Strategies for solving constraints in type and effect systems

Jurriaan Hage[1] and Bastiaan Heeren[2]

[1] Department of Information and Computing Sciences, Universiteit Utrecht P.O.Box 80.089, 3508 TB Utrecht, The Netherlands jur@cs.uu.nl
[2] Faculty of Computer Science, Open Universiteit Nederland, Bastiaan.Heeren@ou.nl

Abstract. Turning type and effect deduction systems into an algorithm is a tedious and error-prone job, and usually results in an implementation that leaves no room to modify the solving strategy, without actually changing it. We employ constraints to declaratively specify the rules of a type system. Starting from a constraint based formulation of a type system, we introduce special combinators in the type rules to specify in which order constraints may be solved. A solving strategy can then be chosen by giving a particular interpretation to these combinators, and the resulting list of constraints can be fed into a constraint solver; thus the gap between the declarative specification and the deterministic implementation is bridged. This design makes the solver simpler and easier to reuse. Our combinators have been used in the development of a real-life compiler.

Category: D.3.4, Programming Languages, Processors, Compilers.
Keywords: type and effect systems, inference algorithms, constraints, solving strategies

1 Introduction

Volpano and Smith [20] showed how security analysis can be specified as a type and effect system (*type system* for short). Security analysis aims to reject programs that exhibit unsafe behaviour, i.e., when sensitive information may be copied to a location reserved for less sensitive information. Therefore, it is considered to be a validating program analysis, and the implementer must not only implement the analysis, but also provide sensible feedback in case the analysis fails. Providing this feedback can be a time consuming and arduous task.

To improve feedback we may investigate the kinds of mistakes programmers make and use that information to construct a heuristics that can help in finding what is the most likely source of a mistake, cf. [5]. However, this is quite an undertaking and by its nature largely language and analysis specific. Therefore, it would be nice to have a more generic solution to the generation of good feedback that can be more easily reused for different analyses, or even different programming languages. If needs be, heuristics can then be added later on as a refinement.

In this paper, we describe a framework that can be used by compiler builders to accomplish exactly this. To illustrate it, we show how the framework can be used in the context of type inferencing the polymorphic lambda calculus, i.e., we consider an analysis of the underlying types of, e.g., a Security Analysis [20]. There is nothing in our development, however, that makes assumptions about the analysis or the programming language involved. The framework has been used in the construction of a real-life compiler, the Helium compiler for Haskell [7].

The paper is structured as follows. After a section on motivation and application, we need some preliminaries to introduce types and constraints on types. We then consider a variant of the Hindley-Milner type system [13, 1], which uses assumption sets and sets of constraints. In Section 5 we introduce a modified type system which uses many of our combinators, and then consider these combinators in detail. Then, we informally indicate how we can emulate various well-known algorithms and implementations for type inferencing by choosing a suitable semantics for our operators as an illustration of the flexibility of our framework, and in Section 7 we discuss how proofs of soundness can be conducted in our approach. In the last two sections, we discuss related work and present our conclusions.

2 Motivation and applications

In type and effect systems, a program analysis is specified by means of a collection of (type) rules that declaratively specifies properties that a program should have. It is the basis of an algorithm that builds a derivation tree for the program and verifies that it satisfies these rules (typically, the "best" possible derivation tree).

A standard text on the subject [14] illustrates the distinction well: compare the deduction system in Table 5.2 with the standard algorithm W in Table 5.8. The algorithm also exhibits a number of drawbacks: Getting all the details correct, e.g., applying the obtained substitutions in all the right places, is not an easy task even for such a simple analysis of such a small language. How the abstract syntax tree of the program is traversed is fixed once and for all, and this can seriously bias the error messages that result.

For example, consider the program:

```
abs2 x = if x 0 then -(2*x) else 2*x
```

Because W traverses the tree from left-to-right, there is a bias in discovering mistakes towards the end of the program. In this case, the condition indicates that x should be a function. Algorithm W will then complain that it cannot multiply a function, although there is much more evidence that x is an integer and that the condition is incorrect. Algorithm W will never take into account that the first use of x might be wrong, and, moreover, when explaining the mistake in the then-part, will not use information from the else-part, or even explain why it concluded that the type of x is boolean. It simply does not have this kind of information.

