_\ell = (\mathcal{A}_2 \preceq [x{:}\sigma_v]) \quad c_2 = (\beta \equiv \tau_2) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{T}_{\mathcal{C}1} \vdash e_1 : \tau_1 \qquad \mathcal{M}, \mathcal{A}_2, \mathcal{T}_{\mathcal{C}2} \vdash e_2 : \tau_2 \end{array}}{\mathcal{M}, \mathcal{A}_1 +\!\!+ \mathcal{A}_2\backslash x, \mathcal{T}_\mathcal{C} \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \beta}\ (\textsc{Let})$$

Fig. 5. The type rules

fresh type variable assigned to the application itself. Of course, when the constraint is solved, it may very well be that some of them have been replaced by a more complicated type, but this depends solely on which constraints have been solved up that point. Also, we have made sure each constraint enforces as little as possible. The constraints c_i $(i = 1, 2, 3)$ in the application rule could for instance have been captured by the single constraint $\tau_1 \equiv \tau_2 \to \beta_3$. This opens the way for fine-grained control over when a certain fact is checked.

In Section 4 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 can be formulated as follows.

Theorem 2. *Let $\mathcal{M}, \mathcal{A}, \mathcal{T}_\mathcal{C} \vdash e : \tau$ for a closed expression e, and let $\mathcal{C}$ be a list of constraints $\mathcal{C}$ that obeys the strict nodes in $\mathcal{T}_\mathcal{C}$. If $(\mathcal{C}, [], []) \to^* (\emptyset, S, \Sigma)$, then (S, Σ) are a solution for the Hindley-Milner type system.*

Proof. The proof is similar to that of Theorem 4.16 in [5], again with the note that implicit instance constraints can be replaced by separate generalization and instantiation constraints. A notable difference with the proof of Theorem 4.16 is that here we have to take the flattening into account. However, because of Theorem 1, we only need to point out that the use of the $\ll$ operator in the rule for the let as we have defined it, together with the semantics of *flatten* which respects

this order no matter the chosen treewalk, ensures that solving $\mathcal{C}$ in the given order does not block on a side condition for either the generalization or the instantiation constraint. □

A corollary is that if any of the algorithms for implementing the Hindley-Milner type system fails, then ours fails as well. The only difference between them is then they might fail 'at different points'. We shall show for some these algorithms that we can choose a treewalk such that they fail 'at the same time', i.e., while essentially solving the same constraint.

7 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 here the classic algorithms, $\mathcal{W}$ [1] and $\mathcal{M}$ [7], but also $\mathcal{G}$ [8] and $\mathcal{U}_{\text{AE}}$ [6].

Consider first the standard algorithm $\mathcal{W}$ ([1]).

$$\mathcal{W} :: (\mathit{TypeEnvironment}, \mathit{Expression}) \to (\mathit{Substitution}, \mathit{Type})$$

$$\begin{array}{lll}
\mathcal{W}(\Gamma, x) & = \textbf{let } \forall\overline{a}.\tau = \Gamma(x) & \\
& \quad \textbf{in } (id, [\overline{a} := \overline{\beta}]\tau) & \textit{fresh } \overline{\beta} \\
\mathcal{W}(\Gamma, \lambda x \to e) & = \textbf{let } (S, \tau) = \mathcal{W}(\Gamma\backslash x \cup \{x : \beta\}, e) & \textit{fresh } \beta \\
& \quad \textbf{in } (S, S\beta \to \tau) & \\
\mathcal{W}(\Gamma, e_1\ e_2) & = \textbf{let } (S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) & \\
& \qquad (S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2) & \\
& \qquad S_3 \quad\ = \mathit{mgu}(S_2\tau_1, \tau_2 \to \beta) & \textit{fresh } \beta \\
& \quad \textbf{in } (S_3S_2S_1, S_3\beta) &
\end{array}$$

$$\begin{array}{lll}
\mathcal{W}(\Gamma, \textbf{ let } x = e_1 \textbf{ in } e_2) & = \textbf{let } (S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) & \\
& \qquad \sigma \quad\ = \mathit{gen}(S_1\Gamma, \tau_1) & \\
& \qquad (S_2, \tau_2) = \mathcal{W}(S_1\Gamma\backslash x \cup \{x : \sigma\}, e_2) & \\
& \quad \textbf{in } (S_2S_1, \tau_2) &
\end{array}$$

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* treewalk with spreading corresponds to Algorithm $\mathcal{W}$. Note that although spreading is used, the bottomUp treewalk makes sure that the algorithm never fails at an identifier.

