Polymorphic Type Reconstruction Using Type Equations*

Venkatesh Choppella

Indian Institute of Information Technology and Management Kerala
Thiruvananthapuram, Kerala 695 581, India
choppell@iiitmk.ac.in

Abstract. The W algorithm of Milner [Mil78] and its numerous variants [McA98,LY98,YTMW00] implement type reconstruction by building type substitutions. We define an algorithm W^E centered around building type equations rather than substitutions. The design of W^E is motivated by the belief that reasoning with substitutions is awkward. More seriously, substitutions fail to preserve the exact syntactic form of the type equations they solve. This makes analysing the source of type errors more difficult. By replacing substitution composition with unions of sets of type equations and eliminating the application of substitution to environments, we obtain an algorithm for type reconstruction that is simple and also useful for type error reconstruction. We employ a sequentiality principle for unifier composition and a constructive account of mgu-induced variable occurrence relation to design W^E and prove its correctness. We introduce syntax equations as a formal syntax for progam slices. We use a simple constraint generation relation to relate syntax equations with type equations to trace program slices responsible for a type error.

1 Introduction

The Damas-Milner type system [DM82,Dam85], also known as the Hindley-Milner type system, is the basis for type reconstruction in higher-order, polymorphically typed functional languages like ML [MTH90] and Haskell [PJH99]. Type reconstruction in Damas-Milner is implemented using Milner's principal type algorithm W [Mil78]. An important practical concern affecting the usability of these languages has been the issue of intelligent type error diagnosis, that is, locating the elements of the source program that contribute to the type error in an ill-typed program. The problem of type error diagnosis has led to several proposals for modifying W [McA98,LY98,YTMW00].

The W algorithm and the above mentioned variants compute the principal type of an expression by building substitutions, which are maps from type variables to types. Each type variable is a placeholder for the type of a subexpression

* Part of this work was done when the author was at Oak Ridge National Laboratory, Oak Ridge TN, USA, managed by UT-Battelle, LLC for the U.S. Department of Energy under contract number DE-AC05-00OR22725.

P. Trinder, G. Michaelson, and R. Peña (Eds.): IFL 2003, LNCS 3145, pp. 53–68, 2004.

of the program expression. It is intuitively appealing, however, to consider dividing the process of building solution substitutions into two phases: an initial phase in which type constraints are constructed and accumulated followed by a second phase in which these constraints are solved to obtain a solution substitution. Such a separation of phases, for example, is the basis of Wand's proof [Wan87] of Hindley's theorem, in which the typability problem for simply-typed lambda calculus is reduced to term unification [CF58,Hin69]. Viewing type assignments as solutions to type equations was encouraged by Milner himself [Mil78], and later by Cardelli [Car87] as well, mainly through examples. In this paper, we present an algorithm W^E that relies on a limited separation of the generation of type equations from their solution. Using the algorithm for unification source-tracking developed earlier [Cho02,CH03], we show how this algorithm may be used for tracking the source of type errors.

The substitutions computed by the Milner W algorithm and others are solutions (*unifiers*) of sets of type equations, yet these equations are never made explicit in these algorithms. Since substitutions lose information about the exact form of the term equations they solve, it is difficult to reconstruct source information from substitutions alone. Therefore, we seek a type reconstruction algorithm centered around the computation of type equations with the following property: The equations should have an mgu that is trivially related to the mgu computed by W. Otherwise, the non-unifiability of these equations, which indicates untypability, should be diagnosable independent of the Damas-Milner type system, using unification source-tracking [CH03], for example.

Separating the generation of type equations from their solution is, however, easier said than done for Damas-Milner type reconstruction. The difficulty may be traced to the non-compositional behavior of the W algorithm and its variants: in the expression $\mathsf{let}\ x\ =\ e\ \mathsf{in}\ e'$, the type of e is required to compute the type of e'. This non-compositional behavior is due to the absence of the principal typing (as opposed to the principal type) property of the Damas-Milner type system [Wel02].

To be sure, separation *is* possible, either by generating type *inequations*, or by unfolding all the let bindings in the original program. But these approaches move the type reconstruction problem outside the realm of unification and the Damas-Milner regime respectively, and are also unsatisfactory from a practical point of view. The solution of type inequations requires semi-unification rather than ordinary unification [Hen93], while the unfolding of let bindings reduces the problem to typability in the Curry-Hindley calculus at the cost of an increase in program size in practice, and an exponential increase in the theoretical worst case [KMM91].

Our approach offers a middleground in which the type equation generation phase is continued until a let expression $\mathsf{let}\ x\ =\ e\ \mathsf{in}\ e'$ is encountered. The type equations for e' refer to the type of e, which is obtained by solving the type equations generated by e. By solving the equations at let boundaries, we avoid both the problem of proliferation of type equations caused by let unfolding and the need to generate type inequations.

1.1 Summary of Contributions and Outline of Paper

The main contributions of this paper are:

1. A sequentiality principle for unifier composition that relates unifier composition with union of term equations (Theorem 1, Section 2).
2. A constructive characterization of variable occurrences in terms computed by applying most general unifiers (Lemmas 1 and 2, Section 2.2). This result relies on the unification path framework developed earlier [CH03]. It is used to provide a formal, constructive interpretation of a common implementation mechanism for identifying non-generic variables. (Section 4.2).
3. A type equation based polymorphic type reconstruction algorithm W^E for Damas-Milner (Section 4).
4. An application of the unification source-tracking algorithm developed in [CH03] to extract type equation slices from the output of W^E (Section 5).
5. A simple framework for error diagnosis in the Damas-Milner type system. The framework consists of *syntax equations* (Section 5.1), which are a formal notation for expressing program slicing information, *type equations*, and a *constraint generation relation* relating syntax equations to type equations (Section 5.2). Type equation slices computed in (4) are mapped back to syntax equation slices generating a type error.

