Parametric Type Inferencing for Helium

Bastiaan Heeren and Jurriaan Hage
{bastiaan,jur}@cs.uu.nl

Inst. of Information and Computing Sci., Univ. Utrecht, P.O.Box 80.089, 3508 TB Utrecht, Netherlands

Abstract. `Helium` is a compiler for a large subset of `Haskell` under development at Universiteit Utrecht. A major design criterion is the ability to give superb error messages. This is especially needful for novice functional programmers. In this paper we document the implementation of the `Helium` type inferencer. For purposes of experimentation with various methods of type inferencing, the type inferencer can be parameterized in a number of ways. Among the instances we find not only standard algorithms such as $\mathcal{M}$ and $\mathcal{W}$, but also more global type inferencers based on type graphs.

1 Introduction

One of the main drawbacks of learning to program a high-level polymorphic functional language such as `Haskell`, is that the type error messages are often too complex for novices to understand. One of the reasons is that the standard local algorithms such as $\mathcal{M}$ [DM82] and $\mathcal{W}$ [LY98] were developed mainly for speed, and not for returning good error messages. There exist many examples of ill-typed programs, where a more global approach would yield better error messages.

One of the drawbacks of developing your own type inferencer for a large language such as `Haskell`, is that eventually a compiler is needed to do real experimentation, for instance in a classroom setting. At Universiteit Utrecht, Arjan IJzendoorn recently started to work on a compiler for a subset of `Haskell` called `Helium`. Only language constructs of `Haskell` for which informative error messages could be given, are included.

As a side-effect, the compiler gives us an opportunity to compare the standard algorithms with our own in a classroom setting, and we plan to do so in the near future in freshmen courses on functional programming. In this paper we go into the implementation of our type inferencing method, which is based on a clear separation between collecting and solving type constraints. This separation also allows us to emulate well-known type inferencing algorithms such as $\mathcal{W}$ and $\mathcal{M}$, by modification of the order in which the type constraints are solved. This provides a way to compare these standard algorithms to global heuristics which may use elaborate data structures such as type graphs (see Section 5). In this way we hope to gain insight into the quality of the error messages and the computational penalties to be paid.

Apart from this main goal we also plan to show the reader how the type constraints can be collected using the attribute grammar system `UU_AG` developed at Universiteit Utrecht by Doaitse Swierstra et al. (documentation can be found at [SBL]). Algorithms such as $\mathcal{W}$ and $\mathcal{M}$ always drag some representation of a substitution around, which is continually updated to reflect newly found information. In contrast, the relation between the typing rules and their implementation in the `UU_AG` system is straightforward and makes the implementation easy to maintain. This is certainly profitable, because it is likely that new constructs will be added to `Helium` at a later stage.

The proof that our inferencing rules, combined with a constraint solver, is equivalent to the type inferencing rules of Hindley-Milner [DM82], with appropriate restrictions on the expression language, can be found in Heeren, Hage and Swierstra [HHS02]. We point out that the expression language in the current paper is much larger. A discussion of the validity of the new typing rules lies outside the scope of the current paper. Instead, we focus on more pragmatic aspects such as the efficiency of the implementation and its amenability to experimentation.

The compiler is structured as follows. As most compilers, the `Helium` compiler can be divided into a number of phases. The first of these phases consists of lexical scanning and parsing, which is done using Daan Leijen and Erik Meijer's collection of monadic parser combinators Parsec [LM01]. The resulting abstract syntax trees, in a format called UHA which is a local standard at Universiteit Utrecht, are then semantically checked using the attribute grammar system `UU_AG`. If the program passes all the tests, then the UHA is desugared, normalized and optimized into a bare language called ASM, which can be executed by the Lazy Virtual Machine of Daan Leijen. In this paper we are only interested in explaining the type inferencing which is part of semantic analysis.

The paper is structured as follows. After some preliminaries to fix notation, we give the bottom-up type inference rules for our complete expression language, also explaining how these type rules are programmed in the `UU_AG` system. We then continue by showing how to deal with multiple constraint solvers in our compiler, including a greedy one (which itself can be instantiated to behave as algorithms like $\mathcal{W}$ and $\mathcal{M}$), as well as a solver which tries to find a minimal set of errors based on a global analysis based on a type graph. Before we conclude in the final section, we give an example comparing Hugs and GHC to the `Helium` type inferencer.

