Proofs about a Folklore Let-Polymorphic Type Inference Algorithm

OUKSEH LEE and KWANGKEUN YI
Korea Advanced Institute of Science and Technology

The Hindley/Milner let-polymorphic type inference system has two different algorithms: one is the *de facto* standard Algorithm $\mathcal{W}$ that is bottom-up (or context-insensitive), and the other is a "folklore" algorithm that is top-down (or context-sensitive). Because the latter algorithm has not been formally presented with its soundness and completeness proofs, and its relation with the $\mathcal{W}$ algorithm has not been rigorously investigated, its use in place of (or in combination with) $\mathcal{W}$ is not well founded. In this article, we formally define the context-sensitive, top-down type inference algorithm (named "$\mathcal{M}$"), prove its soundness and completeness, and show a distinguishing property that $\mathcal{M}$ always stops earlier than $\mathcal{W}$ if the input program is ill typed. Our proofs can be seen as theoretical justifications for various type-checking strategies being used in practice.

Categories and Subject Descriptors: D.3.3 [**Programming Languages**]: Language Constructs and Features—*data types and structures*; F.3.3 [**Logics and Meaning of Programs**]: Studies of Program Constructs—*type structure*

General Terms: Algorithms, Languages, Theory

Additional Key Words and Phrases: Type error, type inference algorithm

1. INTRODUCTION

Algorithm $\mathcal{W}$, which is the standard presentation of the Hindley/Milner let-polymorphic type inference system, fails late if the input program has a type error. Because the algorithm fails only at an application expression where its two subexpressions (function and argument) have conflicting types, an erroneous expression is often successfully type-checked long before its consequence collides at an application expression. This "bottom-up" Algorithm $\mathcal{W}$ thus reports the whole application expression as the problem area, implying some of its subexpressions are ill typed. Such a large type-error message does not help the programmer to find the cause of the type problem.

A different type inference algorithm, which has been used in an early ML compiler [Leroy 1993], can cure this problem. This folklore algorithm carries a type

This work is supported in part by Korea Science and Engineering Foundation grant KOSEF 961-0100-001-2, by Korea Ministry of Information and Communication grant 96151-IT2-12, by Samsung Electronics Corp., by LG Information & Communications, and by KAIST Center for Artificial Intelligence Research.
Authors' addresses: Department of Computer Science, KAIST, Taejon 305-701, Korea; email: {cookcu; kwang}@cs.kaist.ac.kr.

```
#let rec fac n = if n = 0 then 1 else n *(fac(n=1));;
Toplevel input:
>let rec fac n = if n = 0 then 1 else n *(fac(n=1));;
>         ^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^^
This expression has type int -> int,
but is used with type bool -> int.
```

(a) from $\mathcal{W}$ (CamlLight 0.71)

```
#let rec fac n = if n = 0 then 1 else n *(fac(n=1));;
Toplevel input
>let rec fac n = if n = 0 then 1 else n *(fac(n=1));;
>                                            ^
Expression of type 'a -> 'a -> bool
cannot be used with type 'a -> 'a -> int
```

(b) from $\mathcal{M}$ (CamlLight 0.61)

Fig. 1. Different type-error messages from $\mathcal{W}$ and $\mathcal{M}$.

constraint (or an expected type) implied by the context of an expression down to its sub-or-sibling expressions. For example, for an application expression "$e_1\, e_2$" with a type constraint, say of `int`, the type constraint for e_1 is $\alpha \to \texttt{int}$ and the constraint for e_2 is the type that the α becomes after the type inference of e_1. For a constant or a variable expression, its type must satisfy the type constraint that the algorithm has carried to that point. Because of this "top-down" nature we name this algorithm "$\mathcal{M}$."

In this article we formally define algorithm $\mathcal{M}$, prove its soundness and completeness, and show that it finds type errors earlier than $\mathcal{W}$. This property implies that this algorithm in combination with $\mathcal{W}$ can generate *strictly* more informative type-error messages than either of the two algorithms alone can.

As an example to show the difference of the two algorithms, see Figure 1. The program is a factorial function whose recursive call is mistakenly "`fac(n=1)`," instead of "`fac(n-1)`." Algorithm $\mathcal{W}$ (CamlLight 0.71 [Leroy 1995] and SML/NJ 0.93 [MacQueen and Appel 1993]) reports the whole definition as the problem area because the algorithm fails to unify the argument type `bool` inferred from the recursive call "`fac(n=1)`" with the type `int` inferred from the argument use "`if n = 0`$\cdots$." On the other hand, algorithm $\mathcal{M}$ (CamlLight 0.61 [Leroy 1993]) pinpoints the operator "`=`" as the problem spot. This exact error message is possible because the type constraint of the function's argument is `int` when the argument "`(n=1)`" of the recursive call is type-checked.

2. THE $\mathcal{M}$ ALGORITHM

2.1 Overview

Algorithm $\mathcal{M}$ carries a type constraint from the context of an expression and stops when the expression cannot satisfy the current type constraint. Consider the following expression:

$$\underbrace{\texttt{(fn x => x+1)}}_{e_1}\ \underbrace{\texttt{(}\overbrace{\texttt{(fn y => if y then true else false)}}^{e_3}\ \overbrace{\texttt{false}}^{e_4}\texttt{)}}_{e_2}$$

The expression e_1 must be a function expression; thus $\mathcal{M}$ infers its type with the constraint $\alpha \to \beta$. The inference will succeed with substitution $\{\alpha \mapsto \mathtt{int}, \beta \mapsto \mathtt{int}\}$. This imposes the constraint that the argument expression e_2 must be type `int`. Thus $\mathcal{M}$ infers the type of e_2 with the constraint `int`. This, in turn, makes $\mathcal{M}$ infer the type of e_3 with the constraint $\gamma \to \mathtt{int}$. But the `then`-branch expression in the function's body is boolean; thus $\mathcal{M}$ stops at the `true` expression with a type error.

One characteristic of $\mathcal{M}$ is that a type constraint that is derived from the current context dominates in subsequent steps. For example, given an expression "`f(false,1,2)`" where the type of the `f` is $\alpha \times \alpha \times \alpha \to \alpha$, $\mathcal{M}$ reports that the `1` must have the `bool` type, because the constraint from the "`false`" expression forces the subsequent sibling expressions to have the same type. In comparison, Johnson and Walz's unification algorithm [Johnson and Walz 1986] reports that the "`false`" expression must have the type `int`, because it selects the most "popular" types if multiple, conflicting types are bound to a type variable.