Section 2 presents a constructive view of unification. Section 3 briefly reviews the Milner W algorithm. Section 4 defines W^E and sketches its correctness. Section 5 shows how W^E can be used for tracking type errors. Section 6 compares our approach with published variants of W and other related work. Section 7 concludes with pointers to future work.

Proofs of all the results of this paper are included in an accompanying technical report [Cho03b].

2 A Constructive View of Unification

Term unification is at the heart of Damas-Milner type reconstruction. In this section we first introduce a sequentiality principle for term unification. This principle is used to justify the correctness of replacing substitution operations with generation of term equations. Using examples, we then briefly review the constructive approach to term unification offered by the unification path framework of Choppella and Haynes [CH03]. We use this framework to formulate a constructive account of the occurrence of variables in solutions computed by most general unifiers. We assume familiarity with the basic concepts of term unification, including terms, substitutions, idempotent substitutions, term equations, unifiers, and most general unifiers (mgus).

If E is a system of term equations and s is a substitution, then sE denotes the set of equations $\{s\tau \stackrel{?}{=} s\tau' \mid \tau \stackrel{?}{=} \tau' \in E\}$. $vars(S)$ denotes the set of variables occurring in the syntactic entity S, where S represents a term, substitution, term equation, or aggregates of these objects. If E is a set of term equations and s is

a most general unifier (mgu) of E, then $ind(s, E)$ denotes the set of independent variables of s (that is, all variables unchanged by s) also occurring in E.

Our first result relates mgu composition with the union of term equations and forms the basis of our reformulation of W.

Theorem 1 (Sequentiality of unifier composition).
If s_1 is a unifier (mgu) of E_1, and s_2' is a unifier (respectively mgu) of $s_1 E_2$, then $s_2' s_1$ is a unifier (respectively mgu) of $E_1 \cup E_2$.

A consequence of the sequentiality of unifier composition is the "left-to-right" bias of W [McA98]. Theorem 1 suggests that a way out of this sequentiality is to replace unifier composition with the symmetric operation of term equation union.

2.1 Unification Paths

A system of term equations E is efficiently represented using a *unification graph* (also denoted E) using structure sharing: variable nodes are shared; constructor nodes may be shared. E is unifiable if and only if the quotient graph $E/\sim$ under the unification closure $\sim$ of E is acyclic and homogenous (Paterson and Wegman [PW78]). Thus, the unifiability of E depends on the connectivity properties of $E/\sim$.

Unification source-tracking consists of witnessing the connectivity in the quotient graph $E/\sim$ in terms of a special connectivity relation in the "source" graph E. This special connectivity relation is defined using the idea of *unification paths* introduced in [Cho02]. Unification paths are defined over the labeled directed graph (LDG) underlying E (also denoted E). The LDG underlying E is obtained by labeling each projection edge from a constructor vertex labeled f to its i^{th} child with the symbol f_i. Equational edges in E are oriented arbitrarily and labeled ϵ, the empty string. The inverse E^{-1} of E is the LDG obtained by reversing the orientation and inverting the label of each edge of E. The inverse of a label f_i is f_i^{-1}; the inverse of ϵ is ϵ. Each inverted projection symbol f_i^{-1} is treated as an open parenthesis symbol whose matching closed parenthesis symbol is f_i. A *unification path over E* is any labeled path p in $E \cup E^{-1}$ whose label $l(p)$ is a suffix of a balanced string over these parenthesis symbols. The formal relation between unification paths in E and paths in $E/\sim$ and an extension of the unification algorithm to compute unification paths is presented in [CH03]. In the rest of this section, we summarize the relation between unification paths, unification closure and non-unifiability.

Let u, v be vertices in the unification graph of a system of term equations E. $E \models p : u \rightsquigarrow v$ ($E \models u \rightsquigarrow v$) denotes that p is a (there is a) unification path from u to v over the unification graph of E, respectively. Thus, $\rightsquigarrow$ is a reachability relation. In the framework of unification paths, unification closure is a special case of unification path reachability: $E \models u \sim v$ if and only if $E \models p : u \rightsquigarrow v$ and $l(p)$ is a balanced parentheses string. Unification failure is also a special case of reachability. For a clash, $E \models u \sim v$, for some constructor vertices u and v

with different labels. For a cycle, $E \models p : u \rightsquigarrow u$, for some variable u in E and path p such that $l(p)$ is an unbalanced suffix of a balanced parentheses string. The following example illustrates the idea of unification paths:

Example 1. Consider the system of (named) term equations E

$$e : t_7 \stackrel{?}{=} t_8 \to t_6 \qquad f : t_7 \stackrel{?}{=} t_1 \qquad g : t_8 \stackrel{?}{=} t_5$$
$$h : t_{10} \stackrel{?}{=} t_{11} \to t_9 \qquad j : t_{10} \stackrel{?}{=} t_5 \to t_6 \qquad k : t_{11} \stackrel{?}{=} t_5 \to t_6$$

The LDG underlying the unification graph of E is shown in Figure 1. Constructor vertices are identified by circles containing a constructor symbol. Projection edges originate from constructor edges and are identified by solid arrows. Equational edges are named and are identified by open arrows. The names of the left and right projection edges originating from a constructor vertex targeted by an equational edge y are assumed to be y_1 and y_2 respectively. To reduce clutter, these names are omitted. The labels on these edges are also omitted, but are equal to $\to_1$ and $\to_2$, respectively. Each $\to_i$ for $i = 1, 2$ may be thought of as a closed parenthesis symbol whose open parenthesis symbol is $\to_i^{-1}$. The label ϵ on each equational edges is also omitted. An edge y in E^{-1} corresponds to an edge y^{-1} in E with the direction of y and its label inverted. The thick brush edges highlight specific unification paths of interest. The quotient graph with respect to the unification closure of E is shown in Figure 2. The vertex set of $E/\sim$ is the set of equivalence classes of $\sim$.

