# Debugging Type Errors (Full version)

Karen L. Bernstein and Eugene W. Stark *
Department of Computer Science
State University of New York at Stony Brook
Stony Brook, NY 11794-4400 USA

November 8, 1995

**Abstract**

Compilers for programming languages such as Standard ML are able to find many programming errors at compile time, however the diagnostic messages from the type inference algorithm do not always clearly identify the source of type errors. We argue that an extended type definition, which assigns types to open expressions as well as closed expressions, can serve as the basis for a programming environment that helps programmers debug type errors. We present such a type definition, which is closely related to the Damas/Milner definition, but which in addition provides principal typings for open expressions. We present an algorithm that performs type inference with respect to our type system and give a simple direct proof of its correctness. Finally we describe a prototype implementation.

**Keywords**: Programming environments, Type Inference

## 1 Introduction

The Standard ML type system permits ML [MTH90] compilers to identify certain kinds of programming errors at compile time by inferring a "most general" or "principal" type for each closed expression. One problem for ML programmers, especially novice programmers, is that the source of these type errors can be hard to find. In this paper, we present a conservative extension to the current type system which provides support for tracking down the source of type errors.

Generating useful diagnostic messages for type errors is a difficult problem, which has been the subject of several papers [Wan86, JW86, DB94]. One issue is that the source of the error can be quite distant from where the symptom of the error (and the error message) occurs. Another problem is that some of the inferred types, especially for subexpressions internal to the program, are complicated and can be hard to understand. Finally, a large number of type constraints can contribute to the inferred type for a single variable and in general reasoning about large sets of constraints is hard.

We propose to extend the current ML programming environment in order to provide additional support for finding the source of type errors. The Standard ML type inference algorithm infers principal types only for *closed* expressions; that is, for expressions with no free variables. In this paper, we suggest that inferred types for *open* expressions would be a valuable tool for

*Research supported in part by NSF grant CCR-9320846

"debugging" type errors and in Section 2 we give an example of how this extension can provide a kind of "breakpoint" facility for extracting type information from inside of programs.

Extending ML type inference to open expressions is problematic because the ML type system does not directly support a notion of principal type for such expressions. The usual type inference algorithm for ML takes a expression and a *type environment* as parameters and returns a *principal type*, where a type environment is a mapping that assigns types to all variables that occur free in the expression, and the principal type is "most general" in the sense that all other possible types for the expression are instances of it. An obvious thing to try is to modify the algorithm so that it takes an open expression as a parameter and returns a *principal typing*, consisting of a type for the expression together with an assignment of types to free variables, in such a way that all other typings of the expression are instances of the principal typing. This does not work, however, since such principal typings do not exist.

As an example, consider the ML expression:

$$\text{fn } a \Rightarrow ((f\ a, f\ 1), (g\ a, g\ \mathit{true}))$$

The $\lambda$-bound variable $a$ imposes a constraint that $f$ and $g$ be functions that take the same type of parameter. In addition, $f$ is a function that takes an integer parameter and $g$ is a function that takes a boolean parameter. We can see that there is no way to unify these three constraints. One solution is to generalize $g$, so that $g$'s parameter is polymorphic:

$$[f\colon \text{int} \to \beta,\ g\colon \forall\alpha.\alpha \to \gamma] \vdash \text{fn } a \Rightarrow ((f\ a, f\ 1), (g\ a, g\ \mathit{true})) : (\beta * \beta) * (\gamma * \gamma)$$

Another solution is to generalize $f$, so that $f$'s parameter is polymorphic:

$$[f\colon \forall\alpha.\alpha \to \beta,\ g\colon \text{bool} \to \gamma] \vdash \text{fn } a \Rightarrow ((f\ a, f\ 1), (g\ a, g\ \mathit{true})) : (\beta * \beta) * (\gamma * \gamma)$$

Since there is no valid typing having both of these as a common instance, neither typing is principal.

In this paper, we present a type system which does support principal typings and is still very closely related to the ML type system. In this type system, a typing for an expression is a pair consisting of a type for the expression and a *set* of *monomorphic* (quantifier-free) types for each variable that occurs free in the expression. We call a mapping from program variables to sets of monomorphic types an *assumption environment.* With this type system the expression in the previous example has the following principal typing:

$$\begin{aligned}&[f\colon \{\alpha \to \beta, \text{int} \to \beta'\},\ g\colon \{\alpha \to \gamma, \text{bool} \to \gamma'\}]\\ &\qquad \rhd \text{fn } a \Rightarrow ((f\ a, f\ 1), (g\ a, g\ \mathit{true})) : (\beta * \beta') * (\gamma * \gamma')\end{aligned}$$

This same solution was also used by Shao and Appel [SA93] in order to define the "minimum" interface so that compilation units never need to be recompiled unless their implementation changes. A major difference between their work and ours is that they are interested in a *compiler* interface and we are interested in a *user* interface. In their work it is most important that the interface be able to identify *every* case where the compilation unit needs to be recompiled; in our case it is more important that the interface be of practical use to a programmer who may know little about the implementation of the compiler.

In Section 2, we present an example debugging session with our proposed programming environment. In Section 3, we give a type definition with assumption environments and prove that the type definition is very closely related to the Standard ML type definition. We also present a version of the type inference algorithm presented by Shao and Appel [SA93] and sketch a simple direct proof of the correctness of the algorithm which shows that the algorithm computes principal typings for our type definition. A complete proof can be found in the appendix. We feel that our proof has the advantage of being simpler and more straightforward that the one given by Shao and Appel for their algorithm. In Section 4, we discuss our prototype implementation (in the ML Kit [BRTT93]) and in the conclusion we discuss some possible future work.

The type inference algorithm for the ML type discipline (Algorithm W) was presented by Damas and Milner in the paper, *Principal Type Schemes for Functional Programs* [DM82], and was proven correct in Damas' PhD dissertation [Dam85]. Most of the research in this area has concentrated on improving the diagnostic messages generated by the type inference algorithm. Wand [Wan86] presented a type inference algorithm that keeps a record of the pieces of code that contribute to each type deduction. This information is then used to explain why the type error happened. Duggan and Bent present a refinement of Wand's approach that addresses issues such as repeated explanations and aliasing of type variables as well as some practical implementation issues such as path compression [DB94]. Johnson and Walz apply a maximal flow technique to the set of type constraints in order to identify the most likely source of the type error [JW86].

Jim [Jim95] demonstrates the value of principal typings for a variety of problems (although not the problem we discuss in this paper) and suggests the use of a variant of *rank 2 intersection types* (called $P_2$) as a type system that is closely related to the ML type system and supports principal typings. The type system we describe is a restriction of the rank 2 intersection types and therefore seems to be a more conservative choice of a type system for our purposes.

## 2 An Example Debugging Session

By extending the type inference algorithm to infer types for open terms, we can provide an enhanced programming environment that assists programmers in finding the source of type errors. We propose to include type information with the error messages for unbound identifiers. We will show how such an enhancement would be useful by considering the following simple program, which computes the average of a list of numbers and has several type errors:

```
fun avg numList = let
       val sum   = fold op + (0,numList)
       val count = fold inc (0,numList)
    in (sum/count) end
```

We will assume that this program is part of a larger program, that `inc` is defined somewhere else in the program and has type `int -> int`, and that `fold` is a library function that has the type:

```
('a * 'b -> 'b) -> 'a list -> 'b -> 'b
```

Recall also that in Standard ML the keyword "op" is used to inform the parser that the usual infix status of the immediately following operator is to be ignored for this occurrence.

First, the enhanced programming environment allows the programmer to reason about the program in a bottom-up fashion. Let us say that the programmer would like to cut and paste the definition of `sum` to the ML prompt in order to see if the definition seems correct. Usually, by trying this, the programmer would generate the following error message, since `numList` is a parameter to the function `avg`.

```
- val sum  = fold op + (0,numList);
Error: unbound variable: numList
```

In our enhanced environment, the error message is annotated with type information and type information is computed for the function `sum`:

```
- val sum  = fold op + (0,numList);
Error: unbound variable: numList: int list
val sum = fn : int list -> int
```

From the type elaboration, the programmer can see that the definition of sum assumes that numList is a list of integers and as a result sum is a function from a list of integers to an integer value. Notice that the programmer was able to break the information hiding imposed by the **let** construct, but did not define any new bindings at the top-level; all of the type information is just a part of the error message.

