ON THE UNIFICATION OF SUBSTITUTIONS IN TYPE-INFERENCE

BRUCE J. MCADAM[*]

Technical Report ECS–LFCS–98–384
Department of Computer Science
University of Edinburgh

March 26, 1998

Abstract

Traditional type-inference and type-checking algorithms work well with correctly typed programs, but their results when given programs containing type-errors can be unpredictable. This leads to a problem with implementations of type-checkers: they are often inaccurate when announcing the apparent locations of mistakes in programs, tending to notice problems towards the end of the the program even if the source occurs much earlier. This is a particular problem with programming languages with Hindley-Milner type-systems such as used in Standard ML.

A common technique in type-inference and type-checking is *unification*. Unifying types creates a substitution which is applied to a type-environment. The substitutions which have been applied to the type-environment can influence the detection of type-errors in different subexpressions of the program.

This paper defines the operation of *unifying substitutions* and shows how type-inference algorithms can be modified to use this operation to delay the application of substitutions to the type-environment. This removes the bias to detecting errors towards the end of the program. Two different type-inference algorithms for Hindley-Milner type-inference are modified in this way and the potential for improved error reporting is shown.

[*] e-mail: bjm@dcs.ed.ac.uk, WWW: http://www.dcs.ed.ac.uk/home/bjm/

1 Introduction

Many modern programming languages are equipped with a *type-system* which is a set of rules for assigning types to parts of programs. The type-system is implemented in compilers and used for several purposes. Programs which do not have types can be rejected by compilers; the user can be informed of the types of their programs (assuring them of one aspect of correctness); and the compiler uses type information to optimise code. The focus of this paper is the first of these uses of type-systems: the rejection of programs without types (which, therefore, have type *errors* or *inconsistencies*). A difficulty many programmers find in programming is that when their program is rejected they are not given enough information to easily locate and repair the mistakes they have made.

Many type-inference algorithms (for different type-systems) make use of a *unification* operation. This takes types as parameters (usually a pair of types) and returns a substitution mapping type-variables to types which can make two or more types equal. Substitutions are applied to free type-variables in the type-environment before type-checking further subexpressions.

This paper deals primarily with the family of *Hindley-Milner* type-systems used in functional programming languages such as Standard ML. Though these provide polymorphic types, much of their type-inference is concerned with the use of unification to find the types of monomorphic parts of the program (such as function parameters and particular instances of polymorphic-functions). The process of type-checking programs in implicitly typed languages such as Standard ML is a process of *type-inference*: reconstructing the type constraints the programmer would have been required to give in an explicitly typed language (this is the task of unification and substitution). In Section 2 we shall see that informing the user of the reason for type-checking failure (and of possible locations where the programmer may have made a mistake) is an inherently difficult task in these systems. With Hindley-Milner type-inference, users often complain that the 'location' of a problem as given by the compiler (for example when it reports "cannot apply `f` to `x` on line 3") is only tenuously related to the location at which they actually made a mistake (e.g. another use of `f` several pages ago). Section 3 examines the type-inference algorithm and shows why the apparent location of errors can be wrong — it has a subtle *left-to-right bias* towards detecting problems late in the code and this bias is caused by the way unification and substitutions are used.

The solution to this problem of the conventional inference algorithm is a new type-inference algorithm designed from a pragmatic perspective. The key idea applied here is that the algorithm should be *symmetric*, treating subexpressions identically so that there is no bias causing errors to tend to be reported in one part of a program rather than another. The new algorithm rests upon the novel concept of the unification of substitutions to allow the symmetric treatment of subexpressions. Section 4 introduces the operation of *unifying substitutions* and discusses how it will remove the left-to-right bias from type-inference.

Section 5 presents a variation of the classic type-inference algorithm for Hindley-Milner type-systems and shows that the bias has been eliminated.

Programmers frequently view their programs in a slightly different way from the semantics of languages (and from compilers). In functional programming languages programmers often write curried applications which are a succession of applications in a row. When a programmer mentally parses his program he will see the curried expression as a whole and any mistakes are likely to be related to this structure whereas the compiler sees a hierarchy of applications and error messages are based on this view of the code. It is desirable to make the type-checker view the program in a similar manner to the programmer, so that error messages relate to the programmer's view of the program. Section 6 shows how the new algorithm given in Section 5 can be extended so that it treats entire curried expressions symmetrically. As functional programming languages also use constructions such as tuples, and because the type-inference rules for these are similar to those for applications, the algorithm can also be extended to treat components of tuples symmetrically.

There are many other type inference algorithms which use unification and have a left-to-right bias. Some of these are discussed in Section 7. Further extensions to this idea are discussed in Section 8 and finally a summary of this paper's conclusions can be found in Section 9.

2 Motivation

As mentioned in the introduction, type-systems play an important role in software development both from the perspective of the programmer (who has simple static properties of programs machine-checked) and the compiler (which can optimise on the basis of types). One important aspect of type-checking is that it allows the compiler to reject badly typed programs — these are not guaranteed to have a meaningful interpretation in the language semantics, so there may be run-time errors.

The problem with this is that the burden of correcting the program lies in the programmer's hands and often there is little help the compiler can give. Compilers typically tell the programmer at which point in the program the error was found, but generally this is not the part of the program in which the programmer made the mistake and it is unclear why type-checking failed at this point.

2.1 An Inherent Problem with Hindley-Milner

Two key features of Hindley-Milner type-systems are of particular interest to this paper.

Implicit typing means that the programmer need not say what type an expression or identifier should have, the type-system (and inference algorithm) can infer most types with only a small number of typing assertions needed.

Polymorphic typing means that types can be flexible, for example a function might take a value of any type, α, and return a value of the same type. In this case the function can take type $\alpha \to \alpha$ for any type α. It is not necessary for the programmer to specify that a function is polymorphic, type-inference will discover this fact.

From the point of view of finding the location of mistakes in a program, these features are weaknesses. The only way to detect a mistake is to find an inconsistency between two parts of the program, whereas in explicit type-systems the inconsistency is typically between the use of a name and the declaration of its type. So in Hindley-Milner based languages rather than being able to establish that either the use of an identifier or its declaration is incorrect (and more often than not it is the use), there are three possibilities: the first expression may be incorrect, the second may be incorrect, or the problem may lie in the context they are in (e.g. surrounding expressions, or the way they are connected). Because of polymorphism some program fragments which contain errors in them will have still have a type — but not the type the programmer expected. This can lead to a cascading effect as spurious errors are announced later.

2.2 Examples

Let us first consider a simple λ-calculus example. The function should take a real number and return the sum of the original number and its square root.

$$\lambda \mathsf{a}.\mathsf{add}\ \mathsf{a}\ (\mathsf{sqrt}\ \mathsf{a})$$

The error message from the type-checker is

> Cannot apply $\mathsf{sqrt} : \mathsf{real} \rightarrow \mathsf{real}$ to $\mathsf{a} : \mathsf{int}$.

The programmer, seeing this message, is confused as a should be a real so the problem is not that sqrt is being applied to a, it is that something else causes a to appear to be an int. The problem will become apparent if we look at the type-environment the expression is checked inside:

$$\begin{aligned} \mathsf{add} &: \mathsf{int} \rightarrow \mathsf{int} \rightarrow \mathsf{int} \\ \mathsf{sqrt} &: \mathsf{real} \rightarrow \mathsf{real} \end{aligned}$$

The programmer's mistake has been to use integer addition where he actually wants to add two real numbers.