Choosing a different implementation of the type system, like algorithm M [9], does not help: one fixed order will be exchanged for another. However, if the compiler by the nature of its implementation easily allows different traversals over the abstract syntax tree, we may experiment with several such traversals, see what they come up with, and use that information to come up with a better diagnosis of the problem. The design of our framework naturally allows this.

We propose to do the following: consider any type and effect system, say Security Analysis [20]. Separate the type and effect system into two different parts: a declarative specification in terms of constraints that need to be satisfied (notationally close to the usual type deduction rules), and a solver for the kinds of constraints used in the specification. The analysis process then becomes a matter of traversing the abstract syntax of the program, generating the constraints for the program and feeding the constraints to the solver, so that it can decide whether the constraints are consistent.

Our main conceptual contribution is to impose a layer of ordering combinators on top of the constraint language that allows to indicate

– that certain constraints essentially belong together,
– that a programmer may want to choose at compile time in which order particular subsets of constraints should be solved,
– or that certain constraints must always be considered in some fixed order.

Before constraints are solved, a particular solving strategy is chosen by selecting a semantics for the ordering combinators, ensuring that a list of constraints results that can be fed into the solver. Operationally, the ordering process is a third phase that takes place between the generation of constraints in the abstract syntax tree and solving the constraints. The important point here is that different strategies can be used without changing the compiler.

The flexibility obtained in this way can be used in a number of ways: First, there is no need to choose at compiler construction time a fixed strategy to solve constraints, e.g., this decision can be postponed until experimentation with the compiler has shown what works best on average. Moreover, the flexibility of the framework can even be passed on to the programmers, to let them decide for themselves what works best for them. The framework can also be used directly in a setting in which a compiler "learns" to apply the best ordering, based on a training session with a programmer.

3 Preliminaries

The running example of this paper describes type inference for the Hindley-Milner type system [13], and we assume the reader has some familiarity with this type system and the associated inference algorithm [1]. 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).

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

The function $\mathit{ftv}(\sigma)$ returns the free type variable of its argument, and is defined 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:

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

A type is an instance of a type scheme, written as $\tau_1 < \forall \overline{a}.\tau_2$, if there exists a substitution S such that $\tau_1 = S\tau_2$ and $\mathit{domain}(S) \subseteq \overline{a}$. For example, $a \to \mathit{Int} < \forall ab.a \to b$ by choosing $S = [b := \mathit{Int}]$.

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 specify the generalization of 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.

The reason we have constraints to explicitly represent generalization and instantiation is the same as why, e.g., Pottier and Remy do [16]: otherwise we would be forced to (make a fresh) duplicate of the set of constraints every single time we use a polymorphically defined identifier. Such duplication must be avoided, both for reasons of efficiency and because errors might be duplicated, if the polymorphic definition itself is inconsistent. As we shall see later, solving these types of constraints induces a certain amount of bias, which, in the interest of efficiency, is unavoidable.

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,

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

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).

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

For a constraint set $\mathcal{C}$, we start with the solution $(\lambda\mathcal{C}, \mathit{id}, \mathit{id})$ and apply the following rewrite rules until the set of constraints is empty (signifying success, in which case the substitutions are returned), or it is not empty and none of the rules of apply (signifying error, in which case we return $(\top, \top)$). Note that in these rules, $\cup$ denotes a pattern matching operator.

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

where the standard algorithm *mgu* is used to find a most general unifier of two types [17] and the function *actives* computes the set of type variables that may still change whilst solving $\mathcal{C}$. (See p.48 of [6] for a formal definition.)

That the solving process imposes a certain bias is implicit in the side conditions for the generalization and instantiation constraints. To solve an instantiation constraint, the right hand side must be 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, which is exactly what *actives* determines.