Suppose now that the programmer decides that numList is actually supposed to be a list of real values. However, even after the programmer modifies the program by replacing the integer literal "0" with the real literal "0.0" appropriately, there still is a type error:

```
Error: operator and operand don't agree (tycon mismatch)
  operator domain: 'Z * 'Y -> 'Z
  operand:          int -> int
  in expression:
    fold inc
```

The programmer can tell that there is a problem in the definition of count where fold is being applied to inc, but the cause is not obvious. The programmer now inserts a "breakpoint" (an unbound identifier, in this case 'b') to extract the type information at the location reported by the compiler:

```
fun avg numList = let
      val sum   = fold op + (0.0,numList)
      val count = fold (b inc) (0.0,numList)
   in (sum/count) end;
Error: unbound variable: b: (int -> int) -> real * real -> real
val avg = fn : real list -> real
```

Now the programmer can see that the function inc (the input for breakpoint b) is a function of type int -> int and that fold expects a parameter (the output for breakpoint b) of type real * real -> real. By extracting type information from inside the program, the programmer realizes that the definition of inc is not appropriate.

This proposed programming environment is a conservative extension to the current environment; the only changes are type annotations to some error messages. As a result, the changes are unobtrusive and easy to ignore when they are not needed. However, these type annotations can be quite useful. They allow the programmer to use the type inference algorithm to write and debug programs more effectively. The new environment allows the programmer to evaluate type information for pieces of the program as well as to extract type information from inside the program.

# 3 Type Definition

For this paper, we are interested in the programming language mini-ML, which is basic lambda calculus extended with a polymorphic "let" construct. The expressions in the language are defined by the following grammar, where $a$ ranges over a countable set of variables.

$$e ::= a \mid e_1\ e_2 \mid \text{fn}\ a \Rightarrow e \mid\ \textbf{let}\ a = e_1\ \textbf{in}\ e_2$$

We will denote the set of free variables of the programming language expression $e$ by $fv(e)$.

$$\frac{\tau \prec \Gamma(a)}{\Gamma \vdash a\colon \tau}\ VAR$$

$$\frac{\Gamma \vdash e_1\colon \tau' \to \tau \quad \Gamma \vdash e_2\colon \tau'}{\Gamma \vdash e_1\ e_2\colon \tau}\ APP$$

$$\frac{\Gamma + [a\colon \tau'] \vdash e\colon \tau}{\Gamma \vdash \text{fn}\ a \Rightarrow e\colon \tau' \to \tau}\ ABS$$

$$\frac{\Gamma \vdash e_1\colon \tau_1 \quad \Gamma + [a\colon \sigma] \vdash e_2\colon \tau_2 \quad \sigma = Gen(\Gamma, \tau_1)}{\Gamma \vdash\ \textbf{let}\ a = e_1\ \textbf{in}\ e_2\ : \tau_2}\ LET$$

Figure 1: ML Type Definition

*Simple types* (written $\tau$) include type variables $\alpha$, base types **int** and **bool** and function types. *Type schemes* (written $\sigma$) are simple types with some universally quantified type variables. The type expressions for our language are defined by the following grammar.

$$\tau ::= \alpha \mid \text{int} \mid \text{bool} \mid \tau_1 \to \tau_2 \qquad \sigma ::= \tau \mid \forall\alpha.\sigma$$

We will denote the set of type variables that occur free in the type scheme $\sigma$ by $tyvars(\sigma)$. We will treat type schemes that are equivalent up to the renaming of bound variables as equal.

A *type environment* $\Gamma$ is a finite mapping from variables to type schemes. For any type environment $\Gamma$ we write $\Gamma + [a\colon \sigma]$ for the type environment that maps the variable $a$ to the type scheme $\sigma$ and otherwise behaves like $\Gamma$. We will denote the domain of $\Gamma$ by $dom(\Gamma)$. A *substitution* $\theta$ is a finite mapping from type variables to simple types. We will denote the substitution that maps the type variable $\alpha$ to the type $\tau$ by $[\tau/\alpha]$. Substitutions are extended from type variables to type schemes and type environments in the usual way. The type scheme $\sigma = \forall\alpha_1...\alpha_n.\tau$ is *a generic instance* of the type scheme $\sigma' = \forall\beta_1...\beta_m.\tau'$ (written $\sigma \prec \sigma'$) if and only if $\tau = [\tau_i/\beta_i]\tau'$ for some types $\tau_1, ..., \tau_m$, and none of the type variables $\alpha_j$ are free in $\sigma'$. Let $Gen(\Gamma, \tau)$ be the type scheme $\forall\alpha_1, ...\alpha_n.\tau$ where $\{\alpha_1, ..., \alpha_n\} = tyvars(\tau) \setminus tyvars(\Gamma)$.

The type definition given in Figure 1 is based on the type definition from Tofte's dissertation [Tof87]. For a more complete discussion of the type definition see Reade's textbook [Rea89] or Mitchell's handbook article [Mit90].

## 3.1 Type Definition with Assumption Environments

In this section, we present a type definition with assumption environments. This type definition admits principal typings and in addition is very closely related to the ML type system. We will see (Theorem 2) that for any expression in the programming language there is a distinguished typing that subsumes all other typings and also (Theorem 1) that a type can be proven for a programming language expression under the ML type system if and only the same type can be proven under the type definition with assumption environments. Furthermore, there is a close relationship between the corresponding type environments and assumption environments for which the type can be proven.

An *assumption environment* $\Delta$ is a finite mapping of variables to sets of simple types (not type schemes). We will write $\Delta = \Delta_1 \cup \Delta_2$ to mean that for all variables $a$, $\Delta(a) = \Delta_1(a) \cup \Delta_2(a)$. We will write $\Delta(a) = \emptyset$ if $a$ is not in the domain of $\Delta$. We will write $\Delta \backslash a$ to mean the assumption

$$\frac{}{[a\colon \{\tau\}] \rhd a\colon \tau}\ VAR$$

$$\frac{\Delta_1 \rhd e_1\colon \tau' \to \tau \quad \Delta_2 \rhd e_2\colon \tau'}{\Delta_1 \cup \Delta_2 \rhd e_1\ e_2\colon \tau}\ APP$$

$$\frac{\Delta \rhd e\colon \tau \quad \Delta(a) \subseteq \{\tau'\}}{\Delta \setminus a \rhd \mathrm{fn}\ a \Rightarrow e\colon \tau' \to \tau}\ ABS$$

$$\frac{\Delta_1 \rhd e_1\colon \tau_1 \quad \Delta_2 \rhd e_2\colon \tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \rhd\ \mathbf{let}\ a = e_1\ \mathbf{in}\ e_2 : \tau_2}\ LET'$$

$$\frac{\Delta_1 \rhd e_1\colon \tau_1 \quad \Delta_2 \rhd e_2\colon \tau_2 \quad \theta_\tau \tau_1 = \tau,\ \ \text{all } \tau \in \Delta_2(a) \quad a \in fv(e_2)}{(\bigcup_{\tau \in \Delta_2(a)} \theta_\tau \Delta_1) \cup (\Delta_2 \setminus a) \rhd\ \mathbf{let}\ a = e_1\ \mathbf{in}\ e_2 : \tau_2}\ LET$$

Figure 2: Type Definition with Assumption Environments

environment $\Delta$ with $a$ removed from the domain. For any assumption environment $\Delta$ we write $\Delta + [a\colon \pi]$ for the assumption environment that maps the variable $a$ to the set of simple types $\pi$ and otherwise behaves like $\Delta$. We will denote the domain of $\Delta$ by $dom(\Delta)$. Substitutions are extended from type variables to assumption environments in the usual way. An assumption environment $\Delta$ is a *generic instance* of type environment $\Gamma$ (written $\Delta \prec \Gamma$) if and only if for all variables $a$ in the domain of $\Delta$ and for all types $\tau$ in $\Delta(a)$, the type $\tau$ is a generic instance of $\Gamma(a)$. Notice that with this definition, if $\Delta$ is a generic instance of $\Gamma$ then the domain of $\Delta$ is contained in the domain of $\Gamma$.

Figure 2 gives our type definition with assumption environments. Notice that even though the type definition has a polymorphic let construct, it does not use type schemes, only simple types. This is possible because of the use of assumption environments. Rule [LET] is actually a rule scheme, whose correct application requires the existence of a collection of substitutions $\{\theta_\tau : \tau \in \Delta_2(a)\}$ such that the indicated relationships are satisfied.