Clearly in this case the error message is inappropriate as there is equal evidence that a should be a real as there is that it should be an int. The type-checker has incorrectly given priority to the information derived from the leftmost subexpression — it has a left-to-right bias. It would have been more informative in the example if the type-checker has pointed out that there was an inconsistency *between* the two subexpressions, instead ofwrongly claiming that either was internally inconsistent.

The second example is a short fragment of a Standard ML program which is intended to create the list `[1, 2, 3, 4, 5, 6, 7]` by flattening a list of lists onto another list

```
List.foldl  (op @)  [[1, 2], [3, 4]]  [5, 6, 7]
```

The error message is

```
Cannot apply:  List.foldl (op @) [[1, 2], [3, 4]]
          to:  [5, 6, 7]
Required argument type:  int list list list
Actual argument type:    int list
```

A programmer could be perplexed by this message if they think that `List.foldl` expects a list and then a start value, i.e. that the type-environment contains

```
List.foldl : ((’a * ’b) -> ’b) -> ’a list -> ’b
```

(Both `’a` and `’b` are intended to be `int list`). `List.foldl` in fact expects its parameters in the other order:

```
List.foldl : (’a * ’b) -> ’b) -> ’b -> ’a list
```

This is an easy mistake to make as different libraries provide different variations of this fold function.

The problem is caused by instantiating the type of the lists taken by `@` as `int list list` as soon as the list of lists has been seen. The compiler then complains about the final application instead of an inconsistency in the use of `@`. A better error message would tell the programmer that the parameters are incompatible with each other when they are all put together.

A classic example used to illustrate the monomorphism of function parameters is

$$\lambda I.(I3, I\ \mathsf{true})$$

The programmer has written a function which expects the identity function as a parameter, this is not possible in Hindley-Milner type-systems as parameters cannot be used polymorphically. When a compiler is faced with this expression it type-checks from left to right, first establishing that I must have a type of the form $\mathsf{int} \rightarrow \beta$ and then finding that I cannot, therefore, be applied to $\mathsf{true} : \mathsf{bool}$. The user will be given an error message of the form

Cannot apply $\mathsf{I} : \mathsf{int} \rightarrow \alpha$ to $\mathsf{true} : \mathsf{bool}$.

This message implies that there is an inconsistency inside $I\ \mathsf{true}$, whereas there is actually an inconsistency in the use of I between the two subexpressions. The algorithm in this paper will find this inconsistency in the uses of I. Type-checking tuples is similar to type-checking curried expressions and is discussed later.

The examples here have been selected to avoid unnecessary complexity and illustrate the key idea that curried expressions should be treated symmetrically. In real programs this is much more important as expressions can be extremely large and it is not so easy to detect and repair the mistake given the error message.

The next section explains where in the type-inference algorithm these problems arise.

3 Type-Systems and the Inference Algorithm

An introductory discussion of the type-systems and the inference algorithm can be found in [Car87]. Proofs of the algorithm W’s correctness can be found in [Dam85].

3.1 Types, Type-Schemes and Type-Environments

In this paper we will consider types which are built from type-variables ($\alpha, \beta \ldots$) (hence types are polymorphic); type constants (int, real and others); and the function type constructor →.

The form of polymorphism in Hindley-Milner type-systems arises from the use of *type-schemes*. These are types with some (or all) of their type-variables universally quantified, for example $\forall \alpha . \alpha \rightarrow \beta$. A type, τ', is a *generic instance* of a type scheme, $\forall \alpha_1 \cdots \alpha_n . \tau$, if τ' can be obtained from τ by (consistently) substituting types for the type-variables $\alpha_1 \cdots \alpha_n$. In this paper, we will not be particularly concerned with type-schemes.

Type-inference starts from a type-environment, Γ, which is a map from identifiers to type-schemes. Γ can be augmented with new terms, for example after a declaration, and can have terms removed from it (Γ_x is Γ with any term for x removed).

Type-schemes are obtained from types by *closing* a type under a type-environment. $\overline{\Gamma}(\tau)$ (the closure of τ under Γ) is the type scheme $\forall \alpha_1 \cdots \alpha_n . \tau$ where $\alpha_1 \cdots \alpha_n$ are the type-variables occuring in τ but which do not occur free in Γ. In particular, closing a type under a type-environment with no free type-variables results in every type-variable in the type being universally quantified.

Figure 1 Components of type-systems.

Component	Values
Type-Variables	$\alpha, \beta, \ldots$
Types	$\tau ::= \quad \alpha \quad \mid \quad \tau_0 \rightarrow \tau_1 \quad \mid \quad \mathsf{int} \quad \mid \quad \mathsf{real} \quad \mid \ldots$
Type-Schemes	$\sigma = \forall \alpha_0 \cdots \alpha_n . \tau$
Type-Environments	$\Gamma = \{\alpha_0 \mapsto \sigma_0, \cdots, \alpha_n \mapsto \sigma_n\}$

3.2 Type-Systems

Hindley-Milner type-systems are formulated as non-deterministic transition systems. In this paper, we will look at a simple λ-with-let-calculus (as in Figure 2) and will be particularly interested in the rule for deriving types of applications. The type-system is in Figure 3.

Figure 2 Syntax of the λ-with-let-calculus.

$$\begin{array}{lcl} e & ::= & x \\ & \mid & e_0 e_1 \\ & \mid & \lambda x . e \\ & \mid & \mathsf{let}\ x = e_0\ \mathsf{in}\ e_1 \end{array}$$

Figure 3 Type derivation rules.

$$\frac{\Gamma(x) > \tau}{\Gamma \vdash x : \tau}$$

$$\frac{\Gamma \vdash e_0 : \tau' \rightarrow \tau \qquad \Gamma \vdash e_1 : \tau'}{\Gamma \vdash e_0 e_1 : \tau}$$

$$\frac{\Gamma_x \cup \{x : \tau_0\} \vdash e : \tau_1}{\Gamma \vdash \lambda x.e : \tau_0 \rightarrow \tau_1}$$

$$\frac{\Gamma \vdash e_0 : \tau_0 \qquad \Gamma_x \cup \{x : \overline{\Gamma}(\tau_0)\} \vdash e_1 : \tau_1}{\Gamma \vdash \ \textsf{let}\ x = e_0\ \textsf{in}\ e_1 : \tau_1}$$

The type-inference rule for $e_0 e_1$ states that if (given the type-environment Γ) subexpression e_0 has type $\tau' \rightarrow \tau$ (it is a function), and then similarly that e_1 has type τ' under the same type-environment (it is a suitable argument for the function), then the application of e_0 to e_1 has type τ. The non-determinism in this case arises from the function argument type τ'. If we are attempting to show $\Gamma \vdash e_0 e_1 : \tau$, there is no way of knowing what τ' to use in the sub-derivation for each of e_0 and e_1.

