Strategies for solving constraints in program analysis

Jurriaan Hage

Bastiaan Heeren

Department of Information and Computing Sciences, Utrecht University

Technical Report UU-CS-2006-055

www.cs.uu.nl
ISSN: 0924-3275

Abstract

Type and effect systems are typically specified as a system of logical deduction rules, which specify what are valid types and effects in a program. Converting such a system into an algorithm is typically a lot of error-prone work, and usually results in an implementation that performs its task inflexibly, leaving no room to modify the solving strategy, without changing the algorithm.

In this paper we take a constraint-based approach to declaratively specify a type system. Our contribution is to enrich the type rules by the use of a fixed set of combinators. By choosing a semantics for these combinators, we obtain particular orderings of the constraints, which can then be fed into a constraint solver. Thus they bridge the gap between the declarative specification and a deterministic implementation of type systems. The main advantages are that the issue of the order in which constraints should be solved is factored out of the solver, making the solver simpler and more amenable to reuse; the user of the solver can flexibly choose a particular solving strategy; and it makes the implementation of type and effect systems a much easier and less error-prone task. Our combinators have been implemented into a fully functional compiler.

1. Introduction

The Hindley-Milner type system lies at the basis of many type inference algorithms and implementations that exist for polymorphic, functional languages such as Haskell and ML. These algorithms and implementations all rely on the unification of types, and each implementation typically adheres to a particular strategy (or order) in which unifications are performed.

In the literature, there is usually a distinction made between the specification of a type system (we use the name type system for short, but what we say in this paper applies to type and effect systems in general), often in the form of a collection of logical deduction rules, and an algorithm that constructs a derivation tree for these logical deduction rules that is guaranteed to satisfy these rules (typically, the algorithm even constructs the "best" possible derivation tree). A standard text on the subject [13] illustrates the distinction well, e.g., compare Table 5.2 with Table 5.8. These tables also illustrate that obtaining an algorithm from a specification is, even for this relatively simple example, a lot error-prone of work, because it is easy to forget a substitution or to compose them in the wrong order. Note also that the algorithm traverses the abstract syntax tree in a fixed manner, and there is no room to change the order in which unifications are performed.

Over the last ten years, this situation has improved quite a bit, because many researchers choose to specify the type system in terms of constraints. This effectively maps the analysis problem, performed on what might be a complex programming language with many language constructs, onto a much simpler language of constraints, cf. the book by Pierce [14] and the description of the essence of ML type inference by Pottier and Rémy [15]. Deriving an algorithm that generates these constraints is not so hard, and attention can be focused on the implementation of a solver. A pleasant side-effect is that typically only a few different types of constraints are used, and solvers are much more likely to be reused.

However, the description of the essence of ML type inference by Pottier and Rémy [15] also reveals a limitation. The constraint solver they describe in Section 10.6 is unnecessarily complicated, because it deals not only with solving (parts of) constraints, but also in which order this should happen. A further motivation of our work comes from the fact that the solving strategy is now hard-coded inside the rules, and there is no possibility to influence this order, without actually changing the solver (with the danger that other, unwanted changes are introduced).

A question that now may arise, is why the solving strategy matters at all. There are a few reasons. In many cases, it is much easier to construct an efficient solver if we know that some constraints will be fed into the solver in a specific order (an example of this will be given in this paper). This already holds for relatively simple analyses such as type inference for a polymorphic functional languages, but becomes even more important if more language features are added. For example, the rank-n extension to Haskell proposed by Peyton Jones et al. crucially depends on the order in which unifications are performed [8]. In some systems for generating type inference algorithms for high-level specifications, such as TinkerType [11] and Ruler [3], the order in which unifications are performed is either fixed, or it has to be hard-coded by means of composing the correct substitutions, similar to the construction of type inference algorithms as described in the textbook mentioned earlier [13]. We think our work can serve as a source of inspiration to increase the level of abstraction for these systems.

Furthermore, in many type inferencers, the order of performing unifications (i.e., solving constraints) crucially determines which unification shall be blamed for an inconsistency, and through it, what error message will be given. The result of using different solving strategies in that case gives different views on what might be the reason for the inconsistency. The kind of system we propose allows different solving strategies to be applied in parallel. The resulting information can be used in various ways: When different solving strategies blame the same unification, then this gives additional evidence that that unification is indeed to blame. The inherent flexibility of our approach can also be used in a compiler that can learn to apply the "best" ordering, based on a training session with a programmer. Thus our ideas open up a host of new possiblities for future implementations of program analyses, that we think ought to be investigated.

Concretely, we present a set of (high-level) combinators that can be used to specify inside the type rules (which specify which constraints should be generated), what are valid and invalid orderings for solving the constraints. We illustrate our combinators by specifying a variant of the Hindley-Milner type system, but stress that the combinators are not in any way tailored towards this application. Indeed, they are even fully independent from the chosen constraint language. We then illustrate the flexibility of our approach by showing how many of the existing implementations of the Hindley-Milner type system can be obtained by a suitable choice of semantics for the combinators.