In Example 3.1, we can see some of the ways that proofs with this type definition are different from proofs in the ML type system. The [VAR] rule in this type definition is more restrictive than the usual type definition in that the type in the assumption environment must correspond exactly to the type of the variable. The [APP] rule requires the same correspondence of the types of the subexpressions as the ML type system; however, now the [APP] rule accumulates the assumption environments for the subexpressions.

**Example 3.1**

$$\frac{\dfrac{}{\begin{array}{r}[f\colon \{(\alpha \to \alpha) \to (\alpha \to \alpha)\}] \\ \rhd f : (\alpha \to \alpha) \to (\alpha \to \alpha)\end{array}}\ VAR \qquad \dfrac{}{[f\colon \{\alpha \to \alpha\}] \rhd f : \alpha \to \alpha}\ VAR}{[f\colon \{(\alpha \to \alpha) \to (\alpha \to \alpha),\ \alpha \to \alpha\}] \rhd f\ f : \alpha \to \alpha}\ APP$$

In Example 3.2, we see how the the [ABS] rule forces all assumptions on the variable $a$ to be identical. The purpose of the [LET] rule is to make sure that the definition of the variable $a$ in expression $e_1$ is consistent with how the variable $a$ is used in expression $e_2$. The use of $a$ is consistent if there exists some substitution that maps the type of the expression $e_1$ to the type of the occurrence of $a$.

**Example 3.2** *Notice that* $[\alpha/\alpha'](\alpha' \to \alpha') = \alpha \to \alpha$ *and* $[(\alpha \to \alpha)/\alpha'](\alpha' \to \alpha') = (\alpha \to \alpha) \to (\alpha \to \alpha)$ *and therefore the necessary substitution conditions for the [LET] are true.*

$$\dfrac{\dfrac{\dfrac{}{[a\colon\{\alpha'\}] \rhd a\colon\alpha'}\ VAR}{[\,] \rhd \text{fn } a \Rightarrow a : \alpha' \to \alpha'}\ ABS \qquad \dfrac{\textit{see Example } 3.1}{[f\colon\{(\alpha \to \alpha) \to (\alpha \to \alpha),\ \alpha \to \alpha\}] \rhd f\ f : \alpha \to \alpha}}{[\,] \rhd \ \mathbf{let}\ f = \text{fn } a \Rightarrow a \ \mathbf{in}\ f\ f : \alpha \to \alpha}\ LET$$

The [LET'] rule is the trivial case of the let definition, where the variable $a$ does not occur free in the $e_2$. In this case, the rule simply accumulates the assumption environments for the subexpressions.

In Example 3.3, we see that if there are variables that occur free in expression $e_1$ of the [LET] definition, then not only does the assumption environment required for the whole let-expression accumulate all the assumptions required about the free variables of the body $e_2$ (other than the bound variable $a$), but in addition it includes all assumptions required about the free variables of $e_1$, where the latter have been suitably strengthened to take into account the consequences of requiring that all the uses of the bound variable $a$ in the body match its definition. We can see here that the size of the assumption environment can in general be exponential in the nesting depth of the [LET] construct.

**Example 3.3** *Notice that* $[(\alpha \to \alpha)/\beta]\beta \to \beta = (\alpha \to \alpha) \to (\alpha \to \alpha)$ *and* $[\alpha/\beta]\beta \to \beta = \alpha \to \alpha$ *and therefore the necessary substitution conditions for the [LET] are true.*

$$\dfrac{\dfrac{\textit{see Example } 3.1}{\begin{array}{r}[g\colon\{(\beta \to \beta) \to (\beta \to \beta),\ \beta \to \beta\}] \\ \rhd g\ g : (\beta \to \beta)\end{array}} \qquad \dfrac{\textit{see Example } 3.1}{\begin{array}{r}[f\colon\{(\alpha \to \alpha) \to (\alpha \to \alpha),\ \alpha \to \alpha\}] \\ \rhd f\ f : (\alpha \to \alpha)\end{array}}}{\begin{array}{r}[g\colon\{((\alpha \to \alpha) \to (\alpha \to \alpha)) \to ((\alpha \to \alpha) \to (\alpha \to \alpha)),\ (\alpha \to \alpha) \to (\alpha \to \alpha),\ \alpha \to \alpha\}] \\ \rhd\ \mathbf{let}\ f = g\ g\ \mathbf{in}\ f\ f\ : (\alpha \to \alpha)\end{array}}\ LET$$

Theorem 1 states that there is a close correspondence between the ML type definition and our type definition with assumption environments. We can view assumption environments as simply more a concrete version of type environments, if we consider generalization a form of abstraction.

**Theorem 1** *If* $\Gamma \vdash e\colon\tau$ *then there exists an assumption environment* $\Delta$ *such that* $\Delta \prec \Gamma$ *and* $\Delta \rhd e\colon\tau$*. Conversely, if* $\Delta \rhd e\colon\tau$ *and* $\Delta \prec \Gamma$ *then* $\Gamma \vdash e\colon\tau$*.*

In the context of the second part of the theorem, observe that for every assumption environment $\Delta$, the type environment $\Gamma$ that maps each $a \in dom(\Delta)$ to the type scheme $\forall\alpha.\alpha$, has the property $\Delta \prec \Gamma$.

**Proof Outline:** Both directions are by induction on the height of a proof tree, and are straightforward once several simple lemmas about the properties of generic instance ($\prec$) are proven. In the forward direction, one builds the assumption environments $\Delta$ required for a proof that $\Delta \rhd e : \tau$ by collecting the assumptions about the free variables that actually appear in [VAR] rules in a given proof that $\Gamma \vdash e : \tau$. For the converse direction, it is merely necessary to observe, that if $\Delta \prec \Gamma$, then $\Gamma$ is already strong enough to imply all the assumptions about free variables appearing in a proof of $\Delta \rhd e : \tau$. See the appendix for the complete proof. □

## 3.2 Type Inference Algorithm

In this section, we present the type inference algorithm that takes a programming language expression as a parameter and returns the principal typing as defined by the type definition with

$$\frac{\beta \text{ is new}}{[a\colon\{\beta\}] \trianglerighteq a\colon\beta}\ VAR$$

$$\frac{\Delta_1 \trianglerighteq e_1\colon\tau_1 \quad \Delta_2 \trianglerighteq e_2\colon\tau_2 \quad \theta = \mathcal{U}\{\tau_1 = \tau_2 \rightarrow \beta\} \quad \beta \text{ is new}}{\theta(\Delta_1 \cup \Delta_2) \trianglerighteq e_1\ e_2\colon\theta\beta}\ APP$$

$$\frac{\Delta \trianglerighteq e\colon\tau \quad \theta = \mathcal{U}\{\beta = \tau \mid \tau \in \Delta(a)\} \quad \beta \text{ is new}}{\theta(\Delta \setminus a) \trianglerighteq \text{fn } a \Rightarrow e\colon\theta\beta \rightarrow \theta\tau}\ ABS$$

$$\frac{\Delta_1 \trianglerighteq e_1\colon\tau_1 \quad \Delta_2 \trianglerighteq e_2\colon\tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \trianglerighteq \textbf{let } a = e_1 \textbf{ in } e_2 : \tau_2}\ LET'$$

$$\frac{\Delta_1 \trianglerighteq e_1\colon\tau_1 \quad \Delta_2 \trianglerighteq e_2\colon\tau_2 \quad \theta = \mathcal{U}\{[\tau_1]_\tau = \tau \mid \tau \in \Delta_2(a)\} \quad a \in fv(e_2)}{\theta\left((\bigcup_{\tau\in\Delta_2(a)}[\Delta_1]_\tau) \cup (\Delta_2 \setminus a)\right) \trianglerighteq \textbf{let } a = e_1 \textbf{ in } e_2 : \theta\tau_2}\ LET$$

Figure 3: Type Inference Algorithm

assumption environments. The algorithm we present is essentially the same as the algorithm presented by Shao and Appel [SA93]. Our main contribution is a simple direct proof of the correctness of the algorithm and a presentation that avoids some of the complexity of their *Polyunify* subroutine.

We first recall some standard facts about unification (see [MM82]). A *constraint* is an equation of the form $\tau = \tau'$, where $\tau$ and $\tau'$ are type expressions. A *constraint set* is a finite set of constraints. A *unifier* for a constraint set is a substitution $\theta$, such that $\theta\tau = \theta\tau'$ for each constraint $\tau = \tau'$ in the set. A constraint set is *unifiable* if it has a unifier. A unifier $\theta$ is a *most general unifier* for a constraint set if and only if for every unifier $\theta'$ for the constraint set, $\theta'$ is a substitution instance of $\theta$. Most general unifiers are unique up to renaming of variables. A *unification algorithm* accepts a constraint set as input and either outputs a most general unifier for the constraint set or else indicates that the constraint set is not unifiable. Unification algorithms exist. We assume that a unification algorithm has been chosen, and we write $\mathcal{U}$ for the function it computes.