3.3 Substitutions

As well as types, type-schemes and type-environments, the type-inference algorithm makes extensive use of *substitutions*. A substitution is a finite mapping from type-variables to types. Substitutions are denoted by a set of mappings, $\{\alpha_1 \mapsto \tau_1, \cdots \alpha_n \mapsto \tau_n\}$. A substitution represents a means of refining types (and of refining our knowledge of the forms of types associated with expressions). If we know that a certain type (containing type-variables) is associated with an expression, and that a substitution is also associated with it then we can apply the substitution to the type to refine it. Both Damas's algorithm and the new algorithm in this paper work by refining types using substitutions.

All substitutions must meet a well-formedness criteria.

Definition 1 (Well-formedness) *A substitution S is well-formed iff* $\mathrm{dom}(S) \cap FV(S) = \{\}$.

This restriction prevents us from getting 'infinite' types (i.e. types which contain themselves).

Substitutions can be combined. $S_1 S_0$ is the substitution which has the same effect as applying first S_0 and then S_1. $S_1 S_0$ exists iff it is well formed, so as well as both conjuncts being well formed it is necessary that $FV(S_1) \cap \mathrm{dom}(S_0) = \{\}$.

We define an ordering on types: $\tau > \tau'$ iff $\exists S : S\tau = \tau'$. A type-environment, Γ, has an instance, Γ', iff $\exists S : S\Gamma = \Gamma'$.

3.4 The Inference Algorithm

The inference algorithm, W, is a deterministic simulation of the derivation rules. For a particular type-environment and expression, it attempts to find a type for the expression and a substitution of types for type-variables such that the expression has the type under the substituted type-environment. The algorithm traverses the structure of the expression building up substitutions and types. This paper is concerned with the case of the algorithm which handles function applications:

$$\begin{array}{lcl}
W(\Gamma, e_0 e_1) & = & \textbf{let} \\
 & & \quad (S_0, \tau_0) = W(\Gamma, e_0) \\
 & & \quad (S_1, \tau_1) = W(S_0\Gamma, e_1) \\
 & & \quad \tau_0' = S_1\tau_0 \\
 & & \quad V = U(\tau_0', \tau_1 \rightarrow \beta) \textbf{ for new } \beta \\
 & & \textbf{in} \\
 & & \quad (VS_1S_0, V\beta)
\end{array}$$

The most significant part of this case is the use of *unification*. The algorithm U returns the most general substitution which when applied to each of its parameters will produce the same type, for example $U(\mathsf{int} \rightarrow \alpha,\ \beta \rightarrow \mathsf{real}) = \{\alpha \mapsto \mathsf{real},\ \beta \mapsto \mathsf{int}\}$. A survey of applications and techniques for unification can be found in [Kni89]. Type inference fails if no unifier exists. Inference could also fail in either of the recursive calls. When inference fails, implementations print error messages indicating a problem with the subexpression of current interest.

The result of type-inference shown to the programmer is a polymorphic type-scheme (providing inference succeeds). If W returns (S, τ) then the type-scheme is $\overline{S\Gamma}(\tau)$. Since Γ typically does not have any free type-variables, all type-variables in the result type will normally be universally bound.

The action of the algorithm satisfies two theorems.

Theorem 1 (Soundness of W) *If $W(\Gamma, e)$ succeeds with (S, τ) then there is a derivation of $S\Gamma \vdash e : \tau$.*

Theorem 2 (Completeness of W) *Given (Γ, e) let Γ' be an instance of Γ and η be a type-scheme such that $\Gamma' \vdash e : \eta$.*
Then

1. *$W(\Gamma, e)$ succeeds*

2. *If $W(\Gamma, e) = (P, \pi)$ then for some R: $\Gamma' = RP\Gamma$, and η is a generic instance of $R\overline{P\Gamma}(\pi)$.*

Proofs of these theorems can be found in [Dam85]. We will revisit these theorems after the algorithm is modified later in the paper.

4 New Algorithm

We have already seen some examples demonstrating the left-to-right bias of W and have seen how the algorithm works, so we now know why the bias arises (in the case of application), the problem is caused as the substitution from a left-hand subexpression is applied to the type-environment before traversing the right-hand expression.

The objective of the new algorithm is to allow us to infer types and substitutions for each subexpression independently. The new algorithm U_S deals with combining substitutions, the next section shows how to modify W to make use of it and Section 6 shows how to further extend the algorithm to produce better error messages (and suggestions of how to correct programs). The algorithm can be applied to other type-inference algorithms and other type-systems as well (as shown in Section 7).

4.1 The Idea — Unifying Substitutions

To treat the subexpressions e_0 and e_1 independently in a modified version of W, the recursive calls must be $W(\Gamma, e_0)$ and $W(\Gamma, e_1)$. This will yield two result pairs: (S_0, τ_0) and (S_1, τ_1). It is necessary then to

- check that the two substitutions are consistent
- apply terms from S_0 to τ_1 and from S_1 to τ_0 so that the resulting types have no free type-variable in the domain of either substitution, and
- return a well-formed substitution containing entries from both S_0 and S_1.

The second of these operations cannot be done simply by computing $S_0\tau_1$ and $S_1\tau_0$ because this could leave unwanted free type variables, likewise the third of these is not simply S_1S_0 or S_0S_1. The essence of these three operations can be summarised in these two requirements:

- check the substitutions are consistent, and if they are
- create a substitution which contains the effect of both.

We must *unify* the two substitutions.

4.2 Examples

Before we look at the algorithm for unifying substitutions, it will be worthwhile seeing some examples.

The simplest case is where the two substitutions are completely independent.

$$\begin{aligned} S_0 &= \{\alpha \mapsto \mathsf{int}\} \\ S_1 &= \{\beta \mapsto \gamma\} \\ U_S(S_0, S_1) &= \{\alpha \mapsto \mathsf{int}, \beta \mapsto \gamma\} \end{aligned}$$

If the domains of S_0 and S_1 contain a common element, we must unify the relevant types:

$$\begin{array}{rcl} S_0 & = & \{\alpha \mapsto \mathsf{int} \rightarrow \beta\} \\ S_1 & = & \{\alpha \mapsto \gamma \rightarrow \mathsf{real}\} \\ U_S(S_0, S_1) & = & \{\beta \mapsto \mathsf{real}, \gamma \mapsto \mathsf{int}\} \end{array}$$

Note that equivalent results cannot be obtained simply by composing the substitutions ($S_0 S_1$ or $S_1 S_0$). That example would have occured inside the lambda term $\lambda(f, x).(f\,1) + ((f\,x) + 0.1)$. S_0 is the substitution produced from $f\,1$ and S_1 comes from $(f\,x) + 0.1$ (α is the type-variable related to f).

Substitution unification can fail, for example with

$$\begin{array}{rcl} S_0 & = & \{\beta \mapsto \alpha \rightarrow \mathsf{real}\} \\ S_1 & = & \{\beta \mapsto \mathsf{real} \rightarrow \mathsf{real}, \alpha \mapsto \mathsf{int}\} \end{array}$$

There is an inconsistency between the instantiations of α in this case.

Unification could also fail with an *occurs* error.

$$\begin{array}{rcl} S_0 & = & \{\alpha \mapsto \mathsf{int} \rightarrow \beta\} \\ S_1 & = & \{\beta \mapsto \mathsf{int} \rightarrow \alpha\} \end{array}$$

Clearly the two substitutions here imply that α and β should be infinite types.