2.2 Notation

We use conventional notation. Vector $\vec{\alpha}$ is a shorthand for $\{\alpha_1, \cdots, \alpha_n\}$, and $\forall\vec{\alpha}.\tau$ is for $\forall\alpha_1 \cdots \alpha_n.\tau$. Equality of type schemes is up to renaming of bound variables. For a type scheme $\sigma = \forall\vec{\alpha}.\tau$, the set $\mathit{ftv}(\sigma)$ of free type variables in σ is $\mathit{ftv}(\tau) \setminus \vec{\alpha}$, where $\mathit{ftv}(\tau)$ is the set of type variables in type τ. For a type environment Γ, $\mathit{ftv}(\Gamma) = \bigcup_{x \in dom(\Gamma)} \mathit{ftv}(\Gamma(x))$. A substitution $\{\tau_i/\alpha_i \mid 1 \le i \le n\}$ substitutes type τ_i for type variable α_i. We write $\{\vec{\tau}/\vec{\alpha}\}$ as a shorthand for a substitution $\{\tau_i/\alpha_i \mid 1 \le i \le n\}$, where $\vec{\alpha}$ and $\vec{\tau}$ have the same length n and $R\vec{\alpha}$ for $\{R\alpha_1, \cdots, R\alpha_n\}$. For a substitution S, the support $\mathit{supp}(S)$ is $\{\alpha \mid S\alpha \ne \alpha\}$, and the set $\mathit{itv}(S)$ of involved type variables is $\{\alpha \mid \beta \in \mathit{supp}(S), \alpha \in \{\beta\} \cup \mathit{ftv}(S\beta)\}$. For a substitution S and a type τ, $S\tau$ is the type resulting from applying every substitution component τ_i/α_i in S to τ. Hence, $\{\}\tau = \tau$. For a substitution S and a type scheme σ, $S\sigma = \forall\vec{\beta}.S\{\vec{\beta}/\vec{\alpha}\}\tau$, where $\vec{\beta} \cap (\mathit{itv}(S) \cup \mathit{ftv}(\sigma)) = \emptyset$. For a substitution S and a type environment Γ, $S\Gamma = \{x \mapsto S\sigma \mid x \mapsto \sigma \in \Gamma\}$. The composition of substitutions S followed by R is written as RS, which is $\{R(S\alpha)/\alpha \mid \alpha \in \mathit{supp}(S)\} \cup \{R\alpha/\alpha \mid \alpha \in \mathit{supp}(R) \setminus \mathit{supp}(S)\}$. Two substitutions S and R are equal if and only if $S\alpha = R\alpha$ for every $\alpha \in \mathit{supp}(S) \cup \mathit{supp}(R)$. For a substitution P and a set of type variables V, we write $P|_V$ for $\{\tau/\alpha \in P \mid \alpha \notin V\}$. The notation $\forall\vec{\alpha}.\tau' \succ \tau$ means that there exists a substitution S such that $S\tau' = \tau$ and $\mathit{supp}(S) \subseteq \vec{\alpha}$. We write $\Gamma + x{:}\sigma$ to mean $\{y \mapsto \sigma' \mid x \ne y, y \mapsto \sigma' \in \Gamma\} \cup \{x \mapsto \sigma\}$. $\mathit{Clos}_\Gamma(\tau)$ is the same as $\mathrm{Gen}(\Gamma,\tau)$ in Damas and Milner [1982], i.e., $\forall\vec{\alpha}.\tau$, where $\vec{\alpha} = \mathit{ftv}(\tau) \setminus \mathit{ftv}(\Gamma)$.

2.3 Algorithm Definition

The source language, its Hindley/Milner-style let-polymorphic type system, and Algorithm $\mathcal{W}$ are shown in Figure 2. Algorithm $\mathcal{M}$ is shown in Figure 3.

Algorithm $\mathcal{M}$ returns a substitution from three components: an expression, a type environment, and a type constraint. The inferred type of the expression is achieved by applying the result substitution to the type constraint of the expression. The type constraints are just types. Note that the algorithm does not unify types at application expressions. Instead, it unifies at constant, variable, and lambda

Abstract Syntax

$$
\begin{array}{llcll}
\textit{Expr} & e & ::= & () & \text{constant} \\
 & & | & x & \text{variable} \\
 & & | & \lambda x.e & \text{function} \\
 & & | & e\ e & \text{application} \\
 & & | & \texttt{let}\ x \texttt{=} e\ \texttt{in}\ e & \\
 & & | & \texttt{fix}\ f\ \lambda x.e & \\
\textit{Type} & \tau & ::= & \iota & \text{constant type} \\
 & & | & \alpha & \text{type variable} \\
 & & | & \tau \to \tau & \text{function type} \\
\textit{TypeScheme} & \sigma & ::= & \tau \mid \forall \vec{\alpha}.\sigma & \\
\textit{TypeEnv} & \Gamma & \in & \textit{Var} \xrightarrow{\text{fin}} \textit{TypeScheme} & \text{type environment}
\end{array}
$$

$$
\begin{array}{ll}
\text{(CON)} & \Gamma \vdash () : \iota \\[1ex]
\text{(VAR)} & \dfrac{\Gamma(x) \succ \tau}{\Gamma \vdash x : \tau} \\[2ex]
\text{(FN)} & \dfrac{\Gamma + x{:}\tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2} \\[2ex]
\text{(APP)} & \dfrac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2} \\[2ex]
\text{(LET)} & \dfrac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x{:}\,\textit{Clos}_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2 : \tau_2} \\[2ex]
\text{(FIX)} & \dfrac{\Gamma + f{:}\tau \vdash \lambda x.e : \tau}{\Gamma \vdash \texttt{fix}\ f\ \lambda x.e : \tau}
\end{array}
$$

$$\mathcal{W}\colon \textit{TypEnv} \times \textit{Expr} \to \textit{Subst} \times \textit{Type}$$

$$
\begin{array}{lll}
\mathcal{W}(\Gamma, ()) & = & (id, \iota) \\
\mathcal{W}(\Gamma, x) & = & (id, \{\vec{\beta}/\vec{\alpha}\}\tau) \ \text{ where } \Gamma(x) = \forall\vec{\alpha}.\tau,\ \text{new } \vec{\beta} \\
\mathcal{W}(\Gamma, \lambda x.e) & = & \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma + x{:}\beta, e),\ \text{new } \beta \\
 & & \text{in } (S_1, S_1\beta \to \tau_1) \\
\mathcal{W}(\Gamma, e_1\ e_2) & = & \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \\
 & & \qquad (S_2, \tau_2) = \mathcal{W}(S_1\Gamma, e_2) \\
 & & \qquad S_3 = \mathcal{U}(S_2\tau_1, \tau_2 \to \beta),\ \text{new } \beta \\
 & & \text{in } (S_3S_2S_1, S_3\beta) \\
\mathcal{W}(\Gamma, \texttt{let}\ x \texttt{=} e_1\ \texttt{in}\ e_2) = & & \\
 & & \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma, e_1) \\
 & & \qquad (S_2, \tau_2) = \mathcal{W}(S_1\Gamma + x{:}\,\textit{Clos}_{S_1\Gamma}(\tau_1), e_2) \\
 & & \text{in } (S_2S_1, \tau_2) \\
\mathcal{W}(\Gamma, \texttt{fix}\ f\ \lambda x.e) & = & \text{let } (S_1, \tau_1) = \mathcal{W}(\Gamma + f{:}\beta, \lambda x.e),\ \text{new } \beta \\
 & & \qquad S_2 = \mathcal{U}(S_1\beta, \tau_1) \\
 & & \text{in } (S_2S_1, S_2\tau_1)
\end{array}
$$

Fig. 2. The language, its type inference rule, and Algorithm $\mathcal{W}$. Every new type variable is distinct from each other, and the set *New* of new type variables introduced at each recursive call to $\mathcal{W}(\Gamma, e)$ satisfies $\textit{New} \cap \textit{ftv}(\Gamma) = \emptyset$.

expressions.