If $\tau$ is a type, and $x$ is an element of a given finite index set, then we write $[\tau]_x$ to denote the type obtained from $\tau$ by "tagging" each of the type variables in $\tau$ with the subscript $x$. Since we assume a countably infinite collection of type variables, such a tagging function clearly exists and is computable. We extend the $[\tau]_x$ notation to assumption environments in the obvious way.

Figure 3 presents our type inference algorithm. Though it is presented in the form of inference rules, it is in fact a bottom-up algorithm for computing a typing given an expression. Notice how the unifying substitutions are applied to specialize the assumption environments as we apply each of the rules to infer a typing for an expression from typings for the subexpressions. The most interesting rule is [LET]. The constraint set that is being unified in this rule expresses the requirement that each use of the bound variable $a$ match its definition. The "taggings" $[\tau_1]_\tau$ of the type expression $\tau_1$ are used to allow each occurrence of $a$ in the body to have a type that matches its definition, without forcing all of these types to be equal.

As a result of Theorem 2 below, we know that every programming language expression has a principal typing and that the principal typing is computed by the type inference algorithm.

**Theorem 2** *If $\Delta \rhd e\colon\tau$ then there exists an assumption environment $\Delta'$, type $\tau'$, and substitution $\theta$, such that $\Delta' \unrhd e\colon\tau'$, with $\theta\Delta' = \Delta$ and $\theta\tau' = \tau$. Conversely, if $\Delta \unrhd e\colon\tau$ then $\Delta \rhd e\colon\tau$.*

**Proof Outline:** The proof of both directions is again by induction on the height of a proof tree. For the forward direction, given a proof of $\Delta \rhd e\colon\tau$, we inductively construct a proof of $\Delta' \unrhd e\colon\tau'$, together with the connecting substitution $\theta$. The "most general" property of the unifier computed by $\mathcal{U}$ is used at each stage to "factor out" the connecting substitution for the next stage.

Conversely, a straightforward induction shows that if we have a proof of $\Delta \unrhd e\colon\tau$ then by systematically applying the unifying substitutions used in it to the proof tree we obtain a proof of $\Delta \rhd e\colon\tau$. See the appendix for the complete proof. □

In Example 3.4, we see a similar proof to that in Example 3.1. Notice that the typing in this example is principal and any provable typing of the programming language expression (i.e. Example 3.1) is a substitution instance of the typing presented in the example.

**Example 3.4**

$$\frac{\dfrac{}{[f\colon\{\alpha_1\}] \unrhd f:\alpha_1}\,VAR \quad \dfrac{}{[f\colon\{\alpha_2\}] \unrhd f:\alpha_2}\,VAR \quad [\alpha_2 \rightarrow \alpha_3/\alpha_1] = \mathcal{U}\{\alpha_1 = \alpha_2 \rightarrow \alpha_3\}}{[f\colon\{\alpha_2 \rightarrow \alpha_3,\ \alpha_2\}] \unrhd f\ f:\alpha_3}\,APP$$

# 4 A Prototype Implementation

The main obstacle to a direct, practical implementation of the algorithm of Figure 3 is that the assumption environment can be large – in general the size can be exponential in the nesting depth of the [LET] construct. In their paper, Shao and Appel [SA93] suggest an optimization for removing "isolated" and "redundant" assumptions in order to reduce the size of the assumption environment. As the size explosion arises from the tagging operation performed on $\Delta_1$ in the conclusion of the [LET] rule, which effectively copies the information in $\Delta_1$ a number of times equal to the number of types in the set $\Delta_2(a)$, another idea for avoiding the explosion would be to try to postpone actually "multiplying out" these copies until forced to do so. Thus, it seems that there are reasonable possibilities for avoiding the size explosion in practical situations, and that an efficient, direct implementation of the algorithm of Figure 3 is a worthwhile goal for future research. However, in our attempt to investigate the suitability of principal typings for debugging type errors, we took the alternative approach of introducing simple modifications to Algorithm W so that, in case there are unbound variables, the type information produced is an approximation to the assumption environment of the principal typing.

An approximate implementation has practical value as well, since the user might find it very difficult to understand the implications of a principal typing containing an exponential number of assumptions about an unbound variable. For example, consider the following simple program:

```
let
   fun k a b = a
   val g = k (h 1) (h true)
in
   let val f = k (g 1) (g true)
   in (f 1, f true) end
end
```

The only variable that occurs free in this expression is h, but if we were to compute and print the entire principal typing, the error message would be:

```
Error: unbound variable: h: int -> int -> int -> 'a
Error: unbound variable: h: int -> int -> bool -> 'b
Error: unbound variable: h: int -> bool -> int -> 'a
Error: unbound variable: h: int -> bool -> bool -> 'b
Error: unbound variable: h: bool -> int -> int -> 'a
Error: unbound variable: h: bool -> int -> bool -> 'b
Error: unbound variable: h: bool -> bool -> int -> 'a
Error: unbound variable: h: bool -> bool -> bool -> 'b
val it = 'a * 'b
```

Our original design goal was to produce messages that are easy to understand for the novice programmer, yet unobtrusive when they are not needed. As the example above shows, this goal is not satisfied by error messages annotated with the complete type information, when the assumption environment is very large. Fortunately, in practical situations, the assumption environment often only contains one type for each occurrence of the unbound variable. Notice that this was case in the sample debugging session given in Section 2. Our modified Algorithm W produces the following more manageable output for this example:

```
Error: unbound variable: h: 'int -> 'a
Error: unbound variable: h: 'bool -> 'b
val it = 'e * 'f
```

Our modified version of Algorithm W works as follows: each time an unbound program variable is encountered, a new type variable is generated and assigned as the type for the unbound variable. When the traversal of the expression is complete, the normal unification procedure performed by Algorithm W will in general have resulted in a refinement of the type of the unbound variable from a type variable to some larger type expression. Then, instead of simply reporting "Unbound variable", the system also prints out the final type assigned to that variable. Each type in the principal typing will be a substitution instance of the type produced by the modified Algorithm W. What is omitted by the modified Algorithm W is the propagation of constraints imposed by the body of a let-expression on free variables appearing in the definition part. This avoids the multiplicative effect described above and still produces information of practical value to the user.

Our prototype implementation demonstrates the usefulness of our proposed programming environment. However, it is not clear how much of an issue large assumption environments really are in practice. In the near future, we intend to do a direct implementation of the of the algorithm of Figure 3 in order to investigate these issues.

# 5 Conclusions

In this paper we presented a novel solution to the problem of diagnosing type errors. We presented an unobtrusive extension to the programming environment that facilitates finding the source of type errors. We were able to give a simple description of the programming environment, by means of a type definition and a corresponding type inference algorithm. We also described a prototype implementation of the programming environment which only requires minor changes to the current implementation.

We plan to implement a more realistic version of the programming environment, perhaps in SML of New Jersey. In addition, the programming environment that we propose does not eliminate the need for diagnostic messages and it would be interesting to integrate the work described in this paper with the existing work on diagnostics. Another possible extension would be to describe a similar programming environment for modules.

## References


[BRTT93] Lars Birkedal, Nick Rothwell, Mads Tofte, and David N. Turner. *The ML Kit, Version 1*, 1993.

[BS95] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, Computer Science Department, 1995. http://www.cs.sunysb.edu/~stark/REPORTS/INDEX.html.

[Dam85] Luis Damas. *Type Assignment in Programming Languages.* PhD thesis, University of Edinburgh, Edinburgh, U.K., 1985.

[DB94] Dominic Duggan and Frederick Bent. Explaining type inference. Technical Report CS-94-14, University of Waterloo, Waterloo, Canada, 1994. http://nuada.uwaterloo.ca/dduggan/papers.html.

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

[Jim95] Trevor Jim. What are principal typings and what are they good for? Technical Report MIT/LCS/TM-532, Massachusetts Institute of Technology, Laboratory for Computer Science, August 1995.

[JW86] Greg F. Johnson and Janet Walz. A maximum-flow approach to anomaly isolation in unification-based incremental type inference. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 44–57. Association for Computing Machinery, ACM Press, January 1986.