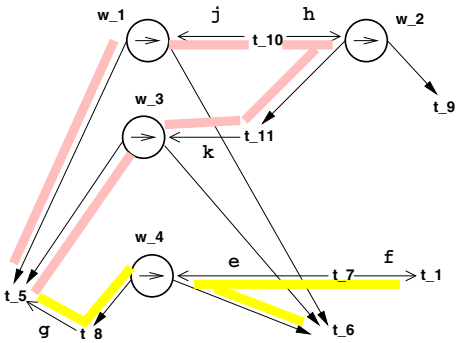

Fig. 1. Unification graph of the set of term equations of Example 1.

The element $t_6 \sim t_9$ of $\sim$ is witnessed by the unification path $j_2^{-1} j^{-1} h h_2$ whose label is $\to_2^{-1} \epsilon\epsilon \to_2$, which simplifies to the balanced string $\to_2^{-1} \to_2$. E is non-unifiable because the quotient graph $E/\sim$ has a cycle. The cycle in the quotient graph corresponds to the unification cycle $j_1^{-1} j^{-1} h h_1 k k_1$ highlighted in the source graph E. The label of this path is $\to_1^{-1} \to_1 \to_1$ after simplification.

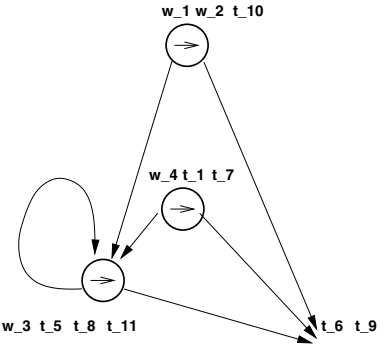

Fig. 2. Quotient graph modulo the unification closure of the unification graph in Figure 1.

2.2 Unification Paths, mgu's and Variable Occurrences

The use of unification paths is not limited to witnessing unification failure. In this section, we show that when a system of equations is unifiable, unification paths may be used as witnesses to the variable occurrence relation imposed by any most general unifier for that system of equations. This witness construction is used in Section 4.2 to track the source of non-generic type variables computed during Damas-Milner type reconstruction.

Lemma 1 (Reachability and mgu-induced occurrence relation). *Let E be a unifiable set of term equations whose mgu is s. If $t' \in vars(E)$ and $t \in ind(s, E)$, then t occurs in $s(t')$ if and only if $E \models t' \rightsquigarrow t$.*

Example 2. Continuing Example 1, consider the system E' consisting of the equations $\{e, f, g\}$. The unification graph of E' is a subgraph of E. The following substitution s' is an mgu of E': $\{t_1 \mapsto t_5 \to t_6, t_7 \mapsto t_5 \to t_6, t_8 \mapsto t_5\}$. Note that $ind(s', E') = \{t_5, t_6\}$. Both variables t_5 and t_6 occur in $s'(t_1)$. This occurrence is witnessed by the reachability of t_5 and t_6 from t_1 in E' via the unification paths $f^{-1}ee_1g$ and $f^{-1}ee_2$ respectively, highlighted in Figure 1.

It is useful to extend the notion of reachability relative to an arbitrary set of variables. The set of variables *in E reachable from V* is defined as:

$$reachable(E, V) \stackrel{\text{def}}{=} \{t \in vars(E) \mid \exists t' \in V : E \models t' \rightsquigarrow t\}$$

Example 3. Continuing Example 2, let $V = \{t_1\}$. The set $reachable(E', V)$ is $vars(E') = \{t_1, t_5, t_6, t_7, t_8\}$. It is simple to verify that every variable in E' is indeed reachable from t_1 via a unification path.

Lemma 2 (Reachability and variable occurrences). *If E is a set of term equations, s is an idempotent mgu of E, and V is any set of variables, then*

$$vars(sV) \cap ind(s, E) = reachable(E, V) \cap ind(s, E) \tag{1}$$

Lemma 2 implies that if $t \in E$ is an independent variable of s and occurs in $s(V)$ (lhs of (1)), then this occurrence can be *witnessed* by a unification path over E from some variable in V to t (rhs). This result is used in Section 4.2 to constructively characterize the non-genericity of type variables.

