Constraint Based Type Inferencing in Helium

Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra
{bastiaan,jur,doaitse}@cs.uu.nl

Institute of Information and Computing Science, Universiteit Utrecht,
P.O.Box 80.089, 3508 TB Utrecht, Netherlands

Abstract. The Helium compiler implements a significant subset of the functional programming language Haskell. One of the major motivations for developing it was to yield understandable and appropriate type error messages. The separation between the generation, the ordering and the solving of constraints on types has led to a flexible framework which has been employed successfully in a classroom setting. Among its many advantages are the possibility to plug in heuristics for deciding the most likely source of a type error, the generation of multiple type error messages during a single compilation, and the possibility for programmers to tune the type inference process and resulting messages to their own needs without having to know any of the details of the implementation.

1 Introduction

At Universiteit Utrecht, the `Helium` compiler was developed with a special focus on generating understandable error messages for novice functional programmers [ILH]. The compiler implements a large subset of `Haskell`, the most serious omission being that of type classes, an omission which is currently being dealt with. The compiler has been succesfully employed in an educational setting and the feedback obtained from the students is very positive.

The generation of "good" type error messages for higher-order functional programming languages such as `Haskell` is a lively field, mainly because novice users have a large problem understanding those generated by most exisiting compilers [HW03,Chi01,YMT00,Sul,McA00]. However, too little of this research has ended up in prominent compilers and interpreters available for the language. In that sense, external tools have an advantage, because they do not have to be built into a compiler.

Every programmer has his own style of programming, like top-down, bottom-up or something in between. A programmer should be able to choose the style of type inferencing which suits him best, based on his style of programming. Therefore, we think that there is not a single deterministic type inference process which works best in all cases for everybody, and advocate the use of a more flexible system that can be tuned by the programmer. To make this workable, some mechanism should be present to make this possible without having to delve into the innards of the compiler. A drawback of existing compilers is that they are rigid, i.e., it is not possible for a programmer to change the way the type

inference process works, unless he is willing to inspect and modify the compiler itself; given the complexity of compilers, this is usually a very difficult task.

The flexibility we sought for has been obtained in the implementation of the type inference process of `Helium` by taking a constraint based approach dividing the process into three distinct phases:

1. the generation of constraints in the abstract syntax tree,
2. the ordering of constraints in the tree into a list, and
3. the solving of constraints.

For the latter phases, multiple instantiations are possible. This separation has resulted in a flexible framework, which has been relatively easy to implement using an attribute grammar system – an abstraction layer on top of `Haskell` –, developed by Swierstra et al. [SBL]. In fact, we plan to show that the translation of the type rules to an attribute grammar is quite straightforward.

The generality of our framework also allows the emulation of well-known type inference algorithms such as $\mathcal{W}$ [DM82] and $\mathcal{M}$ [LY98], by choosing an appropriate order in which the type constraints should be solved. This makes it easy to compare these standard algorithms with type inference processes which use global heuristics and elaborate data structures such as type graphs (see Section 4.3). In this way we can gain insight into the computational penalties to be paid, and it has the additional benefit that the standard ways of type inferencing are available to those who have grown used to it.

The paper is organized as follows. After a short tutorial on type inferencing, and introducing the sorts of constraints we need, we give the type inference rules for our language in Section 3. After showing how these can be transformed into an exectuable specification in the attribute grammar system in Section 4.1, we consider ways of ordering the constraints in Section 4.2. Section 4.3 considers various types of solvers, greedy ones which can act similar to algorithms like $\mathcal{W}$ and $\mathcal{M}$, and a more global one which tries to find a minimal set of errors based on a global analysis of a type graph. We conclude in Section 5 by shortly discussing type inference directives described in [HHS03]. This facility gives programmers, and more specifically the developers of combinator libraries, the tool to control the behaviour of the compiler by specifying how the type inference process should behave on certain sets of expressions, and a way to provide tailormade error messages for the combinator language. This, we feel, is an important innovation, that has become feasible through the underlying constraint based approach.

2 Preliminaries

Types and substitutions

Many existing programming language have a type system,

> a tractable syntactic method for proving the absence of certain program behaviors by classifying phrases according to the kinds of values they compute. Pierce [Pie02]

Reasons for having a type system are many: source documentation, generating more efficient code, and finding program errors early. Tractability is essential for the usefulness of type systems within a compiler. Usually this is obtained by ensuring that the method can be performed in a syntax directed manner, i.e., determined by the structure of the abstract syntax tree. In fact, we generate constraints from our program in such a manner.

In many programming languages with such a type system, it is necessary that the programmer annotates identifiers and procedures or methods with their types. It is then the task of the compiler to verify that these types are consistent and do not lead to run-time errors. The field which addresses this kind of problem is called type checking.

In the case of a language like `Haskell` and `ML`, many of these annotations can be dropped since Damas and Milner showed how to inference polymorphic let expressions [DM82]. For some language constructs, polymorphic lambda abstractions for instance, `Haskell` needs externally supplied types to make the type inferencing process decidable. For more information, the reader might consult Chapter 22 and 23 of Pierce [Pie02] for its constraint based development of a type system for a polymorphic functional language.

The syntax of *types* and *type schemes* is given by:

(type)	$\tau := \alpha \quad \mid \quad T\ \tau_1\ \ldots\ \tau_n$	where $\mathit{arity}(T) = n$
(type scheme)	$\sigma := \forall \vec{\alpha}.\tau$	

A type can be either a type variable or a type constructor applied to a number of types. The arity of each type constructor is fixed. Typical examples are $\rightarrow$ *Int Bool*, and $\rightarrow a\ (\rightarrow b\ a)$, the function space constructor being a binary type constructor. In the following, we use the standard infix notation $\tau_1 \rightarrow \tau_2$ for function types, and a special notation for list and product types. The set of type constructors can be extended with user defined data types such as `Maybe`.

A type scheme $\forall \vec{\alpha}.\tau$ is a type in which a number of type variables $\vec{\alpha} = \alpha_1, \ldots, \alpha_n$, the *polymorphic* type variables, are bound to a universal quantifier. The free type variables are called *monomorphic*. Note that n may be zero, in which case a type scheme is simply a type. Although the type variables have an implicit order in any given type scheme, the order itself is not important. For this reason we may view the vector $\vec{\alpha}$ as a set when the need arises.

The set of *free type variables* of a type τ is denoted by $\mathit{ftv}(\tau)$ and simply consists of all type variables in τ. Additionally, $\mathit{ftv}(\forall \vec{\alpha}.\tau) = \mathit{ftv}(\tau) - \vec{\alpha}$, where we use $-$ to denote set difference.

A substitution, usually denoted by $\mathcal{S}$, is a mapping of type variables to types. For type variables $D = \{\alpha_1, \ldots, \alpha_n\}$ and types $\tau_1, \ldots, \tau_n$ the substitution mapping α_i to τ_i is denoted by $[\alpha_1 := \tau_1, \ldots, \alpha_n := \tau_n]$. Implicitly we assume all type variables not in D are mapped to themselves. As usual, a substitution only replaces free type variables, so the quantified type variables in a type scheme are not affected by a substitution.

Generalizing a type τ with respect to a set of type variables M entails the quantification of the type variables in τ that do not occur in M.

$$\mathit{generalize}(M, \tau) \quad =_{\text{def}} \quad \forall \vec{\alpha}.\tau \qquad \text{where } \vec{\alpha} = \mathit{ftv}(\tau) - M$$

An instantiation of a type scheme is obtained by replacing the quantified type variables with fresh type variables.

$$\mathit{instantiate}(\forall \alpha_1 \ldots \alpha_n.\tau) \quad =_{\text{def}} \quad [\alpha_1 := \beta_1, \ldots, \alpha_n := \beta_n]\tau \quad \text{where } \beta_1, \ldots, \beta_n \text{ are fresh}$$

A type τ_1 is a *generic instance* of a type scheme $\sigma = \forall \vec{\alpha}.\tau_2$, denoted $\tau_1 \prec \sigma$, if there exists a substitution $\mathcal{S}$ with $\{\beta \mid \beta \neq \mathcal{S}(\beta)\} \subseteq \vec{\alpha}$ such that $\tau_1 = \mathcal{S}\tau_2$.