2 Preliminaries

Types and substitutions

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

(type)	$\tau := \alpha \;\mid\; T\ \tau_1\ \ldots\ \tau_n$	where $arity(T) = n$
(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 tuple types. The set of type constructors can be extended with user defined data types such as `Maybe`, but we do not include it here.

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, $\mathit{ftv}(\forall\vec{\alpha}.\tau) = \mathit{ftv}(\tau) - \vec{\alpha}$.

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. For completeness sake we note that the substitution $\top$ maps every type into a special error type also denoted by $\top$. This is simply to cope with the inability to unify types.

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.

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

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

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

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:

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

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 inferencing 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 before we start inferencing; this occurs for instance when an explicit type for the expression was given. In general, the (polymorphic) type of a declaration in 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. 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 look for the minimal substitution that satisfies each constraint in the set. 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. Since in general $\mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau)$ is not equal to $\mathcal{S}(\mathit{generalize}(M, \tau))$, implicit and explicit instance constraints really have different semantics. However, for every implicit instance constraint there comes a point where it can be transformed into an explicit instance constraint and from there into an equality constraint. 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 to solve. As a result we have to make sure that we find a substitution $\mathcal{S}$ that satisfies c, or $\mathcal{S}\alpha_3 \prec \mathit{generalize}(\mathcal{S}\{\alpha_5\}, \mathcal{S}(\alpha_1 \to \alpha_2))$ where the latter is equal to $\forall \beta_1\beta_2.\beta_1 \to \beta_2$, because α_1 and α_2 are not known to be monomorphic. A most general substitution to satisfy this constraint is $\mathcal{S} = [\alpha_3 := \alpha \to \beta]$, where α and β are fresh type variables. 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 is in fact monomorphic, because it is equal to a type variable of which we know that it is monomorphic (and monomorphic type variables can never become polymorphic again).

Although, it might seem that this problem does not occur for $c' = \alpha_5 \equiv \alpha_4$, take note that there may be other constraints through which this constraint makes either α_1 or α_2 monomorphic. The only safe way is to postpone solving c until both its set of monomorphic variables and the type on the right-hand side can no longer change.

However, to simply demand that α_5 is not anymore present in any of the constraints is too coarse a view. If the constraint set contains a constraint of the form $\alpha_3 \leq_{\{\alpha_5\}} \alpha_2 \to \alpha_1$ or even $\alpha_3 \leq_{\emptyset} \alpha_5 \to \alpha_1$, then there is really no problem.

Lemma 1 in [HHS02] states collecting constraints according to our type inference rules always results in a set of constraints that can be solved: equality constraints and explicit instance constraints can be solved unconditionally. If we have only implicit instance constraints left, then the implicit instance constraint generated by the leftmost-innermost let-expression for which we still need to solve such a constraint, always fulfills the condition for being solvable, as explained in the previous example.

expr = lit | var | constructor
  | expr expr$^+$
  | **if** expr **then** expr **else** expr
  | 'λ' pat$^+$ '→' expr
  | **case** expr **of** alt^s
  | **let** decl^s **in** expr
  | '(' expr$^\ell$ ')' | '[' expr$^\ell$ ']'
  | expr '::' typescheme

pat = lit | var | constructor pat*
  | '(' pat$^\ell$ ')' | '[' pat$^\ell$ ']'
  | var '@' pat | '_'
alt = pat '→' expr
decl = fb^s | var '::' typescheme
fb = var pat* '=' rhs
rhs = expr (**where** decl^s)$^?$

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

We conclude that if we take the set of constraints and order them in a fashion compatible with the conditions discussed in the previous paragraph, then we can solve these constraints in that order.

3 Bottom-Up Type Inference Rules

In this section we give the type inference rules for a large part of Helium. The part of Helium we consider is given in Figure 1 where we have concentrated on the 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.

In general, our approach to inferencing types is to label every subexpression with a fresh type variable (usually called β) and to generate constraints for the restrictions to be imposed at this point. Usually the type for an expression is simply a type variable (during the solving process we find out exactly what type it represents), although in some cases it is advantageous to return a type expression containing several new variables. An expression of such kind is denoted $\langle\beta_1,\ldots,\beta_n\rangle$.