Consider the variable case ($\mathcal{M}$.2). The current type constraint ρ and the x's type $\Gamma(x)$ must agree: $\mathcal{U}(\rho, \{\vec{\beta}/\vec{\alpha}\}\tau)$. For the lambda case ($\mathcal{M}$.3), the first thing is to check if the current type constraint ρ is a function type. The unification

$$\mathcal{M}: \mathit{TypEnv} \times \mathit{Expr} \times \mathit{Type} \to \mathit{Subst}$$

$$\begin{array}{lll}
\mathcal{M}(\Gamma, (), \rho) = & \mathcal{U}(\rho, \iota) & (\mathcal{M}.1)\\
\mathcal{M}(\Gamma, x, \rho) = & \mathcal{U}(\rho, \{\vec{\beta}/\vec{\alpha}\}\tau) \text{ where } \Gamma(x) = \forall\vec{\alpha}.\tau, \text{ new } \vec{\beta} & (\mathcal{M}.2)\\
\mathcal{M}(\Gamma, \lambda x.e, \rho) = & \text{let } S_1 = \mathcal{U}(\rho, \beta_1 \to \beta_2), \text{ new } \beta_1, \beta_2 & (\mathcal{M}.3)\\
& \quad\; S_2 = \mathcal{M}(S_1\Gamma + x{:}\, S_1\beta_1, e, S_1\beta_2) & (\mathcal{M}.4)\\
& \text{in } S_2S_1 & \\
\mathcal{M}(\Gamma, e_1\ e_2, \rho) = & \text{let } S_1 = \mathcal{M}(\Gamma, e_1, \beta \to \rho), \text{ new } \beta & (\mathcal{M}.5)\\
& \quad\; S_2 = \mathcal{M}(S_1\Gamma, e_2, S_1\beta) & (\mathcal{M}.6)\\
& \text{in } S_2S_1 & \\
\multicolumn{2}{l}{\mathcal{M}(\Gamma, \texttt{let } x = e_1 \texttt{ in } e_2, \rho) =} & \\
& \text{let } S_1 = \mathcal{M}(\Gamma, e_1, \beta), \text{ new } \beta & (\mathcal{M}.7)\\
& \quad\; S_2 = \mathcal{M}(S_1\Gamma + x{:}\, \mathit{Clos}_{S_1\Gamma}(S_1\beta), e_2, S_1\rho) & (\mathcal{M}.8)\\
& \text{in } S_2S_1 & \\
\mathcal{M}(\Gamma, \texttt{fix } f\ \lambda x.e, \rho) = & \mathcal{M}(\Gamma + f{:}\,\rho, \lambda x.e, \rho) & (\mathcal{M}.9)
\end{array}$$

Fig. 3. Algorithm $\mathcal{M}$. Every new type variable is distinct from each other, and the set *New* of new type variables introduced at each recursive call to $\mathcal{M}(\Gamma, e, \rho)$ satisfies $\mathit{New} \cap (\mathit{ftv}(\Gamma) \cup \mathit{ftv}(\rho)) = \emptyset$.

$$\mathcal{U}(\rho, \beta_1 \to \beta_2), \text{ new } \beta_1, \beta_2 \qquad (\mathcal{M}.3)$$

does this job. For the application case ($\mathcal{M}.5$), the current type constraint ρ becomes the range part of the new constraint $\beta \to \rho$ for the function expression e_1:

$$\mathcal{M}(\Gamma, e_1, \beta \to \rho), \text{ new } \beta. \qquad (\mathcal{M}.5)$$

The constraint for the argument expression e_2 is the type $S_1\beta$. For the `let` case ($\mathcal{M}.7$), the type constraint for the binding expression e_1 is null,

$$\mathcal{M}(\Gamma, e_1, \beta), \text{ new } \beta, \qquad (\mathcal{M}.7)$$

because no constraint about the type of e_1 is available. The constraint for the `let`-body e_2 is the type $S_1\rho$. For the `fix` case ($\mathcal{M}.9$), the constraint of the expression $\lambda x.e$ is the same as the given constraint ρ.

3. SOUNDNESS AND COMPLETENESS

Algorithm $\mathcal{M}$ is sound and complete with respect to the let-polymorphic type inference system of Figure 2.

THEOREM 1 (SOUNDNESS OF $\mathcal{M}$). *Let e be an expression and Γ be a type environment. If there exists a type ρ such that $\mathcal{M}(\Gamma, e, \rho) = S$, then $S\Gamma \vdash e : S\rho$.*

The proof uses Lemma 1 and Lemma 2.

LEMMA 1 [DAMAS AND MILNER 1982]. *If S is a substitution and $\Gamma \vdash e : \tau$, then $S\Gamma \vdash e : S\tau$.*

LEMMA 2 [MILNER 1978]. *Let S be a substitution, Γ be a type environment, and τ be a type. $S\,\mathit{Clos}_\Gamma(\tau) = \mathit{Clos}_{S'\Gamma}(S'\tau)$, where $S' = S\{\vec{\beta}/\vec{\alpha}\}$, $\vec{\alpha} = \mathit{ftv}(\tau) \setminus \mathit{ftv}(\Gamma)$ and $\vec{\beta}$ is new.*

PROOF OF THEOREM 1. We prove by structural induction on e.

—**case** $()$: $S\rho = S\iota = \iota$. So $S\Gamma \vdash () : S\rho$ by (CON).

—**case** x: $S\rho = S\{\vec{\beta}/\vec{\alpha}\}\tau \prec S\Gamma(x)$. By (VAR), $S\Gamma \vdash x : S\rho$.

—**case** $\lambda x.e$:

(1) By induction, ($\mathcal{M}$.4) implies

$$S_2 S_1 \Gamma + x{:}\, S_2 S_1 \beta_1 \vdash e : S_2 S_1 \beta_2.$$

(2) By (FN), $S_2 S_1 \Gamma \vdash \lambda x.e : S_2 S_1 \beta_1 \to S_2 S_1 \beta_2$; that is, by ($\mathcal{M}$.3),

$$S_2 S_1 \Gamma \vdash \lambda x.e : S_2 S_1 \rho.$$

—**case** $e_1\ e_2$:

(1) By induction, ($\mathcal{M}$.5) implies $S_1\Gamma \vdash e_1 : S_1(\beta \to \rho)$. By Lemma 1, we can apply S_2 to both sides.

$$S_2 S_1 \Gamma \vdash e_1 : S_2 S_1 \beta \to S_2 S_1 \rho$$

(2) By induction, ($\mathcal{M}$.6) implies

$$S_2 S_1 \Gamma \vdash e_2 : S_2 S_1 \beta.$$

Hence by the (APP) rule, $S_2 S_1 \Gamma \vdash e_1\ e_2 : S_2 S_1 \rho$.

—**case** $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$: Let $S_2' = S_2\{\vec{\beta}/\vec{\alpha}\}$, where $\vec{\alpha} = \mathit{ftv}(S_1\beta) \setminus \mathit{ftv}(S_1\Gamma)$, and let $\vec{\beta}$ be new type variables.