3 The Milner *W* Algorithm

The syntax of program and type expressions and the Milner W algorithm for computing principal types are shown in Figure 3.

$$x \in Var$$

$$e \in Exp \ ::= \ x \mid \lambda x.\ e \mid @\ e\ e \mid \mathsf{let}\ x = e\ \mathsf{in}\ e$$

$$\begin{aligned} t &\in TyVar \\ \tau &\in Ty \quad ::= t \mid \tau \to \tau \\ \sigma &\in TySch ::= \tau \mid \forall t.\sigma \end{aligned}$$

$$\begin{aligned} s &\in TySubst = TyVar \xrightarrow{\text{fin}} Ty \\ A &\in TyEnv \ = Var \xrightarrow{\text{fin}} TySch \end{aligned}$$

```
1  W(A, e) =
2  case e of
3      x_i :
4                  let ∀ᾱ.τ = A(x)
5                      in ⟨Id, τ[ᾱ/ᾱ′]⟩ where ᾱ′ new
6      @_i e_j e_k:
7                  let ⟨s_j, τ_j⟩ = W(A, e_j) and ⟨s′_k, τ_k⟩ = W(s_j A, e_k)
8                          and u = mgu{s′_k(τ_j) ≟ τ_k → t}
9                      in ⟨u s′_k s_j, u(t)⟩ where t new
10     λ_i x_j.e_k:
11                 let ⟨s_k, τ_k⟩ = W(A[x : t], e_k)
12                     in ⟨s_k, s_k(t) → τ_k⟩ where t new
13     let_i x_j = e_k in e_l:
14                 let ⟨s_k, τ_k⟩ = W(A, e_k) and ᾱ = vars(τ_k) − FV(s_k A)
15                         and⟨s′_l, τ_l⟩ = W((s_k A)[x : ∀ᾱ.τ_k], e_l)
16                     in ⟨s′_l s_k, τ_l⟩
```

Note: W fails if the mgu on line 8 does not exist.

Fig. 3. The syntax of program and type expressions and the principal type algorithm W of the Damas-Milner type system DM.

We abbreviate $\forall t_1 \ldots \forall t_n.\tau$ by $\forall t_1, \ldots, t_n.\tau$ and $t_1, \ldots, t_n$ by $\overline{t}$. The set of *free* type variables in a type scheme $\forall \overline{t}.\tau$ is equal to $vars(\tau) - \overline{t}$. The set of free variables $FV(A)$ in a type environment A is the union of all free variables in all type schemes in the range of A. The *closure* $clo(A, \tau)$ of the type τ with

respect to the type environment A denotes the type scheme $\forall\overline{\alpha}.\tau$, where $\overline{\alpha} = vars(\tau) - FV(A)$.

The following example of untypability illustrates the well-known restriction of monomorphic types imposed on λ-bound variables.

Example 4. In the core ML expression $\lambda z.$ let $y = \lambda x.\ @zx$ in $@yy$, although y is let-bound, y can only be assigned a monomorphic type. This in turn makes the application $@yy$ and hence the expression e untypable. More precisely, since z is λ-bound, the type τ_z of z is monomorphic. This type is assigned to the subexpression $\lambda x.\ @zx$ as well. The type assigned to the let-bound variable y is the closure $clo([z : \tau_z], \tau_z)$, which is τ_z. This monomorphic type τ_z is in turn assigned to y causing $@yy$ to be untypable.

The use of substitutions (composition and application to terms and type environments) in W is pervasive. In the next section, we reformulate W to minimize the use of substitutions.

4 W^E: Polymorphic Type Reconstruction Using Type Equations

The algorithm W^E (Figures 4 and 5) is based on the construction of type equations rather than substitutions. W^E takes a type environment A and an expression e. It returns a pointed set of type equations $\langle t, E\rangle$ consisting of a type variable t, a placeholder for the type of e, and a set E of type equations generated for e.

```
1  W^E(A, e_i) =
2  case e_i of
3     x_i :
4                  let  ∀ᾱ.τ = A(x)
5                       and τ_i = τ[ᾱ/ᾱ']
6                       and d_i = {t_i ≐? τ_i}
7                       and E_i = d_i
8                   in ⟨t_i, E_i⟩
9                            where t_i, ᾱ'  new
10    @_i e_j e_k:
11                 let ⟨t_j, E_j⟩ = W^E(A, e_j)
12                      and ⟨t_k, E_k⟩ = W^E(A, e_k)
13                      and d_i = {t_j ≐? t_k → t_i}
14                      and E_i = E_j ∪ E_k ∪ d_i
15                  in ⟨t_i, E_i⟩ where  t_i  new
```

Fig. 4. Algorithm W^E: VAR and APP cases.

16 $\lambda_i\ x_j.e_k$:
17 **let** $\langle t_k, E_k\rangle = W^E(A[x : t_j], e_k)$
18 **and** $d_i = \{t_i \stackrel{?}{=} t_j \to t_k\}$
19 **and** $E_i = E_k \cup d_i$
20 **in** $\langle t_i, E_i\rangle$ **where** t_i, t_j new
21 $\mathsf{let}_i\ x_j = e_k\ \mathsf{in}\ e_l$:
22 **let** $\langle t_k, E_k\rangle = W^E(A, e_k)$
23 **and** $s_k = mgu(E_k)$
24 **and** $\tau_k = s_k(t_k)$
25 **and** $\overline{\alpha} = vars(\tau_k) - FV(s_k(A))$
26 **and** $\langle t_l, E_l\rangle = W^E(A[x : \forall\overline{\alpha}.\tau_k], e_l)$
27 **and** $d_i = \{t_i \stackrel{?}{=} t_l, t_j \stackrel{?}{=} t_k\}$
28 **and** $E_i = E_k \cup E_l \cup d_i$
29 **in** $\langle t_i, E_i\rangle$ **where** t_i, t_j new
Note: W^E fails if the mgu on line 23 does not exist.

Fig. 5. Algorithm W^E ABS and LET cases.