4.3 Formal Definition

A substitution, S', unifies substitutions, S_0 and S_1, if $S'S_0 = S'S_1$. In particular the most general unifier of a pair of substitutions is S' such that

$$(S'S_0 = S'S_1) \wedge (\forall S'' : (S''S_0 = S''S_1) \rightarrow (\exists R : S'' = RS'))$$

i.e. S' unifies S_0 and S_1, and S' can be augmented to be equivalent to any other unifying substitution.

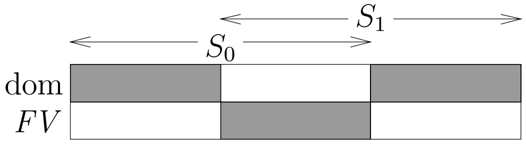
The unified substitution, $S'S_0$, has the effect of both S_0 and S_1 since $(S'S_0\alpha < S_0\alpha)$ and $(S'S_0\alpha < S_1\alpha)$, for all α.

4.4 Algorithm U_S

Algorithm U_S computes the most general unifier of a pair of substitutions.

To see how the algorithm works, note that the domain of the result consists of three parts as shown in Figure 4. The algorithm you are about to see deals with each of the three parts of the domain separately.

Figure 4 The domain of $U_S(S_0, S_1)$ consists of three parts. The disjoint parts of the domains of S_0 and S_1, and the free variables in their ranges where their domains overlap.



The free variables in the range of the unifier are free in either S_0 or S_1 and are in the domains of neither.

Here is the algorithm, commented in italics.

```
$U_S(S_0, S_1)$ = **let**
        *First split the domains into three parts:*
        $D_0 = \mathrm{dom}(S_0) - \mathrm{dom}(S_1)$        $T_0 = \{\alpha \mapsto S_0\alpha : \alpha \in D_0\}$
        $D_1 = \mathrm{dom}(S_1) - \mathrm{dom}(S_0)$        $T_1 = \{\alpha \mapsto S_1\alpha : \alpha \in D_1\}$
        $D_\cap = \mathrm{dom}(S_0) \cap \mathrm{dom}(S_1)$
        *Remember:* $FV(T_0) \cap \mathrm{dom}(S_0) = \{\}$*, similarly for* $T_1$*.*
        *Start with* $T_0$ *and add terms for* $D_1$ *one at a time,*
        *always producing well formed substitutions:*
        $S'_0 = T_0$        $\{\alpha_1 \mapsto \tau_1 \cdots \alpha_n \mapsto \tau_n\} = T_0$
        $S'_{i+1}$ = **let**
            *Remove elements of dom*$(S'_i)$ *from* $\tau_{i+1}$*,*
            *and remove* $\alpha_{i+1}$ *from* $S'_i$*:*
            $\tau'_{i+1} = S'_i\tau_{i+1}$
            **If** $\alpha_{i+1} \in FV(\tau'_{i+1})$ **terminate (occurs error)**
            **in** $\{\alpha_{i+1} \mapsto \tau'_{i+1}\}S'_i$
        $S'_n$ *is the unifier for* $T_0$ *and* $T_1$*.*
        *Now deal with items in* $D_\cap = \{\beta_1 \cdots \beta_m\}$*:*
        $U_0 = S'_n$
        $U_{i+1}$ = **let**
            $\tau_0 = U_iS_0\beta_{i+1}$        $\tau_1 = U_iS_1\beta_{i+1}$
            **If** $\beta_{i+1} \in FV(\tau_0) \cup FV(\tau_1)$ **terminate (occurs error)**
            $V = U(\tau_0, \tau_i)$
            **in** $VU_i$
    **in** $U_m$
```

4.5 Verification of U_S

We must show that U_S does indeed compute the most general unifier of a pair of substitutions, the propositions in this section are similar to those you will see in Section 5.1 (in both cases they are a pair of soundness and completeness results, showing that the algorithm never gives a wrong answer and that if an answer exists then the algorithm will give an answer).

Theorem 3 *For any pair of substitutions, S_0 and S_1, if $U_S(S_0, S_1)$ succeeds then it returns a unifying substitution.*

Theorem 4 *If S'' unifies S_0 and S_1 then*

1. *$U_S(S_0, S_1)$ succeeds returning S', and*
2. *there is some R such that $S'' = RS'$.*

Proofs of both these propositions can be found in Appendix A

4.6 Unifying Sets of Substitutions

U_S can be extended to take a set of substitutions and to return the most general unifier of the entire set. We will write $U_S\{S_0, S_1 \cdots S_n\}$ for the application of this extended version of U_S.

A naïve implementation based on the pair-U_S function is shown below.

```
$U_S\{S_0 \cdots S_n\}$  =  let
                                Generate sequence of substitutions $S'_0 \cdots S'_1$
                                $S'_0 = S_0$
                                $S'_{i+1} = U_S(S'_i, S_{i+1})$
                            in $S'_n$
```

This algorithm will not generally be suitable as if it fails, we are unable to say accurately which pair of substitutions conflicted (it has a left-to-right bias which allows us only to say that some substitution, S_i, conflicts with some other substitution before it whereas we would wish to be able to tell exactly which pair conflicted). A real implementation should be an extended version of the pair-U_S function. The disjoint areas of the domains can be treated as before, while the overlapping areas will need to be handled by a type-unification algorithm which takes a set of types rather than a pair of types.

4.7 Implementation Note

The proofs regarding algorithm U_S show that the most general unifier of a pair of substitutions is computable. This algorithm is not, however, particularly efficient and is not suitable for the representation of substitutions used in many compilers.

Rather than having explicit substitutions passed as parameters and returned from functions, it is common to use references to implement substitutions. A type-variable is represented by a reference, and to apply a substitution the reference is updated to point to a type. This is intended to be more efficient than representing types as data-types and substitutions explicitly as (say) lists. The substitution is applied to every type as soon as it is created.

There has been some discussion of whether it is worthwhile implementing substitutions using references. The Glasgow Haskell compiler [PW93] found ‘spectacular speedups’ were attained by using monadic arrays in the compiler, whereas Tofte [Tof89] found that for a small project written in Standard ML the type-checker was more efficient without imperative data structures.

This representation of types and substitutions, however, represents a *greedy* strategy which is incompatible with the principle being applied here. The intention of the algorithm to be introduced in the next situation is to avoid applying substitutions until this is essential (in order to check that two subexpressions are consistent with each other and to find the result type of their application). The intention of using references on the other hand is to have substitutions applied to every type as soon as the substitution is created.

The great advantage in representing substitutions explicitly is that the type-checker need not apply them immediately and can choose which substitutions to apply to any type. This gives the potential for many variations on conventional algorithms, for example a type-checker could backtrack in order to find possible locations for mistakes. A second advantage is that a substitution contains important information which could help the programmer debug their program, explicit substitutions allow easier manipulation and analysis of this information.

Though it is not possible to represent all types and substitutions using references with the algorithm in this paper, the techniques found in other type-checkers could be applied within U_S to speed it up.