[Mit90] John C. Mitchell. Type systems for programming languages. In *Handbook of Theoretical Computer Science*, volume B, pages 367–458. Elsevier Science Publishers, 1990.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Transactions on Programming Languages and Systems*, 4(2):258–282, April 1982.

[MTH90] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML.* MIT Press, Cambridge, MA, 1990.

[Rea89] Chris Reade. *Elements of Functional Programming.* Addison-Wesley Publishing Company, 1989.

[SA93] Zhong Shao and Andrew Appel. Smartest recompilation. In *Twentieth Annual ACM Symposium on Principles of Programming Languages.* Association for Computing Machinery, ACM Press, January 1993.

[Tof87] Mads Tofte. *Operational Semantics and Polymorphic Type Inference.* PhD thesis, University of Edinburgh, Edinburgh, U.K., November 1987.

[Wan86] Mitchell Wand. Finding the source of type errors. In *Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 38–43. Association for Computing Machinery, ACM Press, January 1986.

# A Proofs

**Lemma A.1** *Some basic properties of generic instance ($\prec$) that follow immediately from the definitions.*

*Recall that we are using the following naming conventions: $\Gamma$ for type environments, $\Delta$ for assumption environments, $\tau$ for simple types, $\sigma$ for type schemes, $\theta$ for substitutions, $a$ for variables and $\pi$ for sets of simple types.*

1. $\Delta_1 \prec \Gamma$ and $\Delta_2 \prec \Gamma$ *if and only if* $\Delta_1 \cup \Delta_2 \prec \Gamma$
2. *If* $\Delta \prec \Gamma$ and $\Gamma(a) = \tau$ *then* $\Delta(a) \subseteq \{\tau\}$
3. *If* $\Delta \prec \Gamma$ *then* $(\Delta \setminus a) \prec (\Gamma \setminus a)$
4. *For* $\sigma = Gen(\Gamma, \tau')$, $\tau \prec \sigma$ *if and only if* $\exists\theta,\ \theta\tau' = \tau$ *and* $dom(\theta) \cap tyvars(\Gamma) = \emptyset$
5. *If* $\Delta \prec \Gamma$ *and* $(dom(\theta) \cap tyvars(\Gamma)) = \emptyset$ *then* $\theta\Delta \prec \Gamma$
6. *If* $\Delta \prec \Gamma$ *and* $a \notin dom(\Delta)$ *then* $\Delta \prec \Gamma + [a{:}\, \sigma]$
7. *If* $\Delta \prec \Gamma$ *and for all* $\tau \in \pi$, $\tau \prec \sigma$ *then* $\Delta + [a{:}\, \pi] \prec \Gamma + [a{:}\, \sigma]$
8. *If* $\Delta \prec \Gamma$ *and* $dom(\Gamma) = dom(\Delta)$ *then* $tyvar(\Gamma) \subseteq tyvar(\Delta)$

**Lemma A.2** *For all $\Delta$, $e$ and $\tau$ such that $\Delta \triangleright e{:}\, \tau$,*

$$a \in dom(\Delta) \iff a \in fv(e)$$

**Proof:** Straightforward structural induction.

**Lemma A.3** *Substitution Lemma. If $\Delta \triangleright e{:}\, \tau$ then $\theta\Delta \triangleright e{:}\, \theta\tau$.*

**Proof:** Straightforward structural induction.

**Theorem 1** *If $\Gamma \vdash e{:}\, \tau$ then there exists an assumption environment $\Delta$ such that $\Delta \prec \Gamma$ and $\Delta \triangleright e{:}\, \tau$. Conversely, if $\Delta \triangleright e{:}\, \tau$ and $\Delta \prec \Gamma$ then $\Gamma \vdash e{:}\, \tau$.*

**Proof:** The first part of the proof is by induction on the height of the proof tree. For our induction hypothesis we will assume that if we have a proof of height less than $k$ that $\Gamma \vdash e{:}\, \tau$ then there exists an assumption environment $\Delta$ such that $\Delta \prec \Gamma$ and $\Delta \triangleright e{:}\, \tau$. We will then demonstrate that this assumption is sufficient to show that if we have a proof of height $k$ that $\Gamma \vdash e{:}\, \tau$ then there exists an assumption environment $\Delta$ such that $\Delta \prec \Gamma$ and $\Delta \triangleright e{:}\, \tau$. For our proof, we will consider each possible bottom rule in the proof tree separately.

**Case 1** [**VAR**] The proof is of the form:

$$\frac{\tau \prec \Gamma(a)}{\Gamma \vdash a{:}\, \tau}\ VAR$$

therefore we also have:

$$\frac{}{[a{:}\, \{\tau\}] \triangleright a{:}\, \tau}\ VAR$$

where it is immediate from the definition of generic instance that $[a{:}\, \{\tau\}] \prec \Gamma$.

**Case 2** [**APP**] The proof is of the form:

$$\frac{\Gamma \vdash e_1{:}\, \tau' \rightarrow \tau \quad \Gamma \vdash e_2{:}\, \tau'}{\Gamma \vdash e_1\ e_2{:}\, \tau}\ APP$$

By the induction hypothesis, there exists $\Delta_1$ and $\Delta_2$ such that $\Delta_1 \rhd e_1 \colon \tau' \rightarrow \tau$ and $\Delta_2 \rhd e_2 \colon \tau'$, where $\Delta_1 \prec \Gamma$ and $\Delta_2 \prec \Gamma$. Therefore we have:

$$\frac{\Delta_1 \rhd e_1 \colon \tau' \rightarrow \tau \quad \Delta_2 \rhd e_2 \colon \tau'}{\Delta_1 \cup \Delta_2 \rhd e_1\ e_2 \colon \tau}\ APP$$

Where by lemma A.1(1), $\Delta_1 \cup \Delta_2 \prec \Gamma$.

**Case 3 [ABS]** The proof is of the form:

$$\frac{\Gamma + [a \colon \tau'] \vdash e \colon \tau}{\Gamma \vdash \text{fn } a \Rightarrow e \colon \tau' \rightarrow \tau}\ ABS$$

By the induction hypothesis, we have $\Delta \rhd e \colon \tau$ where $\Delta \prec \Gamma + [a \colon \tau']$. By lemma A.1(2), $\Delta(a) \subseteq \{\tau'\}$ and we have:

$$\frac{\Delta \rhd e \colon \tau \quad \Delta(a) \subseteq \{\tau'\}}{\Delta \setminus a \rhd \text{fn } a \Rightarrow e \colon \tau' \rightarrow \tau}\ ABS$$

Since we have $\Delta \prec \Gamma + [a \colon \tau']$, by lemma A.1(3) it follows that $(\Delta \setminus a) \prec \Gamma$.

**Case 4 [LET]** The proof is of the form:

$$\frac{\Gamma \vdash e_1 \colon \tau_1 \quad \Gamma + [a \colon \sigma] \vdash e_2 \colon \tau_2 \quad \sigma = Gen(\Gamma, \tau_1)}{\Gamma \vdash \ \textbf{let}\ a = e_1\ \textbf{in}\ e_2 : \tau_2}\ LET$$

By the induction hypothesis, we have that $\Delta_1 \rhd e_1 \colon \tau_1$ and $\Delta_2 \rhd e_2 \colon \tau_2$ where $\Delta_1 \prec \Gamma$ and $\Delta_2 \prec \Gamma + [a \colon \sigma]$. Let us consider the case where $a \notin fv(e_2)$ separately from $a \in fv(e_2)$.

If $a \notin fv(e_2)$ then we have:

$$\frac{\Delta_1 \rhd e_1 \colon \tau_1 \quad \Delta_2 \rhd e_2 \colon \tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \rhd \ \textbf{let}\ a = e_1\ \textbf{in}\ e_2 : \tau_2}\ LET'$$

Since we have $\Delta_2 \prec \Gamma + [a \colon \sigma]$, by lemma A.1(3), $(\Delta_2 \setminus a) \prec \Gamma$. Since $a \notin fv(e_2)$, by lemma A.2 $a \notin dom(\Delta_2)$ and $\Delta_2 \setminus a = \Delta_2$. Therefore we have that $\Delta_2 \prec \Gamma$ and by lemma A.1(1), $\Delta_1 \cup \Delta_2 \prec \Gamma$.