Unlike in W, the role of substitutions in W^E is greatly diminished: Unifiers are computed only at let boundaries (line 23, Figure 5). Substitution composition is replaced by type equation union. Substitutions are not applied to type environments, except to compute generic variables (line 25, Figure 5). However, this application too can be eliminated (see Section 4.2). W computes the type substitution for the application $@_i\ e_j\ e_k$ in a non-compositional way (line 7, Figure 3). This instance of non-compositionality is eliminated when computing type equations in W^E (lines 11-12, Figure 4). Except at let boundaries, W^E neither computes nor applies substitutions. The relation between W^E and W is formally explored in the next section.

Example 5. Assume the expression e of Example 4 is decorated with locations in the following way: $\lambda_0\ z_1.\ \mathsf{let}_2\ \ y_3\ =\ \lambda_4\ x_5.\ @_6\ z_7\ x_8\ \mathsf{in}\ \ @_9\ y_{10}\ y_{11}$. This allows us to refer to values as *attributes* of these locations. Thus, e_0 refers to the expression e at location 0, $\langle t_0, E_0\rangle$ to the pointed set of type equations at location 0 etc. $W^E(\emptyset, e_0)$ returns $\langle t_0, E_0\rangle$, where E_0 is the union of the set E of equations e,f,g,h,j and k of Example 1 and the following equations:

$$a : t_0 \stackrel{?}{=} t_5 \to t_6 \qquad b : t_2 \stackrel{?}{=} t_9 \qquad c : t_3 \stackrel{?}{=} t_4 \qquad d : t_4 \stackrel{?}{=} t_5 \to t_6$$

4.1 Correctness of W^E

We consider a hybrid algorithm W^{SE} obtained by splicing together W and W^E. W^{SE} takes an expression e and a type environment A and returns the tuple $\langle t, E, s, \tau\rangle$ consisting of a type variable t, a set E of type equations, a substitution s, and a type τ. The pair $\langle t, E\rangle$ is exactly that returned by W^E. The type τ is equal to the type returned by W and the substitution s is an extension of the substitution returned by W. The algorithm W^{SE} is given in [Cho03b].

Our goal is to show that t, E, s and τ are related in the following manner: if $W^{SE}(A, e) = \langle t, E, s, \tau \rangle$, then t is a variable in E, s is an mgu of E and $\tau = s(t)$. However, this statement needs to be strengthened before it can be proved as an invariant of W^{SE}.

Lemma 3 (W^{SE} invariants). *Let e be an expression and A a type environment. If $W^{SE}(A, e) = \langle t, E, s, \tau \rangle$, then*

1. *$t \in vars(E)$, s is an mgu of E, and $s(t) = \tau$. Furthermore,*
2. *if σ is any idempotent substitution such that $vars(\sigma)$ is disjoint from $vars(E) - FV(A)$, and $W^{SE}(\sigma A, e) = \langle t^\sigma, E^\sigma, s^\sigma, \tau^\sigma \rangle$, then s^σ is an mgu of σE.*

The proof of invariant (2) relies on Theorem 1 (details are in [Cho03b]). The invariants of W^{SE} are used to relate W and W^E:

Theorem 2 (Relation between W and W^E). *Let e be an expression and A a type environment:*

1. *If $W(A, e) = \langle s, \tau \rangle$, then $W^E(A, e) = \langle t, E \rangle$, E is unifiable with an mgu s', s is a restriction of s', and $s'(t) = \tau$.*
2. *If $W^E(A, e) = \langle t, E \rangle$ and E is unifiable with mgu s', then $W(A, e) = \langle s, \tau \rangle$ for some s and τ such that s is a restriction of s' and $s'(t) = \tau$.*

In W, not all locations of a program are decorated with type variables. This is why the substitution s returned by W is a restriction of the substitution s' returned by W^E.

4.2 Constructive Interpretation of Non-genericity

The algorithm W^E still has one instance where a substitution is applied to a type environment (line 25, Figure 5) in order to compute the generic variable set $\overline{\alpha} = vars(\tau_k) - FV(s_k A)$. To eliminate it, we employ the well-known implementation trick of computing $\overline{\alpha}$ as the difference between $vars(\tau_k)$ and the *non-generic* variables $\overline{\beta} = vars(\tau_k) \cap FV(s_k A)$. This trick relies on the following informal observation: a variable β of $vars(\tau)$ is non-generic if it can be reached from some type variable in A in the "currently computed unification closure of the unification graph constructed so far." Using the results of Section 2.2, this statement can be formalized and tightened. Since s_k is an mgu of E_k and $\overline{\beta} \in ind(s_k, E_k)$, by Lemma 2, each non-generic variable $\beta \in \overline{\beta}$ is in $reachable(E_k, FV(A))$. Hence, for each $\beta \in \overline{\beta}$, there is a type variable $t \in FV(A)$ such that $E_k \models t \rightsquigarrow \beta$. This characterization is constructive because there is a unification path witnessing the reachability of β. It also immediately implies that $t \in vars(E_k)$, and thus reveals the following two bounds on the search for the non-generic variables of τ_k:

1. the type equation space in which to determine the reachability is bounded by E_k.
2. the set of type variables in $FV(A)$ from which to search for reachability to variables in τ_k is bounded by $FV(A) \cap vars(E_k)$.

Example 6. Consider the invocation $W^E(A_4, e_4)$ in the computation $W^E(\emptyset, e_0)$ of Example 5. e_4 is the subexpression $\lambda_4\ x_5.\ @_6\ z_7\ x_8$ of e_0, and $A_4 = [z : t_1]$. This invocation returns $\langle t_4, E_4 \rangle$ (line 22, Figure 5), where E_4 is the type equation set $\{d, e, f, g\}$. The mgu s_4 of E_4 maps t_1,t_4 and t_7 to $t_5 \rightarrow t_6$, and maps t_8 to t_5. Since both $s_4(t_1)$ and $\tau_4 = s_4(t_4)$ (line 24) are equal to $t_5 \rightarrow t_6$, $vars(\tau_4) - FV(s_4(A_4)) = \emptyset$ (line 25). This means that both t_5 and t_6 occurring in τ_4 are non-generic type variables.