(1) By induction, ($\mathcal{M}$.7) implies $S_1\Gamma \vdash e_1 : S_1\beta$. By Lemma 1 we can apply S_2' to both sides.

$$S_2' S_1 \Gamma \vdash e_1 : S_2' S_1 \beta$$

(2) By induction, ($\mathcal{M}$.8) implies $S_2 S_1 \Gamma + x{:}\, S_2\, \mathit{Clos}_{S_1\Gamma}(S_1\beta) \vdash e_2 : S_2 S_1 \rho$. By Lemma 2 and the fact that $S_2' S_1 \Gamma = S_2 S_1 \Gamma$ because S_2' differs from S_2 only on nonfree variables of $S_1\Gamma$,

$$S_2' S_1 \Gamma + x{:}\, \mathit{Clos}_{S_2' S_1 \Gamma}(S_2' S_1 \beta) \vdash e_2 : S_2 S_1 \rho.$$

Hence by the (LET) rule, $S_2' S_1 \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : S_2 S_1 \rho$; that is,

$$S_2 S_1 \Gamma \vdash \texttt{let}\ x = e_1\ \texttt{in}\ e_2 : S_2 S_1 \rho.$$

—**case** $\texttt{fix}\ f\ \lambda x.e$: Let $S = \mathcal{M}(\Gamma + f{:}\,\rho, \lambda x.e, \rho)$.

(1) By induction, ($\mathcal{M}$.9) implies

$$S\Gamma + f{:}\, S\rho \vdash \lambda x.e : S\rho.$$

(2) By (FIX), $S\Gamma \vdash \texttt{fix}\ f\ \lambda x.e : S\rho$. $\square$

Definition 1 [*Damas and Milner 1982*]. Let σ and σ' be type schemes such that $\sigma = \forall\vec{\alpha}.\tau$ and $\sigma' = \forall\vec{\beta}.\tau'$. If there exists a substitution R such that $\mathit{supp}(R) \subseteq \vec{\alpha}$ and $\tau' = R\tau$, and $\mathit{ftv}(\sigma) \subseteq \mathit{ftv}(\sigma')$, we say that σ' is a generic instance of σ, and we write $\sigma \succ \sigma'$. We also write $\Gamma \succ \Gamma'$ if and only if $\mathit{dom}(\Gamma) = \mathit{dom}(\Gamma')$ and $\Gamma(x) \succ \Gamma'(x)$ for all $x \in \mathit{dom}(\Gamma)$.

THEOREM 2 (COMPLETENESS OF $\mathcal{M}$). *Let e be an expression, and let Γ be a type environment. If there exist a type ρ and a substitution P such that $P\Gamma \vdash e : P\rho$, then $\mathcal{M}(\Gamma, e, \rho) = S$ is defined, and there exists a substitution R such that $P|_{New} = (RS)|_{New}$ where New is the set of new type variables used by $\mathcal{M}(\Gamma, e, \rho)$.*

Completeness means that if an expression e has a type τ that satisfies a type constraint ρ (i.e., $\exists P.\tau = P\rho$), then algorithm $\mathcal{M}$ for the expression with the constraint ρ succeeds with substitution S such that the result type $S\rho$ subsumes τ (i.e., the principality, $\exists R.\tau = R(S\rho)$).

The completeness proof uses Lemmas 3–7.

LEMMA 3. *Let S be a substitution, Γ be a type environment, and τ be a type. Then $S\,Clos_\Gamma(\tau) \succ Clos_{S\Gamma}(S\tau)$.*

PROOF. See Appendix A. □

LEMMA 4 [DAMAS AND MILNER 1982]. *Let Γ and Γ' be type environments such that $\Gamma \succ \Gamma'$. If $\Gamma' \vdash e : \tau$, then $\Gamma \vdash e : \tau$.*

LEMMA 5. *If $S = \mathcal{U}(\tau, \tau')$ then $itv(S) \subseteq ftv(\tau) \cup ftv(\tau')$.*

PROOF. By the definition of the unification algorithm [Robinson 1965]. □

LEMMA 6. *If $S = \mathcal{M}(\Gamma, e, \rho)$ then $itv(S) \subseteq ftv(\Gamma) \cup ftv(\rho) \cup New$, where New is the set of new type variables used by $\mathcal{M}(\Gamma, e, \rho)$.*

PROOF. See Appendix B. □

LEMMA 7. *If $itv(S) \cap A = \emptyset$, then $(RS)\nmid_A = R\nmid_A S$.*

PROOF. See Appendix C. □

PROOF OF THEOREM 2. We prove by the structural induction on e. For a rigorous treatment of new type variables, we assume that every new type variable used throughout algorithm $\mathcal{M}$ is distinct from each other, and moreover, the set *New* of new type variables used by each call $\mathcal{M}(\Gamma, e, \rho)$ satisfies $New \cap (ftv(\Gamma) \cup ftv(\rho)) = \emptyset$.

—**case** $()$: Let the given judgment be $P\Gamma \vdash () : P\rho$. By (CON), $P\rho = \iota$. Because $P\iota = \iota = P\rho$, P is a unifier of ρ and ι. $\mathcal{M}(\Gamma, (), \rho)$ succeeds with the most general unifier S of ρ and ι ($\mathcal{M}$.1). Hence there exists a substitution R such that $P = RS$.

—**case** x: Let the given judgment be $P\Gamma \vdash x : P\rho$, and let $\vec{\beta}$ be the new type variables used at ($\mathcal{M}$.2).

First, we prove that a unifier of ρ and $\{\vec{\beta}/\vec{\alpha}\}\tau$ exists, where $\Gamma(x) = \forall\vec{\alpha}.\tau$. By the (VAR) rule,

$$P\Gamma(x) \succ P\rho. \tag{1}$$

Let $\vec{\gamma}$ be type variables such that $(ftv(\Gamma) \cup ftv(\rho) \cup itv(P) \cup \vec{\beta}) \cap \vec{\gamma} = \emptyset$. Then

$$P\Gamma(x) = P\forall\vec{\alpha}.\tau = \forall\vec{\gamma}.P\{\vec{\gamma}/\vec{\alpha}\}\tau. \tag{2}$$

From (1) and (2), there exists a substitution B such that $supp(B) \subseteq \vec{\gamma}$ and

$$BP\{\vec{\gamma}/\vec{\alpha}\}\tau = P\rho. \tag{3}$$

The right-hand side of (3) gives us

$$\begin{aligned} P\rho &= P\{\vec{\gamma}/\vec{\beta}\}\rho && \text{because } ftv(\rho) \cap \vec{\beta} = \emptyset \\ &= BP\{\vec{\gamma}/\vec{\beta}\}\rho && \text{because } supp(B) \subseteq \vec{\gamma} \text{ and } \vec{\gamma} \cap (itv(P) \cup ftv(\rho)) = \emptyset. \end{aligned}$$

The left-hand side of (3) gives us

$$BP\{\vec{\gamma}/\vec{\alpha}\}\tau = BP\{\vec{\gamma}/\vec{\beta}\}\{\vec{\beta}/\vec{\alpha}\}\tau \quad \text{because } \vec{\beta} \cap \mathit{ftv}(\forall\vec{\alpha}.\tau) = \emptyset.$$

Thus $BP\{\vec{\gamma}/\vec{\beta}\}$ is a unifier of ρ and $\{\vec{\beta}/\vec{\alpha}\}\tau$. Note that $(BP\{\vec{\gamma}/\vec{\beta}\})|_{\overline{\vec{\gamma}}}$ is also a unifier because $(\mathit{ftv}(\rho) \cup \mathit{ftv}(\{\vec{\beta}/\vec{\alpha}\}\tau)) \cap \vec{\gamma} = \emptyset$. That is, $(\mathcal{M}.2)$ succeeds with the most general unifier S of ρ and $\{\vec{\beta}/\vec{\alpha}\}\tau$, and there exists a substitution R such that

$$RS = (BP\{\vec{\gamma}/\vec{\beta}\})|_{\overline{\vec{\gamma}}}. \tag{4}$$

Then

$$\begin{aligned}
(RS)|_{\overline{\vec{\beta}}} &= (BP\{\vec{\gamma}/\vec{\beta}\})|_{\overline{\vec{\gamma}\cup\vec{\beta}}} \\
&= (BP)|_{\overline{\vec{\gamma}\cup\vec{\beta}}} \\
&= (B|_{\overline{\vec{\gamma}}}P)|_{\overline{\vec{\beta}}} && \text{by Lemma 7 and because } \mathit{itv}(P) \cap \vec{\gamma} = \emptyset \\
&= P|_{\overline{\vec{\beta}}} && \text{because } \mathit{supp}(B) \subseteq \vec{\gamma}.
\end{aligned}$$

