Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 236 (2009) 163–183
www.elsevier.com/locate/entcs

Strategies for Solving Constraints in Type and Effect Systems

Jurriaan Hage[1]

Department of Information and Computing Sciences
Universiteit Utrecht P.O.Box 80.089
3508 TB Utrecht, The Netherlands

Bastiaan Heeren[2]

Faculty of Computer Science
Open Universiteit Nederland

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.

Keywords: type and effect systems, inference algorithms, constraints, solving strategies

1 Introduction

Volpano and Smith [23] 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.

[1] Email: jur@cs.uu.nl
[2] Email: Bastiaan.Heeren@ou.nl

doi:10.1016/j.entcs.2009.03.021

To provide good feedback one 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. [6] which does exactly that for the domain of Haskell programming. In this case, the user has knowledge only of the program from which the inconsistent set of constraints was derived. Error messages and hints must therefore always be phrased in terms to the source program. Moreover, a mistake made in a Haskell program might be corrected with a small modification on the part of the user, although such a small correction can change the set of constraints in a significant way.

In the constraint programming community there is quite a large body of work devoted to explaining inconsistencies in sets of constraints, cf. [17]. In contrast with our situation, the user is typically given a set of constraints, and needs to, often interactively, find a valuation for the variables that make it consistent, and delete some constraints in case of over-constrainedness. Moreover, the work tends to be of a general nature, applying to different domain areas. This makes the results more generally useful, but also less tailored to our particular domain.

Phrasing and implementing domain-aware heuristics 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 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. Furthermore, while a compiler is being built, it is usually not known what a good solving strategy might be, so we would like to avoid making decisions that determine the solving strategy, and indirectly the feedback provided by the compiler, until it has been completed and experimented with.

Finally, arguably most importantly, the implementation of a type and effect based analysis is quite a bit of work by itself, and, as experience with our students of program analysis shows, it is quite easy to make mistakes. Providing compiler builders with a library in which they can declaratively phrase restraints on the constraint solving order without (1) needing to encode this directly in a low-level implementation, and (2) allowing as much freedom besides, both improves the productivity of the compiler builder and the flexibility of the resulting tool.

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 [23]. There is nothing in our development, however, that makes assumptions about the analysis or the programming language involved. The work is implemented as part of the Top framework [4] and has been used in the construction of a real-life compiler, the Helium compiler for Haskell [8].

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 [15,1] 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 (Section 6). We compare the efficiency with the usual approach of hardcoding the traversal in Section 7, and in Section 8 give a sketch of a proof of soundness. 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 [16] 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. Furthermore, the way the abstract syntax tree of the program is traversed is fixed once and for all, and this determines, for programs that are not type correct, which unification is going to fail, and indirectly influences the error message that results.

For example, consider the type incorrect Haskell program:

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

The inconsistency in this program is caused by the fact that an integer 1 and a boolean *True* are passed to the same function f. Since f is a lambda-bound identifier, its type is monomorphic, and can't handle inputs of different types.

The Hugs Haskell interpreter [9] generates the following error message

```
ERROR "Example2.hs":3 - Type error in application
*** Expression     : f True
*** Term           : True
*** Type           : Bool
*** Does not match : Int
```

The industry strength Haskell compiler GHC [3], on the other hand, provides us with the following.

```
Example2.hs:3:46:
  Couldn't match expected type 'Bool' against inferred type 'Int'
    Expected type: Bool -> t
    Inferred type: Int -> t
  In the expression: if b then (f 1) else (f True)
  In the expression: \ b -> if b then (f 1) else (f True)
```

In short, Hugs puts the blame on the argument to *True*, while GHC puts the

blame on the type of f. The difference between what is reported as the source of the problem is due to a different strategy in solving constraints: Hugs uses a bottom-up approach to solving constraints: first the subexpressions, then the whole expression, while GHC proceeds in a top-down fashion. Objectively, there is no reason to prefer one over the other; it is largely a matter of taste on the part of the programmer. The main advantage of our approach is that with little effort both (and many more) strategies can be provided in a single implementation and that the programmer can himself choose which strategy suits him best. If the compiler by the nature of its implementation easily allows different constraint solving orders, we can 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 now discuss the main lines of our approach. Consider any type and effect system, say Security Analysis [23]. 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. Put differently, a program analysis is performed by interpreting a program written in a constraint language. In that case, the specification describes the mapping from the source program to the constraint language, and the solver is an interpreter for that language.