If $a \in fv(e_2)$ then since $\Delta_2 \prec \Gamma + [a \colon \sigma]$, by lemma A.1(4), we have that for each $\tau$ in $\Delta_2(a)$ there exists some $\theta_\tau$ such that $\theta_\tau \tau_1 = \tau$ and $dom(\theta_\tau) \cap tyvars(\Gamma) = \emptyset$. Therefore we have:

$$\frac{\Delta_1 \rhd e_1 \colon \tau_1 \quad \Delta_2 \rhd e_2 \colon \tau_2 \quad \theta_\tau \tau_1 = \tau,\ \text{ all } \tau \in \Delta_2(a) \quad a \in fv(e_2)}{(\bigcup_{\tau \in \Delta_2(a)} \theta_\tau \Delta_1) \cup (\Delta_2 \setminus a) \rhd \ \textbf{let}\ a = e_1\ \textbf{in}\ e_2 : \tau_2}\ LET$$

Since for each $\tau \in \Delta_2(a)$, $dom(\theta_\tau) \cap tyvars(\Gamma) = \emptyset$, by lemma A.1(5), we have $\theta_\tau \Delta_1 \prec \Gamma$. Since we have that $\Delta_2 \prec \Gamma + [a \colon \sigma]$, by lemma A.1(3), $(\Delta_2 \setminus a) \prec \Gamma$. Since $\Delta_2(a)$ is finite, by repeated application of lemma A.1(1), it follows that $(\bigcup_{\tau \in \Delta_2(a)} \theta_\tau \Delta_1) \cup (\Delta_2 \setminus a) \prec \Gamma$.

**Part 2:** The second part of the proof is also by induction on the height of the proof tree. For our induction hypothesis we will assume that if we have a proof of height less than $k$ that $\Delta \rhd e \colon \tau$ and $\Delta \prec \Gamma$ then $\Gamma \vdash e \colon \tau$. We will then demonstrate that this assumption is sufficient to show that if we have a proof of height $k$ that $\Delta \rhd e \colon \tau$ and $\Delta \prec \Gamma$ then $\Gamma \vdash e \colon \tau$. For our proof, we will consider each possible bottom rule in the proof tree separately.

**Case 1:** [**VAR**] The proof is of the form:

$$\frac{}{[a \colon \{\tau\}] \rhd a \colon \tau}\ VAR$$

For all $\Gamma$ such that $\tau \prec \Gamma(a)$, (that is $[a\colon\{\tau\}] \prec \Gamma$), we have:

$$\frac{\tau \prec \Gamma(a)}{\Gamma \vdash a\colon\tau}\ VAR$$

**Case 2 [APP]** The proof is of the form:

$$\frac{\Delta_1 \rhd e_1\colon\tau' \to \tau \quad \Delta_2 \rhd e_2\colon\tau'}{\Delta_1 \cup \Delta_2 \rhd e_1\ e_2\colon\tau}\ APP$$

By the induction hypothesis, we have that for all $\Gamma_1$ such that $\Delta_1 \prec \Gamma_1$, $\Gamma_1 \vdash e_1\colon\tau' \to \tau$ and for all $\Gamma_2$ such that $\Delta_2 \prec \Gamma_2$, $\Gamma_2 \vdash e_2\colon\tau'$. Choose $\Gamma$ such that $(\Delta_1 \cup \Delta_2) \prec \Gamma$. By lemma A.1(1), $\Delta_1 \prec \Gamma$ and $\Delta_2 \prec \Gamma$, so we have $\Gamma \vdash e_1\colon\tau' \to \tau$ and $\Gamma \vdash e_2\colon\tau'$ and:

$$\frac{\Gamma \vdash e_1\colon\tau' \to \tau \quad \Gamma \vdash e_2\colon\tau'}{\Gamma \vdash e_1\ e_2\colon\tau}\ APP$$

**Case 3 [ABS]** The proof is of the form:

$$\frac{\Delta \rhd e\colon\tau \quad \Delta(a) \subseteq \{\tau'\}}{\Delta \setminus a \rhd \text{fn}\ a \Rightarrow e\colon\tau' \to \tau}\ ABS$$

Choose $\Gamma$ such that $(\Delta \setminus a) \prec \Gamma$. Since $\Delta(a) \subseteq \{\tau'\}$, by A.1(7) we have $\Delta \prec \Gamma + [a\colon\tau']$ and by the induction hypothesis, $\Gamma + [a\colon\tau'] \vdash e\colon\tau$. Therefore:

$$\frac{\Gamma + [a\colon\tau'] \vdash e\colon\tau}{\Gamma \vdash \text{fn}\ a \Rightarrow e\colon\tau' \to \tau}\ ABS$$

**Case 4 [LET']** The proof is of the form:

$$\frac{\Delta_1 \rhd e_1\colon\tau_1 \quad \Delta_2 \rhd e_2\colon\tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \rhd\ \textbf{let}\ a = e_1\ \textbf{in}\ e_2\ \colon\tau_2}\ LET'$$

By the induction hypothesis, we have that for all $\Gamma_1$ such that $\Delta_1 \prec \Gamma_1$, $\Gamma_1 \vdash e_1\colon\tau_1$ and for all $\Gamma_2$ such that $\Delta_2 \prec \Gamma_2$, $\Gamma_2 \vdash e_2\colon\tau_2$. Choose $\Gamma$ such that $(\Delta_1 \cup \Delta_2) \prec \Gamma$. By lemma A.1(1), $\Delta_1 \prec \Gamma$ and $\Delta_2 \prec \Gamma$. Since $a \notin fv(e_2)$, by lemma A.2, $a \notin dom(\Delta)$ and therefore by lemma A.1(6), $\Delta_2 \prec \Gamma + [a\colon\sigma]$. By the induction hypothesis, $\Gamma \vdash e_1\colon\tau_1$ and $\Gamma + [a\colon\sigma] \vdash e_2\colon\tau_2$ and:

$$\frac{\Gamma \vdash e_1\colon\tau_1 \quad \Gamma + [a\colon\sigma] \vdash e_2\colon\tau_2 \quad \sigma = Gen(\Gamma, \tau_1)}{\Gamma \vdash\ \textbf{let}\ a = e_1\ \textbf{in}\ e_2\ \colon\tau_2}\ LET$$

**Case 5 [LET]** The proof is of the form:

$$\frac{\Delta_1 \rhd e_1\colon\tau_1 \quad \Delta_2 \rhd e_2\colon\tau_2 \quad \theta_\tau\tau_1 = \tau,\ \text{ all } \tau \in \Delta_2(a) \quad a \in fv(e_2)}{(\bigcup_{\tau\in\Delta_2(a)} \theta_\tau\Delta_1) \cup (\Delta_2 \setminus a) \rhd\ \textbf{let}\ a = e_1\ \textbf{in}\ e_2\ \colon\tau_2}\ LET$$

By the substitution lemma we can rename type variables such that $tyvars(\Delta_1) \cap tyvars(\Delta_2) = \emptyset$. We can also rename the $\theta_\tau$ appropriately so that $\theta_\tau\tau_1 = \tau$ is true and restrict the $\theta_\tau$ such that $dom(\theta_\tau) \subseteq tyvars(\Delta_1)$ and $ran(\theta_\tau) \subseteq tyvars(\Delta_2)$. Therefore we have that $tyvars(\theta_\tau\Delta_1) \cap dom(\theta_\tau) = \emptyset$.

Choose $\Gamma'$ such that $(\bigcup_{\tau\in\Delta_2(a)} \theta_\tau\Delta_1) \cup (\Delta_2 \setminus a) \prec \Gamma'$ and $dom(\Gamma') = dom((\bigcup_{\tau\in\Delta_2(a)} \theta_\tau\Delta_1) \cup (\Delta_2 \setminus a))$. By lemma A.1(8), $tyvars(\Gamma') \subseteq tyvars((\bigcup_{\tau\in\Delta_2(a)} \theta_\tau\Delta_1) \cup (\Delta_2 \setminus a))$ and therefore $tyvars(\Gamma') \cap dom(\theta_\tau) = \emptyset$. Let $\sigma = Gen(\Gamma, \tau_1)$. By lemma A.1(4), for all $\tau \in \Delta_2(a)$, $\tau \prec \sigma$. Therefore by A.1(7) $\Delta_2 \prec \Gamma' + [a\colon\sigma]$.