In the following we shall often encounter (finite) sets of pairs of the form $x : \tau$, where usually x is a variable and τ a type. For such a set X we define $\mathit{dom}(X) = \{x \mid x{:}\tau \in X\}$ and $\mathit{ran}(X) = \{\tau \mid x{:}\tau \in X\}$.

Constraints

A constraint set, usually denoted by $\mathcal{C}$, is a set of type constraints. We introduce three forms of type constraint:

$$\text{(constraint)} \qquad C := \tau_1 \equiv \tau_2 \quad | \quad \tau_1 \leq_M \tau_2 \quad | \quad \tau \preceq \sigma$$

An equality constraint ($\tau_1 \equiv \tau_2$) reflects that τ_1 and τ_2 should be unified at a later stage of the type inference process. The other two kinds of constraints are used to cope with the polymorphism introduced by let expressions. An explicit instance constraint ($\tau \preceq \sigma$) states that τ has to be a generic instance of σ. This constraint is convenient if we know the type scheme of a variable before we start inferencing its definition; this occurs, for instance, when an explicit type for the variable was given. In general, the (polymorphic) type of a variable introduced by a let expression is unknown and must be inferred before it can be instantiated. To overcome this problem we introduce an implicit instance constraint ($\tau_1 \leq_M \tau_2$), which expresses that τ_1 should be an instance of the type scheme that is obtained by generalizing type τ_2 with respect to the set of monomorphic type variables M, i.e., quantifying over the other type variables, $\mathit{ftv}(\tau_2) - M$. We consider the necessity of the implicit instance constraint when we consider the type rule of the let. Equality constraints on types can be lifted to sets of pairs of types by $(X \equiv Y) = \{\tau_1 \equiv \tau_2 \mid x : \tau_1 \in X,\ x : \tau_2 \in Y\}$ and similarly for $\leq_M$ and $\preceq$.

Once a constraint set has been generated, we can look for the minimal substitution that satisfies each constraint in the set. In the compiler, when we encounter a constraint that cannot be satisfied (or, in general, raises a conflict), we simply dismiss the constraint, and use it to generate an error message instead. In much of the literature, the unification of two types which cannot be unified results in the error substitution $\top$. Needless to say, such a substitution never arises in our case.

Satisfaction of a constraint by a substitution $\mathcal{S}$ is defined as follows:

$$\begin{array}{lll} \mathcal{S} \text{ satisfies } (\tau_1 \equiv \tau_2) & =_{\text{def}} & \mathcal{S}\tau_1 = \mathcal{S}\tau_2 \\ \mathcal{S} \text{ satisfies } (\tau_1 \leq_M \tau_2) & =_{\text{def}} & \mathcal{S}\tau_1 \prec \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau_2) \\ \mathcal{S} \text{ satisfies } (\tau \preceq \sigma) & =_{\text{def}} & \mathcal{S}\tau \prec \mathcal{S}\sigma \end{array}$$

After substitution, the two types of an equality constraint should be syntactically the same. The logical choice for $\mathcal{S}$ in this case is the most general unifier of the two types. For an implicit instance constraint, the substitution is not only applied to both types, but also to the set of monomorphic type variables M. The substitution is applied to the type and the type scheme of an explicit instance constraint, where the quantified type variables of the type scheme are, as usual, untouched by the substitution. The distinction between the explicit and implicit instance constraint is that type variables in the (right hand side of the) latter can still become monomorphic. This information is known at the outset for explicit instance constraints. We illustrate this by an example.

Example 1. Let $c = \alpha_3 \leq_{\{\alpha_5\}} \alpha_1 \to \alpha_2$ be a constraint. We want a substitution $\mathcal{S}$ that satisfies c, i.e., $\mathcal{S}\alpha_3 \prec \mathit{generalize}(\mathcal{S}\alpha_5, \mathcal{S}(\alpha_1 \to \alpha_2))$ where the latter is equal to $\forall\beta_1\beta_2.\beta_1 \to \beta_2$, if α_1, α_2 and α_5 are not touched by $\mathcal{S}$. A most general substitution to satisfy this constraint is $\mathcal{S} = [\alpha_3 := \alpha \to \beta]$, where α and β are fresh type variables.

But what happens if we later encounter $c' = \alpha_5 \equiv \alpha_1$? We have already chosen α_1 to be polymorphic in c, although the constraint c' now tells us that α_1 was in fact monomorphic. It seems then that we considered c too early and should have waited until we were sure that α_1 was polymorphic. A safe approach in this case is to infer let definitions without an explicit type signature before considering their occurrences in the body of the let.

To be more precise, an implicit instance constraint $c = \tau_1 \leq_M \tau_2$ can be solved when τ_2 does not contain any active type variables beside those in M. Activeness can be defined as follows.

$$\begin{array}{lll} \mathit{activevars}(\tau_1 \equiv \tau_2) & =_{\text{def}} & \mathit{ftv}(\tau_1) \cup \mathit{ftv}(\tau_2) \\ \mathit{activevars}(\tau_1 \leq_M \tau_2) & =_{\text{def}} & \mathit{ftv}(\tau_1) \cup (\mathit{ftv}(M) \cap \mathit{ftv}(\tau_2)) \\ \mathit{activevars}(\tau \preceq \sigma) & =_{\text{def}} & \mathit{ftv}(\tau) \cup \mathit{ftv}(\sigma) \end{array}$$

The condition for the implicit instance constraint c to be solvable is then that

$$(\mathit{ftv}(\tau_2) - M) \cap \mathit{activevars}(\mathcal{C}) = \emptyset \ . \tag{1}$$

where $\mathit{activevars}(\mathcal{C}) = \bigcup\{\mathit{activevars}(c) \mid c \in \mathcal{C}\}$.

Let $\mathcal{S}$ be a substitution which satisfies the set of constraints $\mathcal{C}$ and was computed without ever violating the condition (1) for an implicit instance constraint. Lemma 3 of [HHS02] proves that solving $\mathcal{C}$ by considering the constraints of $\mathcal{C}$ in any order (again, without ever violating the condition (1)) results in the same substitution $\mathcal{S}$ (up to type variable renaming). Thus, it follows that there is a lot of freedom when choosing an order, only restricted by the side condition (1).

```
expr  = lit | var | constructor
      | expr expr+
      | if expr then expr else expr
      | 'λ' pat+ '→' expr
      | case expr of alts
      | let decls in expr
      | '(' expr' ')' | '[' expr' ']'
      | expr '::' typescheme

pat   = lit | var | constructor pat*
      | '(' pat' ')' | '[' pat' ']'
      | var '@' pat | '_'
alt   = pat '→' expr
decl  = fb | var '::' typescheme
fb    = var pat* '=' rhs
rhs   = expr (where decls)?
```

Fig. 1. Context-free grammar for the subset of `Helium`

As demonstrated in [HHS02] there are sets of constraints that contain only implicit instance constraints for which the condition is not fulfilled. In the next section we shall see that for sets of constraints generated from a `Helium` program this situation never occurs.

3 The bottom-up type inference rules

In this section we give the type inference rules for a large part of `Helium`. It is important to note that the notational similarity with the Hindley-Milner rules is deceptive. The Hindley-Milner rules form a system of logical deduction which formulate, for a lambda calculus enriched with a let construct, what the valid typings of a given expression are. Constructing the principal type for a given expression under these deduction rules is exactly what an algorithm such as $\mathcal{W}$ does. In this and the next section we formulate a non-deterministic algorithm which is a generalization of $\mathcal{W}$, both by the introduction of more programming constructs as well as by being less deterministic.

The rules given below are thus the first part of an *algorithm*. They specify for a given abstract syntax tree what constraints should be constructed at various nodes in the tree. In fact, the rules are close to an attribute grammar, and we show later how they can transformed into executable code in the UU_AG system.

The part of `Helium` we consider is given in Fig. 1 where we have concentrated on expressions and patterns[1]. In addition, we have alternatives (for case expressions), explicit types and where-clauses. The `Helium` language incorporates a few more constructs such as data type declarations, type synonyms, list comprehension, monadic do-notation, guarded function definitions and a module system. For the sake of brevity we omit these. Note that `Helium` uses the same layout rule as `Haskell`, which implies that the semicolons between multiple declarations in a let expression or where clause are not always necessary.