To summarize, the main contribution of this paper is the message that it makes sense to cleanly decouple the specification (collecting constraints), the ordering of unifications (ordering constraints), and performing the unifications (solving constraints) when developing program analyses as type and effect systems. We give the beginnings of a framework that can handle this in a declarative manner, and show how such a system may be implemented and used.

The combinators have been used to implement a type system for a fully functional compiler, which shows that our approach scales well in practice. The compiler allows the programmer to choose from a fixed range of semantics for the combinators, which allows him to experiment with various well-known type inference algorithms such as $\mathcal{W}$ and $\mathcal{M}$.

This paper is closely related to previously submitted, unrefereed work [1]. The main differences lie in the focus of the paper. In the case of the earlier version, the focus was on generating alternative, "improved" error messages using our combinators. In this paper, we focus on the concept of having a separate ordering phase, and how that would help program analyses in general. The type system mainly serves as a vehicle for explaining and illustrating our combinators. Furthermore, our discussion of phasing and spreading is now more elaborate, and we have added a version of the Hindley-Milner algorithm without combinators, to be able to contrast it to the version which does use our combinators.

The paper is structured as follows. After some preliminaries to settle on notation for types and constraints, we consider a variant of the Hindley-Milner type system, which uses assumption sets and sets of constraints. In Section 4 we introduce a modified type system which uses many of our combinators, and then consider these combinators in detail. Then we show how we can emulate various well-known algorithms for type inferencing by choosing a suitable semantics for our operators as an illustration of the flexibility of our framework. In the last two sections, we discuss related work and present our conclusions.

This is a (rather strongly) revised version of an earlier technical report[5]

2. Preliminaries

The running example of this paper describes type inference for the Hindley-Milner type system, and we assume the reader has some familiarity with this type system [2]. We use a three layer type language: besides mono types (τ) we have type schemes (σ), and ρ's, which are either type schemes or type scheme variables (σ_v).

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

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 free type variables. 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 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 $[a_1 := \tau_1, \ldots, a_n := \tau_n]$ 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:

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

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 $\mathit{domain}(S) \subseteq \overline{a}$. For example, $a \rightarrow Int < \forall ab.a \rightarrow b$ by choosing $S = [b := Int]$.

Types can be related by means of constraints. The following constraints express type equivalence for monomorphic types, generalization and instantiation, respectively.

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

With a generalization constraint we can generalize a type with respect to a set of monomorphic type variables $\mathcal{M}$, and associate the resulting type scheme with a type scheme variable σ_v. 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: for each occurrence of a let-defined identifier, we must then duplicate sets of constraints, and thus much of our work. If such a set is itself inconsistent, then the inconsistency is duplicated as well. We use type scheme variables to function as placeholders for unknown type scheme because we do not want to solve constraints during the constraint generation phase. We shall see shortly that this choice has some implications for the order in which some constraints can be solved.

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

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

Our solution space for solving constraints consists of a pair of mappings (S, Σ), 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 constraint c is satisfied by the substitutions (S, Σ).

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

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.

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

where the standard algorithm *mgu* is used to find a most general unifier of two types [16] and the function $\mathit{actives}$ can be defined as follows:

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

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 solve a generalization constraint, the polymorphic type variables in that type are quantified so that their former identity is lost. Hence, these type variables should play no further role in the future, which can be checked simply by inspecting the positions in which they occur in the constraint set. This property is the one that is checked with the help of the function $\mathit{actives}$.

The final two rules specify the termination of the rewriting process. When the constraint is empty, then we quit and return the obtained substitutions, and if the constraint set is non-empty and none of the other rules can be applied, then this implies that the constraint set was inconsistent and the error solution $(\top, \top)$ is returned.

3. An example type system

Before we actually discuss our combinators in detail, we give by way of example a specification of the Hindley-Milner type system based on the collection of constraints, and assumptions for identifiers. The set of constraints generated for an expression can be used as input to the solver defined as a rewrite system in Section 2.

Type rules for the following expression language (with a non-recursive let) are presented in Figure 1.

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

These rules specify how to construct a constraint tree for a given expression, and are formulated in terms of judgements of the form $\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash e : \tau$. 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 $\mathcal{C}$ 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 keeps track of all the type variables that were introduced in a lambda binding (which in our language are monomorphic). 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. These occurrences are propagated upwards until they arrive at the corresponding binding site, where constraints on their types can be generated, and the assumptions dismissed. Ordinarily, the Hindley-Milner type system uses type environments to communicate the type of a binding to its occurrences. We have chosen to deviate from this, because it turned out to be easier to emulate the type environments in a type system based on assumption, then vice versa (see the discussion on spreading later in this paper). The operator $\cup$ is ordinary set union, 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 (similar to the unique labels that are introduced to be able to refer to analysis data computed for a specific expression [13]). For example, consider the type rule (App). Here, τ_1 is a placeholder for the type of e_1, and is used in the constraint $\tau_1 \equiv \beta_1 \rightarrow \beta_2$. Because of the invariant, we know that τ_1 is actually a type variable, and at this point we have no clue about the type it will become; this will become apparent during the solving process.