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 before certain others.

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 [15], and we assume the reader has some familiarity with this type system and the associated inference algorithm [1], see also Chapter 22 of Pierce's textbook [18] which shares our preference for specifying the analysis in terms of constraints. 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 $[\overline{a} := \overline{\tau}]$.

A type can be generalized to a type scheme, while excluding from the generalization the free type variables of some set $\mathcal{M}$; these remain monomorphic. Dually, we instantiate a type scheme by replacing the bound type variables with fresh type variables:

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

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 [19]: 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,

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

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

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

For a constraint set $\mathcal{C}$, we start with the solution $(\mathcal{C}, id, id)$ 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.

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

where the standard algorithm *mgu* is used to find a most general unifier of two types [20] and the function *actives* computes the set of type variables that may still change whilst solving $\mathcal{C}$:

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

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.

We can now formulate the following result that states that applying the above reduction rules yields the most general solution of a set of constraints.

Theorem 3.1 *If* $(\mathcal{C}, id, id) \rightarrow^* (\emptyset, S, \Sigma)$*, then* $(S, \Sigma) \vdash_s \mathcal{C}$*. In fact,* (S, Σ) *is the most general solution that satisfies* $\mathcal{C}$*.*

Proof. The proof is similar to that of Theorem 4.9 in [7]. Note that the implicit instance constraints in that proof can easily be mapped to a pair of generalization and instance constraints. □

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 [15] formulated in terms of constraints. Such a type system is the basis of a type and effect based analysis, e.g., Security Analysis [23], 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.

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

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 shall return to this issue in later sections.

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

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

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

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

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

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

Fig. 1. Type rules for a simple expression language

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 [16]). 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 given fact is checked. We think the improved control offsets the slight decrease in efficiency due to having slightly more constraints. 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 for building the various kinds of constraint trees. In the remainder of this section, we explain Figure 2 in more detail.

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.

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

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. Choosing the former 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 $\mathcal{T}_{c1}$

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

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

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

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

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

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

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

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, because it must essentially search the set for constraints that may be solved. 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. Our definitions take the form of Haskell programs, i.e., they are executable formal specifications.

5.1 Flattening a Constraint Tree

Our first concern is how to convert a constraint tree into a list, which is then to be fed to a constraint 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* which takes a tree walk (representing the ordering strategy), and a constraint tree. It returns the list of constraints.

```
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, [])], [])
```

A *TreeWalk* is a type synonym for

$$\forall\ a.[a] \to [([a], [a])] \to [a].$$

The first argument $[a]$ corresponds to the constraints belonging to the node itself, the second argument $[([a],[a])]$ contains pairs of lists of constraints, one for each child. 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.

Conceptually, 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.

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

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:

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

Example 5.1 Applying our two tree walks to $t = c_3 \diamond \{\!|\, c_1 \triangledown \mathcal{C}_1^{\bullet}, c_2 \triangledown \mathcal{C}_2^{\bullet} \,|\!\}$ 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.

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

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 5.1 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.

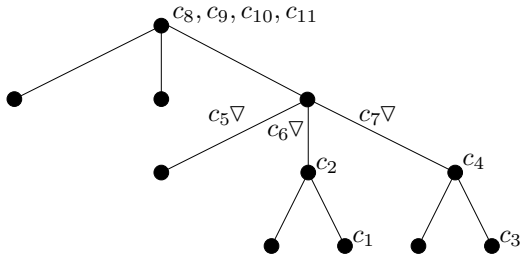

Fig. 3. The constraint tree

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. This combinator essentially doubles the number of traversals!

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

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

Example 5.2 We generate constraints for the expression e from Section 2 repeated 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.