In general, our approach is to label a subexpression (node in the abstract syntax tree) with a fresh type variable (usually called β in the type rules) and

[1] For a non-terminal X we abbreviate $\epsilon \mid (X\ ',')^*X$, a comma separated, possibly empty sequence of X's, by X^{ℓ}. Similarly, X^{s} is equivalent to a semicolon separated sequence of X's: $\mathrm{X}^{s} = \epsilon \mid (X\ ';')^*X$. Also, $X^{?}$ indicates optionality of X.

to generate constraints for the restrictions to be imposed in this node. Often the type for an expression is simply a type variable (during the solving process we discover exactly what type it represents), although in some cases it is advantageous to return a type expression containing a sequence of new variables. A sequence of such kind is denoted $\langle \beta_1, \ldots, \beta_n \rangle$ (see for instance the [FB] type rule in Fig. 4).

The judgements in the type rules for expressions are of the form

$$M,\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau\ .$$

Here $\mathcal{C}$ is a set of constraints, e is the expression, τ is the type of e, M is the set of monomorphic type variables and $\mathcal{A}$ is the so-called *assumption set*. An assumption set collects the type variables that are assigned to the free variables of e. Contrary to the standard type environment used in the Hindley-Milner inference rules, there can be multiple (different) assumptions for a given variable. In fact, for every occurrence of a free variable there will be a different pair in $\mathcal{A}$. As can be seen from the expressions for $\mathcal{A}$ and $\mathcal{C}$ in the rules, there is implicitly a flow of information from the bottom up.

The only piece of information that is passed downwards is the set of *monomorphic variables* M. An implicit instance constraint depends on the context of the declaration. Every node in the abstract syntax tree has a set of monomorphic type variables M. For an arbitrary subexpression, the set M contains exactly the type variables that were introduced by lambda abstractions and other monomorphic binding constructs at a higher level in the abstract syntax tree.

Note that the rules allow for flexibility in coping with unbound identifiers. At any point, those identifiers for which we still have assumptions in our assumption set are unbound in the expression. Also, every assumption corresponds with a unique use of this identifier within the expression. We shall now consider some of the type rules in detail.

The type rule $[\mathrm{Lit}]^{\mathrm{e}}_{\mathrm{BU}}$ expresses that for every literal (such as 1, *False*, 'a'...) we generate the constraint $\beta \equiv \tau$, where τ is the fixed type of the literal (such as *Int*, *Bool*, *Char*,...). For a variable we simply generate a fresh type variable, while for constructors we have to instantiate the type scheme for that constructor. We assume that the type of the constructor is already known.

The rule for application is a basic one. If we have types τ and $\tau_1, \ldots, \tau_n$ for the function and the list of n arguments respectively, then we have to impose the constraint that τ is in fact a function type taking arguments $\tau_1, \ldots, \tau_n$ giving us a result of type β (β is as always fresh). The reason we have chosen for n-ary instead of a binary application is to the stay as close as possible to the source code.

A lambda abstraction abstracts over a number of patterns. These patterns contain variables to which the variables in the body of the lambda can be bound. Similar to assumption sets, each occurrence of such a pattern variable is paired with a unique type variable in $\mathcal{B}$. These type variables are then passed along in $\mathcal{M}$, so that the body of the abstraction is informed about which type variables are definitely monomorphic. The constraints generated for the lambda abstraction

$$\frac{\mathit{literal} : \tau}{M,\ \emptyset,\ \{\beta \equiv \tau\} \vdash^{\mathrm{e}}_{\mathrm{BU}} \mathit{literal} : \beta} \quad [\mathrm{Lit}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{}{M,\ \{x : \beta\},\ \emptyset \vdash^{\mathrm{e}}_{\mathrm{BU}} x : \beta} \quad [\mathrm{Var}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{C : \sigma}{M,\ \emptyset,\ \{\beta \preceq \sigma\} \vdash^{\mathrm{e}}_{\mathrm{BU}} C : \beta} \quad [\mathrm{Con}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} f : \tau \qquad M,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{e}}_{\mathrm{BU}} a_i : \tau_i \ \text{ for } 1 \leq i \leq n}{M,\ \mathcal{A} \cup \bigcup_i \mathcal{A}_i,\ \mathcal{C} \cup \bigcup_i \mathcal{C}_i \cup \{\tau \equiv \tau_1 \to \ldots \to \tau_n \to \beta\} \vdash^{\mathrm{e}}_{\mathrm{BU}} f\ a_1\ \ldots\ a_n : \beta} \quad [\mathrm{App}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A}_1,\ \mathcal{C}_1 \vdash^{\mathrm{e}}_{\mathrm{BU}} e_1 : \tau_1 \qquad M,\ \mathcal{A}_2,\ \mathcal{C}_2 \vdash^{\mathrm{e}}_{\mathrm{BU}} e_2 : \tau_2 \qquad M,\ \mathcal{A}_3,\ \mathcal{C}_3 \vdash^{\mathrm{e}}_{\mathrm{BU}} e_3 : \tau_3}{\begin{array}{r} M,\ \mathcal{A}_1 \cup \mathcal{A}_2 \cup \mathcal{A}_3,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}_3 \cup \{\tau_1 \equiv \mathit{Bool},\ \tau_2 \equiv \beta,\ \tau_3 \equiv \beta\} \\ \vdash^{\mathrm{e}}_{\mathrm{BU}} \textbf{if } e_1 \textbf{ then } e_2 \textbf{ else } e_3 : \beta \end{array}} \quad [\mathrm{If}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{\mathcal{B}_i,\ \mathcal{C}_i \vdash^{\mathrm{p}}_{\mathrm{BU}} p_i : \tau_i \ \text{ for } 1 \leq i \leq n \qquad \mathcal{B} = \bigcup_i \mathcal{B}_i \qquad M \cup \mathit{ran}(\mathcal{B}),\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau}{\begin{array}{r} M,\ \mathcal{A} \backslash \mathit{dom}(\mathcal{B}),\ \bigcup_i \mathcal{C}_i \cup \mathcal{C} \cup (\mathcal{B} \equiv \mathcal{A}) \cup \{\beta \equiv \tau_1 \to \ldots \to \tau_n \to \tau\} \\ \vdash^{\mathrm{e}}_{\mathrm{BU}} (\lambda p_1 \ldots p_n \to e) : \beta \end{array}} \quad [\mathrm{Abs}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{\begin{array}{c} M,\ \mathcal{B}_i,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{d}}_{\mathrm{BU}} d_i \ \text{ for } 1 \leq i \leq n \qquad M,\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau \\ (\mathcal{A}', \mathcal{C}') = \mathit{BindingGroupAnalysis}(M, \mathit{explicits}, \{(\emptyset, \mathcal{A}), (\mathcal{B}_1, \mathcal{A}_1), \ldots, (\mathcal{B}_n, \mathcal{A}_n)\}) \end{array}}{M,\ \mathcal{A}',\ \bigcup_i \mathcal{C}_i \cup \mathcal{C} \cup \mathcal{C}' \cup \{\beta \equiv \tau\} \vdash^{\mathrm{e}}_{\mathrm{BU}} \textbf{let } \mathit{explicits}; d_1; \ldots; d_n \textbf{ in } e : \beta} \quad [\mathrm{Let}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau \qquad M,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{a}}_{\mathrm{BU}} a_i : \langle \phi_i, \psi i \rangle \ \text{ for } 1 \leq i \leq n}{\begin{array}{r} M,\ \mathcal{A} \cup \bigcup_i \mathcal{A}_i,\ \mathcal{C} \cup \bigcup_i \mathcal{C}_i \cup (\{\tau, \phi_1, \ldots, \phi_n\} \equiv \{\beta_1\}) \cup (\{\psi_1, \ldots, \psi_n\} \equiv \{\beta_2\}) \\ \vdash^{\mathrm{e}}_{\mathrm{BU}} (\textbf{case } e \textbf{ of } a_1; \ldots; a_n) : \beta_2 \end{array}} \quad [\mathrm{Case}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{e}}_{\mathrm{BU}} e_i : \tau_i \ \text{ for } 1 \leq i \leq n}{M,\ \bigcup_i \mathcal{A}_i,\ \bigcup_i \mathcal{C}_i \cup \{\beta \equiv (\tau_1, \ldots, \tau_n)\} \vdash^{\mathrm{e}}_{\mathrm{BU}} (e_1, \ldots, e_n) : \beta} \quad [\mathrm{Tuple}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{e}}_{\mathrm{BU}} e_i : \tau_i \ \text{ for } 1 \leq i \leq n}{M,\ \bigcup_i \mathcal{A}_i,\ \bigcup_i \mathcal{C}_i \cup (\{\tau_1 \ldots \tau_n\} \equiv \{\beta_1\}) \cup \{\beta_2 \equiv [\beta_1]\} \vdash^{\mathrm{e}}_{\mathrm{BU}} [e_1, \ldots, e_n] : \beta_2} \quad [\mathrm{List}]^{\mathrm{e}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau}{M,\ \mathcal{A},\ \mathcal{C} \cup \{\beta \preceq \sigma,\ \tau \preceq \sigma\} \vdash^{\mathrm{e}}_{\mathrm{BU}} (e :: \sigma) : \beta} \quad [\mathrm{Typed}]^{\mathrm{e}}_{\mathrm{BU}}$$