Choose $\Gamma$ such that $(\bigcup_{\tau\in\Delta_2(a)} \theta_\tau \Delta_1) \cup (\Delta_2 \setminus a) \prec \Gamma$ ($\Gamma$ does not have the same restriction that $\Gamma'$ had on it's domain). Since $\Gamma$ has a finite domain, by repeated application of lemma A.1(6), we know that $\Delta_1 \prec \Gamma$ and $\Delta_2 \prec \Gamma + [a{:}\,\sigma]$. Therefore by the induction hypothesis, $\Gamma \vdash e_1{:}\,\tau_1$ and $\Gamma + [a{:}\,\sigma] \vdash e_2{:}\,\tau_2$. Therefore we have:

$$\frac{\Gamma \vdash e_1{:}\,\tau_1 \quad \Gamma + [a{:}\,\sigma] \vdash e_2{:}\,\tau_2 \quad \sigma = Gen(\Gamma, \tau_1)}{\Gamma \vdash \textbf{let } a = e_1 \textbf{ in } e_2 : \tau_2}\ LET$$

**Theorem 2** *If $\Delta \triangleright e{:}\,\tau$ then there exists an assumption environment $\Delta'$, type $\tau'$, and substitution $\theta$, such that $\Delta' \trianglerighteq e{:}\,\tau'$, with $\theta\Delta' = \Delta$ and $\theta\tau' = \tau$. Conversely, if $\Delta \trianglerighteq e{:}\,\tau$ then $\Delta \triangleright e{:}\,\tau$.*

**Proof:**

**Part 1:** We can show by induction on the height of the proof that if we have a proof that $\Delta \triangleright e{:}\,\tau$ there exists an assumption environment $\Delta'$, type $\tau'$, and substitution $\psi$, such that $\Delta' \trianglerighteq e{:}\,\tau'$, with $\psi\Delta' = \Delta$ and $\psi\tau' = \tau$.
[**VAR**] The proof is of the form:

$$\frac{}{[a{:}\,\{\tau\}] \triangleright a{:}\,\tau}\ VAR$$

Therefore we have:

$$\frac{\beta \text{ is new}}{[a{:}\,\{\beta\}] \trianglerighteq a{:}\,\beta}\ VAR$$

where $\psi = [\tau/\beta]$.
[**APP**] The proof is of the form:

$$\frac{\Delta_1 \triangleright e_1{:}\,\tau' \to \tau \quad \Delta_2 \triangleright e_2{:}\,\tau'}{\Delta_1 \cup \Delta_2 \triangleright e_1\ e_2{:}\,\tau}\ APP$$

By the induction hypothesis, we know that there exists $\Delta_1'$, $\Delta_2'$, $\tau_1$, $\tau_2$, $\psi_1$ and $\psi_2$ such that

- $\Delta_1' \trianglerighteq e_1{:}\,\tau_1$
- $\Delta_2' \trianglerighteq e_2{:}\,\tau_2$
- $\psi_1\Delta_1' = \Delta_1$
- $\psi_1\tau_1 = \tau' \to \tau$
- $\psi_2\Delta_2' = \Delta_2$
- $\psi_2\tau_2 = \tau'$

We may assume, without loss of generality, that the type variables that occur in $\Delta_1'$ and $\tau_1$ are distinct from the type variables that occur in $\Delta_2'$ and $\tau_2$, since we can rename type variables, if necessary, by the substitution lemma (lemma A.3). Notice that since these sets of type variables are disjoint, the substitutions $\psi_1$ and $\psi_2$ commute.

Let $\psi' = (\psi_1 \circ \psi_2) + [\tau/\beta]$. Now we have that $\psi'\tau_1 = \tau' \to \tau$ and $\psi'(\tau_2 \to \beta) = \tau' \to \tau$ so $\tau_1$ and $\tau_2 \to \beta$ are unifiable and there exists some $\theta$ such that:

$$\frac{\Delta_1' \trianglerighteq e_1{:}\,\tau_1 \quad \Delta_2' \trianglerighteq e_2{:}\,\tau_2 \quad \theta = \mathcal{U}\{\tau_1 = \tau_2 \to \beta\} \quad \beta \text{ is new}}{\theta\Delta_1' \cup \theta\Delta_2' \trianglerighteq e_1\ e_2{:}\,\theta\beta}\ APP$$

Since $\theta$ is the most general unifier for $\tau_1$ and $\tau_2 \to \beta$ we know there exists some $\psi$ such that $\psi' = \psi \circ \theta$. Therefore we have, $\psi(\theta\Delta_1' \cup \theta\Delta_2') = \Delta_1 \cup \Delta_2$ and $\psi(\theta\beta) = \tau$.
[**ABS**] The proof is of the form:

$$\frac{\Delta \triangleright e{:}\,\tau \quad \Delta(a) \subseteq \{\tau'\}}{\Delta \setminus a \triangleright \text{fn } a \Rightarrow e{:}\,\tau' \to \tau}\ ABS$$

First let us consider the trivial case where $\Delta(a)$ is empty and therefore the set $\{\beta = \tau \mid \tau \in \Delta(a)\}$ is trivially unifiable by the empty substitution, *id*. We immediately have:

$$\frac{\Delta \trianglerighteq e{:}\,\tau \quad id = \mathcal{U}\{\beta = \tau \mid \tau \in \Delta(a)\} \quad \beta \text{ is new}}{\Delta \setminus a \trianglerighteq \text{fn } a \Rightarrow e{:}\,\beta \to \tau} \; ABS$$

and $\psi = [\tau'/\beta]$.

Now let us consider the case where $\Delta(a) = \{\tau'\}$. By the induction hypothesis, we know there exists $\Delta_1, \tau_1, \psi_1$ such that

- $\Delta_1 \trianglerighteq e{:}\,\tau_1$
- $\psi_1 \Delta_1 = \Delta$
- $\psi_1 \tau_1 = \tau$

Let $\psi' = \psi_1 + [\tau'/\beta]$. Therefore $\psi'(\Delta_1(a)) = \Delta(a) = \{\tau'\}$ and $\{\beta = \tau \mid \tau \in \Delta_1(a)\}$ is unifiable, so there exists some $\theta$ such that:

$$\frac{\Delta_1 \trianglerighteq e{:}\,\tau_1 \quad \theta = \mathcal{U}\{\beta = \tau \mid \tau \in \Delta_1(a)\} \quad \beta \text{ is new}}{\theta\Delta_1 \setminus a \trianglerighteq \text{fn } a \Rightarrow e{:}\,\theta\beta \to \theta\tau_1} \; ABS$$

Since $\theta$ is the most general unifier of $\{\beta = \tau \mid \tau \in \Delta_1(a)\}$ there exists some $\psi$ such that $\psi' = \psi \circ \theta$. Therefore we have $\psi(\theta\Delta_1 \setminus a) = \Delta \setminus a$ and $\psi(\theta\beta \to \theta\tau_1) = \tau' \to \tau$.
[**LET'**] The proof is of the form:

$$\frac{\Delta_1 \triangleright e_1{:}\,\tau_1 \quad \Delta_2 \triangleright e_2{:}\,\tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \triangleright \textbf{ let } a = e_1 \textbf{ in } e_2 : \tau_2} \; LET'$$

By the induction hypothesis, we know that there exists $\Delta_1'$, $\Delta_2'$, $\tau_1'$, $\tau_2'$, $\psi_1$ and $\psi_2$ such that

- $\Delta_1' \trianglerighteq e_1{:}\,\tau_1'$
- $\Delta_2' \trianglerighteq e_1{:}\,\tau_2'$
- $\psi_1 \Delta_1' = \Delta_1$
- $\psi_1 \tau_1' = \tau_1$
- $\psi_2 \Delta_2' = \Delta_2$
- $\psi_2 \tau_2' = \tau_2$

Therefore we have:

$$\frac{\Delta_1' \trianglerighteq e_1{:}\,\tau_1' \quad \Delta_2' \trianglerighteq e_2{:}\,\tau_2' \quad a \notin fv(e_2)}{\Delta_1' \cup \Delta_2' \trianglerighteq \textbf{ let } a = e_1 \textbf{ in } e_2 : \tau_2'} \; LET'$$

We may assume, without loss of generality, that the type variables that occur in $\Delta_1'$ and $\tau_1'$ are distinct from the type variables that occur in $\Delta_2'$ and $\tau_2'$, since we can rename type variables, if necessary, by the substitution lemma (lemma A.3). Notice that since these sets of type variables are disjoint, the substitutions $\psi_1$ and $\psi_2$ commute.