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

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 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 [9],

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

Fig. 4. The constraints

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 $\mathcal{G}$ [12], 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. Note that if our own type system used type environments, we would still need something like the dual of spreading to emulate the assumption set approach.

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 they bind.

First we adapt *flatten* to handle the new cases:

```
flattenRec down tree =
  case tree of
    ...              → ...
    (ℓ, c) ▽° t      → flattenRec down (c ◇ t)
    (ℓ, c) ≪° t      → flattenRec down ([c]• ≪ t)
    ℓ°               → flattenRec down ⦃ ⦄
```

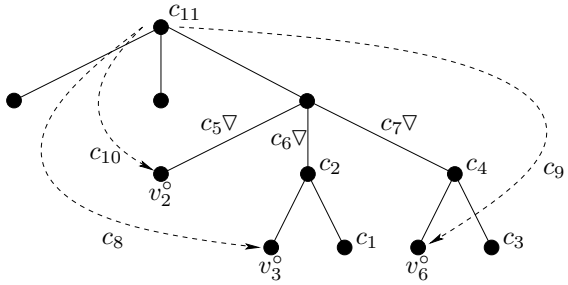

Fig. 5. Constraint tree with type constraints that have been spread

It may surprise the reader to see that the behaviour of *flatten* is to ignore the spreading annotation, e.g., $\ll^\circ$ will be interpreted as $\ll$. This is because *flatten* does not apply the spreading, but is responsible for moving constraints deeper into the tree, until they end up at their destination label. The output of *spread*, in which some constraints may have been moved from binding site to use site, can then be used as input to *flatten*.

$$
\begin{array}{l}
\mathit{spread} :: \mathit{ConstraintTree} \to \mathit{ConstraintTree} \\
\mathit{spread} = \mathit{spreadRec}\ [\,] \\
\quad \textbf{where} \\
\qquad \mathit{spreadRec} :: [(\mathit{Label}, \mathit{Constraint})] \to \mathit{ConstraintTree} \to \mathit{ConstraintTree} \\
\qquad \mathit{spreadRec}\ \mathit{list}\ \mathit{tree} = \\
\qquad\quad \textbf{case}\ \mathit{tree}\ \textbf{of} \\
\qquad\qquad \{\!\bullet\ t_1, \ldots, t_n\ \bullet\!\} \to \{\!\bullet\ \mathit{map}\ (\mathit{spreadRec}\ \mathit{list})\ [t_1, \ldots, t_n]\ \bullet\!\} \\
\qquad\qquad c \diamond t \to c \diamond \mathit{spreadRec}\ \mathit{list}\ t \\
\qquad\qquad c \triangledown t \to c \triangledown \mathit{spreadRec}\ \mathit{list}\ t \\
\qquad\qquad t_1 \ll t_2 \to \mathit{spreadRec}\ \mathit{list}\ t_1 \ll \mathit{spreadRec}\ \mathit{list}\ t_2 \\
\qquad\qquad (\ell, c) \triangledown^\circ t \to \mathit{spreadRec}\ ((\ell, c) : \mathit{list})\ t \\
\qquad\qquad (\ell, c) \ll^\circ t \to \mathit{spreadRec}\ ((\ell, c) : \mathit{list})\ t \\
\qquad\qquad \ell^\circ \to [\,c \mid (\ell', c) \leftarrow \mathit{list}, \ell == \ell'\,]^\bullet
\end{array}
$$

The output of *spread* can be passed on to *flatten* to compute the final list of constraints.

Example 5.3 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.

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

The *bottomUp* tree walk after spreading leads to reporting the constraint c_4: without

spreading type constraints, c_9 is reported.

6 Emulating Existing Algorithms

To further illustrate the flexibility of the 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 [10], 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 [8] 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 [9] 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 [12], and Algorithm $\mathcal{U}_{\text{AE}}$ by Jun et al. [24], are considered in a technical report [5].

7 Efficiency

In this section we shortly address the question how strongly the use of our library impacts performance when compared with hard-coding the traversal as in algorithm W. But what do we base our comparison on? Is it running-time, space consumption, or the number of constraints we have considered? And do we distinguish between well-typed and type incorrect programs?