Fig. 2. The Bottom-Up type inference rules for expressions

itself specify that the type of each identifier is equal to the type of each of its occurrences in the body (expressed by $\mathcal{B} \equiv \mathcal{A}$), and that the resulting type is an appropriate function type.

For $[\text{Let}]^{\text{e}}_{\text{BU}}$ we take into account that the types of some definitions are explicitly given. Essentially, a let expression can be used to introduce an abbreviation. For instance, the let expression `let i = \x -> x in i i` has the same semantics as the unfolded code `(\x -> x)(\x -> x)`. Note that it is essential that the type of both lambda abstractions should have a common form, $\forall a.\ a \to a$, but that the instances of this type may differ, e.g., $b \to b$ for the latter and $(b \to b) \to (b \to b)$ for the former abstraction. It is possible to cope with let expressions without resorting to constraints other than equality constraints, but this has two significant drawbacks:

- The number of constraints increases, in fact it may explode exponentially, because for every occurence of a definition an independent sets of constraints is generated.
- If let definition contains an error, it is much more difficult to discover this fact, because of the independence between the various set of constraints, already mentioned in the previous point.

Since our focus is the generation of good error messages and efficiency is important, we have chosen to introduce special constraints for the let construct (the implicit instance constraints introduced in Section 2).

In `Helium`, the binding groups are determined in part by the given explicit typings. It is the function *BindingGroupAnalysis* that generates the necessary implicit and explicit instance constraints based on the sets of assumptions (the $\mathcal{A}$s) and the variables introduced by the patterns occuring in the left-hand sides (the $\mathcal{B}$s). Obviously, the function also needs to know the set of monomorphic type variables at this point. In our formulation, the body of the let is considered to be just another binding group that does not introduce new variables.

We explain the working of the function *BindingGroupAnalysis* by an example. If we have two mutually recursive definitions in a let expression of the form $f = \ldots g \ldots$ and $g = \ldots f \ldots$, then without explicit types for f or g, the definitions belong to the same binding group and we generate equality constraints between the type variables for the definition of g and every single use in f and g. Consequently, all occurrences of f and g in these definitions are monomorphic. However, if the let expression includes an explicit polymorphic type for f, then f and g are no longer in the same binding group and f may be used polymorphically in g and vice versa. In that case, we generate an explicit instance constraint, based on the explicit type of f, for each use of f in g, and if g is not explicitly typed itself, we generate an implicit instance constraint for every use of g in f. The same applies to uses of f in the body of f. In other words, if an explicit type for f is given, then polymorphic recursion is allowed and we depart from the type system of Hindley-Milner [DM82].

`Helium` expressions can be explicitly typed, in which case both the type τ of the expression itself as well as the returned type have to be an instance of the

explicitly mentioned type σ. The check that σ is not more general than the type τ is postponed until a later stage. We leave the remaining rules to the reader.

$$\frac{literal : \tau}{\emptyset,\ \{\beta \equiv \tau\} \vdash^{\mathrm{p}}_{\mathrm{BU}} literal : \beta} \quad [\mathrm{Lit}]^{\mathrm{p}}_{\mathrm{BU}}$$

$$\frac{}{\{x : \beta\},\ \emptyset \vdash^{\mathrm{p}}_{\mathrm{BU}} x : \beta} \quad [\mathrm{Var}]^{\mathrm{p}}_{\mathrm{BU}}$$

$$\frac{C : \sigma \qquad \mathcal{B}_i,\ \mathcal{C}_i \vdash^{\mathrm{p}}_{\mathrm{BU}} p_i : \tau_i \ \text{ for } 1 \leq i \leq n}{\bigcup_i \mathcal{B}_i,\ \bigcup_i \mathcal{C}_i \cup \{\beta_1 \preceq \sigma,\ \beta_1 \equiv \tau_1 \to \ldots \to \tau_n \to \beta_2\} \vdash^{\mathrm{p}}_{\mathrm{BU}} C\ p_1 \ldots p_n : \beta_2} \quad [\mathrm{Con}]^{\mathrm{p}}_{\mathrm{BU}}$$

$$\frac{\mathcal{B}_i,\ \mathcal{C}_i \vdash^{\mathrm{p}}_{\mathrm{BU}} p_i : \tau_i \ \text{ for } 1 \leq i \leq n}{\bigcup_i \mathcal{B}_i,\ \bigcup_i \mathcal{C}_i \cup \{\beta \equiv (\tau_1, \ldots, \tau_n)\} \vdash^{\mathrm{p}}_{\mathrm{BU}} (p_1, \ldots, p_n) : \beta} \quad [\mathrm{Tuple}]^{\mathrm{p}}_{\mathrm{BU}}$$

$$\frac{\mathcal{B}_i,\ \mathcal{C}_i \vdash^{\mathrm{p}}_{\mathrm{BU}} p_i : \tau_i \ \text{ for } 1 \leq i \leq n}{\bigcup_i \mathcal{B}_i,\ \bigcup_i \mathcal{C}_i \cup (\{\tau_1 \ldots \tau_n\} \equiv \{\beta_1\}) \cup \{\beta_2 \equiv [\beta_1]\} \vdash^{\mathrm{p}}_{\mathrm{BU}} [p_1, \ldots, p_n] : \beta_2} \quad [\mathrm{List}]^{\mathrm{p}}_{\mathrm{BU}}$$

$$\frac{\mathcal{B},\ \mathcal{C} \vdash^{\mathrm{p}}_{\mathrm{BU}} p : \tau}{\mathcal{B} \cup \{x : \beta\},\ \mathcal{C} \cup \{\tau \equiv \beta\} \vdash^{\mathrm{p}}_{\mathrm{BU}} x@p : \beta} \quad [\mathrm{As}]^{\mathrm{p}}_{\mathrm{BU}}$$

$$\frac{}{\emptyset,\ \emptyset \vdash^{\mathrm{p}}_{\mathrm{BU}} \_ : \beta} \quad [\mathrm{Wc}]^{\mathrm{p}}_{\mathrm{BU}}$$

Fig. 3. The Bottom-Up type inference rules for patterns

We now proceed with patterns in `Helium`, see Fig. 3. Patterns occur in left-hand sides of function definitions, in lambda abstractions, and in the left-hand sides of case alternatives. The variables introduced in a pattern are, together with their fresh type variable, passed upwards in the set of bindings $\mathcal{B}$. The rules for literals, variables, lists, and tuples are the same as for expressions, except that we do not need to pass a set of monomorphic variables down into the pattern. The constructor rule combines the function application and constructor rules for expressions into one. The as-pattern allows us to bind nontrivial patterns to variables. The corresponding rule simply equates the type of the pattern with the type of the variable using a fresh type variable. Wildcards do not introduce new restrictions, so we only give them a dummy type β.