5 The New Version of W

Now that we know what it means to unify two substitutions and have seen that this is possible, so let us now look at the new algorithm, W'. This differs from W only in the case for applications

$$
\begin{array}{rcl}
W'(\Gamma, e_0e_1) & = & \textbf{let} \\
 & & \quad (S_0, \tau_0) = W'(\Gamma, e_0) \\
 & & \quad (S_1, \tau_1) = W'(\Gamma, e_1) \\
 & & \quad S' = U_S(S_0, S_1) \\
 & & \quad \tau_0' = S'\tau_0 \qquad \tau_1' = S'\tau_1 \\
 & & \quad V = U(\tau_0', \tau_1' \to \beta) \textbf{ for new } \beta \\
 & & \textbf{in} \\
 & & \quad (VS'S_0, V\beta)
\end{array}
$$

As stated earlier, the algorithm treats e_0 and e_1 symmetrically and features U_S in an analogous manner to (and in addition to) U.

Now that we have explored W' informally and given the algorithm, we will proceed to examine it in a formal manner in the next section .

5.1 Verification of W'

Algorithm W' should produce the same results as W. To verify this it is necessary to prove the soundness and completeness theorems for W'. These theorems are the same as those Damas proved for W.

The algorithm is sound if every answer it gives is a type for the parameter expression under the type-environment obtained from applying the substitution to the original type-environment.

Theorem 5 (Soundness of W') *If $W'(\Gamma, e)$ succeeds with (S, τ) then there is a derivation of $S\Gamma \vdash e : \tau$.*

This theorem is proved in Appendix B. Now that we know the algorithm never gives unreasonable answers, we must show that if a type exists for an expression, the algorithm returns a type which is at least as general as the type known to exist. This is the completeness result.

Theorem 6 (Completeness of W') *Given Γ and e, let Γ' be an instance of Γ and η be a type-scheme such that $\Gamma' \vdash e : \eta$.*
Then

1. *$W'(\Gamma, e)$ succeeds*
2. *If $W'(\Gamma, e) = (P, \pi)$ then for some R: $\Gamma' = RP\Gamma$, and η is a generic instance of $R\overline{P\Gamma}(\pi)$.*

The proof of this theorem can be found in Appendix B.

Because W' satisfies the same soundness and completeness theorems as W, and we know that the solutions of these theorems are unique (from the principal type-scheme theorem of [DM82]) we know that W' always produces the same results as W.

Corollary 1 (W' and W are equivalent) *For any pair, (Γ, e), $W(\Gamma, e)$ succeeds and returns (S, τ) if and only if $W'(\Gamma, e)$ succeeds and returns (S, τ).*

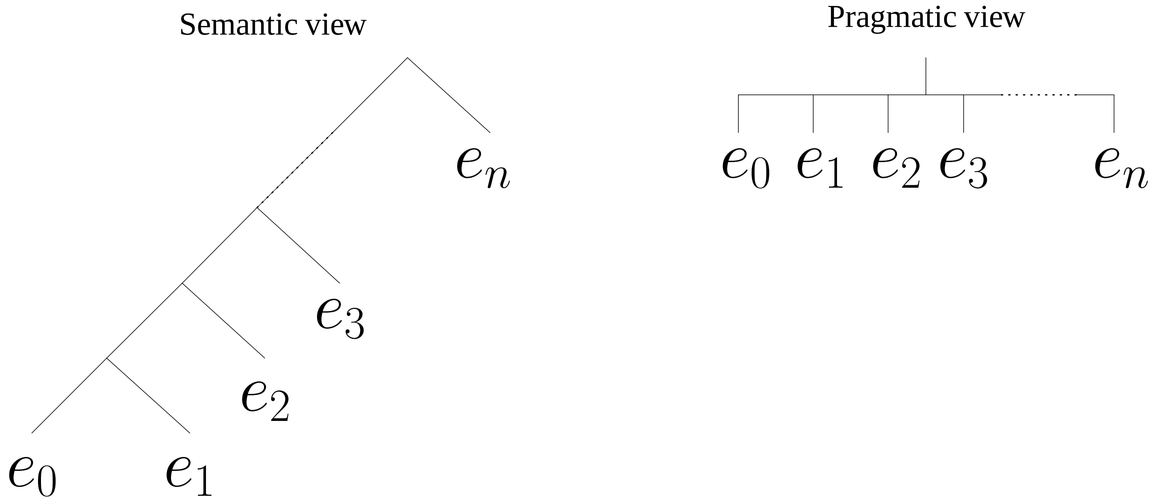
6 Curried Expressions and Tuples

The intention of W' is to type-check subexpressions independently so that the compiler can make a better estimate of the source of inconsistencies. It does this at a coarse granularity, considering only the two components of an application. This section extends W' so that it examines the program in more detail by looking at more than two subexpressions at a time.

6.1 Curried Expressions

Computations in functional programs are composed of curried applications with the form $e_0 e_1 \cdots e_n$, i.e. a sequence of values to be applied in succession. Standard ML and the typed λ-calculus treat these expressions as a hierarchy of applications $(\cdots(((e_0 e_1) e_2) e_3) \cdots) e_n$ as shown in the first part of Figure 5. This section extends the principle of W' (looking at subexpressions independently) to curried applications with many terms. The second part of the figure shows how the user views curried expressions, the modified W' will use this pattern for its recursive calls.

Figure 5 The language semantics (and compiler) view curried expressions differently from the programmer. The algorithm W'' takes the second view of expressions.



6.2 Algorithm for Curried Expressions

Given a curried expression $e_0 e_1 \cdots e_n$, we can independently infer a substitution and type for each subexpression. Then the n substitutions can be unified, and failure at this point indicates an inconsistency between two (or more) subexpressions. The substitutions for the applications and the final result for the whole expression can then be evaluated. Failure at this point would indicate that some argument is not suitable.

The first part of the algorithm is a routine to collect all the subexpression results into a list

$$\begin{aligned} \text{collect}(\Gamma, e) &= \text{append}(\text{collect}(\Gamma, e_0), W''(\Gamma, e_1)) \ \textbf{if}\ e = e_0 e_1 \\ &= W''(\Gamma, e)\ \textbf{otherwise} \end{aligned}$$

The new algorithm W'' first generates a unified substitution (using the version for U_S taking sets of substitutions mentioned earlier), then checks that the application is valid and returns the result type,

$$
\begin{array}{rcl}
W''(\Gamma, e_0 e_1) & = & \textbf{let} \\
 & & \quad (S_0, \tau_0) \cdots (S_n, \tau_n) = \text{collect}(\Gamma, e_0 e_1) \\
 & & \quad S' = U_S\{S_0 \cdots S_n\} \\
 & & \quad V = U(S'\tau_0, S'(\tau_1 \rightarrow \tau_2 \cdots \tau_n \rightarrow \beta))\ \textbf{new}\ \beta \\
 & & \textbf{in} \\
 & & \quad (VS', V\beta)
\end{array}
$$

6.3 Tuples and Records

Tuples are used in programming languages not only as a way of representing data structures such as vectors but also as temporary constructions used to pass parameters to functions. For example the Standard ML basis library provides the function `List.take` which expects a pair of an integer and a list, the pair is a transient value used only to pass two values to a function. So, in programming languages with record and tuple types, the situation for programmers becomes complicated and it is often necessary to remember the arrangement of parameters for many functions (it is not sufficient simply to know what information a function requires, but also to know the order of parameters and whether any of them should be together as a tuple).