4 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 [13] formulated in terms of constraints. Such a type system is the basis of a type and effect based analysis, e.g., Security Analysis [20], in which annotations are attached to the types, and constraints between the annotations need to be satisfied in order for the program to be valid for the analysis.

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

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

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

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

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

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

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

Fig. 1. Type rules for a simple expression language

$$e \quad ::= \quad x \ \mid \ e_1\ e_2 \ \mid \ \lambda x \to e \ \mid \ \textbf{let } x = e_1 \textbf{ in } e_2 \ \mid \ \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3$$

These rules specify how to construct a constraint set 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; here we use the notation $\mathcal{A}\backslash x$ to denote the removal of all assumptions about x from $\mathcal{A}$. 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 assumptions, then vice versa (see the discussion on spreading later in this paper). Theorem 4.14 and 4.15 in [6] show that the above formulation of the type system is equivalent to the original Hindley-Milner type system.

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 [14]). 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 $\tau_1 \equiv \beta_1 \to \beta_2$. Because of the invariant, we know that τ_1 is actually a type variable. At constraint generation time, 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 $\tau_1 \equiv \tau_2 \to \beta_3$. 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.

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 3 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 subexpressions of e. The specification of this solver is highly non-deterministic. During an actual run of the implementation, choices will be made to make the process deterministic.

5 The constraint-tree combinators

In this section we again consider the type system of Section 4 and introduce the combinators that we can use in these type rules to give extra structure to the sets of constraints.

The combinators we introduce form an additional layer of syntax on top of the syntax of constraints. Terms in this layered language are essentially *constraint trees*, giving us added structure to exploit. Formally, the type system of Figure 2 essentially differs from Figure 1 in that we construct constraint trees $\mathcal{T}_c$, instead of constraint sets $\mathcal{C}$, and use special combinators, e.g., for building the various kinds of constraint trees. In the remainder of this section, we explain Figure 2 in more detail. Comparing the two figures already indicates the "price" that needs to be paid for the added flexibility that comes from using the combinators.

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

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

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

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

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

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

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

Fig. 2. Type rules for a simple expression language, with ordering combinators

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.

$$\mathcal{T}_c \quad ::= \quad \{\!\!\{ \mathcal{T}_{c1}, \ldots, \mathcal{T}_{cn} \}\!\!\} \ \mid \ c \Diamond \mathcal{T}_c \mid c \triangledown \mathcal{T}_c \mid \mathcal{T}_{c1} \ll \mathcal{T}_{c2}$$

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 $c \Diamond \mathcal{T}_c$ 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, e.g., the constraint that the condition of an if-then-else expression must have type *Bool*. For this reason, we used $c_i \triangledown \mathcal{T}_{ci}$ $(i = 1, 2, 3)$ in the rule (Cond) in Figure 1, instead of $c_1 \Diamond c_2 \Diamond c_3 \Diamond \{\!\!\{ \mathcal{T}_{c1}, \mathcal{T}_{c2}, \mathcal{T}_{c3} \}\!\!\}$. 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 final 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 3, 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 the side conditions for the instantiation and generalization constraints altogether and thereby speed up and simplify the solving process considerably.

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

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

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 $\mathcal{C} \underline{\Diamond} \{ \}$.

In the remaining part of this section, we discuss how to flatten constraint trees to a constraint list, and how to use spreading to simulate type systems that use type environments instead of sets of type assumptions.

5.1 Flattening a constraint tree

Our first concern is how to convert a constraint tree into a list, to be fed into a solver. This is done by choosing a particular semantics for some of the combinators (excluding $\ll$ and its variants which have a fixed semantics). Indeed, the flexibility of our framework derives from the fact that we can choose the semantics of the combinators differently for every single compilation. This then yields different but equally valid solving orders. It is essential to note that we change neither the constraint generating process, nor the solving process. We simply make use of degrees of freedom left open by the specification of the type rules.