Finally, consider the rules in Fig. 4 for the remaining constructs that we deal with in this paper. The rules for function bindings and declarations are rather subtle. A declaration of a function consists of m functions bindings, all starting with the same function identifier, here f. Each function binding consists of the function name, a list of patterns p_i (here numbered from 1 to $n-1$ to fit better with the $[\mathrm{Decl}]^{\mathrm{d}}_{\mathrm{BU}}$ rule) and a right-hand side. The rule is very similar to that for the lambda abstraction except that we do not return a function type, but construct a type sequence of n types ($n-1$ parameters plus the type of the right-hand side). The type sequences for the various function bindings are collected in $[\mathrm{Decl}]^{\mathrm{d}}_{\mathrm{BU}}$. Now we can see the reason why we did not immediately construct a

$$\frac{\mathcal{B},\ \mathcal{C}_1 \vdash^{\mathrm{p}}_{\mathrm{BU}} p : \tau_1 \qquad M \cup \mathit{ran}(\mathcal{B}),\ \mathcal{A},\ \mathcal{C}_2 \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau_2}{M,\ \mathcal{A} \backslash \mathit{dom}(\mathcal{B}),\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup (\mathcal{B} \equiv \mathcal{A}) \vdash^{\mathrm{a}}_{\mathrm{BU}} (p \to e) : \langle \tau_1, \tau_2 \rangle} \quad [\mathrm{Alt}]^{\mathrm{a}}_{\mathrm{BU}}$$

$$\frac{\begin{array}{c} M,\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{e}}_{\mathrm{BU}} e : \tau \qquad M,\ \mathcal{B}_i,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{d}}_{\mathrm{BU}} d_i \ \text{ for } 1 \le i \le n \\ (\mathcal{A}', \mathcal{C}') = \mathit{BindingGroupAnalysis}(M, \mathit{explicits}, \{(\emptyset, \mathcal{A}), (\mathcal{B}_1, \mathcal{A}_1), \ldots, (\mathcal{B}_n, \mathcal{A}_n)\}) \end{array}}{M,\ \mathcal{A}',\ \mathcal{C} \cup \bigcup_i \mathcal{C}_i \cup \mathcal{C}' \vdash^{\mathrm{rhs}}_{\mathrm{BU}} (e\ \mathbf{where}\ \mathit{explicits}; d_1; \ldots; d_n) : \tau} \quad [\mathrm{RHS}]^{\mathrm{rhs}}_{\mathrm{BU}}$$

$$\frac{M,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{fb}}_{\mathrm{BU}} fb_i : \langle \tau_{1,i}, \ldots, \tau_{n,i} \rangle \ \text{ for } 1 \le i \le m}{\begin{array}{l} M,\ \{f : \beta_1 \to \ldots \to \beta_n\},\ \bigcup_i \mathcal{A}_i,\ \bigcup_i \mathcal{C}_i \cup \bigcup_j \{\beta_j \equiv \tau_{j,i} \mid 1 \le i \le m\} \vdash^{\mathrm{d}}_{\mathrm{BU}} fb_1; \ldots; fb_m \\ \quad \text{where } f \text{ is the single function being declared by the function bindings } fb_i \end{array}} \quad [\mathrm{Decl}]^{\mathrm{d}}_{\mathrm{BU}}$$

$$\frac{\mathcal{B}_i,\ \mathcal{C}_i \vdash^{\mathrm{p}}_{\mathrm{BU}} p_i : \tau_i \ \text{ for } 1 \le i \le n-1 \qquad \mathcal{B} = \bigcup_i \mathcal{B}_i \qquad M \cup \mathit{ran}(\mathcal{B}),\ \mathcal{A},\ \mathcal{C} \vdash^{\mathrm{rhs}}_{\mathrm{BU}} rhs : \tau_n}{M,\ \mathcal{A} \backslash \mathit{dom}(\mathcal{B}),\ \bigcup_i \mathcal{C}_i \cup \mathcal{C} \cup (\mathcal{B} \equiv \mathcal{A}) \vdash^{\mathrm{fb}}_{\mathrm{BU}} (f\ p_1\ \ldots\ p_{n-1} = rhs) : \langle \tau_1, \ldots, \tau_n \rangle} \quad [\mathrm{FB}]^{\mathrm{fb}}_{\mathrm{BU}}$$

Fig. 4. The remaining Bottom-Up type inference rules

function type: the types of the first pattern in each of the function bindings have to be the same, and similar for the other patterns and the right-hand sides. Of course, this could have been done by equating the function types, but our way seems to us more intuitive and more amenable to generating good type error messages.

Example 2. A small example is given in Figure 5 showing the constraints generated for the expression $\lambda f\ x \to f\ (f\ x)$. If we assume the set of monomorphic variables of the root to be empty, then the subexpressions of the body of the lambda all have $\{\tau_0, \tau_1\}$. Note that below every node in the tree we have indicated the fresh type variable introduced at that point.

4 The type inference process

The type inference process consists of three different phases. The first of these generates constraints in the nodes of the abstract syntax tree (using the rules of the previous section), generating a Rose tree[2] where each node is labelled with a collection of constraints.

The second phase consists of traversing the abstract syntax tree and ordering the constraints found in the tree. Our implementation contains an infrastructure for building traversals and the most important ones are predefined.

The final phase solves the flattened constraint tree. The solver may be a greedy solver which continually updates a substitution, or a more global solver which builds elaborate data structures such as type graphs in order to be able to decide in a more global fashion which constraints are likely to be responsible for the inconsistency. In both cases, the outcome includes a list of unsatisfiable

[2] A tree datastructure where each node has an arbitrary number of children.

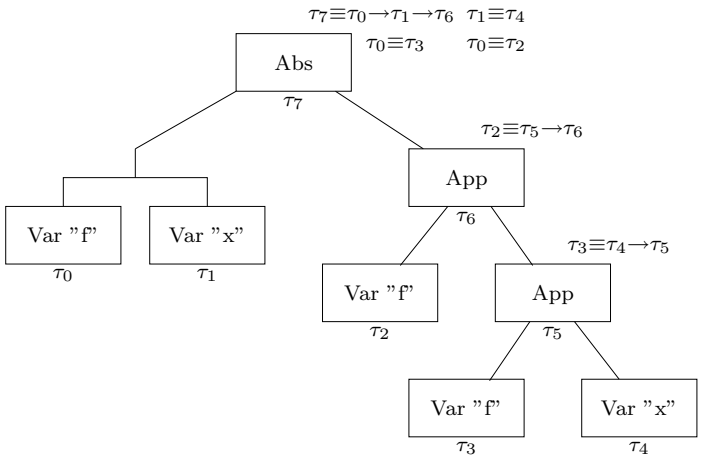

Fig. 5. Constraints for the expression $\lambda f\ x \to f\ (f\ x)$

constraints, each with a type error message which explains to the programmer what the problem is.

4.1 Collecting the constraints using uu_ag

In this section we explain how the type inference rules of the previous section can be implemented easily in the UU_AG system. We assume that the reader is familiar with attribute grammars in general. For more information on the UU_AG system consult [SBL]. The main focus of this section is not to show how it works exactly, but only to give the reader a flavour of what it looks like and to highlight the correspondence between the type rules and the code. The advantage of using attribute grammars at this point is that we only need to formulate how values depend on each other, and the attribute grammar system automatically determines how these values can actually be computed.

All fragments of code are shown in Fig. 6 (slightly modified from the actual code). First we define the attributes of the non-terminals in the abstract syntax tree, found in the **ATTR** section. Besides the top-down (inherited) and the bottom-up (synthesized) aspects, there are chained attributes that are passed along in both directions.