The new type-inference algorithm can be extended to handle any arrangement of subexpressions, such as tuples. This is a trivial matter involving changing the second part of W'' so that it collates the types into a tuple or record instead of returning the final return type.

Given the types and substitutions for subexpressions, it should be possible for an automatic program repair system to attempt to rearrange the subexpressions to try to find a similar program which can be typed. For example `List.take` expects a tuple of type `int * 'a list`, it is easy to see how `List.take [1, 2, 3, 4] 2` can be converted by looking at the types of subexpressions and the obvious ways to rearrange them.

6.4 Example error messages

Recall the examples from earlier. Previously the expression

$$\lambda a.\mathsf{add}\ a(\mathsf{sqrt}\ a)$$

Would have complained

> Cannot apply $\mathsf{sqrt} : \mathsf{real} \rightarrow \mathsf{real}$ to $a : \mathsf{int}$

(Recall that add is integer addition but sqrt is real number square-root).

A modified version of W would respond

> add a requires a to have type int, but sqrt a requires a to have type real so they are inconsistent.

The expression:

$$\lambda I.(I\,3, I\,\mathsf{true})$$

Would have produced a message of the form

> Cannot apply I to true since I has type $\mathsf{int} \rightarrow \alpha$.

In a modified version of W' (one which can deal with tuples), the message would be produced when the substitutions from $I3$ and I true fail to unify. The message would be like

> $I\,3$ is inconsistent with I true since I is applied to an int in the former and a bool in the latter.

7 U_S and Other Algorithms

U_S can be applied to other algorithms which make use of subsitution.

7.1 Unification using U_S

We can write a symetric unification algorithms which makes use of U_S.

$$\begin{array}{rcl}
U(\alpha, \beta) & = & \{\alpha \mapsto \beta\} \\
U(\alpha, \tau) & = & \{\alpha \mapsto \tau\} \\
U(\tau, \alpha) & = & \{\alpha \mapsto \tau\} \\
U(\tau_0 \rightarrow \tau_1, \tau'_0 \rightarrow \tau'_1) & = & \textbf{let} \\
& & \qquad S_0 = U(\tau_0, \tau'_0) \qquad S_1 = U(\tau_1, \tau'_1) \\
& & \textbf{in} \\
& & \qquad (U_S(S_0, S_1))S_0
\end{array}$$

One advantage of implementing U this way is that any error information generated by U and U_S is likely to be in the same format. This does, of course, also eliminate the left-to-right bias which can occur in unification. Note also that there is no need for an occurs check in U as this is handled by the checks in U_S.

7.2 Another Type-Inference Algorithm

An alternative type-inference algorithm for Hindley-Milner type-systems is M, [LY98]. This is a top-down algorithm which attempts to check that an expression has a type suitable for its context. The algorithm takes type-environment, Γ, expressions, e, and target type, τ, and returns substitution, S, such that $S\Gamma \vdash e : S\tau$. The case of the algorithm for application expressions is shown below

$$
\begin{array}{rcl}
M(\Gamma, e_0e_1, \tau) & = & \textbf{let} \\
 & & \qquad S_0 = M(\Gamma, e_0, \beta \rightarrow \tau) \\
 & & \qquad S_1 = M(S_0\Gamma, e_1, S_0\beta) \\
 & & \textbf{in } S_1S_0
\end{array}
$$

It is clear that this algorithm suffers from the same left-to-right bias as W but it is a simple matter to change M to remove the bias:

$$
\begin{array}{rcl}
M'(\Gamma, e_0e_1, \tau) & = & \textbf{let} \\
 & & \qquad S_0 = M'(\Gamma, e_0, \beta \rightarrow \tau) \\
 & & \qquad S_1 = M'(\Gamma, e_1, \beta) \\
 & & \textbf{in } (U_S(S_1, S_0))S_0
\end{array}
$$

If the inference $M'(\Gamma, e_0, \beta \rightarrow \tau)$ fails, this implies that e_0 is not a function, or does not have the correct return type. The inference $M'(\Gamma, e_1, \beta)$ will fail if and only if Γ and e_1 are inconsistent (there is no typing for $\Gamma \vdash e_1$). If the unification fails then either e_1 is not a suitable argument for e_0, or there is some other inconsistency between them.

8 Future Work

The implementation of U_S is deficient in two respects. It is not clear how efficient the implementation is, and whether it will be suitable for type-checking large programs. It is therefore necessary to experiment with it and to investigate means for improving its efficiency (possibly by implementing substitution using references). The implementation also does not gather extensive error reporting information, so it is necessary to look into ways to gather information about parts of programs which conflict with each other (perhaps in the style of Duggan and Bent [DB96]).

Having considered type-checking applications using two independent recursive calls, the next logical step is to consider type-checking let. . . in. . . end declarations using two independent recursive calls. After that, it should be possible to follow the example of curried expressions and type-check nested declarations as if they are a special form of syntactic expression (as declarations are in Standard ML).

Although this paper has concentrated on mistakes within applications (with the implicit assumption that there have not been errors in earlier declarations) many of the errors programmers make come from declarations. For example consider

```
let
  val sum = List.foldl (op +)
in
  (sum [1, 2, 3]) * (sum [4, 5, 6])
end
```

The programmer's intention has been to create a local list addition function using `List.foldl`, but has forgotten to put the initial value (the definition should read `List.foldl (op +) 0`). W would complain about the two applications of `sum` even though these applications are consistent with each other. From the programmer's perspective it would be more constructive to know about the conflict between the two (internally consistent) subexpressions `val sum = List.foldl (op +)` and `(sum [1, 2, 3]) * (sum [4, 5, 6])`. This case is more complex because of Hindley-Milner *let polymorphism*, usually we find constraints in the form of substitutions which force type-variables to become more specific types, but to type-check the use of a declared name without knowing the type of the declaration it is necessary to produce constraints which say 'the type must be more general then x' (where x is the type, in this case `int list` → `int`).

9 Conclusions

This paper has presented modifications of Damas and Milner's type-checking algorithm W. The new algorithms differ from W in that they check subexpressions independently so that information derived from one does not cause spurious errors in another. The algorithm W' relies on the novel idea of unifying substitutions. The two algorithms each applied the same principle (unifying substitutions) at different levels of granularity — simple applications or curried applications. The principle can easily be extended to other constructions in functional programming languages such as tuples.

The algorithm is intended to form the basis for type-checkers which provide better error messages, messages which indicate likely areas where the programmer may have made a mistake and which indicate how the mistake may be corrected. In order for the algorithm to be of practical benefit in informing users of the location of errors in code, it will be necessary to look further into deriving useful information when U_S fails. Duggan and Bent [DB96] have considered a modification of conventional unification which tracks the origin of type-variables. It may be possible to apply their ideas to U_S. A second extension of these ideas is to look into ways to automatically try to fit subexpressions together so that a type can be derived, for example rearranging the parameters to a curried function so that the application is typeable.

Acknowledgements

Thanks to Ian Stark and Stephen Gilmore for commenting on early versions of this report. This work is supported by an EPSRC research studentship.

References

[Car87] Luca Cardelli. Basic polymorphic type-checking. *Science of Computer Programming*, 8(2):147–172, 1987.

