Generalizing Hindley-Milner Type Inference Algorithms

Article · August 2002

Source: CiteSeer

Bastiaan Heeren

Open Universiteit Nederland

Jurriaan Hage

Utrecht University

S. Doaitse Swierstra

Utrecht University

Generalizing Hindley-Milner Type Inference Algorithms

Bastiaan Heeren

Jurriaan Hage

Doaitse Swierstra

institute of information and computing sciences, utrecht university

technical report UU-CS-2002-031

www.cs.uu.nl

Generalizing Hindley-Milner Type Inference Algorithms

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

July 8, 2002

Abstract

Type inferencing according to the standard algorithms $\mathcal{W}$ and $\mathcal{M}$ often yields uninformative error messages. Many times, this is a consequence of a bias inherent in the algorithms. The method developed here is to first collect constraints from the program, and to solve these afterwards, possibly under the influence of a heuristic. We show the soundness and completeness of our algorithm. The algorithms $\mathcal{W}$ and $\mathcal{M}$ turn out to be deterministic instances of our method, giving the correctness for $\mathcal{W}$ and $\mathcal{M}$ with respect to the Hindley-Milner typing rules for free. We also show that our algorithm is more flexible, because it naturally allows the generation of multiple messages.

1 Introduction

Type systems are indispensable in modern higher-order, polymorphic languages. An important contribution to the popularity of `Haskell` and `ML` is their advanced type system, which enables detection of ill-typed expressions at compile-time. Modern language processors use type inference techniques that are derived from the algorithm $\mathcal{W}$ proposed by Milner [Mil78], and are based on the unification of types.

Because the error messages of most compilers and interpreters are often hard to interpret, programmer productivity is hampered. Also, programmers who are new to the language are likely to be discouraged from using it. Unfortunately, it is not straightforward to change unification-based systems to produce clear type error messages. One serious problem is that type conflicts might be detected far away from the site of the error. Another problem is that the location where an inconsistency is detected is influenced by the order in which types are unified. Unification-based type systems have a bias to report type conflicts near the end of the program. This *left-to-right bias* is caused by the way unification and substitution are used. McAdam [McA98] proposes a modification of $\mathcal{W}$ in which substitutions are unified, such that this bias is removed.

Consider the following program:

$$(\lambda x \rightarrow x + 1)\ ((\lambda y \rightarrow\ \mathbf{if}\ y\ \mathbf{then}\ \mathit{True}\ \mathbf{else}\ \mathit{False})\ \mathit{False})$$

Algorithm $\mathcal{M}$, which is described in detail by Lee and Yi [LY98], will report that the expression *True* is not compatible with the type *Int*. Following the advice of the report, a novice user might change *True* into some integer resulting in another type error, because then the first

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

occurrence of *False* will be in error. Algorithm $\mathcal{W}$ will report that the argument to the function $(\lambda x \rightarrow x + 1)$ is of type *Bool* and not *Int* as expected. This does not help much in finding out what exactly is wrong. In this paper we present an approach to type inferencing to remedy this shortcoming.

In our method, a set of constraints on types is generated for an expression. These constraints are typically generated locally, although they can describe global properties. The separation of *constraint generation* (the specification of the analysis) and *constraint resolution* (the implementation) is standard in this field of research [Aik99]. Since we are no longer forced to solve the constraints while they are generated, the system does not necessarily have a left-to-right bias: the order in which the constraints are solved is (almost) arbitrary. Sulzmann et al.[SMZ99] describe a different approach to express the Hindley-Milner type system in constraint form.

Heuristics can be used to remove inconsistencies in the final set of constraints, or, viewed in another way, heuristics can be used to determine the order in which the constraints are solved. At the end of the paper we give an example of such a heuristic, which was inspired by the approach of Johnson and Walz in [WJ86]. However, the larger part of the paper concentrates on formally describing our method and proving it correct. We also indicate how $\mathcal{W}$ and $\mathcal{M}$ can be understood as deterministic instances of our general method.

Several papers present algorithms to capture information about the deductive steps of a type inference algorithm to construct a better explanation for a type conflict [Wan86, BS93, DB96, McA00]. Tracing the reason for a deduction is similar to collecting constraints on types.

In Section 6 we explain how to construct a type graph from the collected constraints. Other techniques have been proposed to store type information in a graph [GVS96, Cho95]. However, these two systems cannot handle polymorphism. The graph presented by McAdam [McA00] can contain let-expressions, but the polymorphic instances of a declaration are obtained by duplicating parts of the graph. This approach can result in computing the type of an expression exponentially many times.

This paper is organised as follows. In the next section, an expression and type language is presented, for which we give the standard Hindley-Milner type inference rules and algorithm $\mathcal{W}$ in Section 3. In Section 4 we give our Bottom-Up type inference rules for collecting type constraints, and show how the latter can be solved. After proving the soundness and completeness of our algorithm in Section 5, we give examples of heuristics for deciding which constraints have to be removed in order to make a constraint set consistent in Section 6. Finally, we summarize our findings and indicate where our research shall go in the future.

2 Preliminaries

We first introduce a small, let-polymorphic, functional language which is the core of popular languages like `Haskell` and `ML`.

$$\text{(expression)} \qquad E \ := \ x \ \mid \ E_1\ E_2 \ \mid \ \lambda x \rightarrow E \ \mid \ \textbf{let}\ x = E_1\ \textbf{in}\ E_2$$

To keep things simple, recursive declarations in a let-construct are not permitted. In other words, the scope of a variable declared in a let expression is limited to the body and does not include the definition part. We deliberately restrict ourselves to this small expression language since extensions, such as recursion, patterns, and explicit type definitions, can be added in a straightforward way. However, the expressions, that serve as example throughout this paper, can contain literals. A literal is a value with its own constant type, e.g. *True* has type *Bool*, and *1* has type *Int*. The syntax of *types* and *type schemes* is given by:

$$\begin{array}{lrcl} \text{(type)} & \tau & := & \alpha \mid \mathit{Int} \mid \mathit{Bool} \mid \mathit{String} \mid \tau_1 \rightarrow \tau_2 \\ \text{(type scheme)} & \sigma & := & \forall \vec{\alpha}.\tau \end{array}$$

A type can be either a type variable, a type constant, or a function type. This rather basic representation of types can easily be extended to include other types, such as lists and user defined data types. However, these extensions do not touch the essence of the inference algorithms.

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. 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 $\mathit{freevars}(\tau)$ and simply consists of all type variables in τ. Additionally, $\mathit{freevars}(\forall\vec{\alpha}.\tau) = \mathit{freevars}(\tau) - \vec{\alpha}$.

A substitution, usually denoted by $\mathcal{S}$, is a mapping of type variables to types. For a set of type variables $\{\alpha_1, \ldots, \alpha_n\}$ we write $[\alpha_1 := \tau_1,\ \alpha_2 := \tau_2,\ \ldots,\ \alpha_n := \tau_n]$. When a substitution is applied, the type variables that are not in the domain of the substitution are to remain unchanged. Consequently, the empty substitution, written as [ ], behaves in a way similar to the identity function. As usual, a substitution only replaces free type variables, so the quantified type variables in a type scheme are not affected by a substitution.

We assume substitutions to be *idempotent*, which implies that $\mathcal{S}(\mathcal{S}\tau) = \mathcal{S}\tau$. For instance, $[\alpha_1 := \alpha_2,\ \alpha_2 := \mathit{Bool}]$ is considered to be an invalid substitution. Every substitution can be transformed into an equivalent idempotent substitution if it fulfils the *occur check* (see [BN98]). For our purposes, substitutions that do not pass the occur check are defined to be equal to the error substitution, denoted by $\top$. We define $\top\tau$ to be equal to the error type, which we also denote by $\top$. The composition of substitution $\mathcal{S}_1$ followed by substitution $\mathcal{S}_2$ is written as $(\mathcal{S}_2 \circ \mathcal{S}_1)$ and is again a substitution. If either of the two equals $\top$, then the composition equals $\top$ as well.

A type environment, usually denoted by Γ, maps variables to their corresponding type schemes, and provides a context in which type inferencing takes place. By $(\Gamma\backslash x)$ we denote the type environment Γ where the variable x is removed from the domain of Γ:

$$\Gamma\backslash x \quad =_{def} \quad \{y\!:\!\sigma \mid y\!:\!\sigma \in \Gamma,\ x \neq y\}$$

We lift the notion of free type variables to environments Γ, by taking the union of the set of free type variables occurring in the type schemes, which are in the range of Γ. The environment $\mathcal{S}\Gamma$ is equal to $\{x\!:\!S\sigma \mid x\!:\!\sigma \in \Gamma\}$.

Generalizing a type τ with respect to a type environment Γ entails the quantification of the type variables that are free in τ but do not occur in Γ:

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

In the literature, generalization is sometimes referred to as determining the closure of a type. In fact, these different notations, e.g. $\overline{\Gamma(\tau)}$ in [DM82] and $\mathit{Clos}_\Gamma(\tau)$ in [LY98], all express the

$$\frac{\tau \prec \Gamma(x)}{\Gamma \ \vdash_{\mathrm{HM}} \ x : \tau} \qquad [\textsc{Var}]_{\textsc{hm}}$$

$$\frac{\Gamma \ \vdash_{\mathrm{HM}} \ e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \ \vdash_{\mathrm{HM}} \ e_2 : \tau_1}{\Gamma \ \vdash_{\mathrm{HM}} \ e_1 \ e_2 : \tau_2} \qquad [\textsc{App}]_{\textsc{hm}}$$

$$\frac{\Gamma \backslash x \cup \{x : \tau_1\} \ \vdash_{\mathrm{HM}} \ e : \tau_2}{\Gamma \ \vdash_{\mathrm{HM}} \ \lambda x \rightarrow e : \tau_1 \rightarrow \tau_2} \qquad [\textsc{Abs}]_{\textsc{hm}}$$

$$\frac{\Gamma \ \vdash_{\mathrm{HM}} \ e_1 : \tau_1 \qquad \Gamma \backslash x \cup \{x : \mathit{generalize}(\Gamma, \tau_1)\} \ \vdash_{\mathrm{HM}} \ e_2 : \tau_2}{\Gamma \ \vdash_{\mathrm{HM}} \ \mathbf{let} \ x = e_1 \ \mathbf{in} \ e_2 : \tau_2} \qquad [\textsc{Let}]_{\textsc{hm}}$$

Figure 1: Hindley-Milner type inference rules

same. An instantiation of a type scheme is obtained by the replacement of the quantified type variables by fresh type variables:

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

A type τ_1 is a *generic instance* of a type scheme $\sigma = \forall \vec{\alpha}.\tau_2$ if there exists a substitution $\mathcal{S}$ with $\mathcal{S}\beta = \beta$ for all $\beta \in \mathit{freevars}(\sigma)$ such that τ_1 and $\mathcal{S}\tau_2$ are syntactically equal.

For two types τ_1 and τ_2, $\mathit{mgu}(\tau_1, \tau_2)$ returns the most general unifier, which is a substitution. By definition, it holds for a unifier $\mathcal{S}$ that $\mathcal{S}\tau_1 = \mathcal{S}\tau_2$. Note that if the types τ_1 and τ_2 cannot be unified, $\mathit{mgu}(\tau_1, \tau_2) = \top$.