Note that all elements in a judgement for an expression turn up as attributes, with M an i-attribute and the remaining ones ($\mathcal{A}$, ...) s-attributes. Additionally, the chained attribute *unique* provides a counter to generate fresh type variables. Instead of using a list to collect the type constraints, a Rose tree that follows the shape of the abstract syntax tree is constructed, in which the nodes are decorated with any number of constraints. In this way we retain some flexibility in the order in which the constraints will be considered later. Hence, this part

```
ATTR Expr [ mono   : Types                                   (inherited)
          | unique : Int                                     (chained)
          | aset   : Assumptions                             (synthesized)
            ctree  : ConstraintTree
            beta   : Type  ]

SEM Expr
  | If
      lhs    . aset    = @guard.aset ++ @then.aset ++ @else.aset                    (1)
             . ctree   = Node [ [@guard.beta ≡ boolType] `add` @guard.ctree        (2)
                              , [@then.beta ≡ @beta]     `add` @then.ctree
                              , [@else.beta ≡ @beta]     `add` @else.ctree ]
      guard . unique = @lhs.unique + 1                                            (3)
      loc    . beta    = TVar @lhs.unique                                         (4)

SEM Expr
  | Lambda
      lhs    . aset    = removeKeys (map fst @pats.bset) @expr.aset
             . ctree   = [ beta ≡ foldr (→) @expr.beta @pats.betas ] `add`
                         Node [ @pats.ctree, @binds `spread` @expr.ctree ]
      pats   . unique = @lhs.unique + 1
      expr   . mono   = map snd @pats.bset  ++ @lhs.mono
      loc    . beta    = TVar @lhs.unique
             . binds   = [ τ1 ≡ τ2  | x1 == x2
                                    , (x1,τ1) ← @pats.bset, (x2,τ2) ← @expr.aset ]
```

Fig. 6. Code fragments of the attribute grammar

of the code only generates the constraints, while fixing the order and actually solving the constraints is considered later.

Due to space restrictions we limit ourselves to giving the semantic functions for the conditional expression and the lambda abstraction. The three sub-expressions of a conditional are referred to as *guard*, *then* and *else*. Consider the attribute *unique*. We use it to generate a new variable in the local attribute *beta*. Note that *beta* is also the name of a synthesized attribute of the if-node. The reason we introduce it as a local attribute is because we also need it for *cset*. The value of *unique* is threaded through the tree, incremented along the way every time we introduce a new type variable.

The semantic rules are given in the **SEM** section of Fig. 6. Here, the syntax for referring to an attribute is @*child.attribute*, where **lhs** and **loc** are special keywords to refer to inherited attributes of the father node and attributes that are defined locally, respectively. Obviously, the subtrees for the condition, the then-part and the else-part will contain the constraints generated for those parts. However, the if construct has some constraints of its own: the constraint that the result of the condition is of type boolean, the constraint that the result of the then-part is equal to the result of the conditional as a whole, and similarly for the else-part. It would be quite natural to put these constraints into a set

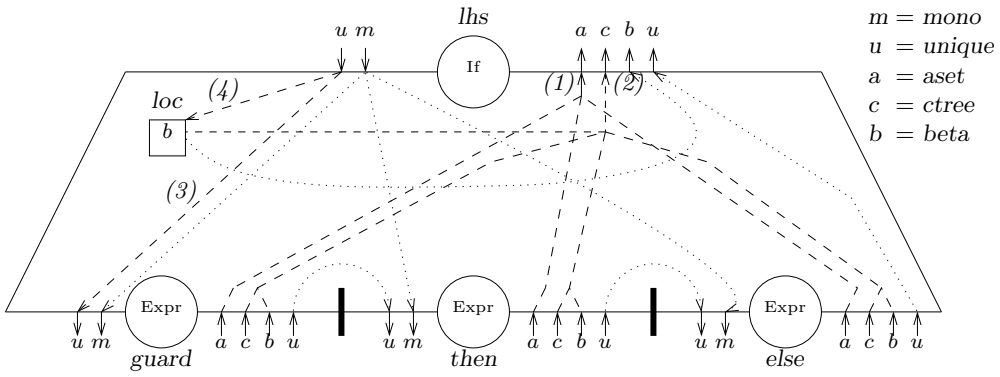

Fig. 7. Dependencies between the attributes for the conditional

of constraints attached to the if-node. However, to retain as much flexibility as possible, we have chosen differently. In addition to being able to add a constraint to the set of constraints of the if-node, it is also possible to add a constraint to one of the subtrees, and this is what happens here. The three type constraints for a conditional are added to the constraint trees of their corresponding subexpression with the function *add*. We consider the uses of this facility in more detail in the next section.

The dependencies between the various attributes for the conditional are made more explicit in Fig. 7. The number following a semantic equation refers to the correspondingly numbered dashed lines in Fig. 7; the dotted edges represent the passing of unmodified attributes. For instance, *mono* is passed on unchanged to all three children. We do not have to write the code for passing these attributes ourselves. Instead, the compiler generates these copy rules automatically.

For lambda abstractions, the assumptions concerning the bound variables are removed from the assumption set, and the type variables that are introduced in the patterns are inserted into the set of monomorphic type variables that is passed to the body. A constraint is constructed for each matching combination of a tuple from the assumption set and from the binding set. This set of constraints, which is the local attribute *binds*, is added to the constraint tree with the function *spread*.

4.2 Flattening the constraint tree

After the generation phase we have obtained a Rose tree in which a node contains a set of constraints, and a list of children. Each child is a subtree containing its own constraints as well as a separate collection of constraints put there by its parent.

The location where most type inferencers detect an inconsistency for an ill-typed expression strongly depends on the order in which types are unified. By

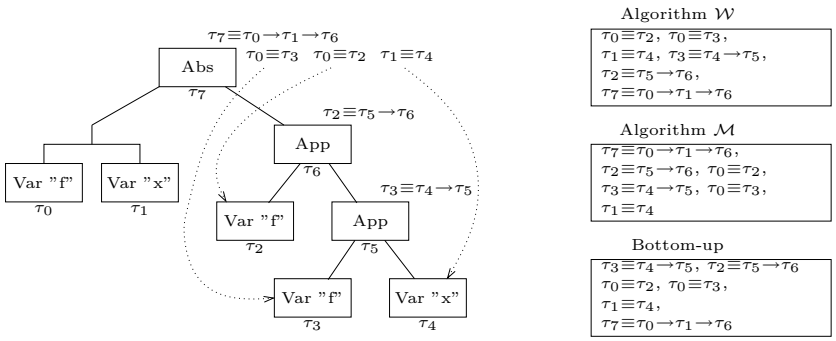

Fig. 8. Spreading the constraints for $\lambda f\ x \to f\ (f\ x)$

specifying how to flatten the constraint tree we can imitate several type inference algorithms, each with its own properties and characteristics.

We flatten a constraint tree by defining a treewalk over it that puts the constraints in a certain order. Besides the standard preorder and postorder treewalks, one can think of more experimental ones such as a right-to-left treewalk. In our current implementation we use the same treewalking strategy in each node of a constraint tree, in the sense that we cannot specify a treewalk going left-to-right in application nodes and right-to-left in the case a list node; it is a straightforward extension to use different strategies depending on the nonterminal in the abstract syntax tree. A treewalk takes the various collections of constraints available in a node: the constraints of the node itself, the collections collected from the subtrees and the collections of constraints added (by means of the function *add*, see Fig.6) to each of the substrees, and orders these collections in some fashion, usually without changing the order within the collections themselves.

Special care is taken for inserting a constraint that corresponds to the binding of a variable to a pattern variable; these constraints are inserted with the function *spread*. Instead of adding the constraint to the node where the actual binding takes place, a constraint may be mapped onto the occurence of the bound variable. Fig. 8 shows the spreading of three constraints. The constraint orders of three treewalks are shown on the right: a postorder treewalk with spreading ($\mathcal{W}$), a preorder treewalk with spreading ($\mathcal{M}$), and a postorder treewalk without spreading. The main motivation for spreading is that the way the standard algorithms $\mathcal{W}$ and $\mathcal{M}$ solve their constraints corresponds to spreading. To be able to mimick these algorithms it has been included.

Finally, we want to point out that any treewalk for ordering the constraints should adhere to the principle that it only generates of lists of constraints so that when we encounter an implicit instance constraint, the condition (1) is satisfied. This is guaranteed by solving the constraints arising from the definitions in a let, before continuing with the constraints arising from the body. An implicit instance

constraints can only depend on definitions higher up in the tree. Remember that we consider an entire binding group at the time and polymorphic recursion is only allowed when explicit types are given.