We could have replaced c_i $(i = 1, 2, 3)$ in the type rule (App) with a single constraint $\tau_1 \equiv \tau_2 \rightarrow \beta_3$. Decomposing this constraint, however, opens the way for fine-grained control over when a certain fact is checked. Something similar has been done in the conditional rule, where we have explicitly associated the constraint that the condition is of boolean type with the constraints generated for the condition. As we shall see later, there is a good reason for that.

For any given expression e we can, based on the rules of Figure 1, determine the set of constraints that need to be satisfied to ensure type correctness of e. The rewrite rules of Section 2 can then be used to determine whether the set is indeed consistent, and if so, the substitution will allow us to reconstruct the types of all the identifiers and subexpressions in e. The specification of this solver is highly non-deterministic, and in an actual implementation, choices will be made to make the process deterministic. Indeed, in many implementations these choices are made once and for all, while in our case we want to be able to choose the order for every expression independently.

4. The constraint-tree combinators

In the previous section we have given a version of the Hindley-Milner type system that uses sets of constraints and assumptions to declaratively specify the type system. We are now ready to introduce the combinators that we can use in these type rules to give extra structure to these sets of constraints.

Simply put, the combinators we introduce form an additional layer of syntax on top of the syntax of constraints. In this way, we are able to build *constraint trees* (instead of constraint sets), of which we can exploit the additional structure. The type system that results can be found in Figure 2, where the only differences are that we construct constraint trees $\mathcal{T}_c$, instead of constraint sets $\mathcal{C}$, and use special combinators for building the various kinds of constraint trees. In the remainder of this section, we shall explain the notation used in these type rules, but comparing Figure 1 to Figure 2 already indicates the "price" that needs to be paid for the added flexibility that comes from using the combinators. We feel this price is not very high.

$$\boxed{\mathcal{M}, \mathcal{A}, \mathcal{C} \vdash e : \tau}$$

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

$$\frac{\begin{array}{c} c_1 = (\tau_1 \equiv \beta_1 \to \beta_2) \quad c_2 = (\beta_1 \equiv \tau_2) \quad c_3 = (\beta_2 \equiv \beta_3) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \quad \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2 \end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{c_1, c_2, c_3\} \vdash e_1\ e_2 : \beta_3}\ (\textsc{App})$$

$$\frac{\begin{array}{c} c_1 = (\tau_1 \equiv \mathit{Bool}) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \quad \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2 \quad \mathcal{M}, \mathcal{A}_3, \mathcal{T}_{c3} \vdash e_3 : \tau_3 \end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3, \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{c_1, c_2, c_3\} \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \beta}\ (\textsc{Cond})$$

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

$$\frac{\begin{array}{c} c_1 = (\sigma_v := \textsc{Gen}(\mathcal{M}, \tau_1)) \quad \mathcal{C}_\ell = (\mathcal{A}_2 \preceq [x : \sigma_v]) \quad c_2 = (\beta \equiv \tau_2) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{C}_1 \vdash e_1 : \tau_1 \quad \mathcal{M}, \mathcal{A}_2, \mathcal{C}_2 \vdash e_2 : \tau_2 \end{array}}{\mathcal{M}, \mathcal{A}_1 \cup (\mathcal{A}_2 \backslash x), \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_\ell \cup \{c_1, c_2\} \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \beta}\ (\textsc{Let})$$

Figure 1. Type rules for a simple expression language

$$\boxed{\mathcal{M}, \mathcal{A}, \mathcal{T}_c \vdash e : \tau}$$

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

$$\frac{\begin{array}{c} c_1 = (\tau_1 \equiv \beta_1 \to \beta_2) \quad c_2 = (\beta_1 \equiv \tau_2) \quad c_3 = (\beta_2 \equiv \beta_3) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{T}_{c1} \vdash e_1 : \tau_1 \quad \mathcal{M}, \mathcal{A}_2, \mathcal{T}_{c2} \vdash e_2 : \tau_2 \end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2, c_3 \lozenge \lbag c_1 \triangledown \mathcal{T}_{c1}, c_2 \triangledown \mathcal{T}_{c2} \rbag \vdash e_1\ e_2 : \beta_3}\ (\textsc{App})$$

$$\frac{\begin{array}{c} \mathcal{T}_c = \lbag c_1 \triangledown \mathcal{T}_{c1}, c_2 \triangledown \mathcal{T}_{c2}, c_3 \triangledown \mathcal{T}_{c3} \rbag \\ c_1 = (\tau_1 \equiv \mathit{Bool}) \quad c_2 = (\tau_2 \equiv \beta) \quad c_3 = (\tau_3 \equiv \beta) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{T}_{c1} \vdash e_1 : \tau_1 \quad \mathcal{M}, \mathcal{A}_2, \mathcal{T}_{c2} \vdash e_2 : \tau_2 \quad \mathcal{M}, \mathcal{A}_3, \mathcal{T}_{c3} \vdash e_3 : \tau_3 \end{array}}{\mathcal{M}, \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3, \mathcal{T}_c \vdash \textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 : \beta}\ (\textsc{Cond})$$