[Dam85] Luis Manuel Martins Damas. *Type Assignment in Programming Languages*. PhD thesis, Department of Computer Science, The University of Edinburgh, April 1985. CST–33–85.

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

[DM82] Luis Damas and Robin Milner. Principal type-schemes for functional programs. In *Ninth Annual Symposium on Principles of Programming Languages*, pages 207–212. Association of Computing Machinery, 1982.

[Kni89] Kevin Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21(1):93–124, 1989.

[LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems*, 1998. Accepted for publication.

[PW93] S. Peyton Jones and P. Wadler. Imperative functional programming. In *ACM Symposium on Principles of Programming Languages*, number 20, pages 71–84. ACM, ACM Press, January 1993.

[Tof89] Mads Tofte. Four lectures on Standard ML. Technical Report ECS–LFCS–89–73, LFCS, March 1989.

A Proofs for U_S

Theorem 3 *For any pair of substitutions, S_0 and S_1, if $U_S(S_0, S_1)$ succeeds then it returns a unifying substitution.*

Proof

First show that S'_n unifies S_0 and S_1 over the domain $D_0 \cup D_1$.

Proposition For all S'_i, $\forall \alpha \in D_0 \cup \{\alpha_1 \cdots \alpha_i\} : S'_i$ is well formed (the α_j terms are from D_1), and $S'_i S_0 \alpha = S'_i S_1 \alpha$.

Proof by induction on i

Base case $S'_0 = T_0$

Only concerned with type-variables in domain of S_0 and not in domain of S_1, so $S'_0 S_0 \alpha = S_0 \alpha$ and $S'_0 S_1 \alpha = S_0 \alpha$.

Induction step $S'_{i+1} = \{\alpha_{i+1} \mapsto S'_i \tau_{i+1}\}$

$\alpha_{i+1} \notin FV(S'_i \tau_{i+1})$ (as otherwise the algorithm fails with an occurs error)

So the result is well formed.

For all α in $\mathrm{dom}(S_0) \cup \{\alpha_1 \cdots \alpha_i\}$: $S'_{i+1} S_0 \alpha = S'_i S_0 \alpha = S'_i S_1 \alpha$ (by induction hypothesis) $= S'_{i+1} S_1 \alpha$

For α_{i+1}: $S'_{i+1} S_1 \alpha = S'_{i+1} \tau_{i+1} = S'_i \tau_{i+1}$ and $S'_{i+1} S_0 \alpha_{i+1} = S'_i \tau_{i+1}$

Now show that U_m unifies over the domain $\mathrm{dom}(S_0) \cup \mathrm{dom}(S_1) \cup FV(S_0) \cup FV(S_1)$.

Proposition For all U_i, U_i is well formed, and $\forall \alpha \in D_0 \cup D_1 \cup FV(S_0) \cup FV(S_1) \cup \{\beta_1 \cdots \beta_i\} : U_i S_0 \alpha = U_i S_1 \alpha$

Proof induction on i

Base case U_0

Leaves all but $D_0 \cup D_1$ unchanged, unifies on $D_0 \cup D_1$.

Induction step $U_{i+1} = V U_i$

Need only consider β_{i+1} since by induction hypothesis everything else is unified by U_i.

Since V unifies $S_0 \beta_{i+1}$ and $S_0 \beta_{i+1}$, U_{i+1} unifies over the appropriate domain.

So U_m unifies S_0 and S_1 over the domain $\mathrm{dom}(S_0) \cup \mathrm{dom}(S_1) \cup FV(S_0) \cup FV(S_1)$ and leaves all other type variables unchanged. Thus U_m unifies S_0 and S_1.

□

Theorem 4 *If S'' unifies S_0 and S_1 then*

1. *$U_S(S_0, S_1)$ succeeds returning S', and*

2. *there is some R such that $S'' = RS'$.*

Proof

The proof follows a similar pattern to the previous result.

First show a property for the substitutions $S'_0 \cdots S'_n$ — that they do exist (there are no occurs errors) and how they relate to S''.

Proposition Each S'_i exists (there are no occurs errors) and for each S'_i there is a substitution X_i such that $\forall \alpha \in D_0 \cup \{\alpha_1 \cdots \alpha_i\} : X_i S'_i S_0 \alpha = S'' S_0 \alpha$

Proof by induction on i

Base Case $S'_0 = \{\alpha \mapsto S_0\alpha : \alpha \in D_0\}$

Know that $\forall \alpha : S''S_0\alpha \geq S_0\alpha$

And $\forall \alpha : S'_0 S_0 \alpha = S_0 \alpha$

So X_0 exists.

Induction Step $S'_{i+1} = \{\alpha_{i+1} \mapsto S'_i \tau_{i+1}\} S'_i$

First show there is no occurs error by showing $\alpha_{i+1} \notin FV(S'_i\tau_{i+1})$.

Since $S''S_1$ and X_i exist, the occurs error cannot happen.

And that $S'_i\tau_{i+1} > S''S_0\alpha$

Know $\tau_{i+1} > S''S_0\alpha$ and terms of S'_i only effect the limited domain (which is $\subseteq \mathrm{dom}(S'')$). So, by I.H. this holds.

So, by I.H. result holds for appropriate domain and X_{i+1} exists.

Also, we can see that $\mathrm{dom}(S'_n) = D_0 \cup D_1$.

Now show a similar result for the sequence $U_1 \cdots U_m$

Proposition Each U_i exists (the unification succeeds and there are no occurs errors); and for each U_i, there is a Y_i such that $\forall \alpha \in D_0 \cup D_1 \cup \{FV(S_0\beta_x) \cup FV(S_1\beta_x) \cup \{\beta_x\} : 0 \leq x \leq i\}.Y_i U_i S_0 \alpha = S'' S_0 \alpha$

Proof Induction on i

Base Case $U_0 = S'_n$

This comes directly from the previous result.

Induction Step $U_{i+1} = VU(U_iS_0\beta_{i+1}, U_iS_1\beta_{i+1})$

Must show that U succeeds. Since $S''S_0\beta_{i+1} = S''S_1\beta_{i+1}$, there must be a type which is the unification of the two types $U_iS_0\beta_{i+1}$ and $U_iS_1\beta_{i+1}$, so U succeeds.

There is no occurs error (for similar reason to those in the previous proof)

Since U is the most general unifier, Y_{i+1} exists.

So U_S succeeds and, $Y_mU_mS_0\alpha = S''S_0\alpha$ for all α in the domain above. Since all other type-variables are invariant under $Y_mU_mS_0$, it is trivial to provide a substitution, R', such that $R'Y_mS_0 = S''S_0$. So R exists and is $R'Y_m$.

□

B Proofs for W'

Theorem 5 (Soundness of W') *If $W'(\Gamma, e)$ succeeds with (S, τ) then there is a derivation of $S\Gamma \vdash e : \tau$.*

Proof Damas [Dam85] gives a proof of this theorem for W by induction on the structure of e. As W' differs from W only in the case of application, it is sufficient to present this case only.