4.3 Solving the constraints

In this section we consider a number of approaches to solving the collected constraints. We describe the general characteristics of a constraint solver by listing all the operations that it should be able to perform. We continue by explaining how various greedy solving methods can be implemented within this framework. These greedy algorithms can be tuned quite easily by specifying a small number of parameters. Finally, we spend some time on discussing how the constraints can be solved in a more global way using type graphs which enables us to remove the left-to-right bias inherent in tree based online algorithms such as $\mathcal{W}$ and $\mathcal{M}$.

Note that "solving" in this context means that the constraint is taken from the list of constraints and dealt with. Solving does not mean to imply that the constraint is consistent with the information we have obtained so far. Directly, or at some later point, we may decide that the constraint is a source of an inconsistency and generate an appropriate error message for it.

A type class for solving constraints To pave the way for multiple implementations of a solver, we present a `Haskell` type class. Since it is convenient to maintain a state while solving the constraints, we have implemented this class using a *State* monad that provides a counter for generating unique type variables, a list of reported inconsistencies, and a substitution or something equivalent. A type *solver* can solve a set of constraints, in which each constraint is carrying additional *info*, if it is an instance of the following class.

```
class Solver solver info where
  initialize        ::                                    State solver info ()
  makeConsistent    ::                                    State solver info ()
  unifyTypes        :: info → Type → Type →               State solver info ()
  newVariables      :: [Int] →                            State solver info ()
  findSubstForVar   :: Int →                              State solver info Type
```

By default, the functions *initialize*, *makeConsistent*, and *newVariables* do nothing, that is, leave the state unchanged. If *unifyTypes* is called with two non-unifiable types it can either deal with the inconsistency immediately, or postpone it and leave it to *makeConsistent*. The function *applySubst*, which performs substitution on types, is defined in terms of a more primitive function called *findSubstForVar*. A function *solve* can now be defined which takes an initial value for the type variable counter, and a collection of constraints, where each constraint is labeled with *info*, and results in a *State* which describes the result of solving the collection of constraints. After initialization, the constraints are solved one after another resulting in a possibly inconsistent state. Calling *makeConsistent*

will remove possible inconsistencies and, as a side effect, add error messages to the state.

We consider now how the three types of constraints may be solved. An equality constraint is solved by unification of the two types. The type scheme of an explicit instance constraint is instantiated and the state is informed about the fresh type variables that are introduced. An implicit instance constraint is solved by first making the state consistent, and subsequently applying the substitution to the type and the monomorphic type variables. Only then may we solve the constraint by transforming it into an explicit instance constraint and solving it.

Note that due to the lazy semantics of `Haskell` the list of constraints generated by a treewalk is only constructed insofar we actually solve the constraints. Whenever an error is encountered with the kind of solver that terminates once it has seen a type error, the other constraints are not even computed. Needless to say, laziness imposes its own penalties.

Greedy constraint solving The most obvious instance of the type class *Solver* is a substitution. The implementation of *unifyTypes* then simply returns the most general unifier of two types. The result of this unification is incorporated into the substitution. When two types cannot be unified, we immediately deal with the inconsistency. As a result, *makeConsistent* can be the default skip function, because *unifyTypes* always results in a consistent state: if adding a constraint results in an inconsistent state, then it is ignored instead, although an appropriate error message is generated (and added to the state). After the discovery of an error, we can choose to continue solving the remaining constraints, which can lead to the detection of more type errors.

For efficiency reasons, we represent a substitution by a mutable array in a strict state thread, also because the domain of the substitution is dense. Instead of maintaining an idempotent substitution, we compute the fixpoint in case of a type variable lookup.

Constraint solving with type graphs Because we are aiming for an unbiased method to solve type constraints, we discuss an implementation which is based on the construction of a *type graph* inspired by the path graphs described in [Por88]. The type graph allows us to perform a global analysis of a set of constraints, which makes type inferencing a non-local operation. Because of its generality, adding heuristics for determining the "correct" type error is much easier. It is important to note that although the constraint solver based on type graphs does consider the constraints in the order that the treewalk generated them, this does not mean that an early constraint has a smaller chance of being held responsible for an inconsistency. This is completely determined by a number of heuristics which work on the type graph.

Each vertex in the type graph corresponds to a subterm of a type in the constraint set. A composed type has an outgoing edge labelled with (i) to the vertex that represents the i^{th} subterm. For instance, a vertex that represents a function type has two outgoing edges. All occurrences of a type variable in the

constraint set share the same vertex. Furthermore, we add undirected edges labelled with information about why two (sub)terms are unified. For each equality constraint, an edge is added between the vertices that correspond to the types in the constraint. Equivalence of two composed types propagates to equality of the subterms. As a result, we add *derived* (or *implied*) edges between the subterms in pairwise fashion. For example, the constraint $\tau_1 \to \tau_1 \equiv \mathit{Bool} \to \tau_2$ enforces an equality between τ_1 and *Bool*, and between τ_1 and τ_2. Therefore, we add a derived edge between the vertex of τ_1 and the vertex of *Bool*, and similar for τ_1 and τ_2. For each derived edge we can trace the constraints responsible for its inclusion. Note that adding an edge can result in the connection of two equivalence classes, and this might lead to the insertion of more derived edges. Whenever a connected component of the type graph contains two or more different type constructors, we have encountered a type error. Infinite types can also be detected, but we skip the details.

Example 3. Consider the following ill-typed program.

$f\ 0\ y = y$
$f\ x\ y = \mathbf{if}\ y\ \mathbf{then}\ x\ \mathbf{else}\ f\ (x-1)\ y$

The set of type constraints that is collected for this program is as follows.

#1	$\tau_0 \equiv \tau_2 \to \tau_3 \to \tau_1$				
#2	$\mathit{Int} \equiv \tau_4$	#3	$\tau_4 \equiv \tau_2$	#4	$\tau_5 \equiv \tau_3$
#5	$\tau_5 \equiv \tau_6$	#6	$\tau_6 \equiv \tau_1$	#7	$\tau_7 \equiv \tau_2$
#8	$\tau_8 \equiv \tau_3$	#9	$\tau_8 \equiv \tau_{10}$	#10	$\tau_{10} \equiv \mathit{Bool}$
#11	$\tau_7 \equiv \tau_{11}$	#12	$\tau_{11} \equiv \tau_9$	#13	$\tau_0 \equiv \tau_{13}$
#14	$\tau_{17} \preceq \mathit{Int} \to \mathit{Int} \to \mathit{Int}$	#15	$\tau_7 \equiv \tau_{16}$	#16	$\mathit{Int} \equiv \tau_{18}$
#17	$\tau_{17} \equiv \tau_{16} \to \tau_{18} \to \tau_{15}$	#18	$\tau_{15} \equiv \tau_{14}$	#19	$\tau_8 \equiv \tau_{19}$
#20	$\tau_{13} \equiv \tau_{14} \to \tau_{19} \to \tau_{12}$	#21	$\tau_{12} \equiv \tau_9$	#22	$\tau_9 \equiv \tau_1$

Fig. 9 depicts the type graph for this set. The shaded area indicates a number of type variables that are supposed to have the same type. The graph is clearly inconsistent, because of the presence of both *Int* and *Bool*. Applying the heuristic of Walz and Johnson [WJ86] to measure the proportion of each constant, would result in cutting off the boolean. Cutting off the boolean removes the inconsistency, but even then there are various constraints which can be removed to obtain this effect. If #10 is chosen – this amounts to saying that the type of y is not a boolean, then it would generate the following type error message.

```
(2,12): Type error in conditional
 expression       : if y then x else f (x - 1) y
 term             : y
   type           : Int
   does not match : Bool
```

If instead #12 is chosen, which corresponds to the fact that the type of the then-branch of a conditonal should have the same type as the conditional as