3 The Hindley-Milner Type Inference Rules

Damas and Milner [DM82] present a set of inference rules, which is shown in Figure 1. There is exactly one rule for each of the four language constructs in the expression language. These rules deal with judgements of the form $\Gamma \ \vdash_{\mathrm{HM}} \ e : \tau$, expressing that expression e can be assigned a type τ under the type environment Γ. However, given e and Γ, there may exist multiple types that validate the assertion. Fortunately, there is exactly one type scheme of which all valid types are generic instances, and this is referred to as the *principle type scheme* of the expression. For instance, the identity function has the principle type scheme $\forall \alpha . \alpha \rightarrow \alpha$, and can therefore also be assigned the types $\mathit{Bool} \rightarrow \mathit{Bool}$ and $(\tau \rightarrow \tau) \rightarrow \tau \rightarrow \tau$. The rules provide an easy-to-use framework to construct a proof that a certain type can be assigned to an expression, but they cannot classify a type as invalid for a given expression, nor do they suggest an approach on how to find a valid judgement.

In the literature, several algorithms are discussed that compute the principle type (scheme) of an expression under a type environment. The best known algorithm is the bottom-up algorithm $\mathcal{W}$ shown in Figure 2. The correctness of this algorithm with respect to the type inference rules is proven in [DM82]. A second implementation of the Hindley-Milner type system is a folklore top-down algorithm named $\mathcal{M}$. Algorithm $\mathcal{M}$ has the property that it stops earlier for an ill-typed expression than $\mathcal{W}$ does. Earlier means that it has visited fewer

$$\mathcal{W} :: \mathit{TypeEnvironment} \times \mathit{Expression} \rightarrow \mathit{Substitution} \times \mathit{Type}$$

$$\begin{array}{lcllcl}
\mathcal{W}(\Gamma, x) & = & \multicolumn{4}{l}{([\,], \mathit{instantiate}(\sigma)), \text{ where } (x\!:\!\sigma) \in \Gamma} \\
\mathcal{W}(\Gamma, \lambda x \rightarrow e) & = & \textbf{let} & (\mathcal{S}_1, \tau_1) & = & \mathcal{W}(\Gamma\backslash x \cup \{x\!:\!\beta\}, e), \text{ fresh } \beta \\
 & & \textbf{in} & \multicolumn{3}{l}{(\mathcal{S}_1, \mathcal{S}_1\beta \rightarrow \tau_1)} \\
\mathcal{W}(\Gamma, e_1\ e_2) & = & \textbf{let} & (\mathcal{S}_1, \tau_1) & = & \mathcal{W}(\Gamma, e_1) \\
 & & & (\mathcal{S}_2, \tau_2) & = & \mathcal{W}(\mathcal{S}_1\Gamma, e_2) \\
 & & & \mathcal{S}_3 & = & \mathit{mgu}(\mathcal{S}_2\tau_1, \tau_2 \rightarrow \beta), \text{ fresh } \beta \\
 & & \textbf{in} & \multicolumn{3}{l}{(\mathcal{S}_3 \circ \mathcal{S}_2 \circ \mathcal{S}_1, \mathcal{S}_3\beta)} \\
\mathcal{W}(\Gamma, \textbf{let } x = e_1 \textbf{ in } e_2) & = & \textbf{let} & (\mathcal{S}_1, \tau_1) & = & \mathcal{W}(\Gamma, e_1) \\
 & & & (\mathcal{S}_2, \tau_2) & = & \mathcal{W}(\mathcal{S}_1\Gamma\backslash x \cup \{x\!:\!\mathit{generalize}(\mathcal{S}_1\Gamma, \tau_1)\}, e_2) \\
 & & \textbf{in} & \multicolumn{3}{l}{(\mathcal{S}_2 \circ \mathcal{S}_1, \tau_2)}
\end{array}$$

Figure 2: Algorithm $\mathcal{W}$

nodes in the abstract syntax tree, and does not refer to the amount of work that has been done. The proof of this property and the correctness of $\mathcal{M}$ is given in [LY98].

Although type inference algorithms are primarily designed to find the principle type scheme of an expression, they are also used to detect ill-typed expressions. As a result, the reported error message and its associated location strongly depend on the order in which the type inference algorithm unifies types. Combining the error sites of $\mathcal{W}$ and $\mathcal{M}$ can lead to more informative messages. In general, an analysis of contradicting sites provides a better insight into what is the most likely origin of a type error. In the next section we introduce a new set of typing rules that allows global type inferencing, providing even more flexibility.

4 Type Inferencing with Constraints

In this section we discuss a type inference algorithm that differs from algorithm $\mathcal{W}$ in two aspects. First, a set of constraints on types is collected instead of constructing a substitution. Postponing the unification of types paves the way for a more global approach to type inferencing allowing for a more finely tuned method of choosing the invalidating constraints. The second difference is the absence of a type environment Γ under which the type inferencing takes place. Instead, an assumption set is used to record the type variables that are assigned to the occurrences of free variables. The bottom-up construction of both the constraint set and the assumption set is a compositional computation that follows the shape of the abstract syntax tree.

4.1 Type Constraints

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

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

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 sorts of constraints are used to cope with

$$\{x:\beta\},\ \emptyset\ \vdash_{\mathrm{BU}}\ x:\beta \qquad [\textsc{Var}]_{\mathrm{BU}}$$

$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\mathrm{BU}}\ e_1:\tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\mathrm{BU}}\ e_2:\tau_2}{\mathcal{A}_1\cup\mathcal{A}_2,\ \mathcal{C}_1\cup\mathcal{C}_2\cup\{\tau_1\equiv\tau_2\rightarrow\beta\}\ \vdash_{\mathrm{BU}}\ e_1\ e_2:\beta} \qquad [\textsc{App}]_{\mathrm{BU}}$$

$$\frac{\mathcal{A},\ \mathcal{C}\ \vdash_{\mathrm{BU}}\ e:\tau}{\mathcal{A}\backslash x,\ \mathcal{C}\cup\{\tau'\equiv\beta \mid x:\tau'\in\mathcal{A}\}\ \vdash_{\mathrm{BU}}\ \lambda x\rightarrow e:(\beta\rightarrow\tau)} \qquad [\textsc{Abs}]_{\mathrm{BU}}$$

$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\mathrm{BU}}\ e_1:\tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\mathrm{BU}}\ e_2:\tau_2}{\mathcal{A}_1\cup\mathcal{A}_2\backslash x,\ \mathcal{C}_1\cup\mathcal{C}_2\cup\{\tau'\leq_M\tau_1 \mid x:\tau'\in\mathcal{A}_2\}\ \vdash_{\mathrm{BU}}\ \mathbf{let}\ x=e_1\ \mathbf{in}\ e_2:\tau_2} \qquad [\textsc{Let}]_{\mathrm{BU}}$$

Figure 3: Bottom-Up type inference rules

polymorphism that is 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 type inferencing an expression. 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 polymorphic type variables.

4.2 The Bottom-Up Type Inference Rules

The Bottom-Up type inference rules (henceforth abbreviated as the Bottom-Up rules), given in Figure 3, deal with judgements of the form $\mathcal{A},\ \mathcal{C}\ \vdash_{\mathrm{BU}}\ e:\tau$. Assumption set $\mathcal{A}$ records the type variables that are assigned to the free variables of e. Contrary to the standard type environment Γ, as used in the Hindley-Milner inference rules, there can be multiple (different) assumptions for a given variable. The rules are such that they allow for flexibility in coping with unbound identifiers. Advantages of this property are discussed in Nikhil [Nik85] and Bernstein [Ber95].

The inference rule for a variable is very straightforward: a fresh type variable β is introduced and returned as the type. We assume that at any time there are infinitely many fresh type variables available. The fact that β was assigned to the variable is recorded in the assumption set. The constraint set is empty.

A new type variable β is introduced to represent the type of an application of two expressions. An equality constraint ensures that the domain and the range of the type of the first expression match with the type of the second expression and β respectively. Furthermore, the collected constraints for the subexpressions are passed on unchanged, and the two assumption sets are merged.

The fresh β in the inference rule for a lambda abstraction represents the type of the lambda bound variable. An equality constraint is generated for each type variable in the assumption set that is associated with the variable that is bound by the lambda. The assumptions that concern this variable are removed from the assumption set.

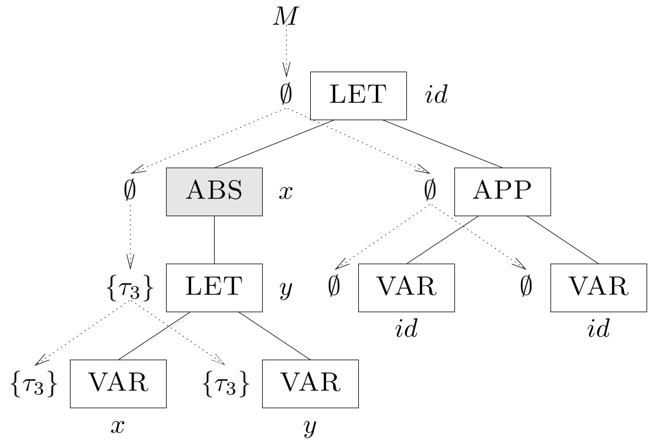


Figure 4: The top-down computation of the monomorphic sets

Unsurprisingly, it is the let-expression that introduces polymorphism and brings in some difficulties. Inferring a type for a let-expression implies a specific order in which the types of the two subexpressions have to be computed. This order is reflected in the Hindley-Milner inference rules: the inferred type of the declaration is generalized before it is added to the type environment under which the type of the body is inferred. An implicit instance constraint is generated for each variable in the body that becomes bound. Although there is no order in the set of constraints, an implicit instance constraint requires some constraints to be solved before it becomes solvable. In Section 4.3 we discuss why this is necessary.

An implicit instance constraint depends on the context of the declaration. In particular, it depends on the monomorphic type variables of unbound variables. Every node in the abstract syntax tree has a set of monomorphic type variables M. To compute the monomorphic sets, a single top-down computation is sufficient. For an arbitrary expression, the set M contains exactly the type variables that were introduced by a lambda abstraction at a higher level in the abstract syntax tree. We have left the distribution of M implicit in the Bottom-Up rules. We confine ourselves to an example.

Example 1 Consider the following expression:

$$\begin{array}{ll} \textbf{let } id = \lambda x \rightarrow & \textbf{let } y = x \\ & \textbf{in } y \\ \textbf{in } id\ id & \end{array}$$

In Figure 4 we have depicted the abstract syntax tree for this expression, and indicated the top-down computation of the monomorphic sets. The type variable τ_3 is introduced for the variable x in the lambda abstraction rooted at the marked node in the tree. The type of the declaration for id is polymorphic in τ_3, whereas the type of y is monomorphic in τ_3. This follows from the fact that the set of monomorphic type variables is empty in the outer let, whereas it contains τ_3 in the inner let.