First of all, we typically generate more, but smaller constraints: essentially what we have done is to decompose complicated unifications into the smallest unifications possible. This induces some memory overhead, but it is small. In terms of time, there can be little difference since for a type correct program we make exactly the

same comparisons. For a type incorrect program, Lee and Yi proved that algorithm M considers the least number of constraints before finding an error, and algorithm W the most [11]. They actually advocate using something in the middle: algorithm M complains too soon (and therefore gives little context in its error message) and algorithm W too late. In practice, however, the differences are hardly noticeable, and we believe that spending a bit more time to come up with a better judgement of the problem easily outweighs the extra expense. Note also that in our particular situation, the programmer can choose the order he prefers, and therefore either choose a slower or faster solving order. This is an additional, albeit somewhat accidental, feature of our work.

We use our framework in the Helium compiler and experience shows that constraint generation and re-ordering only take up a small amount of time. The compiler is not much faster or slower than other compilers for Haskell. It is certainly fast enough for interactive program development with an IDE.

8 Soundness

A soundness proof for a program analysis typically has two parts: first prove the logical deduction system sound with respect to the semantics of the programming language, and then prove the correctness of the algorithm with respect to this deduction system. The soundness of the algorithm then follows by transitivity. In this section we sketch the main steps in conducting such a proof for the type system we have described above. At the end of this section we reflect back on this and indicate how one would proceed in a slightly different but maybe more usual scenario.

The main theorem can be formulated as follows:

Theorem 8.1 *Let $\mathcal{M}, \mathcal{A}, \mathcal{T}_c \vdash e : \tau$ for a closed expression e, and let $\mathcal{C}$ be a list of constraints obtained by flattening $\mathcal{T}_c$. If $(\mathcal{C}, id, id) \rightarrow^* (\emptyset, S, \Sigma)$, then $\emptyset \vdash_{\text{HM}} e : S\tau$, where $\vdash_{\text{HM}}$ denotes the Hindley-Milner type system.*

The proof is similar to that of Theorem 4.16 and its subsidiary results as proved in [7], again with the note that implicit instance constraints can be replaced by separate generalization and instantiation constraints.

Some details of the structure of the full proof of Theorem 4.16 are in order: instead of proving our deduction system, Figure 1, correct with respect to the semantics of language, we have proved it sound and complete with respect to the (sound) Hindley-Milner type system instead [15]. Because our system uses assumption sets and the Hindley-Milner type system uses type environments, we need a few technical lemmas (Lemma 4.11 and 4.12) and Theorem 4.13 to relate the standard Hindley-Milner type system to a slightly modified version that is easier to relate to ours. Then Theorem 4.14 and 4.15 prove the soundness and completeness of Figure 1 with respect to the slightly modified version. This takes care of the logic side of things.

For the algorithmic part we need to show that our algorithm, which (1) generates

a constraint tree, (2) uses an arbitrary treewalk to flatten the tree into a list of constraints and then (3) proceeds to solve these in the order in which they are listed, returns the most general solution for the constraints in the constraint tree. The crucial realization is now that our choice to solve the constraints from a let definition before those of the corresponding let body, ensures that we never block on the side conditions of the rules for the generalization and instantiation constraints. This ensures that we obtain the most general solution if we solve the constraints in the listed order and justifies leaving out those side conditions from the solver, as long as we use a solving order that follows from a valid interpretation of the combinators. Soundness then follows from Theorem 3.1.

In this paper we considered an existing analysis for which a soundness proof was already known. A distinct advantage is that we do not need to provide a semantics of the language. A disadvantage, in our particular situation, is that because our type system is based on assumption sets we needed to do some extra work (about one page) to prove equivalence with the type environment based system. However, it seems that this construction is quite generic and doing the same for a different analysis is not likely to be much different. Proving equivalence with the slightly modified system (Theorems 4.14 and 4.15) takes about two pages.