Consider the type inference rules for expressions in Figure 2. The judgements in these rules are of the form $M,\ \mathcal{A},\ \mathcal{C} \vdash^{e}_{\mathrm{BU}} e : \tau$. Here $\mathcal{C}$ is simply 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 records 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. In fact, the only piece of information that is passed downwards is the set of monomorphic variables M. Note that the rules allow for flexibility in coping with unbound identifiers.

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

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

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

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

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

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

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

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

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

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

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

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

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

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*,...). The need for introducing a type variable even for a literal shall become apparent later. 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). For a conditional, we generate constraints to enforce that the condition e_1 has type *Bool* and e_2 and e_3 must have equal types (enforced through the intermediate type variable β).

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. As in the case of the assumption set, 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. This set is used when generating the implicit instance constraints for let-expression in the body. The constraints generated for the lambda abstraction itself should take into account 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 $[\mathrm{Let}]^{\mathrm{e}}_{\mathrm{BU}}$ we take into account that the types of some definitions are explicitly given. In `Helium`, the binding groups are determined in part by the given explicit typings. 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. When generating implicit instance constraints, we need the set M of monomorphic type variables.

A case expression consists of an expression e and a list of n alternatives. We collect information for each alternative, resulting in a type for the pattern and one for the expression. Of course, the types of the patterns must be the same and equal to the type of the expression e. In addition, the types of the expressions on the right should all agree as well. This is also the resulting type of the case expression.

`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. To save space, we leave the remaining rules to the reader.

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

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

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

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

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

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

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

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

We now proceed with patterns in `Helium`, see Figure 3. Patterns occur in left-hand sides of function definitions, in lambda abstractions, and occur in the left-hand sides of case alternatives. The variables introduced in a pattern are, together with the corresponding fresh type variable, passed bottom-up 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 Figure 4 for the remaining constructs that we deal with in this paper. An alternative, as used in case expressions is a pattern followed by an expression, separated by an arrow. This rule is very similar to the rule for lambda-abstractions. As is to be expected, the rule for right-hand sides $[\mathrm{RHS}]^{\mathrm{rhs}}_{\mathrm{BU}}$ is similar to the rule for let-expressions. The only difference is that right-hand sides themselves are not expressions that may be used as part of larger expressions. For that reason we omit the, in this case unnecessary, introduction of a fresh type variable. The rules for function bindings and declaration need a more careful explanation. 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

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

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

$$\frac{M,\ \mathcal{A}_i,\ \mathcal{C}_i \vdash^{\mathrm{fb}}_{\mathrm{BU}} \mathit{fb}_i : \langle \tau_{1,i}, \ldots, \tau_{n,i} \rangle \ \text{ for } 1 \leq i \leq m}{M,\ \{f : \beta_1 \rightarrow \ldots \rightarrow \beta_n\},\ \bigcup_i \mathcal{A}_i,\ \bigcup_i \mathcal{C}_i \cup \bigcup_j \{\beta_j \equiv \tau_{j,i} \mid 1 \leq i \leq m\} \vdash^{\mathrm{d}}_{\mathrm{BU}} \mathit{fb}_1; \ldots; \mathit{fb}_m} \qquad [\mathrm{Decl}]^{\mathrm{d}}_{\mathrm{BU}}$$

where f is the single function being declared by the function bindings fb_i

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

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

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 construct a 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 this seems to us more intuitive and more amenable to generating good type error messages.