$$\frac{\begin{array}{c} \mathcal{C}_\ell = ([x : \beta_1] \equiv \mathcal{A}) \quad c_1 = (\beta_3 \equiv \beta_1 \to \beta_2) \quad c_2 = (\tau \equiv \beta_2) \\ \mathcal{M} \mathbin{+\!\!+} \mathit{ftv}(\mathcal{C}_\ell), \mathcal{A}, \mathcal{T}_c \vdash e : \tau \end{array}}{\mathcal{M}, \mathcal{A} \backslash x, c_1 \lozenge \mathcal{C}_\ell \underline{\lozenge}^{\circ} \lbag c_2 \triangledown \mathcal{T}_c \rbag \vdash \lambda x \to e : \beta_3}\ (\textsc{Abs})$$

$$\frac{\begin{array}{c} \mathcal{T}_c = (c_2 \lozenge \lbag \mathcal{T}_{c1} \ll [c_1]^{\bullet} \ll (\mathcal{C}_\ell \underline{\lll}^{\circ} \mathcal{T}_{c2}) \rbag) \\ c_1 = (\sigma_v := \textsc{Gen}(\mathcal{M}, \tau_1)) \quad \mathcal{C}_\ell = (\mathcal{A}_2 \preceq [x : \sigma_v]) \quad c_2 = (\beta \equiv \tau_2) \\ \mathcal{M}, \mathcal{A}_1, \mathcal{T}_{c1} \vdash e_1 : \tau_1 \quad \mathcal{M}, \mathcal{A}_2, \mathcal{T}_{c2} \vdash e_2 : \tau_2 \end{array}}{\mathcal{M}, \mathcal{A}_1 \cup (\mathcal{A}_2 \backslash x), \mathcal{T}_c \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \beta}\ (\textsc{Let})$$

Figure 2. Type rules for a simple expression language

Typically, the constraint tree has the same shape as the abstract syntax tree of the expression for which the constraints are generated. 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 *must* be solved before others, and we can encode this in the constraint tree as well.

This results in the four main alternatives for constructing a constraint tree.

$$\mathcal{T}_c \quad ::= \quad \bullet\, \mathcal{T}_{c1}, \ldots, \mathcal{T}_{cn} \,\bullet \;|\; c \lozenge \mathcal{T}_c \mid c \triangledown \mathcal{T}_c \mid \mathcal{T}_{c1} \ll \mathcal{T}_{c2}$$

Note that to minimize the use of parentheses, all combinators to build constraint trees are right associative. With the first alternative we combine a list of constraint trees into a single tree with a root and $\mathcal{T}_{ci}$ as subtrees. The second and third alternatives add a single constraint to a tree. The case $c \lozenge \mathcal{T}_c$ makes constraint c part of the constraint set associated with the root of $\mathcal{T}_c$. The constraint that the type of the body of the let equals the type of the let (see (LET) in Figure 2) is a typical example of this.

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*. For this reason, we wrote $c_i \triangledown \mathcal{T}_{ci}$ $(i = 1, 2, 3)$ in the rule (COND) in Figure 1, instead of $c_1 \lozenge c_2 \lozenge c_3 \lozenge \bullet\, \mathcal{T}_{c1}, \mathcal{T}_{c2}, \mathcal{T}_{c3} \,\bullet$. In both cases, the constraints are generated by the conditional node, but in the former case the constraints are associated with the respective subtree, and in the latter case with the conditional node itself. This choice will give improved flexibility later on.

The last case $(\mathcal{T}_{c1} \ll \mathcal{T}_{c2})$ combines two trees in a strict way: all 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 of a let and those for the body. When one considers the rewrite rules for our constraint language in Section 2, this is not necessary, because the solver can determine that a given generalization constraint may be solved. However, this gives extra work for the solver, and by insisting that the constraints from the definition are solved before the generalization constraints, we can omit to verify this property altogether and speed up the solving process considerably. This is an example where a combinator can be used to forbid certain orderings of the constraints to preserve soundness and completeness of a (simplified) solver.

For brevity, we introduce the underlined version of $\lozenge$ and $\triangledown$, which we use for adding lists of constraints. For instance,

$$[c_1, \ldots, c_n] \,\underline{\lozenge}\, \mathcal{T}_c \quad =_{def} \quad c_1 \lozenge \ldots \lozenge c_n \lozenge \mathcal{T}_c.$$

This also applies to similar combinators to be defined later in this paper. We write $\mathcal{C}^\bullet$ for a constraint tree with only one node: this abbreviates $\mathcal{C} \,\underline{\lozenge}\, \bullet\;\bullet$.

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 constraint trees.