Unification paths from the type variable t_1 of the λ-bound variable z provide a constructive explanation of the non-genericity of t_5 and t_6. These paths were already identified in Example 2. While searching for paths from the free variables of the type environment A_4, due to observation (1), we can limit our search for reachability to E_4. The type environment A_4 contains only $[z : t_1]$. In general, though, the binding of z could be nested arbirtrarily deep, making A_4 much larger. Due to observation (2), the search for source vertices of the unification paths needs to consider only those free variables in the type environment A_4 that occur in E_4.

5 Using W^E for Source-Tracking Type Errors

Any ordinary unification algorithm can be used to compute the mgu (line 23,Figure 5) in W^E. However, when the unification source-tracking algorithm of [CH03] is used, W^E can report type equation slices causing a type error. A type error in Damas-Milner is signaled by non-unifiability of a system of type equations, and the unification source-tracking algorithm is designed to return the equation slices generating the symptom of non-unifiability (clash or cycle).

Example 7. The expression e in Example 5 is untypable because the system of equations E_0 is non-unifiable. When the unification source-tracking algorithm of [CH03] is used as part of W^E, it signals non-unifiability and returns the unification cycle $j_1^{-1}j^{-1}hh_1kk_1$ witnessing this unification failure (see Example 1). This path may be partitioned into the following type equation slices: $t_{10} \stackrel{?}{=} t_{11} \rightarrow \Box$, $t_{10} \stackrel{?}{=} t_5 \rightarrow \Box$, $t_{11} \stackrel{?}{=} t_5 \rightarrow \Box$. Each slice is obtained by erasing (replacing by $\Box$'s) information not relevant to the type error. The type equation slices obtained by dividing the unification path witnessing the non-genericity of t_5 are $t_1 \stackrel{?}{=} t_7$, $t_7 \stackrel{?}{=} t_8 \rightarrow \Box$, $t_8 \stackrel{?}{=} t_5$. The corresponding slices for t_6 are $t_1 \stackrel{?}{=} t_7$, $t_7 \stackrel{?}{=} \Box \rightarrow t_6$.

Type equation slices by themselves are only partly useful for type error diagnosis. We want to be able to identify the slices of the *source program* contributing to the type error. In the next few sections, we present a framework for computing program slices from type equation slices.

5.1 Syntax Equations

We express the syntactic relation between locations of a program expression using a system of *syntax equations*, inspired by the flat system formalism for set

equations of Barwise and Moss [BM96]. Syntax equations encode constraints between various locations of a program. They are a more expressive alternative to using locations as units of program slicing information. Syntax equations are either *local*, relating an expression to its immediate subexpressions, or *referential*, in which a variable occurrence refers to its binding location. Each location i with constructor f and children at locations $i_1, \ldots i_n$ is represented by the equation $i = f(i_1, \ldots, i_n)$. Each variable reference at location i to a λ-bound (respectively let-bound) variable at location j is represented by the syntax equation $i = \lambda\mathsf{var}(j)$ (respectively $i = \mathsf{letvar}(j)$).

Example 8. The decorated expression $\lambda_0\ z_1.\ \mathsf{let}_2\ \ y_3\ \ =\ \ \lambda_4\ x_5.\ @_6\ z_7\ x_8\ \mathsf{in}\ @_9\ y_{10}\ y_{11}$ of Example 5 yields the following syntax equations: The lhs of each equation the is the subexpression at which the equation was generated.

$$\begin{array}{lll} 0 = \lambda(1,2) & 2 = \mathsf{let}(3,4,9) & 4 = \lambda(5,6) \\ 6 = @(7,8) & 7 = \lambda\mathsf{var}(1) & 8 = \lambda\mathsf{var}(5) \\ 9 = @(10,11) & 10 = \mathsf{letvar}(3) & 11 = \mathsf{letvar}(3) \end{array}$$

5.2 A Simple Constraint Generation Relation

We relate each type equation to its source information by defining a *constraint generation relation* relating the syntax equation at location i of a subexpression to the (new) type equations generated by W^E at i. Each element of the constraint generation relation is of the form "syntax equation $\Longrightarrow$ type equation".

Example 9. The constraint generation relation for the decorated expression e of Example 8 is given below:

$$\begin{array}{llll} 0 = \lambda(5,6) & \Longrightarrow a : t_0 \stackrel{?}{=} t_5 \to t_6 & 2 = \mathsf{let}(3,4,9) & \Longrightarrow b : t_2 \stackrel{?}{=} t_9 \\ 2 = \mathsf{let}(3,4,9) & \Longrightarrow c : t_3 \stackrel{?}{=} t_4 & 4 = \lambda(5,6) & \Longrightarrow d : t_4 \stackrel{?}{=} t_5 \to t_6 \\ 6 = @(7,8) & \Longrightarrow e : t_7 \stackrel{?}{=} t_8 \to t_6 & 7 = \lambda\mathsf{var}(1) & \Longrightarrow f : t_7 \stackrel{?}{=} t_1 \\ 8 = \lambda\mathsf{var}(5) & \Longrightarrow g : t_8 \stackrel{?}{=} t_5 & 9 = @(10,11) & \Longrightarrow h : t_{10} \stackrel{?}{=} t_{11} \to t_9 \\ 10 = \mathsf{letvar}(3) & \Longrightarrow j : t_{10} \stackrel{?}{=} t_5 \to t_6 & 11 = \mathsf{letvar}(3) & \Longrightarrow k : t_{11} \stackrel{?}{=} t_5 \to t_6 \end{array}$$