—**case** $\lambda x.e$: Let the given judgment be $P\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2$ where $\tau_1 \to \tau_2 = P\rho$ and New be $\{\beta_1, \beta_2\} \cup \mathit{New}_1$, where β_1 and β_2 are the new type variables used at $(\mathcal{M}.3)$ and where New_1 is the set of the new type variables used by $\mathcal{M}(S_1\Gamma + x{:}\,S_1\beta_1, e, S_1\beta_2)$ at $(\mathcal{M}.4)$.
First, we prove the unification $\mathcal{U}(\rho, \beta_1 \to \beta_2)$ at $(\mathcal{M}.3)$ succeeds. Let P' be $\{\tau_1/\beta_1, \tau_2/\beta_2\} \cup P|_{\overline{\{\beta_1,\beta_2\}}}$. Then P' is a unifier of ρ and $\beta_1 \to \beta_2$ because

$$\begin{aligned}
P'\rho &= P\rho && \text{because } \{\beta_1, \beta_2\} \cap \mathit{ftv}(\rho) = \emptyset \\
&= \tau_1 \to \tau_2 \\
&= P'(\beta_1 \to \beta_2).
\end{aligned}$$

Thus there exists a substitution R_1 such that

$$R_1 S_1 = P'. \tag{5}$$

By the (FN) rule,

$$P\Gamma + x{:}\,\tau_1 \vdash e : \tau_2. \tag{6}$$

To apply induction to $\mathcal{M}(S_1\Gamma + x{:}\,S_1\beta_1, e, S_1\beta_2)$ at $(\mathcal{M}.4)$ and (6), we must prove that there exists a substitution P_1 such that $\tau_2 = P_1S_1\beta_2$ and $P\Gamma + x{:}\,\tau_1 = P_1(S_1\Gamma + x{:}\,S_1\beta_1)$. Such P_1 is the R_1 at (5) because

$$\begin{aligned}
R_1S_1\beta_2 &= P'\beta_2 && \text{by (5)} \\
&= \tau_2
\end{aligned}$$

and

$$\begin{aligned}
R_1(S_1\Gamma + x{:}\,S_1\beta_1) &= P'(\Gamma + x{:}\,\beta_1) \\
&= P\Gamma + x{:}\,\tau_1 && \text{because } \{\beta_1, \beta_2\} \cap \mathit{ftv}(\Gamma) = \emptyset.
\end{aligned}$$

Thus by induction, $(\mathcal{M}.4)$ and (6) imply that there exists a substitution R_2 such that

$$R_1|_{\overline{\mathit{New}_1}} = (R_2S_2)|_{\overline{\mathit{New}_1}}. \tag{7}$$

Let $R = R_2$. Then

$$\begin{aligned}(RS)|_{New} &= (R_2S_2S_1)|_{New} \\ &= ((R_2S_2S_1)|_{New_1})|_{\{\beta_1,\beta_2\}}. \end{aligned} \tag{8}$$

Note that $itv(S_1) \cap New_1 = \emptyset$ because $itv(S_1) \subseteq ftv(\rho) \cup \{\beta_1, \beta_2\}$ by Lemma 5, $ftv(\rho) \cap New_1 \subseteq ftv(\rho) \cap New = \emptyset$, and $\{\beta_1, \beta_2\} \cap New_1 = \emptyset$ by the assumption that all new type variables are distinct from each other. Therefore, by Lemma 7, Eq. (8) becomes

$$\begin{aligned}((R_2S_2S_1)|_{New_1})|_{\{\beta_1,\beta_2\}} &= ((R_2S_2)|_{New_1}S_1)|_{\{\beta_1,\beta_2\}} \\ &= (R_1|_{New_1}S_1)|_{\{\beta_1,\beta_2\}} && \text{by (7)} \\ &= ((R_1S_1)|_{New_1})|_{\{\beta_1,\beta_2\}} && \text{by Lemma 7} \\ &= (R_1S_1)|_{New} \\ &= P'|_{New} && \text{by (5)} \\ &= P|_{New} && \text{because } \{\beta_1,\beta_2\} \subseteq New.\end{aligned}$$

—**case** $e_1\ e_2$: Let the given judgment be $P\Gamma \vdash e_1\ e_2 : P\rho$ and $New = \{\beta\} \cup New_1 \cup New_2$, where β is the new type variable used at $(\mathcal{M}.5)$ and New_1 and New_2 are the sets of the new type variables used by $\mathcal{M}(\Gamma, e_1, \beta \to \rho)$ at $(\mathcal{M}.5)$ and $\mathcal{M}(S_1\Gamma, e_2, S_1\beta)$ at $(\mathcal{M}.6)$, respectively. By the (APP) rule, there exists a type τ such that

$$P\Gamma \vdash e_1 : \tau \to P\rho \tag{9}$$

and

$$P\Gamma \vdash e_2 : \tau. \tag{10}$$

Let $P' = \{\tau/\beta\} \cup P|_{\{\beta\}}$. Then $\tau \to P\rho = P'(\beta \to \rho)$ and $P\Gamma = P'\Gamma$ because $\beta \notin ftv(\Gamma) \cup ftv(\rho)$. Hence, applying induction to $\mathcal{M}(\Gamma, e_1, \beta \to \rho)$ at $(\mathcal{M}.5)$ and (9), there exists a substitution R_1 such that

$$P'|_{New_1} = (R_1S_1)|_{New_1}. \tag{11}$$

Similarly, we can apply induction to $\mathcal{M}(S_1\Gamma, e_2, S_1\beta)$ at $(\mathcal{M}.6)$ and (10) because

$$\tau = P'\beta = R_1S_1\beta \quad \text{because } \beta \notin New_1 \text{ and by (11)}$$

and

$$\begin{aligned}P\Gamma &= P'\Gamma && \text{because } \beta \notin ftv(\Gamma) \\ &= R_1S_1\Gamma && \text{because } ftv(\Gamma) \cap New_1 = \emptyset \text{ and by (11).}\end{aligned}$$

Thus by induction, there exists a substitution R_2 such that

$$R_1|_{New_2} = (R_2S_2)|_{New_2}. \tag{12}$$

Let $R = R_2$. Then

$$\begin{aligned}(RS)|_{New} &= (R_2S_2S_1)|_{New} \\ &= ((R_2S_2S_1)|_{New_2})|_{New_1 \cup \{\beta\}}. \end{aligned} \tag{13}$$

Note that $itv(S_1) \cap New_2 = \emptyset$ because $itv(S_1) \subseteq ftv(\Gamma) \cup ftv(\rho) \cup New_1 \cup \{\beta\}$ by Lemma 6, $(ftv(\Gamma) \cup ftv(\rho)) \cap New_2 \subseteq (ftv(\Gamma) \cup ftv(\rho)) \cap New = \emptyset$, and