4.1 Flattening a constraint tree

At some point, a constraint tree has to be turned into a constraint list, in order to be fed into a solver of some kind. This is done by choosing a particular semantics for the combinators (except $\ll$ and its variants which have a fixed semantics). The flexibility of our work derives from the fact that we can vary the semantics of the combinators, yielding different but equally valid solving orders. It is important to note that to obtain this, we do not need to change either the constraint generating process, or the solving process.

Therefore, our first concern is how to convert a constraint tree into a list: for this, we use the function *flatten* (note that we use Haskell code in this section to define the semantics of our operators). How a tree is converted depends on the tree walk of our choice, which is a parameter of *flatten*. A tree walk specifies the order in which the constraints generated for a single node should be solved.

```
data TreeWalk = TW (∀ a.[a] → [([a], [a])] → [a])
```

```
flatten :: TreeWalk → ConstraintTree → [Constraint]
flatten (TW 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
        • t1, ..., tn • → let pairs = map (flattenRec [ ]) [t1, ..., tn]
                           in (f down pairs, [ ])
        c ◊ t          → flattenRec (down ++ [c]) t
        c ▽ t          → let (C, up) = flattenRec down t
                           in (C, up ++ [c])
        t1 ≪ t2        → let cs1 = flattenTop t1
                                 cs2 = flattenTop t2
                           in (f down [(cs1 ++ cs2, [ ])], [ ])
```

The first argument of the tree walk function corresponds to the constraints belonging to the node itself, the second argument 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 (recursively flattened) subtree, the second element those constraints that the node associates with the subtree. Note that if we did not have both $\lozenge$ and $\triangledown$, then a treewalk would only take the constraints associated with the node itself, and a list containing the lists of constraints coming from the children as a parameter.

The function *flatten* simply traverses the constraint tree, and 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. Of course, the constraint ordering for the strict combinator $\ll$ is fixed and does not depend on the tree walk.

Our first example tree walk is truly bottom-up.

```
bottomUp = TW (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. In a similar way, we define the dual tree walk, which is a top-down approach.

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

EXAMPLE 4.1. Let t be $c_3 \lozenge \bullet\, c_1 \triangledown \mathcal{C}_1^\bullet, c_2 \triangledown \mathcal{C}_2^\bullet \,\bullet$. Flattening this constraint tree with our two tree walks gives us:

$$\begin{array}{lcl} \textit{flatten bottomUp } t & = & \mathcal{C}_1 +\!\!+ \mathcal{C}_2 +\!\!+ [c_1] +\!\!+ [c_2] +\!\!+ [c_3] \\ \textit{flatten topDown } t & = & [c_3] +\!\!+ [c_1] +\!\!+ [c_2] +\!\!+ \mathcal{C}_1 +\!\!+ \mathcal{C}_2 \end{array}$$

Other useful tree walks interleave the associated constraints and the recursively flattened constraint trees for each subexpression of a node. Here, we have two choices to make: do the associated constraints precede or follow the constraints from the corresponding child, and do we put the remaining constraints (those that are not associated with a subexpression) in front or at the end of the list? These two options lead to the following helper-function.

```
variation :: (∀ a.[a] → [a] → [a]) →
             (∀ a.[a] → [a] → [a]) → TreeWalk
variation f g =
  TW (λdown list → f down (concatMap (uncurry g) list))
```

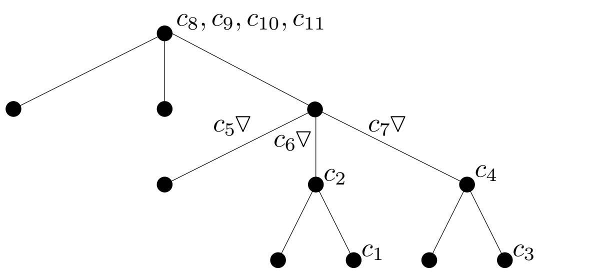

Figure 3. The constraint tree

For both arguments of $\mathit{variation}$, we consider two alternatives: combine the lists in the order given ($+\!\!+$), or flip the order of the lists ($\mathit{flip}\ (+\!\!+)$). For instance, the constraint tree from Example 4.1 can now be flattened in the following way:

$$\mathit{flatten}\ (\mathit{variation}\ (+\!\!+)\ (+\!\!+))\ t = [c_3] +\!\!+ \mathcal{C}_1 +\!\!+ [c_1] +\!\!+ \mathcal{C}_2 +\!\!+ [c_2]$$

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.

```
$\mathit{reversed} :: \mathit{TreeWalk} \to \mathit{TreeWalk}$
$\mathit{reversed}\ (\mathit{TW}\ f) = \mathit{TW}\ (\lambda \mathit{down}\ \mathit{list} \to f\ \mathit{down}\ (\mathit{reverse}\ \mathit{list}))$
```

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

EXAMPLE 4.2. We generate constraints for the expression given below. For this, we use type rules similar to the ones defined in Figure 2, and take the liberty of including a conditional in the example. 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.