Example 2 To illustrate the application of the typing rules, we consider the type inferencing problem for the following expression:

```
λm → let y = m
         in let x = y True
            in x
```

The variable m is introduced in a lambda abstraction, and therefore all the occurrences of m in the body must have the same monomorphic type. Although y is introduced in a let-declaration, it is assigned the (monomorphic) type of m. Moreover, the application of y to the literal *True* requires m to be a function of type $\mathit{Bool} \rightarrow a$ for some a. The abstract syntax tree determines which Bottom-Up rules should be applied. This results in the following deduction tree:

$$\dfrac{\dfrac{\dfrac{}{\{m{:}\tau_1\},\ \emptyset \vdash_{\text{BU}}\ m{:}\tau_1}\,\text{VAR} \qquad \dfrac{\dfrac{\dfrac{}{\{y{:}\tau_2\},\ \emptyset \vdash_{\text{BU}}\ y{:}\tau_2}\,\text{VAR} \quad \dfrac{}{\emptyset,\ \emptyset \vdash_{\text{BU}}\ \mathit{True}{:}\mathit{Bool}}\,\text{LIT}}{\{y{:}\tau_2\},\ \mathcal{C}_1 \vdash_{\text{BU}}\ y\ \mathit{True}{:}\tau_3}\,\text{APP} \qquad \dfrac{}{\{x{:}\tau_4\},\ \emptyset \vdash_{\text{BU}}\ x{:}\tau_4}\,\text{VAR}}{\{y{:}\tau_2\},\ \mathcal{C}_2 \vdash_{\text{BU}}\ \textbf{let}\ x = y\ \mathit{True}\ \textbf{in}\ x{:}\tau_4}\,\text{LET}}{\{m{:}\tau_1\},\ \mathcal{C}_3 \vdash_{\text{BU}}\ \textbf{let}\ y = m\ \textbf{in let}\ x = y\ \mathit{True}\ \textbf{in}\ x{:}\tau_4}\,\text{LET}}{\emptyset,\ \mathcal{C}_4 \vdash_{\text{BU}}\ \lambda m \rightarrow \textbf{let}\ y = m\ \textbf{in let}\ x = y\ \mathit{True}\ \textbf{in}\ x{:}\tau_5 \rightarrow \tau_4}\,\text{ABS}$$

where the constraint sets are given as

$$\begin{array}{lcl@{\qquad}lcl} \mathcal{C}_1 & = & \{\tau_2 \equiv \mathit{Bool} \rightarrow \tau_3\} & \mathcal{C}_2 & = & \mathcal{C}_1 \cup \{\tau_4 \leq_{\{\tau_5\}} \tau_3\} \\ \mathcal{C}_3 & = & \mathcal{C}_2 \cup \{\tau_2 \leq_{\{\tau_5\}} \tau_1\} & \mathcal{C}_4 & = & \mathcal{C}_3 \cup \{\tau_5 \equiv \tau_1\} \end{array}$$

The set at the root of the tree contains four constraints, among which are two implicit instance constraints in which the type variable τ_5 (assigned to the lambda bound variable m) is monomorphic.

4.3 Solving Type Constraints

After the generation of a constraint set, a substitution is constructed that satisfies each constraint in the set. Satisfaction of a constraint by a substitution is defined as follows:

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

After substitution, the two types of an equality constraint should be syntactically equal. For instance, the most general unifier can be chosen to satisfy an equality constraint. For an implicit instance constraint, the substitution is not only applied to both types, but also to the set of monomorphic type variables M. (In Example 3, presented later in this section, it shall become clear why the substitution should also be applied to 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 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.

Note that for two types τ_1 and τ_2

$$\begin{array}{lcl} \mathcal{S} \text{ satisfies } \tau_1 \equiv \tau_2 & \Longleftrightarrow & \mathcal{S} \text{ satisfies } \tau_1 \preceq \tau_2 \qquad (1) \\ \mathcal{S} \text{ satisfies } \tau_1 \equiv \tau_2 & \Longleftrightarrow & \mathcal{S} \text{ satisfies } \tau_1 \leq_{\mathit{freevars}(\tau_2)} \tau_2 \qquad (2) \\ \mathcal{S} \text{ satisfies } \tau_1 \leq_M \tau_2 & \Longleftrightarrow & \mathcal{S} \text{ satisfies } \tau_1 \preceq \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau_2) \qquad (3) \end{array}$$

The first two properties show that every equality constraint can be written as an instance constraint of either type. These properties hold because the only generic instance of a type is the type itself, and because $\mathit{generalize}(\mathit{freevars}(\tau), \tau)$ equals τ for all types τ. Property (3) is justified by the fact that substitution $\mathcal{S}$ is idempotent, from which follows that $\mathcal{S}(\mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau))$ is equal to $\mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau)$.

When applied to a constraint set, a substitution is simply applied to the types and type schemes therein. For implicit instance constraints, we make note of the fact that the substitution also has to be applied to the sets of monomorphic type variables:

$$\mathcal{S}(\tau_1 \leq_M \tau_2) \quad =_{def} \quad \mathcal{S}\tau_1 \leq_{\mathcal{S}M} \mathcal{S}\tau_2$$

First, we define which type variables in a constraint set are active.

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

Next, we present a function, which returns a substitution that satisfies a given a set of constraints. As we shall explain later in more detail, the algorithm assumes that the constraint set always contains a constraint which can be solved. In particular, if it contains only implicit instance constraints, then there is one for which its condition is fulfilled.

$$\begin{array}{lcl}
\multicolumn{3}{l}{\textsc{Solve} :: \mathit{Constraints} \rightarrow \mathit{Substitution}} \\
\textsc{Solve}\ (\emptyset) & = & [\,] \\
\textsc{Solve}\ (\{\tau_1 \equiv \tau_2\} \cup \mathcal{C}) & = & \textsc{Solve}\ (\mathcal{SC}) \circ \mathcal{S} \\
 & & \qquad \textbf{where } \mathcal{S} = \mathit{mgu}(\tau_1, \tau_2) \\
\textsc{Solve}\ (\{\tau_1 \leq_M \tau_2\} \cup \mathcal{C}) & = & \textsc{Solve}\ (\{\tau_1 \preceq \mathit{generalize}(M, \tau_2)\} \cup \mathcal{C}) \\
 & & \qquad \textbf{if } (\mathit{freevars}(\tau_2) - M) \cap \mathit{activevars}(\mathcal{C}) = \emptyset \\
\textsc{Solve}\ (\{\tau \preceq \sigma\} \cup \mathcal{C}) & = & \textsc{Solve}\ (\{\tau \equiv \mathit{instantiate}(\sigma)\} \cup \mathcal{C})
\end{array}$$

Example 3 Consider the constraints that were collected for Example 2. We show how a substitution is constructed from this set.

$$\begin{array}{ll}
 & \textsc{Solve}\ (\{\tau_2 \equiv \mathit{Bool} \rightarrow \tau_3,\ \tau_4 \leq_{\{\tau_5\}} \tau_3,\ \tau_2 \leq_{\{\tau_5\}} \tau_1,\ \tau_5 \equiv \tau_1\}) \\
= & \textsc{Solve}\ (\{\tau_4 \leq_{\{\tau_5\}} \tau_3,\ \mathit{Bool} \rightarrow \tau_3 \leq_{\{\tau_5\}} \tau_1,\ \tau_5 \equiv \tau_1\}) \quad \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\{\tau_4 \leq_{\{\tau_1\}} \tau_3,\ \mathit{Bool} \rightarrow \tau_3 \leq_{\{\tau_1\}} \tau_1\}) \quad \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\{\tau_4 \leq_{\{\tau_1\}} \tau_3,\ \mathit{Bool} \rightarrow \tau_3 \preceq \tau_1\}) \quad \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\{\tau_4 \leq_{\{\tau_1\}} \tau_3,\ \mathit{Bool} \rightarrow \tau_3 \equiv \tau_1\}) \quad \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\{\tau_4 \leq_{\{\tau_3\}} \tau_3\}) \quad \circ [\tau_1 := \mathit{Bool} \rightarrow \tau_3] \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\{\tau_4 \preceq \tau_3\}) \quad \circ [\tau_1 := \mathit{Bool} \rightarrow \tau_3] \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\{\tau_4 \equiv \tau_3\}) \quad \circ [\tau_1 := \mathit{Bool} \rightarrow \tau_3] \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & \textsc{Solve}\ (\emptyset) \quad \circ [\tau_4 := \tau_3] \circ [\tau_1 := \mathit{Bool} \rightarrow \tau_3] \circ [\tau_5 := \tau_1] \circ [\tau_2 := \mathit{Bool} \rightarrow \tau_3] \\
= & [\tau_4 := \tau_3,\ \tau_1 := \mathit{Bool} \rightarrow \tau_3,\ \tau_5 := \mathit{Bool} \rightarrow \tau_3,\ \tau_2 := \mathit{Bool} \rightarrow \tau_3]
\end{array}$$

Notice that the sets of monomorphic type variables are modified while the substitution is constructed. Applying this substitution to $\tau_5 \rightarrow \tau_4$ results in $(\mathit{Bool} \rightarrow \tau_3) \rightarrow \tau_3$, which is the most general type for the expression. If the substitution had not been applied to the monomorphic sets of the implicit instance constraints, an incorrect type would be returned.

The recursive definition of Solve consists of a basic case (the empty constraint set, for which the empty substitution is returned), and a transformation for each of the three types of constraints. Solve does not assume any ordering of the constraints. However, the condition accompanying an implicit instance constraint imposes a certain ordering: all constraints that involve type variables occurring in the right hand side of $\tau_1 \leq_M \tau_2$, but that do not and never will occur in M, should be handled before the implicit instance constraint itself can be handled. Intuitively, this means that before inferring the body of a let, we ought to have finished inferring the types of the let-definitions that are used in the body. As discussed in Example 3, it would have been incorrect to solve the constraint $\tau_2 \leq_{\{\tau_5\}} \tau_1$ before handling $\tau_5 \equiv \tau_1$, because the semantics of $\tau_2 \leq_{\{\tau_5\}} \tau_1$ and $\tau_2 \leq_{\{\tau_1\}} \tau_1$ are quite different. Our side condition prevents this from happening, by insisting that τ_1 should not be active anymore. This is the case after we have solved $\tau_5 \equiv \tau_1$.

It is possible that in a constraint set, two implicit instance constraints depend on each other in this way. A simple example is $\mathcal{C} = \{\tau_1 \leq_\emptyset \tau_2,\ \tau_2 \leq_\emptyset \tau_1\}$, because τ_1 and τ_2 are both active. In this case, our algorithm blocks, because no constraint can be solved and the set of constraints is not empty. Note, however, that the empty substitution satisfies $\mathcal{C}$. It is not difficult to see that for constraint sets generated by our Bottom-Up rules, this kind of circularity is impossible:

Lemma 1 Let $\mathcal{C}$ be a set of constraints generated by the Bottom-Up rules. During the execution of Solve($\mathcal{C}$), every non-empty constraint set passed to Solve contains a constraint that can be solved.

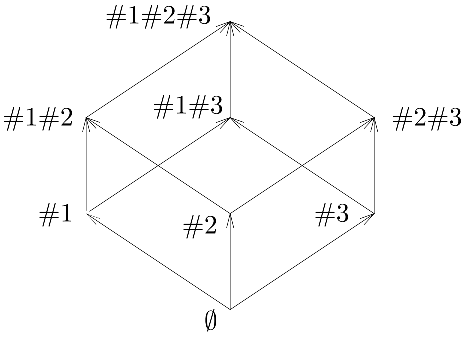
Proof If there are still equality constraints or explicit instance constraints, then we can solve these. Now consider all let-expressions which gave rise to implicit instance constraints that we still have to solve. Among these let-expressions, we can always solve the implicit instance constraints belonging to the outermost, leftmost let-expression. This holds for the following reason: the Bottom-Up rules introduce disjoint sets of type variables in disjoint trees (in particular, the one for the definition of the let and the one for the body of the let), and the only way that these type variables can ever get related is through a type variable that is introduced higher in the tree. This type variable is necessarily monomorphic and hence present in the set of monomorphic type variables at that point. □

Our method of solving the instance constraints introduces new constraints of another type. However, it is easy to show that after solving a constraint the measure function (n_1, n_2, n_3), where n_1, n_2, and n_3 are the number of implicit instance constraints, explicit instance constraints, and equality constraints respectively, strictly decreases with respect to the lexicographic order.

Lemma 2 The function Solve terminates for all inputs.

4.4 Correctness of Algorithm Solve

Lemma 3 If Solve returns a substitution for a constraint set $\mathcal{C}$, then each constraint in $\mathcal{C}$ is satisfied by this substitution. Moreover, this substitution is the most general substitution (up to type variable renaming).