The main function is *flatten* :: *TreeWalk* $\rightarrow$ *ConstraintTree* $\rightarrow$ [*Constraint*] that takes two parameters: a tree walk that represents the ordering strategy to be applied and a constraint tree. It returns a list of constraints to be fed into the constraint solver. For reasons of space, we often omit the actual implementation of functions. These can be found in Section 5.3 of [6]

A *TreeWalk* has the following type: $\forall\ a.[a] \rightarrow [([a],[a])] \rightarrow [a]$. The first argument 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 $\Diamond$ 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.

Intuitively, the higher-order function *flatten* is an iterator that traverses the constraint tree, and uses the *TreeWalk* to 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 ordered in the list. Of course, the constraint ordering for the strict combinator $\ll$ is fixed and does not depend on the chosen tree walk.

We now consider some examples. The first is a tree walk that is fully bottom-up.

$bottomUp = TW\ (\lambda down\ list \rightarrow f\ (unzip\ list) +\!\!+ down)$
  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 top-down tree walk:

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

Example 1. Applying our two tree walks to t be $c_3 \Diamond \left\{ c_1 \bigtriangledown \mathcal{C}_1^{\bullet}, c_2 \bigtriangledown \mathcal{C}_2^{\bullet} \right\}$ gives

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

Some 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.

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

For both arguments of *variation*, we consider two alternatives: combine the lists in the order given $(+\!\!+)$, or flip the order of the lists ($\mathit{flip}\ (+\!\!+)$). For instance, the constraint tree from Example 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, again a higher-order function: it takes a *TreeWalk* and builds the *TreeWalk* which behaves in exactly the same way, except that the children of each node are inspected in reverse order. Of course, this reversal is not applied to nodes with a strict ordering. With this transformer, we can inspect a program from right-to-left, instead of the standard left-to-right order.

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

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

Example 2. We generate constraints for the expression e given below following the type rules of Figure 2. 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.

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

The constructed constraint tree t for e 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

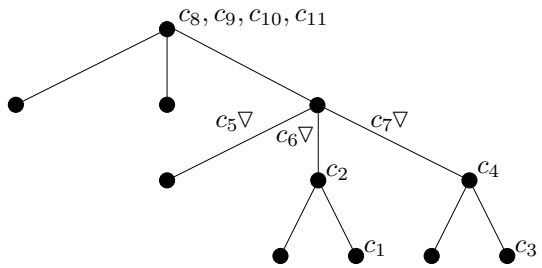

Fig. 3. The constraint tree

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

Fig. 4. The constraints

inconsistency is detected.

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

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:

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

This extension enables us to model inference processes such as the one of Hugs [8], 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.

5.2 Spreading type constraints

Spreading allows to move type constraints from one place in the constraint tree to a different location. In particular, we will consider constraints that relate the definition site and the use site of an identifier. This is necessary, because the type system of Figure 2 uses type assumptions that are propagated from the leaves upwards to the binding sites, whereas most type systems essentially pass down constraints from the binding site of an identifier to all of its uses. In other words, by spreading constraints we can emulate algorithms that use a top-down type environment (usually denoted by Γ), even though our rules use a bottom-up assumption set to construct the constraints.

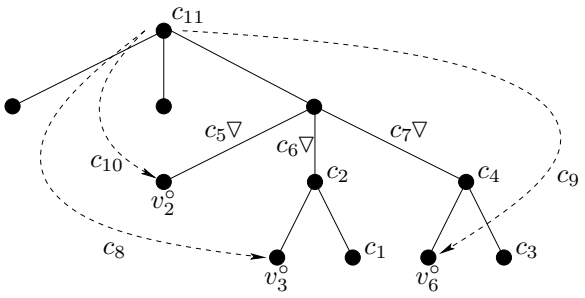

Fig. 5. Constraint tree with type constraints that have been spread

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, as a mapping from *ConstraintTree* to *ConstraintTree* so that in the latter, all constraints have been moved to the corresponding receiver. We can then use this tree as input for *flatten* to compute the list of constraints. It may be that the user does not choose to apply *spread* to the constraint tree (but the type system does use the combinators for spreading), and to apply *flatten* directly. The behaviour of *flatten* will then be to ignore the spreading specific, e.g., $\ll^\circ$ will be interpreted as $\ll$.