$(New_1 \cup \{\beta\}) \cap New_2 = \emptyset$ by the assumption that all new type variables are distinct from each other. Therefore, by Lemma 7, Eq. (13) becomes

$$\begin{array}{rcll} ((R_2 S_2 S_1)|_{New_2})|_{New_1 \cup \{\beta\}} & = & ((R_2 S_2)|_{New_2} S_1)|_{New_1 \cup \{\beta\}} & \\ & = & (R_1|_{New_2} S_1)|_{New_1 \cup \{\beta\}} & \text{by (12)} \\ & = & (R_1 S_1)|_{New} & \text{by Lemma 7} \\ & = & ((R_1 S_1)|_{New_1})|_{New_2 \cup \{\beta\}} & \\ & = & P'|_{New} & \text{by (11)} \\ & = & P|_{New} & \text{because } \beta \in New. \end{array}$$

—**case** `let` $x = e_1$ `in` e_2: Let the given judgment be $P\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 : P\rho$ and $New = \{\beta\} \cup New_1 \cup New_2$, where β is the new type variable used at $(\mathcal{M}.7)$ and where New_1 and New_2 are the sets of the new type variables used by $\mathcal{M}(\Gamma, e_1, \beta)$ at $(\mathcal{M}.7)$ and $\mathcal{M}(S_1\Gamma + x{:}\, Clos_{S_1\Gamma}(S_1\beta), e_2, S_1\rho)$ at $(\mathcal{M}.8)$, respectively. By the (LET) rule, there exists a type τ such that

$$P\Gamma \vdash e_1 : \tau \tag{14}$$

and

$$P\Gamma + x{:}\, Clos_{P\Gamma}(\tau) \vdash e_2 : P\rho. \tag{15}$$

Let $P' = \{\tau/\beta\} \cup P|_{\{\beta\}}$. Then $\tau = P'\beta$ and $P\Gamma = P'\Gamma$ because $\beta \notin ftv(\Gamma)$. Hence by induction, $(\mathcal{M}.7)$ and (14) imply that there exists a substitution R_1 such that

$$P'|_{New_1} = (R_1 S_1)|_{New_1}. \tag{16}$$

Note

$$\begin{array}{rcll} Clos_{P\Gamma}(\tau) & = & Clos_{P'\Gamma}(P'\beta) & \\ & = & Clos_{R_1 S_1 \Gamma}(R_1 S_1 \beta) & \text{by (16) and } (ftv(\Gamma) \cup \{\beta\}) \cap New_1 = \emptyset \\ & \prec & R_1\, Clos_{S_1\Gamma}(S_1\beta) & \text{by Lemma 3} \end{array}$$

and $P\Gamma = P'\Gamma = R_1 S_1 \Gamma$ because $New_1 \cap ftv(\Gamma) = \emptyset$. Thus

$$P\Gamma + x{:}\, Clos_{P\Gamma}(\tau) \prec R_1(S_1\Gamma + x{:}\, Clos_{S_1\Gamma}(S_1\beta)).$$

By Lemma 4 and (15),

$$R_1(S_1\Gamma + x{:}\, Clos_{S_1\Gamma}(S_1\beta)) \vdash e_2 : P\rho.$$

Because $R_1 S_1 \rho = P'\rho = P\rho$,

$$R_1(S_1\Gamma + x{:}\, Clos_{S_1\Gamma}(S_1\beta)) \vdash e_2 : R_1 S_1 \rho. \tag{17}$$

Thus by induction, $(\mathcal{M}.8)$ and (17) imply that there exists a substitution R_2 such that

$$R_1|_{New_2} = (R_2 S_2)|_{New_2}. \tag{18}$$

Let $R = R_2$. Then, using (16) and (18) and following exactly the same steps in the proof for the application expression, we have

$$(RS)|_{New} = P|_{New}.$$

—**case** $\texttt{fix}\ f\ \lambda x.e$: Let the given judgment be $P\Gamma \vdash \texttt{fix}\ f\ \lambda x.e : P\rho$. By the (FIX) rule,

$$P\Gamma + f\colon P\rho \vdash \lambda x.e : P\rho.$$

By induction, $\mathcal{M}(\Gamma + f\colon \rho, \lambda x.e, \rho)$ succeeds with a substitution S, and there exists a substitution R such that $P\restriction_{New} = (RS)\restriction_{New}$, where *New* is the set of new type variables used by $\mathcal{M}(\Gamma + f\colon \rho, \lambda x.e, \rho)$ at ($\mathcal{M}$.9). By ($\mathcal{M}$.9), $\mathcal{M}(\Gamma, \texttt{fix}\ f\ \lambda x.e, \rho)$ also succeeds with the S and $P\restriction_{New} = (RS)\restriction_{New}$. $\square$

4. $\mathcal{M}$ STOPS EARLIER THAN $\mathcal{W}$ DOES

We model the behaviors of the two type inference algorithms by their *call strings*. The call string of $\mathcal{W}(\Gamma, e)$ (written $[\![\mathcal{W}(\Gamma, e)]\!]$) is constructed by starting with the empty call string ϵ and appending a tuple $(\Gamma_1, e_1)^d$ (respectively, $(\Gamma_1, e_1)^u$) whenever $\mathcal{W}(\Gamma_1, e_1)$ is called (respectively, returned). The d or u superscript indicates the *d*ownward or *u*pward movement of the stack pointer when the inference algorithm is recursively called or returned. When the algorithm stops because of a unification failure the call string does not have matching returns (u tuples) for some calls (d tuples). We similarly define call strings for algorithm $\mathcal{M}$.

For example, given an expression

$$\overbrace{\texttt{(}\underbrace{\texttt{(fn x=>x)}}_{e_1}\texttt{ 2)}}^{e},$$

the call string $[\![\mathcal{W}(\Gamma, e)]\!]$ is

$$(\Gamma, e)^d (\Gamma, e_1)^d (\Gamma + \texttt{x} : \beta, \texttt{x})^d (\Gamma + \texttt{x} : \beta, \texttt{x})^u (\Gamma, e_1)^u (\Gamma, \texttt{2})^d (\Gamma, \texttt{2})^u (\Gamma, e)^u.$$

For the ill-typed expression `(1 2)`, the call string is

$$(\Gamma, \texttt{1 2})^d (\Gamma, \texttt{1})^d (\Gamma, \texttt{1})^u (\Gamma, \texttt{2})^d (\Gamma, \texttt{2})^u.$$

Note that tuple $(\Gamma, \texttt{1 2})^u$ is missing because the algorithm stops because of the unification failure at the application.

Note that call strings $[\![\mathcal{W}(\Gamma, e)]\!]$ and $[\![\mathcal{M}(\Gamma, e, \rho)]\!]$ are always finite because, for any expression e, type environment Γ, and a type ρ, at most one call to $\mathcal{W}$ (respectively, $\mathcal{M}$) occurs for each subexpression of e during $\mathcal{W}(\Gamma, e)$ (respectively, $\mathcal{M}(\Gamma, e, \rho)$).