The type equation slices causing the type error in e and the syntax equation slices deriving them are:

$$\begin{array}{ll} \Box = @(10,11) \Longrightarrow t_{10} \stackrel{?}{=} t_{11} \to \Box & 10 = \mathsf{letvar}(3) \Longrightarrow t_{10} \stackrel{?}{=} t_5 \to \Box \\ 11 = \mathsf{letvar}(3) \Longrightarrow t_{11} \stackrel{?}{=} t_5 \to \Box & \end{array}$$

5.3 Limitations of the Simple Constraint Generation Relation

The $\lambda\mathsf{var}$ and local syntax equations generate type equations that are *linear* (each type variable occurs just once). Furthermore, the type variables in these type equations refer to locations occurring in the corresponding syntax equations. This is, however, not true at let variable references. Consider Example 9 in which

the type equations generated at locations 10 and 11 contain variables t_5 and t_6 *not occurring* in the corresponding syntax equations. In general, however, type equations at references to let bindings could contain newly cloned *generic* type variables not occurring anywhere before. The simple generation relation shown here is inadequately equipped to trace the origin of generic type variables. This problem will be addressed in a successor paper [Cho04] where a framework for expressing success proofs (why a type variable is mapped to a certain type by a substitution) will be presented.

6 Related Work

The problem of type error diagnosis has received much attention. A more detailed survey of related work is reported elsewhere [Cho03b].

Lee and Yi [LY98] present a top-down variant of algorithm W that relies on eager application of intermediate substitutions. They prove that this eager application not only generates better error messages, but that their algorithm halts sooner than W for untypable programs. Our algorithm W^E fails *later* than W does, but when used with the unification source-tracking algorithm of [CH03], returns the set of type equations slices that led to the failure. The algorithm U_{AE} of Yang et al. [YTMW00] depends on unifying type assumption environments. McAdam [McA98] uses a special algorithm for unifying substitutions. Our approach of unifying type equations is more natural and simpler. McAdam [McA00] uses an annotated graph structure to directly store program source information with the unification graph. In contrast, our approach separates the extraction of type equation slices (using unification source-tracking) with the extraction of program slices from the type slices (using the constraint generation relation). Haack and Wells [HW03] focus on the generation of minimal program slices which combines the use of a novel unification algorithm with a constraint collecting algorithm due to Damas [Dam85]. Their analysis of type diagnosis is inspired by intersection types, while the recent work of Neubauer and Thiemann [NT03] employs disjoint unions.

Trace-based approaches for type error diagnoses have also been suggested. Early work here is Maruyama et al. [MMA92]. Their trace information is unfortunately too closely dependent on the order of the unifications performed. Recently, Heeren et al. [HHS03] have proposed the use of type inference directives and specialized type rules to control the order of unification and type inference. Our approach, on the other hand, is based on tracing inferences in the connectivity space of the term equation graph rather than the execution sequence of the reconstruction algorithm.

The early work of Wand [Wan86] and Johnson and Walz [JW86,Wal89] correctly identified that the root of the type error problem lay in the source-tracking of unification. Their work focused on retrofitting unification algorithms with source-tracking information, but lacked a formal basis. There has also been a considerable effort in the area of *type explanation*, where the focus is to provide human readable analyses of type errors, often in an interactive environment that

sometimes includes visual navigation of the type graph annotated with various entities [BS94,DB96,YM00,Chi01]. Our approach for computing slicing information has a more formal basis: the unification paths used in our framework encode *proofs* in the unification path logic P^U introduced in [CH03]. Other formal approaches for include the unification logics of Le Chenadec [LC89] and the pushdown automata of Cox [Cox87].

Remy [Rém92] proposed an improvement in the search for non-generic variables using a ranked variant of the Damas-Milner based on levels of nesting of let constructs. In our approach, non-genericity is constructively demonstrated using unification paths. The unification path formalism is also simpler and ties in more naturally with our overall framework for type and error reconstruction.

7 Conclusions and Future Work

We have argued that substitution-based type reconstruction algorithms are limited in their ability to effectively track the source of type errors. This is because substitutions fail to preserve the type equations that they solve. We have formalized a sequentiality principle for unifier composition. This formalism sheds light on how to obtain the type equation based type reconstruction algorithm W^E. We believe that W^E is easier to understand and reason with than other algorithms implementing Damas-Milner. We have used the framework of unification paths developed earlier to build a constructive account of the non-genericty of type variables. We have introduced syntax equations as a new syntactic formalism for expressing program slicing. We have introduced a simple constraint generation relation to relate syntax equations to type equations. W^E, the constraint generation framework, and the unification source-tracking algorithm developed earlier together constitute a simple framework for source-tracking type errors. We have implemented this framework in Scheme for a mini-ML prototype [Cho03a].

A central feature of Damas-Milner is the controlled cloning of existing type variables to create generic type. The origin of these generic type variables, cannot, however, be accurately traced by the constraint generation relation described. This problem is addressed in a forthcoming paper [Cho04]. The algorithm W^E is defined on the core subset of ML. Considerable work is needed to extend this algorithm to the large type systems of current day functional programming languages which typically support features like polymorphic recursion, subtyping, type classes, overloading, reference types etc. Also, our program slicing information is currently text-based. A visual front-end for viewing slicing information would be very useful. We are currently developing a graphical front-end for W^E using existing graph displaying packages.