$\mathcal{C}$	SOLVE($\mathcal{C}$)
$\emptyset$	[ ]
#1	$[v_1 := \mathit{Bool}]$
#2	$[v_2 := v_3 \to v_3]$
#3	$[v_3 := v_1]$
#1#2	$[v_1 := \mathit{Bool}, v_2 := v_3 \to v_3]$
#1#3	$[v_1 := \mathit{Bool}, v_3 := \mathit{Bool}]$
#2#3	$[v_2 := v_1 \to v_1, v_3 := v_1]$
#1#2#3	$[v_1 := \mathit{Bool}, v_2 := \mathit{Bool} \to \mathit{Bool}, v_3 := \mathit{Bool}]$

Figure 5: A partially ordered set on substitutions

Proof The empty substitution is the most general unifier that satisfies the empty constraint set. If we restrict ourselves to equality and explicit instance constraints, then Solve returns the most general substitution that satisfies the constraint set $\mathcal{C}$. This is because we use the most general unifier to solve equality constraints, and because of the fact that although we use function composition to combine our separate substitutions, we apply $\mathcal{S}$ to the remaining constraints before we continue the solving process. Furthermore, the following properties hold:

$$\begin{array}{lcl} \mathcal{S}_1 \text{ satisfies } \{\tau_1 \equiv \tau_2\} & \Longleftrightarrow & \text{for all } \mathcal{S}_2\text{: } \mathcal{S}_2 \circ \mathcal{S}_1 \text{ satisfies } \{\tau_1 \equiv \tau_2\} \\ \mathcal{S}_1 \text{ satisfies } \{\tau \preceq \sigma\} & \Longleftrightarrow & \text{for all } \mathcal{S}_2\text{: } \mathcal{S}_2 \circ \mathcal{S}_1 \text{ satisfies } \{\tau \preceq \sigma\} \end{array}$$

The second, more tricky part of the problem concerns the solving of implicit instance constraints. We have to show that solving such a constraint does not interfere with the solving of the other constraints. We consider the situation $\mathcal{C} \cup \{\tau_1 \leq_M \tau_2\}$ where $(\mathit{freevars}(\tau_2) - M) \cap \mathit{activevars}(\mathcal{C}) = \emptyset$. In other words, the implicit instance constraint is now solvable. We claim that $(\textsc{Solve}(\mathcal{C}'))(\beta) = \beta$ for all $\beta \in \mathit{freevars}(\tau_2) - M$, where $\mathcal{C}' = \mathcal{C} \cup \{\tau_1 \preceq \mathit{generalize}(M, \tau_2)\}$. The reason for this is that if β still exists in $\mathcal{C}'$, then it exists only in the right hand side of another implicit instance constraint without being part of the corresponding monomorphic set. In other words, β continues to be inactive until all the implicit instance constraints containing it have been solved. Now it follows that β is mapped to β, because it is not anymore present in the constraint set. Please note that type variables that have become inactive, i.e., elements of $\mathit{freevars}(\mathcal{C}) - \mathit{activevars}(\mathcal{C})$, can never again become active. In general, the following property is valid for generalization:

$$(\mathit{freevars}(\tau) - M) \cap \mathit{dom}(\mathcal{S}) = \emptyset \quad \Longrightarrow \quad \mathcal{S}(\mathit{generalize}(M, \tau)) = \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau)$$

Consequently, as soon as the condition of an implicit instance constraint is met, we can transform it into an explicit instance constraint. □

Example 4 Consider the quasi-order $\lesssim$ on substitutions as defined by Baader and Nipkow [BN98], where $\mathcal{S}_1 \lesssim \mathcal{S}_2$ denotes that $\exists \mathcal{S} : \mathcal{S}_2 = \mathcal{S} \circ \mathcal{S}_1$. If $\mathcal{S}_1 \lesssim \mathcal{S}_2$, we say that $\mathcal{S}_1$ is more general than $\mathcal{S}_2$. Substitutions are considered to be equal under the renaming of type variables. For instance, $[\tau_1 := \tau_2] \sim [\tau_2 := \tau_1]$. This order describes a complete lattice on substitutions, where the least element $\bot$ is the empty substitution [ ] and the greatest element $\top$ is a special case that represents the error substitution.

Consider the three constraints $\#1 : v_1 \equiv \mathit{Bool}$, $\#2 : v_2 \equiv v_3 \to v_3$, and $\#3 : v_3 \equiv v_1$. The $\#n$ notation is only introduced to label the constraints for later reference. In Figure 5 we have the part of the complete lattice of substitutions in which the various substitutions that can arise by solving the constraints in any order, are given. Note that the part in question forms a complete sublattice.

4.5 A Type Inference Algorithm

The algorithm SOLVE has now been proven to be correct. In this section we give an algorithm, which can be more easily compared to $\mathcal{W}$ and $\mathcal{M}$. In the latter cases, type inferencing takes place in the context of a type environment, where variables are paired with a type scheme. The same effect can be obtained by constructing an extra set of constraints based on the type environment and the assumption set, and pass them on to SOLVE together with the set of constraints collected by the Bottom-Up rules.

For this purpose, the definition of $\preceq$ is lifted to sets:

$$\mathcal{A} \preceq \Gamma = \{\ \tau \preceq \sigma \ \mid\ x\!:\!\tau \in \mathcal{A},\ x\!:\!\sigma \in \Gamma\ \}$$

Because the definition does not restrict the type environment Γ to have at most one type scheme for a variable, the definition is more general than required.

Example 5 Let $\mathcal{A}$ be $\{id\!:\!\tau_6,\ id\!:\!\tau_7,\ f\!:\!\tau_8\}$, and let Γ be $\{id\!:\!\forall\alpha.\alpha \to \alpha,\ f\!:\!\tau_1 \to \tau_1\}$. Then $\mathcal{A} \preceq \Gamma$ is equal to $\{\tau_6 \preceq \forall\alpha.\alpha \to \alpha,\ \tau_7 \preceq \forall\alpha.\alpha \to \alpha,\ \tau_8 \preceq \tau_1 \to \tau_1\}$. Note that the last constraint is semantically equal to $(\tau_8 \equiv \tau_1 \to \tau_1)$.

The following properties can be derived from the definition of $\preceq$.

$$\begin{array}{lcll}
(\mathcal{A}_1 \cup \mathcal{A}_2) \preceq \Gamma & = & (\mathcal{A}_1 \preceq \Gamma) \cup (\mathcal{A}_2 \preceq \Gamma) & (4)\\
\mathcal{A} \preceq (\Gamma_1 \cup \Gamma_2) & = & (\mathcal{A} \preceq \Gamma_1) \cup (\mathcal{A} \preceq \Gamma_2) & (5)\\
\mathcal{A} \preceq \Gamma \backslash x & = & \mathcal{A} \backslash x \preceq \Gamma & (6)\\
\mathcal{A} \preceq \{x\!:\!\tau\} & = & \{\tau' \equiv \tau \mid x\!:\!\tau' \in \mathcal{A}\} & (7)
\end{array}$$

The first two properties express that $\preceq$ distributes over union. Property (6) states that it is irrelevant in which of the two sets an occurrence of a variable is removed. The last property lifts Property (1) to sets.

With these ingredients, we present an algorithm to compute a type for an expression under a type environment Γ.

```
InferType(Γ, e) =
$\mathcal{A}, \mathcal{C} \vdash_{BU} e : \tau$
if $dom(\mathcal{A}) \not\subseteq dom(\Gamma)$ then report undefined variables exist
else  $\mathcal{S} = \text{Solve}(\mathcal{C} \cup \mathcal{A} \preceq \Gamma)$
      return $(\mathcal{S},\ \mathcal{S}\tau)$
```

Besides the inferred type, INFERTYPE also returns a substitution, which gives INFERTYPE and $\mathcal{W}$ the same signature. An advantage of this is that the substitution reveals the types of all expressions that were assigned a fresh type variable by the Bottom-Up rules. Because there is no distinction between the type variables that were introduced while applying the inference rules, and the (monomorphic) type variables that occur in the initial type environment Γ, a substitution can change the types in Γ.

The choice which rule to apply in SOLVE is nondeterministic, and we have just proved that all possible ways of solving the constraints result in the most general substitution that satisfies the constraints.

We are now interested in comparing INFERTYPE with the algorithms $\mathcal{M}$ and $\mathcal{W}$ and show that these are in fact deterministic instances. For the moment we restrict ourselves to $\mathcal{W}$ and show how each constraint generated by our Bottom-Up rules, can be mapped to a node in the abstract syntax tree where $\mathcal{W}$ would "solve" it. Something similar can be done for $\mathcal{M}$ and, in fact, could be done for algorithm $\mathcal{G}$ as defined in [LY00].

We consider the Bottom-Up rules one by one. The case for $[\text{VAR}]_{\text{BU}}$ is trivial, because no constraints are generated. For applications, the equality constraint generated in an application node is solved by $\mathcal{W}$ in the same node by means of unification after the subexpressions have been inferenced (note that $\mathcal{W}$ performs its unifications in postorder). Consider a lambda abstraction $\lambda x \to e$. At this lambda, the bottom-up algorithm generates an equality constraint for each occurrence of x in e. However, $\mathcal{W}$ solves these constraints not in the lambda node, but at the moment that it reaches each of the occurrences of x: $\mathcal{W}$ passes, as it were, the constraints down to the variables by means of the type environment. Hence the constraints generated in the lambda node are mapped to the occurrence of the bound identifier that they belong to. Similar to the lambda abstraction, we can pass the implicit instance constraints generated for x defined by a let expression down to the uses of x in the body of the let.

It is easy to see that after $\mathcal{W}$ has finished inferencing the definition of a let, the condition of each of the implicit instance constraints generated by the let is fulfilled. The line of reasoning is similar to that in Lemma 1. Having mapped the constraints of the Bottom-Up algorithm to (possibly other) nodes in the tree, a postorder traversal (as done by $\mathcal{W}$), can resolve the constraints in the order that it encounters them, where we may assume that the implicit instance constraints are solved "directly", i.e., we convert them to explicit instance constraints, to equality constraints, and then use unification. As a result, we have

Theorem 4 Algorithm $\mathcal{W}$ is a deterministic instance of INFERTYPE.

5 Soundness and Completeness

In this section we consider the soundness and completeness of algorithm INFERTYPE with respect to the Hindley-Milner inference rules. We start with a lemma on the relation between the value of the type environment Γ in a given node of the abstract syntax tree on the one hand, and the types in $\mathcal{S}M$ on the other (where $\mathcal{S}$ satisfies the collected constraints).

Lemma 5 (monomorphic type variables) For a type environment Γ and a typable expression e, let $(\Gamma \ \vdash_{\text{HM}} \ e : \tau)$ and $(\mathcal{A},\ \mathcal{C} \ \vdash_{\text{BU}} \ e : \tau')$ be the corresponding deduction trees. Let $\mathcal{S}$ be a substitution that satisfies $\mathcal{C}$ and $(\mathcal{A} \preceq \Gamma)$, and which unifies the types at corresponding nodes of the two deduction trees. Then for corresponding nodes, it holds that $\mathit{freevars}(\mathcal{S}\Gamma) = \mathit{freevars}(\mathcal{S}M)$.

Proof To simplify the proof, we assume that no shadowing occurs since it is straightforward to rename variables. For example, $(\lambda x \to \lambda x \to x)$ can be transformed into $(\lambda x \to \lambda y \to y)$. At the root of the abstract syntax tree we choose M such that it contains exactly the free type variables in $\mathcal{S}\Gamma$. Each pair of deduction rules, i.e., a Hindley-Milner type rule and its corresponding Bottom-Up deduction rule, preserves this invariant. Both the variable case and the case of function application are trivial.