Example 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.

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

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

Due to space restriction we omit a discussion of a general facility called *phasing* in which constraints are assigned to a phase so that constraints from an earlier phase are considered before those assigned to later phases (see Section 5.3.3 of [6]).

6 Emulating existing algorithms

To further illustrate the flexibility of the (small) set of combinators we have introduced, we show informally how a selection of existing algorithms can be emulated, in the sense that the list of constraints for a given flattening corresponds to the unifications performed by such an algorithm.

Algorithm W [1] proceeds in a bottom-up fashion, and considers subtrees from left-to-right. Second, it treats the let-expression in exactly the same way as we do: first the definition, followed by a generalization step, and then the body. This behavior corresponds to the *bottomUp* tree walk introduced earlier. Furthermore, a type environment is passed down, which means that we have to spread constraints.

Folklore algorithm M [9], 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 or conditional nodes, but for identifiers and lambda abstractions. We note that the Helium compiler [7] provides flags $-M$ and $-W$ to mimic algorithm M and W respectively, showing that our combinators can indeed be used to give control over the constraint solving order to the programmer. Other strategies can be provided easily; that is simply a matter of associating a treewalk with a particular compiler flag.

Spreading type constraints gives constraint orderings that correspond more closely to the type inference process of Hugs [8] and GHC [3]. 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.

Due to restrictions of space, other more advanced algorithms, notably Algorithm $\mathcal{G}$ and one of its instances $\mathcal{H}$ by Lee and Yi [10], and Algorithm $\mathcal{U}_{\mathrm{AE}}$ by Jun et al. [21], are considered in a technical report [4].

7 Soundness

One of the issues in developing program analyses is to prove the soundness of a program analysis. Typically, we then first prove the logical deduction system sound with respect to the semantics of the programming language, and then prove the correctness of the algorithm, usually a variant of algorithm W, with respect to this deduction system. We sketch how such a proof can be conducted for a type system formulated as in Figure 2 (the technical details can be found in [6], Sections 4.4 and 4.5).

To prove the type rules sound amounts to showing that every solution that satisfies the constraints is valid with respect to the language semantics. In our particular case we approached this by proving our type system equivalent to the Hindley-Milner type system [13]. Theorem 4.14 and Theorem 4.15 of [6] give a detailed proof of the equivalence. If there is no other analysis to relate to, a soundness proof must be constructed from scratch. In such a proof, the ordering combinators play no role of importance, so they do not add to the complexity of the proof.

On the other hand, in the correctness proof for the algorithm (always with respect to the type system) the ordering combinators play a large role. For that we need to consider every possible constraint list that results from flattening with any *TreeWalk* that adheres to the restrictions imposed by the $\ll$ combinators. If for each such

constraint list, a *particular* solver returns an answer (okay or error) consistent with the type system, then that particular combination of ordering combinators and solver is sound and complete with respect to the type system. Obviously, the implementations may differ in other operational aspects, e.g., how many constraints were considered before discovering that the constraint set was found to be inconsistent, and which constraint was under consideration when the inconsistency was detected.

In our result, Theorem 4.16 of [6], the correctness of the implementation essentially depends on the interplay between the solver and the use of $\ll$ in our type rules. As explained before, the combinators ensure that generalization constraints and instantiation constraints are guaranteed to be ordered in such a way that the solver does not need to verify the side conditions for solving these constraints (see Section 3). On the one hand, the proof of the correctness result seems more complicated due to the fact that all possible valid orderings must be considered. On the other hand, we have found that such a proof can be more elementary in the sense that abstracting away from a particular implementation such as algorithm W, avoids polluting the proof with particularities of the implementation and focuses on the essence, in our case the interplay between the solver and the use of the $\ll$ combinator.

8 Related work

We are not aware of any work having been done that uses a separate language of ordering combinators as we have done, neither for the type inferencing the polymorphic lambda-calculus nor for other analyses and languages.