References

[BM96] J. Barwise and L. Moss. *Vicious Circles.* CSLI, 1996.

[BS94] M. Beaven and R. Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages*, 1994.

[Car87] L. Cardelli. Basic polymorphic typechecking. *Science of Computer Programming*, 8:147–172, 1987.

[CF58] H. B. Curry and R. Feys. *Combinatory Logic*, volume 1. North Holland, 1958.

[CH03] V. Choppella and C. T. Haynes. Source-tracking Unification. In Franz Baader, editor, *Proceedings of 19th International Conference on Automated Deduction, CADE-19, Miami Beach, USA*, number 2741 in Lecture Notes in Artificial Intelligence, pages 458–472. Springer, 2003.

[Chi01] Olaf Chitil. Compositional explantion of types and debugging of type errors. In *Sixth ACM SIGPLAN International Conference on Functional Programming (ICFP'01)*. ACM Press, September 2001.

[Cho02] Venkatesh Choppella. *Unification Source-tracking with Application to Diagnosis of Type Inference*. PhD thesis, Indiana University, August 2002. IUCS Tech Report TR566.

[Cho03a] Venkatesh Choppella. An implementation of algorithm of W^E. `http://www.iiitmk.ac.in/hyplan/choppell/WE.tar.gz`, October 2003.

[Cho03b] Venkatesh Choppella. Polymorphic Type reconstruction using type equations (full version with proofs). Technical Report SP-06, Indian Institute of Information Technology and Managament, Kerala, Technopark, Thiruvananthapuram, Kerala, October 2003.

[Cho04] Venkatesh Choppella. Source-tracking Damas-Milner using unification path embeddings. Technical report, Indian Institute of Information Technology and Management, Kerala, Technopark, Thiruvananthapuram, Kerala, 2004. In preparation.

[Cox87] P. T. Cox. On determining the causes of non-unifiability. *Journal of Logic Programming*, 4(1):33–58, 1987.

[Dam85] L. Damas. *Type assignment in Programming Languages*. PhD thesis, University of Edinburgh, April 1985.

[DB96] Dominic Duggan and Frederick Bent. Explaining type inference. *Science of Computer Programming*, 27(1):37–83, July 1996.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional languages. In *Proc. 9th ACM Symp. on Principles of Programming Languages*, pages 207–212, January 1982.

[Hen93] F. Henglein. Type inference with polymorphic recursion. *ACM Transactions on Programming Languages and Systems*, 15(2):253–289, April 1993.

[HHS03] Bastiaan Heeren, Jurriaan Hage, and S. Doaitse Swierstra. Scripting the type inference process. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 3–13, Uppsala, Sweden, August 2003. ACM Press.

[Hin69] R. Hindley. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society*, 146:29–60, December 1969.

[HW03] Christian Haack and Joe Wells. Type error slicing in higher order polymorphic languages. In *Proc. of Theory and Practice of Software (TAPAS-2003)*. Springer, 2003.

[JW86] G. F. Johnson and J. A. Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th ACM Symposium on Programming Languages*, pages 44–57, 1986.

[KMM91] P. C. Kanellakis, H. G. Mairson, and J. C. Mitchell. Unification and ML type reconstruction. In J.L. Lassez and G. Plotkin, editors, *Computational Logic: Essays in honor of Alan Robinson*, pages 444–478. MIT Press, 1991.

[LC89] P Le Chenadec. On the logic of unification. *Journal of Symbolic computation*, 8(1):141–199, July 1989.

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

[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*, volume 1595 of *Lecture Notes in Computer Science*, pages 139–154. Springer-Verlag, September 1998 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, 2000.

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

[MMA92] H. Maruyama, M. Matsuyama, and K. Araki. Support tool and strategy for type error correction with polymorphic types. In *Proceedings of the Sixteenth annual international computer software and applications conference, Chicago*, pages 287–293. IEEE, September 1992.

[MTH90] R. Milner, M. Tofte, and R. Harper. *The Definition of Standard ML*. MIT Press, 1990.

[NT03] Matthias Neubauer and Peter Thiemann. Discriminative sum types locate the source of type errors. In *Proceedings of the eighth ACM SIGPLAN international conference on Functional programming*, pages 15–26, Uppsala, Sweden, August 2003. ACM Press.

[PJH99] S. Peyton-Jones and J. Hughes (Eds.). Haskell 98: A non-strict, purely functional language, February 1999. `http://www.haskell.org/onlinereport`.

[PW78] M. Paterson and M. Wegman. Linear unification. *Journal of Computer and System Sciences*, 16(2):158–167, 1978.

[Rém92] D. Rémy. Extension of ML type system with a sorted equation theory on types. Technical Report 1766, INRIA, October 1992.

[Wal89] J. A. Walz. *Extending Attribute Grammars and Type Inference Algorithms*. PhD thesis, Cornell University, February 1989. TR 89-968.

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

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.

[Wel02] J. B. Wells. The essence of principal typings. In *Proc. 29th Int'l Coll. Automata, Languages, and Programming*, volume 2380 of *LNCS*. Springer-Verlag, 2002.

[YM00] Jun Yang and G. Michaelson. A visualisation of polymorphic type checking. *Journal of Functional Programming*, 10(1):57–75, January 2000.

[YTMW00] Jun Yang, Phil Trinder, Greg Michaelson, and Joe Wells. Improved type error reporting. In *Proceeding of Implementation of Functional Languages, 12th International Workshop*, pages 71–86, September 2000.