Case $e = e_0e_1$

By the induction hypothesis, we know $W(\Gamma, e_0) = (S_0, \tau_0)$ and $S_0A \vdash e_0 : \tau_0$; and $W(\Gamma, e_1) = (S_1, \tau_1)$ and $S_1\Gamma \vdash e_1 : \tau_1$
And since $W(\Gamma, e)$ succeeds, that $S' = U_S(S_0, S_1)$, $V = U(S'\tau_0, S'\tau_1 \rightarrow \beta)$
From the definition of U_S, let $S = S'S_0 = S'S_1$, and from that of U we know $VS'\tau_1 \rightarrow \beta = VS'\tau_0$.
And the final result $W(\Gamma, e) = (VS, V\beta)$.
Must show that there is a derivation of $VS'\Gamma \vdash e_0e_1 : V\beta$
The derivation will end

$$\frac{VS\Gamma \vdash e_0 : VS'\tau_1 \rightarrow V\beta \qquad VS\Gamma \vdash e_1 : VS'\tau_1}{VS\Gamma \vdash e_0e_1 : V\beta}$$

We already know (from I.H. above) that derivations of $S_0\Gamma \vdash e_0 : \tau_0$ and $S_1\Gamma \vdash e_1 : \tau_1$ exist, so by proposition 2 of [DM82] derivations of $S'S_0\Gamma \vdash e_0 : S'\tau_0$ and $S'S_1\Gamma \vdash e_1 : S'\tau_1$ also exist.
So derivations of $VS\Gamma \vdash e_0 : VS'\tau_0$ (the type here is $VS\tau_1 \rightarrow V\beta$) and $VS\Gamma \vdash e_1 : VS'\tau_1$ also exist.
So the derivation of $VS\Gamma \vdash e_0e_1 : V\beta$ exists.

The other cases are a simple analogue of ones which have been shown by Damas, so the theorem holds in general. □

Theorem 6 (Completeness of W') *Given Γ and e, let Γ' be an instance of Γ and η be a type-scheme such that $\Gamma' \vdash e : \eta$.*
Then

1. *$W'(\Gamma, e)$ succeeds*
2. *If $W'(\Gamma, e) = (P, \pi)$ then for some R: $\Gamma' = RP\Gamma$, and η is a generic instance of $R\overline{P\Gamma}(\pi)$.*

Proof Damas provides a proof for this theorem for W on the structure of the derivation of $\Gamma' \vdash e : \eta$. As W' differs from W only in the case that $e = e_0e_1$, it is sufficient to present only the inductive step for this case.

Case $e = e_0e_1$

The derivation ends

$$\frac{\Gamma' \vdash e_0 : \tau' \rightarrow \tau \qquad \Gamma' \vdash e_1 : \tau'}{\Gamma' \vdash e_0e_1 : \tau}$$

for some τ'.
By the induction hypothesis we know that $W'(\Gamma, e_0)$ succeeds, call the result (S_0, π_0)
By condition 2 of the induction hypothesis there is a substitution R_0 such that $\Gamma' = R_0 S_0 \Gamma$ and $\tau' \to \tau$ is a generic instance of $R_0 \overline{S_0\Gamma}(\pi_0)$.
Let $\alpha_1 \cdots \alpha_n$ be the generic type-variables in π_0 (these occur in π_0 but are not free in $S_0\Gamma$).
R_0 leaves all α_i unchanged since it is minimal (which means $\mathrm{dom}(R_0) \subseteq FV(S_0\Gamma)$).
Since $\tau' \to \tau$ is a generic instance of $R_0(\forall \alpha_1 \cdots \alpha_n . \pi_0)$ (the scheme here is the closure $\overline{S_0\Gamma}(\pi_0)$), and R_0 leaves all $\alpha_1 \cdots \alpha_n$ unchanged: there are types $\tau_1 \cdots \tau_n$ such that $\tau' \to \tau = (R_0 + \{\alpha_i \mapsto \tau_i\})\pi_0$.
Likewise, for e_1: R_1, $\beta_1 \cdots \beta_m$ and $\tau'_1 \cdots \tau'_m$ exist, and $\tau' = (R_1 + \{\beta_j \mapsto \tau'_j\})\pi_1$.
First, show that $W_S(S_0, S_1)$ succeeds. To do this exhibit a substitution which is a unification of S_0 and S_1.
Note that $R_0 S_0 \Gamma = \Gamma' = R_1 S_1 \Gamma$, so $\forall \alpha \in FV(\Gamma) : (R_0 S_0)\alpha = (R_1 S_1)\alpha$.
And note that $\mathrm{dom}(S_0) \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{new}_0$ (where new_0 is the set of new type variables produced by $W'(\Gamma, e_0)$.). Similarly $\mathrm{dom}(S_1) \subseteq \mathrm{dom}(\Gamma) \cup \mathrm{new}_1$.
Let $S = \{R_0 S_0 \alpha/\alpha : \alpha \in FV(\Gamma)\} + \{R_0 S_0 \alpha/\alpha : \alpha \in \mathrm{new}_0\} + \{R_1 S_1 \alpha/\alpha : \alpha \in \mathrm{new}_1\}$.
S is a unification of S_0 and S_1. Since a unified substitution exists, $U_S(S_0, S_1)$ will terminate and returns some unifying substitution S', and there is some S'' such that $S = S'' S' S_0$.
Now it is necessary to show $U(S'\pi_0, S'\pi_1 \to \beta)$ (β is new) succeeds. To do this, exhibit a unifying substitution.
Let $U_0 = \{\alpha_i \mapsto \tau_i, \beta_j \mapsto \tau'_j, \beta \mapsto \tau\} + S''$.
First show that U_0 is a well formed substitution and then that it is a unifying substitution.
The type variables $\alpha_1 \cdots \alpha_n, \beta_1 \cdots \beta_n$ and β are all distinct. β does not occur in $\mathrm{dom}(S'')$.
Must also show none of $\alpha_1 \cdots \alpha_n, \beta_1 \cdots \beta_n$ are in $\mathrm{dom}(S'')$. $\mathrm{dom}(S'') \subseteq FV(\Gamma) \cup \mathrm{new}_0 \cup \mathrm{new}_1$, so none of the α_i, β_j occur in this domain. So U_0 is a well formed substitution.
We know $\tau' \to \tau = (R_0 + \{\alpha_i \mapsto \tau_i\})\pi_0$ so $\tau' \to \tau = U_0 S' \pi_0$. Also $\tau' = (R_1 + \{\beta_j \mapsto \tau'_j\})\pi_1$ so $\tau' \to \tau = U_0(S'\pi_1 \to \beta)$. So U_0 is unifying substitution for $S'\pi_0$ and $S'\pi_1 \to \beta$.
Since a unifying substitution exists, $U(S'\pi_0, S'\pi_1 \to \beta)$ succeeds and returns V, and there is some substitution V' such that $U_0 = V'V$.
It remains to show that the result $W(\Gamma, e) = (VS, V\beta)$ satisfies condition 2.
Must show that there is some substitution R such that $\Gamma' = RVS'S_0\Gamma$ and τ is a generic instance of $R\overline{VS'S_0\Gamma}(\tau)$. It is clear that such an R can be constructed, so this condition is satisfied.

Since the induction case $e = e_0 e_1$ holds, and the other cases are a simple analogue of ones which have been shown by Damas, the theorem holds in general. □