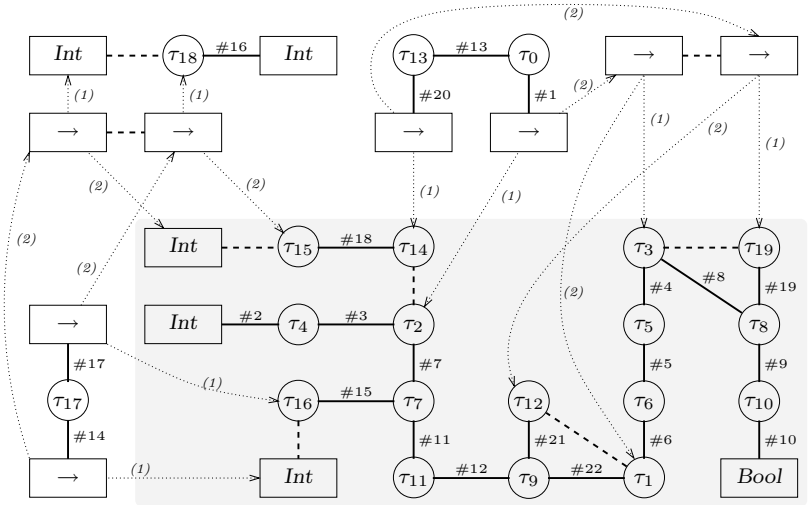

Fig. 9. TypeGraph

whole, then the following message results:

```
(2,19): Type error in conditional
 expression       : if y then x else f (x - 1) y
 term             : x
   type           : Int
   does not match : Bool
```

Constraints have been implemented in such a way in our compiler, that each constraint carries all the information it needs, e.g. position information, to generate an appropriate error message.

We refer the reader to the `Helium` website [ILH] for a list of example programs with the output as generated by the compiler. The type error messages were all generated using the type graph data structure of this section guided by a collection of heuristics. After an inconsistency is detected in the type graph, a large repertoire of heuristics is applied resulting for each heuristic in two things: a list of constraints contributing to an inconsistency and a score proportional to the amount of trust the heuristic has in the result. These values are compared for the various heuristics and the 'best' one is chosen. The impact of a heuristic can be changed by modifying the way the score is computed.

Heuristics fall into two natural classes: the ones which look for special mistakes programmers often make (the first two listed next), and a number of more general ones (the others). The scoring in `Helium` is set up such that the latter are used as tiebreakers in case none of the former return anything useful.

– If the permutation of arguments to a function resolves a type inconsistency, and no other permutation does, then we assume that this is the problem and

generate an error message and a hint for improving the program. This is in fact one example of a host of heuristics which result in easy fixes: forgotten arguments, permuted elements in a pair, too many arguments, etcetera.
- If an integer is used where a float was expected or vice versa, then this is a likely source of an error. Since we do not have for numerals as is the case in `Haskell`, this is an error in `Helium`. Experienced `Haskell` programmers are hence likely to make this type of mistake and we have included a special heuristic to give a hint.
- The proportion of constants in an inconsistent part of the type graph determines which constant is considered to be the error (see the example earlier on).
- Constraints which are part of a cycle are usually not chosen, because they cannot by themselves resolve an inconsistency.
- Definitions from the Prelude are to be trusted more than definitions in the program itself.
- The order in which the constraints were added to the type graph The later it was added, the more likely it is to be considered a source of inconsistencies.

5 Type inference directives

In the previous sections we have already shown that an infrastructure is present to be able to experiment with various type inference processes within the `Helium` compiler. Some of these can be used by setting the appropriate parameters to `Helium`, some can be used by small modifications to the compiler.

To improve the quality of type error messages in functional programming languages, we proposed four techniques in [HHS03] to further influence the behaviour of the constraint based type inference process. These techniques take the form of externally supplied type inference directives, precluding the need to make any changes to the compiler. A second advantage is that the directives are automatically checked for soundness with respect to the underlying type system. These techniques can be used to improve the type error messages reported for a combinator library. More specifically, they can help to generate error messages which are conceptually closer to the domain for which the library was developed.

In [HHS03] we show how this can be obtained within `Helium`. The main reason that this can done at all, is the separation between the generation, the ordering and the solving of constraints. An additional necessity is that it should be possible for type error messages to be overridden by messages supplied by the user of the compiler. We conclude this section with a necessary limited example to give a flavour of how this facility can be used.

Example 4. Novice programmers often have difficulty learning to use higher-order functions such as *map* and *filter*. User-defined type rules can be used to isolate expressions which use these functions and generate specific error messages for them. In our setup this can be formulated in a *.type* file which accompanies the usual *.hs* source file. We now give a much simplified type rule for expressions in which map is applied to two parameters (which can be any expression).

```
  f :: t1 ;     xs :: t2 ;
-------------------------------
   map f xs :: [b] ;;

t1 == t3 -> t4: The first argument to map should be a function
t4 == b     : The result type of the function should match
                 the elements of the resulting list
t2 == [t3]  : The second argument to map should be a list of @t3@s
```

By means of the notation `@t3@`, the programmer can refer to the type to which the type variable `t3` maps during execution. Similar expressions can be used to refer to range information for expressions in occurring in the type rule and so on. The main thing to note is that the list of constraints implies a preference from the side of the programmer about the order in which they should be checked, in the sense the error message for `t4 == b` can only be given, if `t1 == t3 -> t4` was satisfied. Because of this, we can use the fact that `t1 == t3 -> t4` is known to hold when we consider `t4 == b`, so that referring to 'the function' in the error message is guaranteed to make sense. This allows for much more concrete error messages. The impact of this is even more strongly felt when type rules are given for expressions which include complex user defined datatypes.

6 Conclusion and future work

In this paper we have described the `Helium` compiler which deals with a large subset of `Haskell` and can by virtue of a constraint based type inference process generate understandable type error messages. The type inference process consists of three distinct phases:

1. the generation of constraints,
2. the ordering of constraints and, finally,
3. the solving of constraints.

Extra flexibility is obtained by means of various tree walking algorithms and the even more powerful feature of type inference directives which allow a programmer to tune the type inference process to his liking. All of this has been made possible by virtue of a constraint based type inference process, and the way it has been implemented.

We are currently working on extending our type inferencer with type classes and turning the solver into a general implementation for solving constraints. One of the major challenges is to extend the type inference directives into an easy to use programming language with which programmers can tailor a compiler to their own needs. On the one hand we expect to include directives with which one can specify the tree walk to be performed and which heuristics should be applied. On the other hand, the addition of type classes shall give rise to a new set of directives, e.g., specifying that two type classes have to be disjoint. This can help giving good error messages in case of ambuigities that may arise.

References

[Chi01] Olaf Chitil. Compositional explanation of types and algorithmic debugging of type errors. In *Proceedings of the Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*, pages 193–204, September 2001.

[DM82] L. Damas and R. Milner. Principal type schemes for functional programs. In *Principles of Programming Languages (POPL '82)*, pages 207–212, 1982.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, July 2002.

[HHS03] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *International Conference on Functional Programming '03*, 2003. To appear.

[HW03] Christian Haack and J. B. Wells. Type error slicing in implicitly typed higher-order languages. In *Proceedings of the 12th European Symposium on Programming*, pages 284–301, April 2003.

[ILH] Arjan IJzendoorn, Daan Leijen, and Bastiaan Heeren. The Helium compiler. http://www.cs.uu.nl/helium.

[LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transanctions on Programming Languages and Systems*, 20(4):707–723, July 1998.

[McA00] Bruce J. McAdam. Generalising techniques for type debugging. In Phil Trinder, Greg Michaelson, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 49–57. Intellect Books, March 2000.

[Pie02] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, Cambridge, MA, 2002.

[Por88] Graeme S. Port. A simple approach to finding the cause of non-unifiability. In Robert A. Kowalski and Kenneth A. Bowen, editors, *Proceedings of the Fifth International Conference and Symposium on Logic Programming*, pages 651–665, Seatle, 1988. The MIT Press.

[SBL] S. Doaitse Swierstra, Arthur I. Baars, and Andres Loeh. The UU-AG attribute grammar system. http://www.cs.uu.nl/people/arthurb/ag.html.

[Sul] Martin Sulzmann. The Chameleon system, http://www.comp.nus.edu.sg/ sulzmann/chameleon.

[WJ86] J. A. Walz and G. F. Johnson. A maximum flow approach to anomaly isolation in unification-based incremental type inference. In *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 44–57, St. Petersburg, FL, January 1986.

[YMT00] J. Yang, G. Michaelson, and P. Trinder. Helping students understand polymorphic type errors, 2000.