Similarly, we consider the folklore algorithm $\mathcal{M}$ as discussed in detail in [7].

$$\mathcal{M} :: (\mathit{TypeEnvironment}, \mathit{Expression}, \mathit{Type}) \to \mathit{Substitution}$$

$$\begin{array}{ll}
\mathcal{M}(\Gamma, x, \tau_1) & = \textbf{let } \forall\overline{a}.\tau_2 = \Gamma(x) \textbf{ in } \mathit{mgu}(\tau_1, [\overline{a} := \overline{\beta}]\tau_2) \\
\mathcal{M}(\Gamma, \lambda x \to e, \tau) & = \textbf{let } S_1 = \mathit{mgu}(\tau, \beta_1 \to \beta_2) \\
& \qquad S_2 = \mathcal{M}(S_1\Gamma\backslash x \cup \{x : S_1\beta_1\}, e, S_1\beta_2) \\
& \quad \textbf{in } S_2S_1 \\
\mathcal{M}(\Gamma, e_1\ e_2, \tau) & = \textbf{let } S_1 = \mathcal{M}(\Gamma, e_1, \beta \to \tau) \\
& \qquad S_2 = \mathcal{M}(S_1\Gamma, e_2, S_1\beta) \\
& \quad \textbf{in } S_2S_1 \\
\mathcal{M}(\Gamma, \textbf{ let } x = e_1 \textbf{ in } e_2, \tau) & = \textbf{let } S_1 = \mathcal{M}(\Gamma, e_1, \beta) \\
& \qquad \sigma \ = \mathit{gen}(S_1\Gamma, S_1\beta) \\
& \qquad S_2 = \mathcal{M}(S_1\Gamma\backslash x \cup \{x : \sigma\}, e_2, S_1\tau) \\
& \quad \textbf{in } S_2S_1
\end{array}$$

$\mathcal{M}$ is a top-down inference algorithm, which also uses a type environment to pass down information about identifiers. It is dual to $\mathcal{W}$, in the sense that unification takes place in the identifier and lambda node, and not in the application node. These properties determine that spreading in combination with the *topDown* treewalk emulates $\mathcal{M}$. This ensures that constraints generated higher in the tree dominate, which is viewed by Lee and Yi as characteristic for $\mathcal{M}$.

Algorithm $\mathcal{G}$ defined by Lee and Yi [8] is a combination of $\mathcal{M}$ and $\mathcal{W}$ and generalizes both. It actually defines a set of algorithms. The essential idea here is checks that $\mathcal{W}$ and $\mathcal{M}$ make are broken into pieces so that some of these can be checked when arriving at a certain node, some can be checked between two subtree visits, and some can be checked before going back up. This decomposition can be chosen independently for each non-terminal. (Note that the algorithm of $\mathcal{G}$ specified in [8] actually suggests to check all constraints at the end, and perform parts of this check earlier on as well.)

To see how in our system we can model the choices to be made in $\mathcal{G}$, we consider the most interesting of the cases, which is the application node $e_1 e_2$. The formulation of Yee and Li allows us to have θ_1 be equal to either a fresh type variable, a function type in which argument and result are fresh, or a function type in which the argument is fresh, and the result type is equal to ρ ((2) in Fig. 3 of [7]). Afterwards 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 we decide what to pass down: the type β which has become known from inferring the function, or a fresh type variable (4). After visiting e_2, all constraints are checked again to make sure that that which was omitted earlier on, is taken care of.

Instead of exhaustively listing all the possibilities and showing how these can be specified in our system via a certain treewalk, we illustrate by considering algorithm $\mathcal{H}$ from [7]. The corresponding treewalk 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 application rule of our type system in Figure 5.