If there is no other analysis to relate to, a soundness proof must be constructed from scratch. Note that in such a proof, the ordering combinators play no role of importance, so they do not add to the complexity of the proof. Since such a proof needs to conducted anyway, we are no worse off than without the combinators.

Although we have never made the effort, we believe that for a proof from scratch it does not matter whether an assumption set or type environment based formulation is used.

As a final reflective note, the proof of the correctness result seems more complicated due to the fact that all possible flattenings must be considered. On the other hand, such a proof can be more elementary in the sense that abstracting away from a particular fixed order, 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.

9 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 $\mathcal{G}$ [12], 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 [24,14].

Sulzmann presents constraint propagation policies [21] for modeling W and M in the HM(X) framework [22]. 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 [22]), or an unordered set of constraints is collected (e.g., Pierce's first textbook on type systems [18]). 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 [19]. 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 [19]). Our combinators could help here to decouple the traversal of the constraint from the constraint semantics.

10 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. 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 [13] 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.

In future work, we plan to completely phrase (a suitable variation) of Volpano and Smith's Security Analysis in terms of our combinators. The current paper thus far only addresses the specification of the underlying type system. In that case, we shall choose a variant of the type system that uses type environments instead of assumption sets. As always, the main goal will twofold: to obtain flexibility in the order of solving constraints, and to make the transition from specification of the analysis to implementation as painless as possible.

Acknowledgement

The authors thank the anonymous referees for their constructive comments.

References

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

[2] Dijkstra, A. and S. D. Swierstra, *Ruler: Programming Type Rules*, in: *FLOPS*, 2006, pp. 30–46.

[3] GHC Team, "The Glasgow Haskell Compiler," http://www.haskell.org/ghc.

[4] Hage, J., *Top project website*, http://www.cs.uu.nl/wiki/Top.

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

[6] Hage, J. 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*, Lecture Notes in Computer Science **4449** (2007), pp. 199 – 216.

[7] Heeren, B., "Top Quality Type Error Messages," Ph.D. thesis, Universiteit Utrecht, The Netherlands (2005), http://www.cs.uu.nl/people/bastiaan/phdthesis.

[8] Heeren, B., D. Leijen and A. van IJzendoorn, *Helium, for learning Haskell*, in: *ACM Sigplan 2003 Haskell Workshop* (2003), pp. 62–71, http://www.cs.uu.nl/wiki/bin/view/Helium/WebHome.

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

[10] Lee, O. and K. Yi, *Proofs about a folklore let-polymorphic type inference algorithm*, ACM Transactions on Programming Languages and Systems **20** (1998), pp. 707–723.

[11] Lee, O. and K. Yi, *A Generalization of Hybrid Let-Polymorphic Type Inference Algorithms*, in: *Proceedings of the First Asian Workshop on Programming Languages and Systems*, National university of Singapore, Singapore, 2000, pp. 79–88.

[12] Lee, O. 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 (2000).

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

[14] McAdam, B. J., *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*, Lecture Notes in Computer Science **1595** (1998), pp. 139–154.

[15] Milner, R., *A theory of type polymorphism in programming*, Journal of Computer and System Sciences **17** (1978), pp. 348–375.

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

[17] O'Callaghan, B., B. O'Sullivan and E. C. Freuder, *Generating Corrective Explanations for Interactive Constraint Satisfaction*, in: P. van Beek, editor, *Principles and Practice of Constraint Programming (CP'05)*, Lecture Notes in Computer Science **3709** (2005), pp. 445–459.

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

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

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

[21] Sulzmann, M., *A general type inference framework for Hindley/Milner style systems.*, in: *FLOPS*, 2001, pp. 248–263.

[22] Sulzmann, M., M. Odersky and M. Wehr, *Type inference with constrained types*, Research Report YALEU/DCS/RR-1129, Yale University, Department of Computer Science (1997).

[23] Volpano, D. M. and G. Smith, *A Type-Based Approach to Program Security*, in: *Proceedings of the 7th International Joint Conference CAAP/FASE on Theory and Practice of Software Development TAPSOFT'97*, Lecture Notes in Computer Science **1214** (1997), pp. 607–621.

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