4 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. All fragments of code are shown in Figure 5, and are intended to illustrate the implementation techniques used; they are not complete. First we choose a representation for assumption sets and binding sets. Because these sets can become quite large in practice, it is sensible to choose a different and more efficient representation. The function *removeKeys* is used to filter the assumptions in a set. The next step is to define the attributes of the nonterminals 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 the attributes exactly match the elements in a judgement for an expression. 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, where the nodes are decorated with any number of constraints. In this way we retain some flexibility in the order of the constraints.

```
type Assumptions = [(String,Type)]
type Bindings    = [(String,Type)]
removeKeys xs ys = filter ((`notElem` xs) . fst) ys

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 Expression
  | 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. 5. Code fragments of the attribute grammar

Due to space restrictions we limit ourselves to giving the semantic functions for the conditional expression and the lambda abstraction. We start with the conditional, for which we give a pictorial representation of the dependencies between the various attributes in Figure 6. The three sub-expressions of a conditional are referred to as *guard*, *then* and *else*. Consider the attribute *unique*. Its value is used in two different ways. First of all, 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*. After incrementing, *unique* is passed to the first child. The *unique* attribute of the second child depends on the *unique* value coming out of the first child, and similarly for the third child. Finally, the *unique* counter coming out of the third child is passed upwards. In other words, the value of *unique* is threaded through the tree, being incremented along the way.

The semantic rules are given in the **SEM** section of Figure 5. Here, the syntax for referring to an attribute is @*child.attribute*, where **lhs** and **loc** are special keywords to

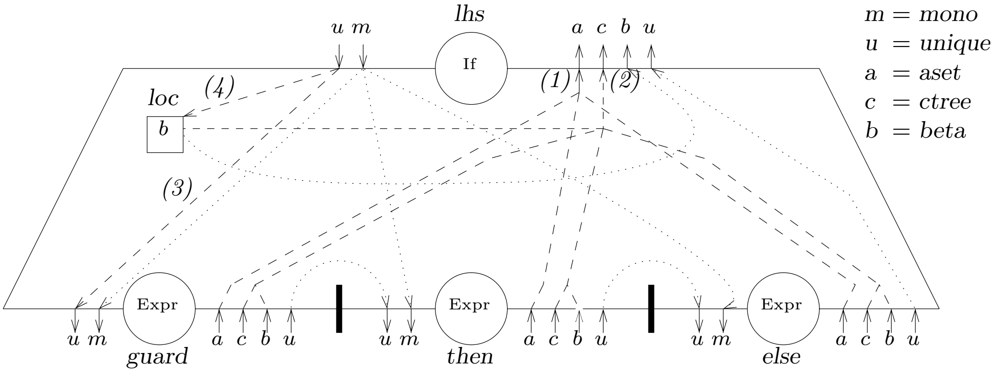

Fig. 6. Dependencies between the attributes for the conditional

refer to inherited attributes of the father node and attributes that are defined locally, respectively. The three type constraints for a conditional are added to the constraint trees of their corresponding subexpression with the function *add*. The number following a semantic equation refers to the correspondingly numbered dashed lines in Figure 6; the dotted edges represent the passing of unmodified attributes. For instance, *mono* is passed on unchanged to the three children. We do not have to write the code for passing these attributes ourselves. Instead, the compiler inserts 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*.

Flattening the constraint tree

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 specifying how to flatten the constraint tree we can imitate several type inferencing algorithms, each with their own properties and characteristics. Among the instances are the well-known algorithm $\mathcal{W}$, and the folklore algorithm $\mathcal{M}$.

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, but it is a straightforward extension to use different strategies depending on the non-terminal in the abstract syntax tree.

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

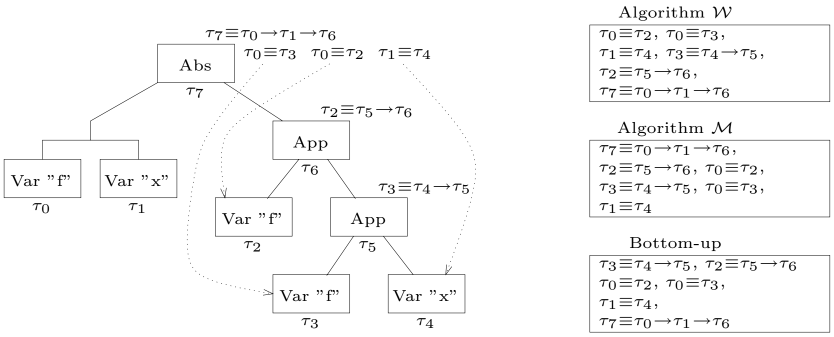
$\tau_7 \equiv \tau_0 \to \tau_1 \to \tau_6$
$\tau_0 \equiv \tau_3$
$\tau_0 \equiv \tau_2$
$\tau_1 \equiv \tau_4$
Abs
τ_7
$\tau_2 \equiv \tau_5 \to \tau_6$
App
τ_6
Var "f"
τ_0
Var "x"
τ_1
$\tau_3 \equiv \tau_4 \to \tau_5$
Var "f"
τ_2
App
τ_5
Var "f"
τ_3
Var "x"
τ_4
Algorithm $\mathcal{W}$
$\tau_0 \equiv \tau_2$, $\tau_0 \equiv \tau_3$,
$\tau_1 \equiv \tau_4$, $\tau_3 \equiv \tau_4 \to \tau_5$,
$\tau_2 \equiv \tau_5 \to \tau_6$,
$\tau_7 \equiv \tau_0 \to \tau_1 \to \tau_6$
Algorithm $\mathcal{M}$
$\tau_7 \equiv \tau_0 \to \tau_1 \to \tau_6$,
$\tau_2 \equiv \tau_5 \to \tau_6$, $\tau_0 \equiv \tau_2$,
$\tau_3 \equiv \tau_4 \to \tau_5$, $\tau_0 \equiv \tau_3$,
$\tau_1 \equiv \tau_4$
Bottom-up
$\tau_3 \equiv \tau_4 \to \tau_5$, $\tau_2 \equiv \tau_5 \to \tau_6$
$\tau_0 \equiv \tau_2$, $\tau_0 \equiv \tau_3$,
$\tau_1 \equiv \tau_4$,
$\tau_7 \equiv \tau_0 \to \tau_1 \to \tau_6$

Fig. 7. Spreading the constraints for $\lambda f\ x \to f\ (f\ x)$

spread. Instead of decorating the node where the actual binding takes place, each single constraint can instead be mapped onto the location of the bound variable. Figure 7 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.

5 Solving the Constraints

In this section we consider a number of implementations for 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 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 inference algorithms such as $\mathcal{W}$ and $\mathcal{M}$.

A type class for solving constraints

To pave the way for multiple implementations to solve a list of constraints, we present a type class. Since it is convenient to maintain a state while solving the constraints, we have implemented this class using a *State* monad that contains a counter to generate 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*. Now we present the algorithm that does the job.