For a lambda abstraction, the set of monomorphic type variables M is extended with the fresh type variable β that was introduced in $[\textsc{Abs}]_{\text{BU}}$. Because $\mathcal{S}(\beta \to \tau)$ equals $\mathcal{S}(\tau_1 \to \tau_2)$, the set of free type variables in $\mathcal{S}(\Gamma \backslash x \cup \{x : \tau_1\})$ and $\mathcal{S}(M \cup \{\beta\})$ are the same. The removal of x and its type from Γ does not alter the set of free type variables, since we do not allow shadowing. For a let-expression, the type environment in which the type of the declaration is inferred is unchanged, and so is the set of monomorphic type variables. The type environment for the body is extended with the closure of the type of the declaration. However, this type scheme cannot introduce free type variables, that is,

$$\mathit{freevars}(\Gamma) = \mathit{freevars}(\Gamma \backslash x \cup \{x : \mathit{generalize}(\Gamma, \tau_1)\}). \qquad \square$$

To prove that the combination of our Bottom-Up rules and the algorithm InferType is sound, we show that if InferType(Γ, e) succeeds with $(\mathcal{S}, \tau)$, then $\mathcal{S}\Gamma \ \vdash_{\text{HM}} \ e : \mathcal{S}\tau$ can be derived. In this proof we use properties of Solve and the Bottom-Up typing rules. An alternative, more detailed proof of this theorem can be found in Appendix A.

Theorem 6 (Soundness) If $\mathcal{A}, \ \mathcal{C} \ \vdash_{\text{BU}} \ e : \tau$, then for all Γ and $\mathcal{S}$ such that

- $\mathit{dom}(\mathcal{A}) \subseteq \mathit{dom}(\Gamma)$
- $\mathcal{S}$ satisfies $\mathcal{C}$
- $\mathcal{S}$ satisfies $\mathcal{A} \preceq \Gamma$

it holds that $\mathcal{S}\Gamma \ \vdash_{\text{HM}} \ e : \mathcal{S}\tau$.

Proof A judgement $\mathcal{A}, \ \mathcal{C} \ \vdash_{\text{BU}} \ e : \tau$ can be found by applying the Bottom-Up rules to any expression e, including ill-typed expressions. The assumption set $\mathcal{A}$ contains one variable-type pair for each free occurrence of a variable in e. Consequently, a type environment Γ should provide a type for each variable present in $\mathcal{A}$. This is ensured by the condition $\mathit{dom}(\mathcal{A}) \subseteq \mathit{dom}(\Gamma)$. The condition $\mathcal{S}$ satisfies $\mathcal{C}$ should be obvious: we can only expect a typing according to the Hindley-Milner rules if the collected constraints are satisfied. The third constraint ensures that the variables present in $\mathcal{A}$ have obtained a type (to be found in $\mathcal{S}$) that is consistent with the types in the type environment Γ. This condition restricts the combinations of $\mathcal{S}$ and Γ for which the conclusion holds. Clearly, choosing $\mathcal{S}(\tau_1) = \mathit{Bool}$ while $x : \tau_1 \in \mathcal{A}$ and $\Gamma(x) = \forall a.a \to a$ is not a valid choice.

The theorem can be proved by induction on the structure of the expression. The base case of the induction is a variable x, for which the judgement $\{x : \beta\}, \ \emptyset \ \vdash_{\text{BU}} \ e : \beta$ can be derived. The empty constraint set does not impose any restriction on the substitution. $\Gamma(x)$ is defined because x is in the domain of the assumption set. Substitution $\mathcal{S}$ should satisfy $\{x : \beta\} \preceq \Gamma$, which implies that $\mathcal{S}\beta$ is a generic instance of $\mathcal{S}\Gamma(x)$. As a result, judgement $\mathcal{S}\Gamma \ \vdash_{\text{HM}} \ x : \mathcal{S}\beta$ is derivable.

The inductive cases are application, lambda abstraction, and let-expressions. For a subexpression e_1, the induction hypothesis states that there are an $\mathcal{A}_1$, $\mathcal{C}_1$, and τ_1, such that $\mathcal{A}_1, \ \mathcal{C}_1 \ \vdash_{\text{BU}} \ e_1 : \tau_1$ is valid. In other words, a subexpression e_1 can always be decorated with an assumption set, a constraint set, and a type, using the Bottom-Up rules. Secondly, the induction hypothesis expresses that for all pairs $(\mathcal{S}, \Gamma)$ that meet the three condition imposed by the theorem, the type $\mathcal{S}\tau_1$ can be derived with the Hindley-Milner inference rules for expression e_1 in type environment $\mathcal{S}\Gamma$. The strategy to construct a proof for the three remaining inductive cases is as follows. Firstly, introduce an arbitrary substitution $\mathcal{S}$, together with a type environment Γ, for which we assume that the three conditions hold. Secondly, fulfil the conditions for each of the subexpressions, where the same substitution is used, but possibly a different type environment. Some planning is required to choose the *right* Γ. By now, a Hindley-Milner judgement is obtained for each subexpression. Thirdly, combine the judgements and apply the corresponding Hindley-Milner inference rule to get the desired judgement.

If the expression at hand is a function e_1 applied to an argument e_2, we can assume that $\mathcal{A}_1,\ \mathcal{C}_1 \vdash_{\mathrm{BU}} e_1 : \tau_1$ and $\mathcal{A}_2,\ \mathcal{C}_2 \vdash_{\mathrm{BU}} e_2 : \tau_2$. It follows from $[\textsc{App}]_{\mathrm{BU}}$ that $\mathcal{A},\ \mathcal{C} \vdash_{\mathrm{BU}} e_1\ e_2 : \tau$ holds, where $\mathcal{A}$ is $\mathcal{A}_1 \cup \mathcal{A}_2$, $\mathcal{C}$ is $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \to \beta\}$, and τ is β. Consider each substitution $\mathcal{S}$ and each type environment Γ such that $\mathcal{S}$ satisfies $\mathcal{C} \cup \mathcal{A} \preceq \Gamma$ and $dom(\mathcal{A}) \subseteq dom(\Gamma)$. In general it is true that if $\mathcal{S}$ satisfies $\mathcal{C}$, then $\mathcal{S}$ also satisfies each subset of $\mathcal{C}$. As a result, $\mathcal{S}$ satisfies $\mathcal{C}_1 \cup (\mathcal{A}_1 \preceq \Gamma)$. In addition $dom(\mathcal{A}_1) \subseteq dom(\Gamma)$, and therefore by induction we acquire the judgement $\mathcal{S}\Gamma \vdash_{\mathrm{HM}} e_1 : \mathcal{S}\tau_1$. In a similar way $\mathcal{S}\Gamma \vdash_{\mathrm{HM}} e_2 : \mathcal{S}\tau_2$ can be obtained. Since $\mathcal{S}$ must satisfy the constraint $\tau_1 \equiv \tau_2 \to \beta$, $\mathcal{S}\tau_1$ is syntactically equal to $\mathcal{S}\tau_2 \to \mathcal{S}\beta$. From this observation we conclude that $\mathcal{S}\Gamma \vdash_{\mathrm{HM}} e_1\ e_2 : \mathcal{S}\beta$ holds, which completes the proof in case the expression is an application.

Decorating the body of a lambda abstraction with the bottom-up algorithm results in the judgement $\mathcal{A},\ \mathcal{C} \vdash_{\mathrm{BU}} e : \tau$. Applying the appropriate rule justifies $\mathcal{A}\backslash x,\ \mathcal{C}' \vdash_{\mathrm{BU}} \lambda x \to e : \beta \to \tau$, where $\mathcal{C}'$ is $\mathcal{C} \cup \{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}$. We consider each $\mathcal{S}$ and Γ such that $dom(\mathcal{A}\backslash x) \subseteq dom(\Gamma)$ and $\mathcal{S}$ satisfies $\mathcal{C}' \cup \mathcal{A}\backslash x \preceq \Gamma$. The first assumption about Γ implies that also $dom(\mathcal{A}) \subseteq dom(\Gamma\backslash x \cup \{x : \tau\})$ holds. Because $\mathcal{S}$ satisfies the constraints $\{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}$, the types associated with x in $\mathcal{A}$ are all equivalent after applying the substitution. In other words, the types that were introduced at the variables, which are bound by the lambda abstraction, are unified. $\mathcal{S}$ also satisfies $\mathcal{A} \preceq \{x : \beta\}$, since $\tau_1 \preceq \tau_2$ (notice the monomorphic type on the right) is equal to $\tau_1 \equiv \tau_2$. Merging two environments as in Property (5) yields that $\mathcal{A} \preceq \Gamma\backslash x \cup \{x : \beta\}$ is satisfied by $\mathcal{S}$. By induction we get $\mathcal{S}(\Gamma\backslash x \cup \{x : \beta\}) \vdash_{\mathrm{HM}} e : \mathcal{S}\tau$, which implies $\mathcal{S}\Gamma \vdash_{\mathrm{HM}} \lambda x \to e : \mathcal{S}(\beta \to \tau)$.

The judgement $\mathcal{A},\ \mathcal{C} \vdash_{\mathrm{BU}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \tau_2$ can be inferred, where $\mathcal{A}$ is $\mathcal{A}_1 \cup \mathcal{A}_2\backslash x$, $\mathcal{C}$ is $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leq_M \tau_1 \mid x : \tau' \in \mathcal{A}_2\}$, and assuming the declaration and the body of the let-expression to have judgements $\mathcal{A}_1,\ \mathcal{C}_1 \vdash_{\mathrm{BU}} e_1 : \tau_1$ and $\mathcal{A}_2,\ \mathcal{C}_2 \vdash_{\mathrm{BU}} e_2 : \tau_2$ respectively. Consider all substitutions $\mathcal{S}$ and all type environments Γ that fulfil the three conditions. By induction we get $\mathcal{S}\Gamma \vdash_{\mathrm{HM}} e_1 : \mathcal{S}\tau_1$. The essential step to prove the let-rule sound is to observe that $\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq \{x : \mathit{generalize}(\mathcal{S}\Gamma, \mathcal{S}\tau_1)\}$. This holds because $\mathcal{S}$ satisfies $\mathcal{C}$, and because, according to Lemma 5, generalizing type τ_1 with respect to the free type variables in $\mathcal{S}\Gamma$ is the same as generalization with respect to the free type variables in $\mathcal{S}M$. By induction we get $\mathcal{S}(\Gamma\backslash x \cup \{x : \mathit{generalize}(\mathcal{S}\Gamma, \tau_1)\}) \vdash_{\mathrm{HM}} e_2 : \mathcal{S}\tau_2$, from which $\mathcal{S}\Gamma \vdash_{\mathrm{HM}} \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 : \mathcal{S}\tau_2$ can be concluded. □

We have shown that every satisfiable constraint set implies the existence of a Hindley-Milner derivation with the same result type. We now turn to the complementary result where we prove the completeness of our method: if there is a successful Hindley-Milner derivation, then our method will infer a type that is at least as general as the type derived using the Hindley-Milner rules. In Appendix B we present a more detailed proof of this theorem.

Theorem 7 (Completeness) If $\Gamma \vdash_{\mathrm{HM}} e : \tau$ then $\mathcal{A},\ \mathcal{C} \vdash_{\mathrm{BU}} e : \tau'$ and there exists an $\mathcal{S}$

such that

- $\mathcal{S}$ satisfies $\mathcal{C}$
- $\mathcal{S}$ satisfies $\mathcal{A} \preceq \Gamma$
- $\mathcal{S}\tau = \mathcal{S}\tau'$

Proof Consider the collection of well-typed expressions under a type environment Γ. For such an expression e, we can construct a derivation tree, where the type τ in the root of the tree is an instance of the (unique) principle type scheme for the expression. Decorating the expression according to the Bottom-Up rules, we obtain the judgement $\mathcal{A},\ \mathcal{C}\ \vdash_{\mathrm{BU}}\ e : \tau'$ for some τ'. The types τ and τ' are independent; τ is derived with Hindley-Milner, whereas a substitution that satisfies each constraint in $\mathcal{C}$ still has to be applied to τ'. In our proof, the substitution $\mathcal{S}$ also bridges the gap between the fact that $S\tau'$ is the principal type scheme of e, whereas this is not necessarily the case for τ in the Hindley-Milner derivation.