```
$\lambda f^{v_0} \to \lambda b^{v_1} \to$ **if** $b^{v_2}$
                    **then** $(f^{v_3}\ \ 1^{v_4})^{v_5}$
                    **else** $(f^{v_6}\ \ \mathit{True}^{v_7})^{v_8}$
```

The constructed constraint tree t for this expression is shown in Figure 3, and the constraints are given in Figure 4. 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, and underline the constraints where the inconsistency is detected.

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

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 syntax tree, then the definition of *flatten* can straightforwardly be generalized to treat different language constructs differently:

$$\mathit{flatten} :: (\mathit{String} \to \mathit{TreeWalk}) \to$$

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

Figure 4. The constraints

$$\mathit{ConstraintTree} \qquad \to [\mathit{Constraint}].$$

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 $\mathcal{G}$ [10], such as exhibiting $\mathcal{M}$-like behavior for one construct and $\mathcal{W}$-like behavior for another. Of course, we could generalize *flatten* even 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. More specifically, by spreading constraints we can also emulate algorithms that use a top-down type environment (usually denoted by Γ), even though our rules may use a bottom-up assumption set to collect the constraints.

The grammar for constraint trees is extended with three cases.

$$\mathcal{T}_c \quad ::= \quad (\ldots) \ \mid\ (\ell, c) \triangledown^{\circ}\, \mathcal{T}_c \ \mid\ (\ell, c) \ll^{\circ} \mathcal{T}_c \ \mid\ \ell^{\circ}$$

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 $\ll^{\circ}$). We expect for every constraint that is spread to have exactly one receiver in its scope. In our particular case, we enforce this by using the generated fresh type variable (see the rule (VAR) in Figure 2) as the receiver, and the fact that the let and lambda rules remove assumptions for identifiers bound at that point.

The function spread is responsible for moving constraints deeper into the tree, until they end up at their destination label. It can be implemented straightforwardly:

```
$\mathit{spread} :: \mathit{ConstraintTree} \to \mathit{ConstraintTree}$
$\mathit{spread} = \mathit{spreadRec}\ [\,]$
  **where**
    $\mathit{spreadRec} :: [(\mathit{Label}, \mathit{Constraint})] \to$
                   $\mathit{ConstraintTree} \to \mathit{ConstraintTree}$
    $\mathit{spreadRec}\ \mathit{list}\ \mathit{tree} =$
      **case** $\mathit{tree}$ **of**
        $⦃\, t_1, \ldots, t_n \,⦄ \to ⦃\, \mathit{map}\ (\mathit{spreadRec}\ \mathit{list})\ [t_1, \ldots, t_n] \,⦄$
        $c \lozenge t$            $\to c \lozenge \mathit{spreadRec}\ \mathit{list}\ t$
        $c \triangledown t$            $\to c \triangledown \mathit{spreadRec}\ \mathit{list}\ t$
        $t_1 \ll t_2$          $\to \mathit{spreadRec}\ \mathit{list}\ t_1 \ll \mathit{spreadRec}\ \mathit{list}\ t_2$
```

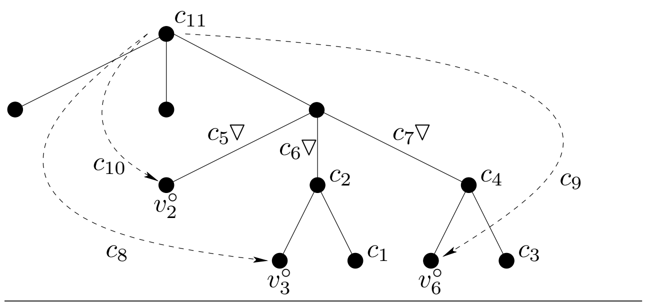

Figure 5. Constraint tree with type constraints that have been spread

$$\begin{array}{ll}(\ell, c)\ \triangledown^\circ\ t & \rightarrow \mathit{spreadRec}\ ((\ell, c) : \mathit{list})\ t \\ (\ell, c) \lll^\circ t & \rightarrow \mathit{spreadRec}\ ((\ell, c) : \mathit{list})\ t \\ \ell^\circ & \rightarrow [c \mid (\mathit{lab1}, c) \leftarrow \mathit{list}, \ell == \mathit{lab1}]^\bullet\end{array}$$

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

EXAMPLE 4.3. Consider the constraint tree t in Figure 3. We spread the type constraints introduced for the 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 is displayed in Figure 5.

$$\begin{array}{l}\mathit{flatten}\ \mathit{bottomUp}\ (\mathit{spread}\ t) \\ \quad = [c_{10}, c_8, c_1, c_2, c_9, c_3, \underline{c_4}, c_5, c_6, c_7, c_{11}] \\ \mathit{flatten}\ \mathit{topDown}\ (\mathit{spread}\ t) \\ \quad = [c_{11}, c_5, c_6, c_7, c_{10}, c_2, c_8, c_1, c_4, c_9, \underline{c_3}] \\ \mathit{flatten}\ (\mathit{reversed}\ \mathit{bottomUp})\ (\mathit{spread}\ t) \\ \quad = [c_3, c_9, c_4, c_1, c_8, \underline{c_2}, c_{10}, c_7, c_6, c_5, c_{11}]\end{array}$$

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

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. One might argue that the dual approach could be taken: to use type environments by default and to emulate assumption sets by means of a combinator. However, spreading downwards turned out to be easier to handle than spreading upwards.

Spreading type constraints gives constraint orderings that correspond more closely to the type inference process of Hugs [7] and GHC [4]. 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.

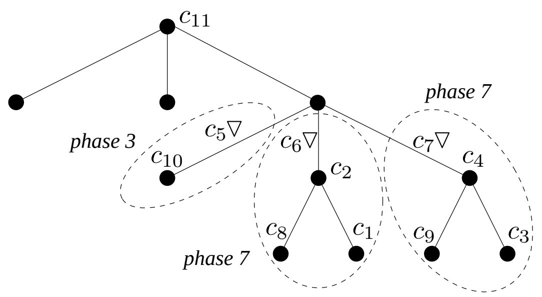

Figure 6. The constraint tree before phasing

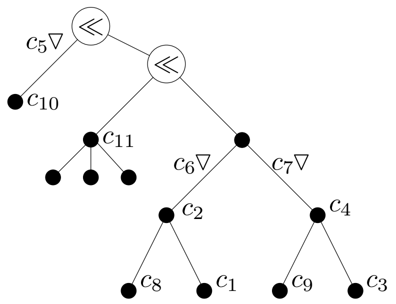

Figure 7. The constraint tree after phasing

4.3 Phasing constraint trees

The essence of phased type inference, a concept introduced in [6], is assigning phase numbers to type constraints. Constraints with a low phase number are solved before those with a high phase number. With phasing we can (conceptually) visit nodes of an abstract syntax tree more than once.

The original motivation for phased type inference was to depart from the fixed unification order for programs that use a domain-specific combinator library. For instance, we could first type the "normal" expressions between the combinators, and check in subsequent phases whether the types of the operands actually match with the combinator types. Other examples of phasing constraints are propagating type signatures by first "pushing down" an expected type, and assigning low phase numbers to constraints that were generated during an earlier compilation. This has the effect of putting the blame on more recently developed pieces of code.

To support phasing, we introduce one new constructor, $\mathit{Phase}\ i\ \mathcal{T}_c$, and a function phase that transforms a constraint tree such that it respects the policy of the ($\lll$) combinator. The implementation of phase is omitted for reasons of space, but we can illustrate its effects by means of an example.
Consider again the expression

$(\lambda f\ b \rightarrow \mathbf{if}\ b\ \mathbf{then}\ f\ 1\ \mathbf{else}\ f\ \mathit{True})$

and its constraint tree after spreading type constraints, shown in Figure 6. 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 7 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.

As an aside, note that traditional type inference algorithms can hardly be extended with support for phasing. The use of constraints makes this a lot easier to do.

5. Emulating inferencing algorithms

To further illustrate the flexibility of the (small) set of combinators we have introduced, we show how we can emulate some of the existing algorithms in the literature, in the sense that the list of constraints for a given flattening corresponds to the unifications performed by such an algorithm. We consider here $\mathcal{W}$ [2] and $\mathcal{M}$ [9], algorithm $\mathcal{G}$ and one of its instances called $\mathcal{H}$ [10].

Algorithm $\mathcal{W}$ proceeds in a bottom-up fashion, and considers the children from left-to-right. Second, it treats the let-expression in exactly the same way as we do: first the definition, followed by generalization, and finally the body. This behavior corresponds to the $bottomUp$ tree walk introduced earlier. Furthermore, we see that a type environment is passed down, which means that we have to spread constraints. The combination of the $bottomUp$ tree walk and spreading ensures that our solver only fails for constraints generated at applications.

Folklore algorithm $\mathcal{M}$, on the other hand, is a top-down inference algorithm for which we should select the $topDown$ tree walk. Spreading with this tree walk implies that we no longer fail at application nodes, but for identifiers and lambda abstractions.

Algorithm $\mathcal{G}$ by Lee and Yi [10] defines a set of algorithms. 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 of $\mathcal{G}$, we focus on the case for applications $e_1\ e_2$. We start with choosing 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 the expected type ρ (see constraint (2) in Fig. 3 of [10]). Next, we can decide to strengthen our demands on the type found for e_1, or postpone this to after considering e_2 (3). Then, we decide on the expected type passed to e_2: the type β (which is now 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 proceed by considering the application case for one instance, namely algorithm $\mathcal{H}$ [10]. 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 from type rule (APP).

6. Related work