```
solve :: Solver solver info ⇒ Int → Constraints info → State solver info ()
solve unique constraints = do setUnique unique
                              initialize
                              mapM solveOne constraints
                              makeConsistent
```

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 adds error messages to the state. The code fragment in Figure 8 shows how to solve a single constraint. (We have omitted *info* from the body of the function for clarity.)

```
solveOne :: Solver solver info ⇒ Constraint info → State solver info ()
solveOne constraint = case constraint of
   t1 ≡ t2 →
      do unifyTypes info t1 t2
   tp ⪯ ts →
      do unique ← getUnique
         let (unique', its) = instantiate unique ts
         setUnique unique'
         newVariables [unique..unique'-1]
         solveOne (tp ≡ its)
   t1 ≤m t2 →
      do makeConsistent
         t2' ← applySubst t2
         m' ← mapM applySubst m
         let scheme = generalize (ftv m') t2'
         solveOne (t1 ⪯ scheme)
```

Fig. 8. *solveOne*

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. Finally, we solve an explicit instance constraint that is constructed from the generalized type.

Please note that due to laziness in `Haskell` the list of constraints generated by a given 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 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 a constraint would result in an inconsistent state, then it is ignored, 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.

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 $\tau_1 \rightarrow \tau_1 \equiv \mathit{Bool} \rightarrow \tau_2$ 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

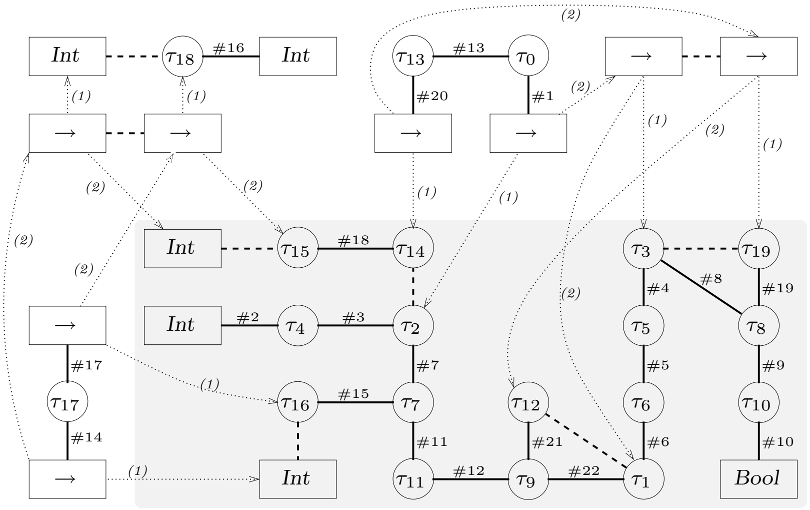

Fig. 9. TypeGraph

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 2. Consider the following ill-typed program.

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

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	$\tau_9 \equiv \tau_1$

Figure 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. The recursive call in the program is responsible for the two derived edges that complete a cycle. In general, edges that are part of a cycle are likely to be considered valid since their removal will not change the equivalence class.