We say that "$\mathcal{W}$ (respectively, $\mathcal{M}$) fails at an expression e" whenever the current argument expression to $\mathcal{W}$ (respectively, $\mathcal{M}$) is e when the unification fails:

Definition 2. Let Γ be a type environment, e be an expression that has a type error, and β be a new type variable. "$\mathcal{W}(\Gamma, e)$ fails at e'" whenever the rightmost unmatching tuple in its call string $[\![\mathcal{W}(\Gamma, e)]\!]$ is $(\Gamma', e')^d$. Similarly, "$\mathcal{M}(\Gamma, e, \beta)$ fails at e'" whenever the rightmost unmatching tuple in its call string $[\![\mathcal{M}(\Gamma, e, \beta)]\!]$ is $(\Gamma', e', \rho')^d$.

Example 1. Consider an expression `((λx.x^"a") true)`. $\mathcal{W}$ fails at the top expression `((λx.x^"a") true)` after it succeeded at every proper subexpression. Meanwhile $\mathcal{M}$ fails at `true`.

Example 2. Consider an expression `(1 (2 3))`. $\mathcal{W}$ fails at `(2 3)`. $\mathcal{M}$ fails at `1`, which is earlier than $\mathcal{W}$ does because the left expression `1` is checked before the right expression `(2 3)`.

Example 3. Consider an expression `((`λ`x.x+1)` λ`y.(1 1))`. $\mathcal{W}$ fails at `(1 1)`. $\mathcal{M}$ fails at `(`λ`y.(1 1))`, which is before it checks the body expression `(1 1)`.

As the above examples indicate, algorithm $\mathcal{M}$ always stops earlier than Algorithm $\mathcal{W}$:

THEOREM 3 (EARLINESS). *Let Γ be a type environment, e be an expression, and β be a new type variable. Then*

$$|[\![\mathcal{M}(\Gamma, e, \beta)]\!]| \leq |[\![\mathcal{W}(\Gamma, e)]\!]|,$$

where $|s|$ is the number of tuples in call string s.

The proof of Theorem 3 uses the completeness of Algorithm $\mathcal{W}$ and Lemmas 8–11.

THEOREM 4 (COMPLETENESS OF $\mathcal{W}$) [DAMAS AND MILNER 1982]. *Given Γ and e, let Γ' be an instance of Γ and σ a type scheme such that $\Gamma' \vdash e : \sigma$. Then $\mathcal{W}(\Gamma, e)$ succeeds, and if $\mathcal{W}(\Gamma, e) = (S, \tau)$ then, for some substitution R, $\Gamma' = RS\Gamma$ and $R\mathit{Clos}_{S\Gamma}(\tau) \succ \sigma$.*

LEMMA 8. *Let Γ and Γ' be type environments and τ be a type. If $\Gamma \succ \Gamma'$, then $\mathit{Clos}_{\Gamma}(\tau) \succ \mathit{Clos}_{\Gamma'}(\tau)$.*

PROOF. See Appendix D. □

LEMMA 9 [DAMAS AND MILNER 1982]. *If $\sigma \succ \sigma'$ then $S\sigma \succ S\sigma'$.*

LEMMA 10. *Let e be an expression, Γ be a type environment, and β be a new type variable. If $[\![\mathcal{W}(\Gamma, e)]\!]$ has $(\Gamma^{\mathcal{W}}, e')^d$ and $[\![\mathcal{M}(\Gamma, e, \beta)]\!]$ has $(\Gamma^{\mathcal{M}}, e', \rho)^d$, then there exists a substitution R such that $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$. (Note that because $\mathcal{W}$ and $\mathcal{M}$ are called only once for each subexpression of e, such $\Gamma^{\mathcal{W}}$ and $\Gamma^{\mathcal{M}}$ are well defined.)*

PROOF. We prove by induction on the length of the prefixes $(\Gamma, e)^d \cdots (\Gamma^{\mathcal{W}}, e')^d$ of $[\![\mathcal{W}(\Gamma, e)]\!]$ and $(\Gamma, e, \beta)^d \cdots (\Gamma^{\mathcal{M}}, e', \rho)^d$ of $[\![\mathcal{M}(\Gamma, e, \beta)]\!]$. Note that the prefixes have the same length because $\mathcal{W}$ and $\mathcal{M}$ check the sub-expressions of e in the same order. The symbols of both algorithms are identified by their superscripts.

—**base case:** When the prefixes are of length 1, they represent the initial calls where e' is e and where $\Gamma^{\mathcal{W}}$ and $\Gamma^{\mathcal{M}}$ are identical. Then the identity substitution R satisfies $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$.

Followings are inductive cases.

—**case** e' **of** e_1 **in "**$\lambda x.e_1$**"**: Let the type environment parameters be $\Gamma^{\mathcal{W}}$ and $\Gamma^{\mathcal{M}}$ when $\lambda x.e$ is visited. By induction, $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$; that is, by Lemma 9, $S_1^{\mathcal{M}} R\Gamma^{\mathcal{W}} \succ S_1^{\mathcal{M}}\Gamma^{\mathcal{M}}$. Let $R_1 = S_1^{\mathcal{M}}(R|_{\{\beta^{\mathcal{W}}\}} \cup \{\beta_1^{\mathcal{M}}/\beta^{\mathcal{W}}\})$. Then

$$\begin{array}{rcll} R_1(\Gamma^{\mathcal{W}} + x{:}\,\beta^{\mathcal{W}}) & = & S_1^{\mathcal{M}} R\Gamma^{\mathcal{W}} + x{:}\,S_1^{\mathcal{M}}\beta_1^{\mathcal{M}} & \text{because } \beta^{\mathcal{W}} \notin \mathit{ftv}(\Gamma^{\mathcal{W}}) \\ & \succ & S_1^{\mathcal{M}}\Gamma^{\mathcal{M}} + x{:}\,S_1^{\mathcal{M}}\beta_1^{\mathcal{M}}. & \end{array}$$

—**case** e' **of** e_1 **in "**$e_1\ e_2$**"**: The call at e_1 occurs with the same environment as the one that accompanied the call at $e_1\ e_2$. Thus the case holds by the induction hypothesis.

—**case** e' **of** e_2 **in "**$e_1\ e_2$**"**:
(1) By the soundness of $\mathcal{M}$, ($\mathcal{M}$.5) implies

$$S_1^{\mathcal{M}}\Gamma^{\mathcal{M}} \vdash e_1 : S_1^{\mathcal{M}}(\beta^{\mathcal{M}} \to \rho^{\mathcal{M}}).$$

(2) By induction, $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$; that is, by Lemma 9, $S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}} \succ S_1^{\mathcal{M}}\Gamma^{\mathcal{M}}$. By Lemma 4,

$$S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}} \vdash e_1 : S_1^{\mathcal{M}}(\beta^{\mathcal{M}} \to \rho^{\mathcal{M}}).$$

(3) By the completeness of $\mathcal{W}$, there exists a substitution R' such that $R'S_1^{\mathcal{W}}\Gamma^{\mathcal{W}} = S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}}$. So, $R'S_1^{\mathcal{W}}\Gamma^{\mathcal{W}} = S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}} \succ S_1^{\mathcal{M}}\Gamma^{\mathcal{M}}$.

—**case** e' **of** e_1 **in "**`let` $x = e_1$ `in` e_2**"**: The call at e_1 occurs with the same environment as the one that accompanied the call at `let` $x = e_1$ `in` e_2. Thus the case holds by the induction hypothesis.