The proof proceeds by induction on the structure of the expressions. Our induction hypothesis is somewhat stronger than the statement of the theorem. The condition that $\mathcal{S}$ satisfies $\mathcal{A} \preceq \Gamma$ is in fact $[\,]$ satisfies $(\mathcal{S}\mathcal{A}) \preceq \Gamma$, in other words, $\mathcal{S}$ is not allowed to modify Γ. The third condition similarly becomes $\tau = \mathcal{S}\tau'$. This simplifies the proof.

If the expression is a single variable x, then the substitution that satisfies the three conditions can be constructed in a straightforward way. Choose $\mathcal{S}$ to be $[\beta \mapsto \tau]$, where β is the fresh type variable assigned to x by the Bottom-Up rules, and τ is the type that is returned by the Hindley-Milner rules, and which therefore is an instance of the type scheme that is provided by Γ for x. Clearly, $\mathcal{S}$ satisfies the empty constraint set, $\mathcal{S}$ satisfies $\{x : \beta\} \preceq \Gamma$, and $\mathcal{S}\beta = \tau$.

A substitution for an application is constructed by composing the substitutions of the two subexpressions, and additionally map the fresh type variable β to the range of the function type that was derived for the function expression. The domains of the substitutions are independent because the type variables mentioned in the two subtrees are disjoint.

The fresh type variable β in the Bottom-Up rule for lambda abstractions is used to unify several type variables that were assigned to the occurrences of the abstracted variable in the scope of the lambda. Substitution $\mathcal{S}$ maps this type variable to the type τ_1, which was assigned to the variable x and used to extend Γ. For the other type variables we will use the substitution obtained by induction. Because $\mathcal{S}$ satisfies $\mathcal{A} \preceq \{x : \tau_1\}$, $\mathcal{S}$ also satisfies $\{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}$.

It is sufficient to combine the two substitutions that can be obtained from the declaration and the body, also because no fresh type variables are introduced. The only interesting condition to verify for this composed substitution $\mathcal{S}$ is to check whether $\mathcal{S}$ satisfies the created implicit instance constraints. The induction hypothesis results in $\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq \{x : \mathit{generalize}(\Gamma, \tau_1)\}$ for the body, and $\mathcal{S}\tau_1' = \tau_1$ for the declaration. Because generalizing over $\mathcal{S}\Gamma$ and $\mathcal{S}M$ is equivalent (Lemma 5), we get that $\mathcal{S}$ satisfies $\{\tau' \preceq \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau_1') \mid x : \tau' \in \mathcal{A}_2\}$. Applying Property (3) results in satisfaction of $\{\tau' \leq_M \tau_1' \mid x : \tau' \in \mathcal{A}_2\}$ by $\mathcal{S}$.

We conclude that if $\Gamma\ \vdash_{\mathrm{HM}}\ e : \tau$ is derivable, then $\textsc{InferType}(\Gamma, e)$ returns the most general type for expression e under type environment Γ. □

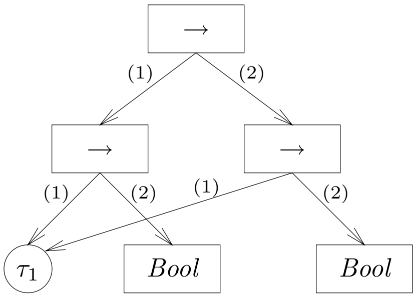


Figure 6: Term graph for $(\tau_1 \rightarrow \mathit{Bool}) \rightarrow \tau_1 \rightarrow \mathit{Bool}$

6 Unbiased Constraint Solving

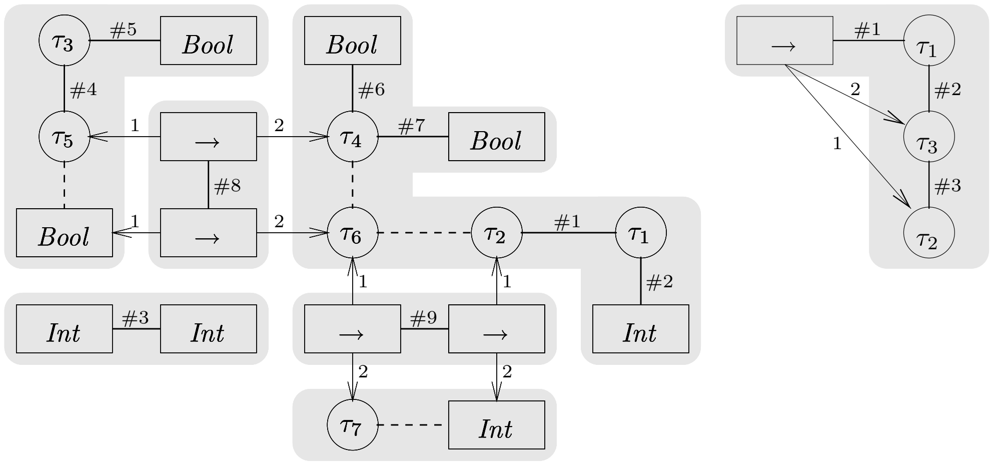
In Section 4 we presented a nondeterministic algorithm to solve a set of constraints that was constructed according to the inference rules. The order in which the constraints are solved determines the location where a type error is detected and reported. In the process of type inferencing, this dependency can result in a bias. In this section we discuss an alternative approach to solve a set of type constraints and for which there is no bias. The method is based on the construction of a *type graph* inspired by the path graphs described by Port [Por88].

6.1 Construction of the Type Graph

We start with replacing explicit instance constraints by equality constraints. Each type scheme is instantiated, which results in the introduction of new type variables in the constraint set. Similar to algorithm SOLVE, an implicit instance constraint imposes a condition when it can be dealt with. We will postpone discussing how these constraints are handled in the graph. For now we consider solving a set of equality constraints.

First, each type that occurs in an equality constraint is added to a directed *term graph*. Each vertex 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. Figure 6 depicts the term graph for $(\tau_1 \rightarrow \mathit{Bool}) \rightarrow \tau_1 \rightarrow \mathit{Bool}$.

In addition to the term graph, a second graph is constructed, which has the same set of vertices, and which identifies the equivalence classes of types. This undirected graph administrates 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. Two types are in the same equivalence class if their corresponding vertices are connected. 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 classes, and this might lead to the insertion of more derived edges.



(a) (b)

Figure 7: Two examples of a type graph

Example 6 Consider the following ill-typed expression that was mentioned in the introduction:

$$(\lambda x \rightarrow x + 1)\ ((\lambda y \rightarrow \textbf{if}\ y\ \textbf{then}\ \mathit{True}\ \textbf{else}\ \mathit{False})\ \mathit{False})$$

Extending the expression language and the typing rules to support + and conditionals results in the assignment of the type τ_7 to this expression, with the following constraints:

$$\begin{array}{llll@{\qquad}llll@{\qquad}llll}
\#1 & \tau_1 & \equiv & \tau_2 & \#4 & \tau_3 & \equiv & \tau_5 & \#7 & \tau_4 & \equiv & \mathit{Bool} \\
\#2 & \tau_1 & \equiv & \mathit{Int} & \#5 & \tau_3 & \equiv & \mathit{Bool} & \#8 & \tau_5 \rightarrow \tau_4 & \equiv & \mathit{Bool} \rightarrow \tau_6 \\
\#3 & \mathit{Int} & \equiv & \mathit{Int} & \#6 & \tau_4 & \equiv & \mathit{Bool} & \#9 & \tau_2 \rightarrow \mathit{Int} & \equiv & \tau_6 \rightarrow \tau_7
\end{array}$$

Figure 7a shows the six equivalence classes for this constraint set, as indicated by the shaded areas. The graph combines the term graph and the path graph, and contains three kinds of edges. First, the directed edges are part of the term graph, and are labelled with the child number. Second, the solid edges are the *initial* equalities that correspond to the equality constraints in the set. These edges are annotated with the constraint number, where each constraint number appears exactly once. Finally, the *derived* equalities are depicted as dashed lines. For instance, the edge between the type variable τ_5 and the type constant *Bool* is the result of the two function types that are put in the same equivalence class by constraint #8. The equivalence classes are the connected components when considering the initial and the derived edges in the path graph.

We now discuss how the algorithm should deal with the implicit instance constraints. Before we can deal with the constraint $\tau_1 \leq_M \tau_2$, all the free type variables in τ_2 that do not occur in M (these are the type variables to be quantified) should be in a *fixed equivalence class*, that is, a consistent equivalence class that is not going to change while handling the remaining constraints. The implicit instance constraint is replaced by the explicit instance constraint $\tau_1 \preceq \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau_2)$, where the substitution $\mathcal{S}$ maps each type variable to the *representative* of its equivalence class. If an equivalence class contains a composed type, then this is the representative of the class. If there are only type variables, then one is chosen to represent the others.

The type graph can also be constructed for unsatisfiable constraint sets. This contributes to the advantage of graph unification over algorithm SOLVE, and over traditional algorithms such as $\mathcal{W}$ and $\mathcal{M}$. Because the constraint set is solved as a whole, the bias that is the result of an imposed order of the unifications is removed completely.

An inconsistency can show up in two ways. The most obvious case is when two different type constructors end up in the same equivalence class, e.g., *Bool* and *Int* in Figure 7a. The *error path* is the path that connects the incompatible types within an equivalence class. In the other case, there is no topological ordering of the equality classes with respect to the directed edges from the term graph, which indicates that there is an infinite type. Also an directed edge between two vertices in the same equivalence class is erroneous. Consider the ill-typed expression $\lambda x \rightarrow x\ x$, and its constraint set

$$\#1 : \tau_1 \equiv \tau_2 \rightarrow \tau_3,\ \#2 : \tau_1 \equiv \tau_3,\ \#3 : \tau_2 \equiv \tau_3.$$

The graph for this constraint set, which is shown in Figure 7b, reveals an infinite type.

6.2 Heuristics to Report Inconsistencies

At least one constraint for each error path should be removed to restore the consistency in an equality graph. Heuristics select the edges to be removed in the graph, and produce an appropriate error message. Traditional algorithms only report the first unification error that is detected, whereas the type graph allows reports of possibly independent errors.

We list a number of heuristics to report an inconsistency.

- Select an ordering of the constraints and report the location where the type inferencer detects an inconsistency. For example, in Figure 7a there are two error paths: $p_1 = \{\#2, \#1, \#9, \#8, \#6\}$ and $p_2 = \{\#2, \#1, \#9, \#8, \#7\}$. The constraint orders associated with the algorithms $\mathcal{W}$ and $\mathcal{M}$ report #9 and #6 respectively, since these are the constraints that complete an error path. However, removing #6 does not break error path p_2, which is reflected by the fact that replacing *True* in the *then*-branch by a value of type *Int*, as is suggested by the error message of $\mathcal{M}$, does not remove the type error. If wanted, we can continue solving the ordered constraints after detecting and reporting an inconsistency.

- Following the approach of Johnson and Walz in [WJ86], we can select the type equation in a constraint set that is the most likely source of error by counting the number of occurrences of each type constructor in an equivalence class. We assign a *removal cost* (or a weight) to each constraint, and then compute the maximal consistent set of constraints for which the total removal cost is minimal. Paths between compatible type constructors should be preserved if possible.

- The expression that is reported in a type error message is the location where one expects that the program should be modified. Consequently, replacing this expression by $\bot$ should remove the type inconsistency, where $\bot$ has the most general type $\forall a.a$. In Example 6, this can be both x (represented by τ_1) and the conditional (τ_6).