```
(2,12): Type error in conditional
Expression     : if y then x else f (x - 1) y
Term           : y
Type           : Int
Does not match : Bool
```

(a)

```
(1,9): Type error in rhs of function binding
Expression     : y
Type           : Bool
Does not match : Int
```

(b)

```
(2,19): Type error in conditional
Expression     : if y then x else f (x - 1) y
Term           : x
Type           : Int
Does not match : Bool
```

(c)

```
ERROR (line 2): Type error in conditional
Expression     : if y then x else f (x - 1) y
Term           : f (x - 1) y
Type           : Bool
Does not match : Int
```

(d)

```
Example.hs:2:
    Couldn't match 'Bool' against 'Int'
       Expected type: Bool
       Inferred type: Int
    In the definition of 'f': if y then x else f (x - 1) y
```

(e)

Fig. 10. Type error messages

6 An Example

In this section we present possible error messages for Example 2. The code in this example is erroneous, but it is not obvious which constraint, or subexpression, is the source of the error. Notwithstanding, we do have the feeling that the reporting of some subexpressions is to be preferred over others.

One option to make the type graph in Figure 9 consistent is to remove constraint #10. Because this constraint was constructed at the node of a conditional to make the guard expression have type Bool, we report the error message in Figure 10(a). Similarly, one might think that #9 is a proper candidate for removal. Together with #19, it corresponds to the binding of the two variables by the pattern variable y in the second function binding. However, it is the context of the pattern variable (the second argument of f, captured by #8) that causes the inconsistency. Another alternative is to consider #6 to be invalid, which results in the error message shown in Figure 10(b). Finally, a conditional expression must have the same type as its then-branch (#12). Figure 10(c) shows a message which focuses on this inconsistency. Contrast this with the fact that the then-branch is a location where most traditional type inference algorithms cannot detect an inconsistency. The three error messages so far indicate that besides being able to report multiple independent type error messages (the removal of multiple edges in a type graph), it can be helpful to have several alternative error messages available for the same error. This makes it more likely that a programmer realizes what his mistake is.

That type classes, and in particular the overloading of numerical literals, obscure error messages and affect the reported location, is illustrated by the following message produced by Hugs.

```
ERROR "Ex.hs" (line 1): Instance of Num Bool required for definition of f
```

Figure 10(d) shows the Hugs error message where the literals are explicitly typed to make a fair comparison, because otherwise type classes would muddle the picture.

The message points to the else-branch, which can be misleading. In the type graph, this corresponds to the removal of constraint #21, which is part of a cycle. This error message is not a good one, because replacing *f (x-1) y* with *undefined* does not even yield a type correct program. The error reported by the Glasgow Haskell Compiler comes closest to #22. The error message (see Figure 10(e)) points to such a large expression that it is difficult to find the actual source of the error. Note that a smaller subexpression could have been indicated (for instance Figure 10(c)).

7 Conclusion and Future Work

In this paper we have shown how type inferencing can be implemented in a generic way to allow for experimentation with various methods of type inferencing. Along the way we have shown how such a general inferencing framework can be made with a minimum of work using the `UU_AG` system, and we described various instantiations of the framework. The example of the previous section shows that flexibility can indeed be obtained.

This year, we plan to use the `Helium` compiler in a concrete educational setting. As a result, we expect to obtain a collection of type-erroneous programs written by novice students. From this collection we can extract various error messages that might be given, and measure to what extent they help in finding the source of the error.

References

[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 Doaitse Swierstra. Generalizing Hindley-Milner type inference algorithms. Technical Report UU-CS-2002-031, Dept. of Comp. Sci, Universiteit Utrecht, 2002. http://www.cs.uu.nl/research/techreps/UU-CS-2002-031.html.

[LM01] Daan Leijen and Erik Meijer. Parsec: Direct style monadic parser combinators for the real world. Technical Report UU-CS-2001-35, Dept. of Comp. Sci, Universiteit Utrecht, 2001. http://www.cs.uu.nl/people/daan/parsec.html.

[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.

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

[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.