Let $\psi = \psi_1 \circ \psi_2$. Therefore we have $\psi(\Delta_1' \cup \Delta_2') = \Delta_1 \cup \Delta_2$ and $\psi\tau_2' = \tau_2$.
[**LET**] The proof is of the form:

$$\frac{\Delta_1 \triangleright e_1{:}\,\tau_1 \quad \Delta_2 \triangleright e_2{:}\,\tau_2 \quad \theta_\tau \tau_1 = \tau, \;\text{ all } \tau \in \Delta_2(a) \quad a \in fv(e_2)}{(\bigcup_{\tau \in \Delta_2(a)} \theta_\tau \Delta_1) \cup (\Delta_2 \setminus a) \triangleright \textbf{ let } a = e_1 \textbf{ in } e_2 : \tau_2} \; LET$$

By the induction hypothesis, we know that there exists $\Delta_1'$, $\Delta_2'$, $\tau_1'$, $\tau_2'$, $\psi_1$ and $\psi_2$ such that

- $\Delta_1' \trianglerighteq e_1{:}\,\tau_1'$
- $\Delta_2' \trianglerighteq e_1{:}\,\tau_2'$

- $\psi_1 \Delta_1' = \Delta_1$
- $\psi_1 \tau_1' = \tau_1$
- $\psi_2 \Delta_2' = \Delta_2$
- $\psi_2 \tau_2' = \tau_2$

We may assume, without loss of generality, that the type variables that occur in $\Delta_1'$ and $\tau_1'$ are distinct from the type variables that occur in $\Delta_2'$ and $\tau_2'$, since we can rename type variables, if necessary, by the substitution lemma (lemma A.3). Notice that since these sets of type variables are disjoint, the substitutions $\psi_1$ and $\psi_2$ commute. We may also assume, without loss of generality, that $dom(\theta_\tau) \subseteq tyvars(\Delta_1)$.

Let $\psi_{\tau'} = \theta_{(\psi_2\tau')} \circ ((\psi_1 \circ [\ ]_{\tau'}^{-1}) \circ \psi_2)$ and let $\psi'$ be the composition of the $\psi_{\tau'}$ for all $\tau' \in \Delta_2'(a)$ where $[\ ]_{\tau'}^{-1}$ is the function that removes the subscript $\tau'$. We have that $\psi'$ unifies $\{[\tau_1']_{\tau'} = \tau' \mid \tau' \in \Delta_2'(a)\}$ and therefore there exists some $\theta$ such that:

$$\frac{\Delta_1' \trianglerighteq e_1 : \tau_1' \quad \Delta_2' \trianglerighteq e_2 : \tau_2' \quad \theta = \mathcal{U}\{[\tau_1']_{\tau'} = \tau' \mid \tau' \in \Delta_2'(a)\} \quad a \in fv(e_2)}{\theta\left((\bigcup_{\tau' \in \Delta_2'(a)} [\Delta_1']_{\tau'}) \cup (\Delta_2' \setminus a)\right) \trianglerighteq \textbf{let } a = e_1 \textbf{ in } e_2 : \theta\tau_2'}\ LET$$

Since $\theta$ is the most general unifier of $\{[\tau_1']_{\tau'} = \tau' \mid \tau' \in \Delta_2'(a)\}$ there exists some $\psi$ such that $\psi' = \psi \circ \theta$. Therefore we have $\psi(\theta\left((\bigcup_{\tau' \in \Delta_2'(a)} [\Delta_1']_{\tau'}) \cup (\Delta_2' \setminus a)\right)) = (\bigcup_{\tau \in \Delta_2(a)} \theta_\tau \Delta_1) \cup (\Delta_2 \setminus a)$ and $\psi(\theta\tau_2') = \tau_2$.

**Part 2:** We can show by induction on the height of the proof that if we have a proof that $\Delta \trianglerighteq e : \tau$ then we can construct a proof that $\Delta \triangleright e : \tau$.

[**VAR**] The proof is of the form:

$$\frac{\beta \text{ is new}}{[a : \{\beta\}] \trianglerighteq a : \beta}\ VAR$$

Therefore we have

$$\frac{}{[a : \{\beta\}] \triangleright a : \beta}\ VAR$$

[**APP**] The proof is of the form:

$$\frac{\Delta_1 \trianglerighteq e_1 : \tau_1 \quad \Delta_2 \trianglerighteq e_2 : \tau_2 \quad \theta = \mathcal{U}(\tau_1 = \tau_2 \to \beta) \quad \beta \text{ is new}}{\theta(\Delta_1 \cup \Delta_2) \trianglerighteq e_1\ e_2 : \theta\beta}\ APP$$

By the induction hypothesis, we know that $\Delta_1 \triangleright e_1 : \tau_1$ and $\Delta_2 \triangleright e_2 : \tau_2$. Therefore by the substitution lemma we have $\theta\Delta_1 \triangleright e_1 : \theta\tau_2 \to \theta\beta$ and $\theta\Delta_2 \triangleright e_2 : \theta\tau_2$.

$$\frac{\theta\Delta_1 \triangleright e_1 : \theta\tau_2 \to \theta\beta \quad \theta\Delta_2 \triangleright e_2 : \theta\tau_2}{\theta(\Delta_1 \cup \Delta_2) \triangleright e_1\ e_2 : \theta\beta}\ APP$$

[**ABS**] The proof is of the form:

$$\frac{\Delta \trianglerighteq e : \tau \quad \theta = \mathcal{U}\{\beta = \tau \mid \tau \in \Delta(a)\} \quad \beta \text{ is new}}{\theta(\Delta \setminus a) \trianglerighteq \text{fn } a \Rightarrow e : \theta\beta \to \theta\tau}\ ABS$$

By the induction hypothesis, we know that $\Delta \triangleright e : \tau$. Therefore by the substitution lemma we have $\theta\Delta \triangleright e : \theta\tau$. By construction $\theta$ is the most general unifier of the elements of $\Delta(a)$, therefore $\theta\Delta(a) \subseteq \{\theta\beta\}$. and we have:

$$\frac{\theta\Delta \triangleright e : \theta\tau \quad \theta\Delta(a) \subseteq \{\theta\beta\}}{\theta\Delta \setminus a \triangleright \text{fn } a \Rightarrow e : \theta\beta \to \theta\tau}\ ABS$$

[**LET'**] The proof is of the form:

$$\frac{\Delta_1 \;\underline{\triangleright}\, e_1 : \tau_1 \quad \Delta_2 \;\underline{\triangleright}\, e_2 : \tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \;\underline{\triangleright}\; \textbf{let } a = e_1 \textbf{ in } e_2 : \tau_2} \; LET'$$

Therefore by the induction hypothesis we immediately have:

$$\frac{\Delta_1 \triangleright e_1 : \tau_1 \quad \Delta_2 \triangleright e_2 : \tau_2 \quad a \notin fv(e_2)}{\Delta_1 \cup \Delta_2 \triangleright \textbf{let } a = e_1 \textbf{ in } e_2 : \tau_2} \; LET'$$

[**LET**] The proof is of the form:

$$\frac{\Delta_1 \;\underline{\triangleright}\, e_1 : \tau_1 \quad \Delta_2 \;\underline{\triangleright}\, e_2 : \tau_2 \quad \theta = \mathcal{U}\{[\tau_1]_\tau = \tau \mid \tau \in \Delta_2(a)\} \quad a \in fv(e_2)}{\theta\left((\bigcup_{\tau \in \Delta_2(a)} [\Delta_1]_\tau) \cup (\Delta_2 \setminus a)\right) \;\underline{\triangleright}\; \textbf{let } a = e_1 \textbf{ in } e_2 : \theta\tau_2} \; LET$$

By the induction hypothesis, we know that $\Delta_1 \triangleright e_1 : \tau_1$ and $\Delta_2 \triangleright e_2 : \tau_2$. By the substitution lemma we have $\theta\Delta_2 \triangleright e_2 : \theta\tau_2$. Let $\theta_\tau = \theta \circ [\,]_\tau$. Therefore we have:

$$\frac{\Delta_1 \triangleright e_1 : \tau_1 \quad \theta\Delta_2 \triangleright e_2 : \theta\tau_2 \quad \theta_\tau \tau_1 = \tau, \text{ all } \tau \in \Delta_2(a) \quad a \in fv(e_2)}{(\bigcup_{\tau \in \Delta_2(a)} \theta_\tau \Delta_1) \cup (\theta\Delta_2 \setminus a) \triangleright \textbf{let } a = e_1 \textbf{ in } e_2 : \tau_2} \; LET$$

$\square$