We are certainly not the first to consider a more flexible approach in solving constraints (or, in most cases, perform unifications) in various orders. Algorithm $\mathcal{G}$ [10], presented by Lee and Yi, can be instantiated with different parameters, yielding the well-known algorithms $\mathcal{W}$ and $\mathcal{M}$ (and many others). Our constraint-based approach has a number of advantages over their generalized algorithm, which essentially selects a number of unifications to be performed early. The soundness of their algorithm follows from the decision to simply perform all unifications before the abstract syntax tree node is left for the final time. This includes unifications which were done during an earlier visit to the node, which is harmless, but not very efficient. Additionally, all these moments of performing unifications add complexity to the algorithm: the application case alone involves five substitutions that have to be propagated carefully. Our constraint-based approach with a constraint ordering phase circumvents this complexity. Instances of algorithm $\mathcal{G}$ are restricted to one-pass, left-to-right traversals with a type environment that is passed top-down: it is not straightforward to further generalize $\mathcal{G}$ to have support for phased type inference strategies (Section 4.3), or algorithms that remove the left-to-right bias [19, 12].

Sulzmann presents constraint propagation policies [17] for modeling $\mathcal{W}$ and $\mathcal{M}$ in the HM(X) framework [18]. First, general type rules are formulated that mention partial solutions of the constraint problem: later, these rules are specialized to obtain $\mathcal{W}$ and $\mathcal{M}$. While interesting soundness and completeness results are discussed for his system, he makes no attempt at defining one implementation that can handle all kinds of propagation policies.

Hindley-Milner's type system has been formulated with constraints several times. Typically, the constraint-based type rules use logical conjunction (e.g., the HM(X) framework [18]), or an unordered set of constraints is collected (e.g., Pierce's first textbook on type systems [14]). Type rules are primarily intended as a declarative specification of the type system, and from this point of view our combinators are nothing but generalizations of $(\wedge)$. However, when it comes to implementing the type rules, our special combinators also bridge the gap between the specification of the constraints and the implementation, which is the solver.

Finally, Pottier and Rémy present constraint-based type rules for ML [15]. Their constraint language contains conjunction (where we use the comma) and *let* constraints (where we use generalization and instantiation constraints). The main drawback of their setup is the specified solver uses a stack, essentially to traverse the constraint, making the specification of the solver as a rewrite system overly complex and rigid (see Figure 10-11 in [15]). Our combinators could help here to decouple the traversal of the constraint from the constraint semantics.

7. Conclusion and future work

In this paper we have advocated the introduction of a separate constraint reordering phase between the phase that generates the constraints and the phase that solves constraints. We have made a beginning in this respect by presenting a number of combinators that can be used directly in the constraint generation rules (which are very similar to the logical deduction rules used in the literature). By way of example, we have given a specification of a constraint based type inferencer for the Hindley-Milner type system, and showed that many well-known algorithms that implement this type system can be effectively emulated by choosing a suitable semantics for our combinators.

The main benefit of our work is the decoupling between the three phases, making each of them much simpler than when considered combined. Other benefits are the higher flexibility, increased re-use of constraint solvers and the possibility to construct simplified or more efficient solvers that can be used if constraints are known to be supplied to the solver in a specific order, e.g., the use of the $\ll$ combinator in the let rule. Finally, our combinators, and others like them, can play a large role in the development of domain specific languages for specifying executable program analyses, such as envisioned in systems such as TinkerType [11] and Ruler [3].

The combinators we described are indeed only the beginning. Once the realization is made that the ordering of constraints is an issue, it is not difficult to come up with a host of new combinators, each with their own special characteristics and uses, and that can be used in combination with existing solvers. For example, combinators can be defined that specify that certain parts of the constraint solving process can be performed in parallel, guaranteeing that the results of these parallel executions can be easily integrated.

References

[1] Anonymous. Anonymous unrefereed workshop 2006. Details omitted for double blind reviewing.

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

[3] A. Dijkstra and S. D. Swierstra. Ruler: Programming type rules. In *FLOPS*, pages 30 – 46, 2006.

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

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

[6] B. Heeren, J. Hage, and S. D. Swierstra. Scripting the type inference process. In *Eighth ACM Sigplan International Conference on Functional Programming*, pages 3 – 13, New York, 2003. ACM Press.

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

[8] S. Peyton Jones, D. Vytiniotis, S. Weirich, and M. Shields. Practical type inference for arbitrary-rank types. *Journal of Functional Programming*, To appear.

[9] 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.

[10] Oukseh Lee and Kwangkeun 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.

[11] M. Y. Levin and B. C. Pierce. Tinkertype: A language for playing with formal systems. *Journal of Functional Programming*, 13(2):295 – 316, March 2003.

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

[13] F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, second printing edition, 2005.

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

[15] 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.

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

[17] Martin Sulzmann. A general type inference framework for hindley/milner style systems. In *FLOPS*, pages 248–263, 2001.

[18] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. Research Report YALEU/DCS/RR-1129, Yale University, Department of Computer Science, April 1997.

[19] J. Yang. Explaining type errors by finding the sources of type conflicts. In Greg Michaelson, Phil Trindler, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.