With the realization that our algorithm can handle reversed orders, we can summarize the above as follows

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

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

Yang describes a type inference algorithm which proceeds by unifying assumption environments, based on a suggestion by Agat and Gustavsson. The main idea is to handle the two subexpressions of an application independently to remove the left-to-right bias present in most algorithms. The types of most identifiers are recorded in an assumption environment, which is constructed bottom-up (synthesized). There is also a type environment, which is used only to gain efficiency. In the same paper a second algorithm $\mathcal{IEI}$ is defined which combines $\mathcal{U}_{\mathrm{AE}}$ with $\mathcal{M}$. At first, an expression is checked with Algorithm $\mathcal{U}_{\mathrm{AE}}$. If this fails at top-level, then $\mathcal{M}$ is used to try and find a smaller expression that is in error.

Example 0.7. Consider the following erroneous expression:

$$\begin{array}{cccccccc} f = \lambda x \rightarrow (\textbf{if } x \textbf{ then } x + x \textbf{ else } 2) * x \\ \quad | \quad\quad | \quad\quad | \;\; | \quad\quad\quad | \\ \quad v_1 \quad\quad v_2 \quad\quad v_3 \;\; v_4 \quad\quad\quad v_5 \end{array}$$

Here we indicated only the type variables introduced for the identifier x. The constraints generated for x are $\{v_3 \equiv v_6, v_4 \equiv v_6\}$. Here v_6 is the fresh type variable to represent the variable x in the expression $x+x$. Only v_6 is visible to rest of the expression, so that when we come to the conditional itself, the constraints $\{v_2 \equiv v_7, v_6 \equiv v_7\}$ are generated and $(x : v_7)$ replaces the assumptions $(x : v_2)$ and $(x : v_6)$. Similarly, we obtain $\{v_5 \equiv v_8, v_7 \equiv v_8\}$ and the assumption $(x : v_8)$. Finally, the lambda node binds the variable x and the constraint $v_1 \equiv v_8$ is generated. In our original set-up, the constraints involving x would be through the type variable introduced for the definition of x: $\{v_1 \equiv v_i \mid i = 2, 3, 4, 5\}$.

The type inference process of $\mathcal{U}_{\mathrm{AE}}$ proceeds in a bottom-up fashion, as to remove and left-to-right that might exist. Indeed, if the constraints are solved in this fashion, then we find the conflicting constraints $v_2 \equiv v_7$ and $v_6 \equiv v_7\}$. This immediately points to the conditional as the source of the problem.

The above behavior can be simulated if we make the following straightforward extension: first of all, the operator $+\!\!+$ that combines assumption environments should be changed: instead of concatenating the assumption sets, it should generate a fresh assumption $(x : \beta)$ for every variable that occurs in both its arguments (say $(x : v_i)$ and $(x : v_j)$). Additionally, the constraints $v_i \equiv \beta$ and $v_j \equiv \beta$ should be added to the set of constraints. Thus it takes two assumption sets and returns the merged assumption set and a set of constraints, which should be added the constraints associated with this node. If we now apply the truly bottom-up treewalk *bottomUp*, then the behavior of our algorithm mimics that of $\mathcal{U}_{\mathrm{AE}}$.

8 Conclusion

In this paper we have shown how various existing algorithms for type inferencing can be emulated by a framework in which there exists an intermediate phase for ordering the constraints between the collecting and solving phase. In practice this gives many benefits in terms of having alternative configurations for the type inference process, and paving the way for global heuristics which combine and compete to determine the most likely sources of an inconsistency (for instance by means of type graphs and heuristics defined thereupon).

We have shown how constraint trees can be built, and that they can be converted into lists of constraints by choosing an appropriate treewalk. The library has been used in developing the Helium compiler for Haskell, showing that it scales up well. Due to the flexibility, the programmer can experiment with various treewalks, to see which fits his way of programming.

We expect the library to be useful for developing type inferencers for other languages besides Haskell as well as for other types of constraint based analysis, especially in those cases where feedback to the programmer is of importance.

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. Bastiaan Heeren and Jurriaan 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.
3. Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.
4. Bastiaan Heeren, Daan Leijen, and Arjan IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York. ACM Press.
5. Bastiaan J. Heeren. *Top Quality Type Error Messages*. PhD thesis, Utrecht University, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/TopQuality.pdf`.
6. 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.
7. 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.
8. Oukseh Lee and Kwangkeun Yi. A generalization of hybrid let-polymorphic type inference algorithms. In *Proceedings of the First Asian Workshop on Programming Languages and Systems*, pages 79–88, National university of Singapore, Singapore, December 2000.
9. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.