- In [Wan86, BS93, DB96], all deductive steps are maintained and interpreted to construct a sensible error message. Precisely the constraints that are mentioned in the error paths contribute to the ill-typedness of an expression, and therefore we claim that the information required for these analyses is captured in the type graph. McAdam [McA00] describes a different graph to store the information.

- Yang [Jun00] claims that conflicting sites should be reported rather than a single location. To incorporate this approach for a type graph only requires tracing the origin of each type constructor.

7 Conclusion

In this paper we have described a method for collecting constraints for type inferencing and have shown how to solve these. We have proved the method to be sound and complete with respect to the inference rules of Hindley-Milner. An important aspect of our method is that it encompasses the well-known algorithms $\mathcal{W}$ and $\mathcal{M}$, corresponding to certain orders in which the collected constraints are solved.

As an illustration of combining heuristics for generating suitable error messages with the constraint collecting Bottom-Up rules, we showed how a type graph can be constructed for an expression. This type graph can be used to generate multiple, independent error reports that are in some sense optimal.

With the main formal algorithm proven correct we can investigate the quality of the corresponding reported errors for various heuristics. We plan to do this by including our type inference algorithms into a `Haskell` compiler developed in our group. This compiler includes a large subset of `Haskell` (but excludes, e.g., type classes) and shall be used mainly in courses which teach `Haskell` to novice functional programmers. As a result, we hope to obtain a variety of programs reflecting the kind of type errors that are made by novice functional programmers.

Another point of investigation is into the resources necessary for implementing a heuristic. The ideal situation would be that the user can choose how many resources may be spent on finding good error messages.

Finally, we are interested in extending our language. One important extension for the type system is the introduction of *type* and *constructor* classes, which provide a way to overload functions. Using type synonyms in reported error messages will increase understanding, but also introduces new problems for the type system.

References

[Aik99] A. Aiken. Introduction to set constraint-based program analysis. In *Science of Computer Programming, 35(1)*, pages 79–111, 1999.

[Ber95] Karen Bernstein. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, November 1995. Technical Report.

[BN98] F. Baader and T. Nipkow. *Term Rewriting and All That.* Cambridge University Press, Cambridge, UK, 1998.

[BS93] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. In *ACM Letters on Programming Languages*, volume 2, pages 17–30, December 1993.

[Cho95] Venkatesh Choppella. Diagnosis of ill-typed programs, 1995. http://citeseer.nj.nec.com/choppella95diagnosis.html.

[DB96] D. Duggan and F. Bent. Explaining type inference. In *Science of Computer Programming 27*, pages 37–83, 1996.

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

[GVS96] M. Gandhe, G. Venkatesh, and A. Sanyal. Correcting errors in the curry system. In *Chandrum V. and Vinay, V. (Eds.): Proc. of 16th Conf. on Foundations of Software Technology and Theoretical Computer Science, LNCS vol. 1180, Springer-Verlag*, pages 347–358, 1996.

[Jun00] Yang Jun. Explaining type errors by finding the sources of type conflicts. In Greg Michaelson, Phil Trindler, and Hans-Wolfgang Loidl, editors, *Trends in Functional Programming*, pages 58–66. Intellect Books, 2000.

[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.

[LY00] Oukseh Lee and Kwangkeun Yi. A generalization of hybrid let-polymorphic type inference algorithms. In *Proceedings of the First Asian Workshop on Programming Languages and Systems*, pages 79–88, National university of Singapore, Singapore, December 2000.

[McA98] Bruce J. McAdam. On the Unification of Substitutions in Type Inference. In Kevin Hammond, Anthony J.T. Davie, and Chris Clack, editors, *Implementation of Functional Languages (IFL '98), London, UK*, volume 1595 of *LNCS*, pages 139–154. Springer-Verlag, September 1998.

[McA00] B. 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.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17:348–375, 1978.

[Nik85] R. S. Nikhil. Practical polymorphism. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 319–333. Springer-Verlag, Berlin, DE, 1985.

[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.

[SMZ99] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/milner style type systems in constraint form. Research Report ACRC–99–009, University of South Australia, School of Computer and Information Science, July 1999.

[Wan86] M. Wand. Finding the source of type errors. In *13th Annual ACM Symp. on Principles of Prog. Languages*, pages 38–43, January 1986.

[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.

A Soundness

We present a detailed proof of Theorem 6, in which we claim that the Bottom-Up type inference rules are sound with respect to the Hindley-Milner rules. We give the four cases for a proof by induction on the structure of the expressions.

A.1 Variable

Consider the following Bottom-Up type inference rule:

$$\{x : \beta\},\ \emptyset\ \vdash_{\mathrm{BU}}\ x : \beta \qquad [\textsc{Var}]_{\textsc{bu}}$$

Choose $\mathcal{S}$ *and* Γ *such that:*

1. $\mathit{dom}(\{x : \beta\}) \subseteq \mathit{dom}(\Gamma)$
2. $\mathcal{S}$ satisfies $\emptyset$
3. $\mathcal{S}$ satisfies $\{x : \beta\} \preceq \Gamma$

Then the following holds:

4.	$\mathcal{S}$ satisfies $\{\beta \preceq \Gamma(x)\}$	$(\preceq)$, (3)
5.	$\mathcal{S}\beta \prec \mathcal{S}\Gamma(x)$	$(\preceq)$, (4)
6.	$\mathcal{S}\Gamma\ \vdash_{\mathrm{HM}}\ x : \mathcal{S}\beta$	$[\textsc{Var}]_{\textsc{hm}}$, (5)

A.2 Application

Consider the following Bottom-Up type inference rule:

$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\mathrm{BU}}\ e_1 : \tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\mathrm{BU}}\ e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \rightarrow \beta\}\ \vdash_{\mathrm{BU}}\ e_1\ e_2 : \beta} \qquad [\textsc{App}]_{\textsc{bu}}$$

Choose $\mathcal{S}$ *and* Γ *such that:*

1. $\mathit{dom}(\mathcal{A}_1 \cup \mathcal{A}_2) \subseteq \mathit{dom}(\Gamma)$
2. $\mathcal{S}$ satisfies $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1 \equiv \tau_2 \rightarrow \beta\}$
3. $\mathcal{S}$ satisfies $(\mathcal{A}_1 \cup \mathcal{A}_2) \preceq \Gamma$

Then the following holds:

4.	$\mathit{dom}(\mathcal{A}_1) \subseteq \mathit{dom}(\Gamma)$	(1)
5.	$\mathcal{S}$ satisfies $\mathcal{C}_1$	(2)
6.	$\mathcal{S}$ satisfies $\mathcal{A}_1 \preceq \Gamma$	Property (4), (3)
7.	$\mathcal{S}\Gamma\ \vdash_{\mathrm{HM}}\ e_1 : \mathcal{S}\tau_1$	induction, (4, 5, 6)
8.	$\mathcal{S}$ satisfies $\{\tau_1 \equiv \tau_2 \rightarrow \beta\}$	(2)
9.	$\mathcal{S}\tau_1 = \mathcal{S}(\tau_2 \rightarrow \beta)$	$(\equiv)$, (8)
10.	$\mathcal{S}\Gamma\ \vdash_{\mathrm{HM}}\ e_1 : \mathcal{S}\tau_2 \rightarrow \mathcal{S}\beta$	(7, 9)
11.	$\mathit{dom}(\mathcal{A}_2) \subseteq \mathit{dom}(\Gamma)$	(1)
12.	$\mathcal{S}$ satisfies $\mathcal{C}_2$	(2)
13.	$\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq \Gamma$	Property (4), (3)
14.	$\mathcal{S}\Gamma\ \vdash_{\mathrm{HM}}\ e_2 : \mathcal{S}\tau_2$	induction, (11, 12, 13)
15.	$\mathcal{S}\Gamma\ \vdash_{\mathrm{HM}}\ e_1\ e_2 : \mathcal{S}\beta$	$[\textsc{App}]_{\textsc{hm}}$, (10, 14)

A.3 Lambda

Consider the following Bottom-Up type inference rule:

$$\frac{\mathcal{A},\ \mathcal{C}\ \vdash_{\text{BU}}\ e : \tau}{\mathcal{A}\backslash x,\ \mathcal{C} \cup \{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}\ \vdash_{\text{BU}}\ \lambda x \rightarrow e : (\beta \rightarrow \tau)} \qquad [\textsc{Abs}]_{\text{BU}}$$

Choose $\mathcal{S}$ and Γ such that:

1. $dom(\mathcal{A}\backslash x) \subseteq dom(\Gamma)$
2. $\mathcal{S}$ satisfies $\mathcal{C} \cup \{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}$
3. $\mathcal{S}$ satisfies $\mathcal{A}\backslash x \preceq \Gamma$

Then the following holds:

4.	$dom(\mathcal{A}) \subseteq dom(\Gamma\backslash x \cup \{x : \mathcal{S}\beta\})$	(1)
5.	$\mathcal{S}$ satisfies $\mathcal{C}$	(2)
6.	$\mathcal{S}$ satisfies $\mathcal{A} \preceq \Gamma\backslash x$	Property (6), (3)
7.	$\mathcal{S}$ satisfies $\{\tau' \equiv \beta \mid x : \tau' \in \mathcal{A}\}$	(2)
8.	$\mathcal{S}$ satisfies $\mathcal{A} \preceq \{x : \mathcal{S}\beta\}$	Property (7), (7)
9.	$\mathcal{S}$ satisfies $\mathcal{A} \preceq (\Gamma\backslash x \cup \{x : \mathcal{S}\beta\})$	Property (5), (6, 8)
10.	$\mathcal{S}(\Gamma\backslash x \cup \{x : \mathcal{S}\beta\})\ \vdash_{\text{HM}}\ e : \mathcal{S}\tau$	induction, (4, 5, 9)
11.	$(\mathcal{S}\Gamma)\backslash x \cup \{x : \mathcal{S}\beta\}\ \vdash_{\text{HM}}\ e : \mathcal{S}\tau$	(10)
12.	$\mathcal{S}\Gamma\ \vdash_{\text{HM}}\ \lambda x \rightarrow e : \mathcal{S}\beta \rightarrow \mathcal{S}\tau$	$[\textsc{Abs}]_{\text{HM}}$, (11)
13.	$\mathcal{S}\Gamma\ \vdash_{\text{HM}}\ \lambda x \rightarrow e : \mathcal{S}(\beta \rightarrow \tau)$	(12)

A.4 Let expression

Consider the following Bottom-Up type inference rule:

$$\frac{\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\text{BU}}\ e_1 : \tau_1 \qquad \mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\text{BU}}\ e_2 : \tau_2}{\mathcal{A}_1 \cup \mathcal{A}_2\backslash x,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leq_M \tau_1 \mid x : \tau' \in \mathcal{A}_2\}\ \vdash_{\text{BU}}\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2 : \tau_2} \qquad [\textsc{Let}]_{\text{BU}}$$

Choose $\mathcal{S}$ and Γ such that:

1. $dom(\mathcal{A}_1 \cup \mathcal{A}_2\backslash x) \subseteq dom(\Gamma)$
2. $\mathcal{S}$ satisfies $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leq_M \tau_1 \mid x : \tau' \in \mathcal{A}_2\}$
3. $\mathcal{S}$ satisfies $(\mathcal{A}_1 \cup \mathcal{A}_2\backslash x) \preceq \Gamma$

Then the following holds:

4.	$dom(\mathcal{A}_1) \subseteq dom(\Gamma)$	(1)
5.	$\mathcal{S}$ satisfies $\mathcal{C}_1$	(2)
6.	$\mathcal{S}$ satisfies $\mathcal{A}_1 \preceq \Gamma$	Property (4), (3)
7.	$\mathcal{S}\Gamma \vdash_{\mathrm{HM}} e_1 : \mathcal{S}\tau_1$	induction, (4, 5, 6)
8.	$dom(\mathcal{A}_2) \subseteq dom(\Gamma\backslash x \cup \{x : generalize(\mathcal{S}\Gamma, \mathcal{S}\tau_1)\})$	(1)
9.	$\mathcal{S}$ satisfies $\mathcal{C}_2$	(6)
10.	$\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq \Gamma\backslash x$	Property (6), (3)
11.	$\mathcal{S}$ satisfies $\{\tau' \leq_M \tau_1 \mid x : \tau' \in \mathcal{A}_2\}$	(2)
12.	$\mathcal{S}$ satisfies $\{\tau' \preceq generalize(\mathcal{S}M, \mathcal{S}\tau_1) \mid x : \tau' \in \mathcal{A}_2\}$	Property (3), (11)
13.	$\mathcal{S}$ satisfies $\{\tau' \preceq generalize(\mathcal{S}\Gamma, \mathcal{S}\tau_1) \mid x : \tau' \in \mathcal{A}_2\}$	Lemma 5, (12)
14.	$\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq \{x : generalize(\mathcal{S}\Gamma, \mathcal{S}\tau_1)\}$	($\preceq$), (13)
15.	$\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq (\Gamma\backslash x \cup \{x : generalize(\mathcal{S}\Gamma, \mathcal{S}\tau_1)\})$	Property (5), (10, 14)
16.	$\mathcal{S}(\Gamma\backslash x \cup \{x : generalize(\mathcal{S}\Gamma, \mathcal{S}\tau_1)\}) \vdash_{\mathrm{HM}} e_2 : \mathcal{S}\tau_2$	induction, (8, 9, 15)
17.	$(\mathcal{S}\Gamma)\backslash x \cup \{x : generalize(\mathcal{S}\Gamma, \mathcal{S}\tau_1)\} \vdash_{\mathrm{HM}} e_2 : \mathcal{S}\tau_2$	(16)
18.	$\mathcal{S}\Gamma \vdash_{\mathrm{HM}} \textbf{let } x = e_1 \textbf{ in } e_2 : \mathcal{S}\tau_2$	$[\text{LET}]_{\text{HM}}$, (7, 17)

B Completeness

We present a detailed proof of Theorem 7, in which we claim that the Bottom-Up type inference rules are complete with respect to the Hindley-Milner rules. We give the four cases for a proof by induction on the structure of the expressions.

B.1 Variable

Consider the following Hindley-Milner type inference rule:

$$\frac{\tau \prec \Gamma(x)}{\Gamma \vdash_{\mathrm{HM}} x : \tau} \qquad [\text{VAR}]_{\text{HM}}$$

We assume that:

1. $\tau \prec \Gamma(x)$

Then the following holds:

2.	$\{x : \beta\},\ \emptyset \vdash_{\mathrm{BU}} x : \beta$	$[\text{VAR}]_{\text{BU}}$
3.	We choose $\mathcal{S}$ to be $[\beta \mapsto \tau]$	
4.	$\mathcal{S}$ satisfies $\emptyset$	
5.	$\mathcal{S}\tau \prec \mathcal{S}\Gamma(x)$	(1)
6.	$\mathcal{S}\beta \prec \mathcal{S}\Gamma(x)$	(3, 5)
7.	$\mathcal{S}$ satisfies $\{\beta \preceq \Gamma(x)\}$	($\preceq$), (6)
8.	$\mathcal{S}$ satisfies $\{x : \beta\} \preceq \Gamma$	($\preceq$), (7)
9.	$\mathcal{S}\beta = \tau$	(3)

Proof completed by (2, 4, 8, 9)

B.2 Application

Consider the following Hindley-Milner type inference rule:

$$\frac{\Gamma \vdash_{\mathrm{HM}} e_1 : \tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash_{\mathrm{HM}} e_2 : \tau_1}{\Gamma \vdash_{\mathrm{HM}} e_1\ e_2 : \tau_2} \qquad [\text{APP}]_{\text{HM}}$$

Induction results in:

1. $\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\text{BU}}\ e_1 : \tau_1'$, $\mathcal{S}_1$ satisfies $\mathcal{C}_1$, (for some $\mathcal{S}_1$)
   $\mathcal{S}_1$ satisfies $\mathcal{A}_1 \preceq \Gamma$, and $\mathcal{S}_1 \tau_1' = \tau_1 \to \tau_2$
2. $\mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\text{BU}}\ e_2 : \tau_2'$, $\mathcal{S}_2$ satisfies $\mathcal{C}_2$, (for some $\mathcal{S}_2$)
   $\mathcal{S}_2$ satisfies $\mathcal{A}_2 \preceq \Gamma$, and $\mathcal{S}_2 \tau_2' = \tau_1$

Then the following holds:

3. $\mathcal{A}_1 \cup \mathcal{A}_2,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1' \equiv \tau_2' \to \beta\}\ \vdash_{\text{BU}}\ e_1\ e_2 : \beta$ $[\textsc{App}]_{\text{BU}}$, (1, 2)
4. We choose $\mathcal{S}$ to be $[\beta \mapsto \tau_2] \circ \mathcal{S}_2 \circ \mathcal{S}_1$
5. $\mathcal{S}$ satisfies $\mathcal{A}_1 \cup \mathcal{A}_2 \preceq \Gamma$ Property (4), (1, 2, 4)
6. $\mathcal{S}\tau_1' = \tau_1 \to \tau_2$ (1, 4)
7. $\mathcal{S}(\tau_2' \to \beta) = \tau_1 \to \tau_2$ (2, 4)
8. $\mathcal{S}$ satisfies $\{\tau_1' \equiv \tau_2' \to \beta\}$ ($\equiv$), (6, 7)
9. $\mathcal{S}$ satisfies $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau_1' \equiv \tau_2' \to \beta\}$ (1, 2, 8)
10. $\mathcal{S}\beta = \tau_2$ (4)

Proof completed by (3, 5, 9, 10)

B.3 Lambda

Consider the following Hindley-Milner type inference rule:

$$\frac{\Gamma\backslash x \cup \{x:\tau_1\}\ \vdash_{\text{HM}}\ e:\tau_2}{\Gamma\ \vdash_{\text{HM}}\ \lambda x \to e:\tau_1 \to \tau_2} \qquad [\textsc{Abs}]_{\text{HM}}$$

Induction results in:

1. $\mathcal{A},\ \mathcal{C}\ \vdash_{\text{BU}}\ e:\tau$, $\mathcal{S}_1$ satisfies $\mathcal{C}$, (for some $\mathcal{S}_1$)
   $\mathcal{S}_1$ satisfies $\mathcal{A} \preceq \Gamma\backslash x \cup \{x:\tau_1\}$, and $\mathcal{S}_1\tau = \tau_2$

Then the following holds:

2. $\mathcal{A}\backslash x,\ \mathcal{C} \cup \{\tau' \equiv \beta \mid x:\tau' \in \mathcal{A}\}\ \vdash_{\text{BU}}\ \lambda x \to e:\beta \to \tau$ $[\textsc{Abs}]_{\text{BU}}$, (1)
3. We choose $\mathcal{S}$ to be $[\beta \mapsto \tau_1] \circ \mathcal{S}_1$
4. $\mathcal{S}$ satisfies $\mathcal{A}\backslash x \preceq \Gamma$ Property (6), (1, 3)
5. $\mathcal{S}$ satisfies $\mathcal{C}$ (1, 3)
6. $\mathcal{S}$ satisfies $\mathcal{A} \preceq \{x:\tau_1\}$ Property (5), (1, 3)
7. $\mathcal{S}$ satisfies $\{\tau' \equiv \tau_1 \mid x:\tau' \in \mathcal{A}\}$ Property (7), (6)
8. $\mathcal{S}$ satisfies $\{\tau' \equiv \beta \mid x:\tau' \in \mathcal{A}\}$ ($\equiv$), (3, 7)
9. $\mathcal{S}$ satisfies $\mathcal{C} \cup \{\tau' \equiv \beta \mid x:\tau' \in \mathcal{A}\}$ (5, 8)
10. $\mathcal{S}(\beta \to \tau) = \tau_1 \to \tau_2$ (1, 3)

Proof completed by (2, 4, 9, 10)

B.4 Let expression

Consider the following Hindley-Milner type inference rule:

$$\frac{\Gamma\ \vdash_{\text{HM}}\ e_1:\tau_1 \quad \Gamma\backslash x \cup \{x:\mathit{generalize}(\Gamma,\tau_1)\}\ \vdash_{\text{HM}}\ e_2:\tau_2}{\Gamma\ \vdash_{\text{HM}}\ \textbf{let}\ x = e_1\ \textbf{in}\ e_2:\tau_2} \qquad [\textsc{Let}]_{\text{HM}}$$

Induction results in:

1. $\mathcal{A}_1,\ \mathcal{C}_1\ \vdash_{\mathrm{BU}}\ e_1 : \tau_1'$, $\mathcal{S}_1$ satisfies $\mathcal{C}_1$, (for some $\mathcal{S}_1$)
   $\mathcal{S}_1$ satisfies $\mathcal{A}_1 \preceq \Gamma$, and $\mathcal{S}_1\tau_1' = \tau_1$
2. $\mathcal{A}_2,\ \mathcal{C}_2\ \vdash_{\mathrm{BU}}\ e_2 : \tau_2'$, $\mathcal{S}_2$ satisfies $\mathcal{C}_2$, (for some $\mathcal{S}_2$)
   $\mathcal{S}_2$ satisfies $\mathcal{A}_2 \preceq (\Gamma\backslash x \cup \{x : \mathit{generalize}(\Gamma, \tau_1)\})$,
   and $\mathcal{S}_2\tau_2' = \tau_2$

Then the following holds:

3. $\mathcal{A}_1 \cup \mathcal{A}_2\backslash x,\ \mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leq_M \tau_1' \mid x : \tau' \in \mathcal{A}_2\}$
   $\vdash_{\mathrm{BU}}$ **let** $x = e_1$ **in** $e_2 : \tau_2'$ [Let]$_{\mathrm{BU}}$, (1, 2)
4. We choose $\mathcal{S}$ to be $\mathcal{S}_2 \circ \mathcal{S}_1$
5. $\mathcal{S}$ satisfies $(\mathcal{A}_1 \cup \mathcal{A}_2\backslash x) \preceq \Gamma$ Property (6), (1, 2, 4)
6. $\mathcal{S}$ satisfies $\mathcal{A}_2 \preceq \{x : \mathit{generalize}(\Gamma, \tau_1)\}$ Property (5), (2, 4)
7. $\mathcal{S}$ satisfies $\{\tau' \preceq \mathit{generalize}(\Gamma, \tau_1) \mid x : \tau' \in \mathcal{A}_2\}$ ($\preceq$), (6)
8. $\mathcal{S}$ satisfies $\{\tau' \preceq \mathit{generalize}(\mathcal{S}M, \mathcal{S}\tau_1') \mid x : \tau' \in \mathcal{A}_2\}$ Lemma 5, (1, 4, 7)
9. $\mathcal{S}$ satisfies $\{\tau' \leq_M \tau_1' \mid x : \tau' \in \mathcal{A}_2\}$ Property (3), (8)
10. $\mathcal{S}$ satisfies $\mathcal{C}_1 \cup \mathcal{C}_2 \cup \{\tau' \leq_M \tau_1' \mid x : \tau' \in \mathcal{A}_2\}$ (1, 2, 4, 9)
11. $\mathcal{S}\tau_2' = \tau_2$ (2, 4)

Proof completed by (3, 5, 10, 11)