We are not the first to consider a more flexible approach in solving constraints. Algorithm $\mathcal{G}$ [10], presented by Lee and Yi, can be instantiated with different parameters, yielding the well-known algorithms W and M (and many others) as instances. Their algorithm essentially allows to consider certain constraints earlier in the type inference process. Our constraint-based approach has a number of advantages: 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 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 extend this to algorithms that remove the left-to-right bias [21, 12].

Sulzmann presents constraint propagation policies [18] for modeling W and M in the HM(X) framework [19]. First, general type rules are formulated that mention partial solutions of the constraint problem: later, these rules are specialized to obtain W and 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 [19]), or an unordered set of constraints is collected (e.g., Pierce's first textbook on type systems [15]). 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 [16]. 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 that 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 [16]). Our combinators could help here to decouple the traversal of the constraint from the constraint semantics.

9 Conclusion and future work

In this paper we have advocated the introduction of a separate constraint ordering phase between the phase that generates the constraints and the phase that solves constraints. We have presented a number of combinators that can be used in the type rules to specify restrictions and, contrarily, degrees of freedom on the order in which constraints may be solved. The freedom can be used to influence the order in which constraints are solved in order to control the decision which constraint will be blamed for an inconsistency, and ultimately, what type error message may result. The restrictions can be used to simplify the solver, so that side conditions do not need to be checked. This may also simplify proofs of correctness, which should follow from the interplay between the use of ordering combinators and the solver. Note that these proofs of soundness should consider all possible solving orders allowed by the ordering combinators.

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 benefits of our work are that choices about the best order to solve constraints can be made much later in the development of the compiler, or not at all, in which case the choice can be made by the programmer who uses the compiler. Different orderings typically yield different error messages, and in this way the programmer can consider alternative views on the inconsistency to discover what is really wrong. The framework is very general and can be applied to other analyses and other programming languages with little effort. We also note that our combinators can play a large role in the development of domain specific languages for specifying executable program analyses, such as envisioned in systems like TinkerType [11] and Ruler [2].

The combinators we described are 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. 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. L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.
2. A. Dijkstra and S. D. Swierstra. Ruler: Programming type rules. In *FLOPS*, pages 30 – 46, 2006.

3. GHC Team. *The Glasgow Haskell Compiler*. `http://www.haskell.org/ghc`.
4. J. Hage and B. Heeren. Ordering type constraints: A structured approach. Technical Report UU-CS-2005-016, Department of Information and Computing Science, Utrecht University, Netherlands, April 2005. Technical Report.
5. J. Hage and B. Heeren. Heuristics for type error discovery and recovery. In Z. Horváth, V. Zsók, and A. Butterfield, editors, *Implementation of Functional Languages – IFL 2006*, volume 4449, pages 199 – 216, Heidelberg, 2007. Springer Verlag.
6. B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, The Netherlands, 2005. `http://www.cs.uu.nl/people/bastiaan/phdthesis`.
7. B. Heeren, D. Leijen, and A. van IJzendoorn. Helium, for learning Haskell. In *ACM Sigplan 2003 Haskell Workshop*, pages 62 – 71, New York, 2003. ACM Press. `http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome`.
8. M. P. Jones et al. *The Hugs 98 system*. OGI and Yale, `http://www.haskell.org/hugs`.
9. O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 20(4):707–723, July 1998.
10. O. Lee and K. Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, Research on Program Analysis System, Korea Advanced Institute of Science and Technology, March 2000.
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 K. Hammond, A. J. T. Davie, and C. Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.
13. R. Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.
14. F. Nielson, H.R. Nielson, and C. Hankin. *Principles of Program Analysis*. Springer Verlag, second printing edition, 2005.
15. B. C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.
16. 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.
17. J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
18. M. Sulzmann. A general type inference framework for Hindley/Milner style systems. In *FLOPS*, pages 248–263, 2001.
19. M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types. Research Report YALEU/DCS/RR-1129, Yale University, Department of Computer Science, April 1997.
20. D. M. Volpano and G. Smith. A type-based approach to program security. In *TAPSOFT '97: Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development*, pages 607–621, London, UK, 1997. Springer-Verlag.
21. 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.