—**case** e' **of** e_2 **in "**`let` $x = e_1$ `in` e_2**"**:
(1) By the soundness of $\mathcal{M}$, ($\mathcal{M}$.7) implies

$$S_1^{\mathcal{M}}\Gamma^{\mathcal{M}} \vdash e_1 : S_1^{\mathcal{M}}\beta^{\mathcal{M}}.$$

(2) By induction, $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$; that is, by Lemma 9, $S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}} \succ S_1^{\mathcal{M}}\Gamma^{\mathcal{M}}$. By Lemma 4,

$$S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}} \vdash e_1 : S_1^{\mathcal{M}}\beta^{\mathcal{M}}.$$

(3) By the completeness of $\mathcal{W}$, there exists a substitution R' such that $R'\tau_1^{\mathcal{W}} = S_1^{\mathcal{M}}\beta^{\mathcal{M}}$ and $R'S_1^{\mathcal{W}}\Gamma^{\mathcal{W}} = S_1^{\mathcal{M}}R\Gamma^{\mathcal{W}}$. Therefore,

$$R'S_1^{\mathcal{W}}\Gamma^{\mathcal{W}} \succ S_1^{\mathcal{M}}\Gamma^{\mathcal{M}}$$

and

$$\begin{array}{rcll} R'\,Clos_{S_1^{\mathcal{W}}\Gamma^{\mathcal{W}}}(\tau_1^{\mathcal{W}}) & \succ & Clos_{R'S_1^{\mathcal{W}}\Gamma^{\mathcal{W}}}(R'\tau_1^{\mathcal{W}}) & \text{by Lemma 3} \\ & = & Clos_{R'S_1^{\mathcal{W}}\Gamma^{\mathcal{W}}}(S_1^{\mathcal{M}}\beta^{\mathcal{M}}) & \\ & \succ & Clos_{S_1^{\mathcal{M}}\Gamma^{\mathcal{M}}}(S_1^{\mathcal{M}}\beta^{\mathcal{M}}) & \text{by Lemma 8.} \end{array}$$

The above two facts imply that this case is proven.

—**case** e' **of** $\lambda x.e$ **in "**`fix` $f\ \lambda x.e$**"**: By induction, there exists a substitution R such that $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$. Let $R' = R\!\downarrow_{\{\beta^{\mathcal{W}}\}} \cup \{\rho^{\mathcal{M}}/\beta^{\mathcal{W}}\}$. Then

$$\begin{array}{rcll} R'(\Gamma^{\mathcal{W}} + f{:}\,\beta^{\mathcal{W}}) & = & R\Gamma^{\mathcal{W}} + f{:}\,\rho^{\mathcal{M}} & \text{because } \beta^{\mathcal{W}} \notin ftv(\Gamma^{\mathcal{W}}) \\ & \succ & \Gamma^{\mathcal{M}} + f{:}\,\rho^{\mathcal{M}}. \ \square & \end{array}$$

LEMMA 11. *Let e be an expression, Γ be a type environment, and β be a new type variable. Suppose $[\![\mathcal{W}(\Gamma, e)]\!]$ has $(\Gamma^{\mathcal{W}}, e')^d$ and $[\![\mathcal{M}(\Gamma, e, \beta)]\!]$ has $(\Gamma^{\mathcal{M}}, e', \rho)^d$. If $\mathcal{W}(\Gamma^{\mathcal{W}}, e')$ fails, then $\mathcal{M}(\Gamma^{\mathcal{M}}, e', \rho)$ fails.*

PROOF. Assume for contradiction that $\mathcal{M}(\Gamma^{\mathcal{M}}, e', \rho)$ succeeds; that is, $\mathcal{M}(\Gamma^{\mathcal{M}}, e', \rho) = S$ is defined. By the soundness of $\mathcal{M}$,

$$S\Gamma^{\mathcal{M}} \vdash e' : S\rho.$$

By Lemma 10, there exists a substitution R such that $R\Gamma^{\mathcal{W}} \succ \Gamma^{\mathcal{M}}$; that is, by Lemma 9, $SR\Gamma^{\mathcal{W}} \succ S\Gamma^{\mathcal{M}}$. By Lemma 4,

$$SR\Gamma^{\mathcal{W}} \vdash e' : S\rho.$$

And, finally, by the completeness of $\mathcal{W}$, $\mathcal{W}(\Gamma^{\mathcal{W}}, e')$ succeeds. It contrasts with the given condition, so the assumption is not true. $\square$

Now we prove Theorem 3.

Proof of Theorem 3. The case that e has no type error is trivially true, because it is obvious that $|[\![\mathcal{W}(\Gamma, e)]\!]| = |[\![\mathcal{M}(\Gamma, e, \beta)]\!]|$.

Let us consider the case that e has a type error. Let $\mathcal{W}(\Gamma, e)$ fail at an application expression $(e_1\ e_2)$. Then

$$[\![\mathcal{W}(\Gamma, e)]\!] = \cdots (\Gamma_1^{\mathcal{W}}, e_1\ e_2)^d \cdots (\Gamma_2^{\mathcal{W}}, e_2)^u$$

because $\mathcal{W}$ fails right after it returned from an application's operand. Suppose for contradiction that

$$|[\![\mathcal{W}(\Gamma, e)]\!]| < |[\![\mathcal{M}(\Gamma, e, \beta)]\!]|.$$

That is,

$$[\![\mathcal{M}(\Gamma, e, \beta)]\!] = \cdots (\Gamma_1^{\mathcal{M}}, e_1\ e_2, \rho_1)^d \cdots (\Gamma_2^{\mathcal{M}}, e_2, \rho_2)^u \cdot \kappa$$

and

$$\kappa \neq \epsilon$$

because the order of visiting subexpressions is the same for $\mathcal{W}$ and $\mathcal{M}$. The next call/return after the return from e_2 is the return from $e_1\ e_2$. So κ is $(\Gamma_1^{\mathcal{M}}, e_1\ e_2, \rho_1)^u \cdot \kappa'$. It means that $\mathcal{M}(\Gamma_1^{\mathcal{M}}, e_1\ e_2, \rho_1)$ succeeds. However, it is impossible by Lemma 11 because $\mathcal{W}(\Gamma_1^{\mathcal{W}}, e_1\ e_2)$ fails. $\square$

5. CONCLUSION

The Hindley/Milner let-polymorphic type inference system has two different algorithms: one is the *de facto* standard $\mathcal{W}$ algorithm [Milner 1978; Damas and Milner 1982] that is bottom-up (or context-insensitive) and the other is a "folklore" algorithm that is top-down (or context-sensitive).

In this article, we formally defined the folklore algorithm (named $\mathcal{M}$), proved its soundness and completeness, and showed that $\mathcal{M}$ always finds type errors earlier (considers a less number of expressions) than $\mathcal{W}$.

Our proofs can be seen as theoretical justifications for various type-checking strategies. For example, a compiler can let the user switch between the two algorithms, which may help in situations where it is hard to find the cause of the type error. The two algorithms will always stop at different expressions reporting different causes of the type problem. Already, a combination of a variant of $\mathcal{M}$ with $\mathcal{W}$ is being implemented in practice [Rideau and Théry 1997]. A compiler can also mix the two algorithms on-the-fly, choosing one algorithm against the other, depending on the current subexpression to type check. For example, when type-checking the definitions of recursive functions the compiler may switch to algorithm $\mathcal{M}$, because it generates a better type diagnostic than $\mathcal{W}$ if the recursive calls have ill-typed arguments (as seen in Figure 1). A variant of this technique is implemented in the SML/NJ compiler system version 110. Similar bidirectional type-checking ideas have been formalized in the setting of subtyping and impredicative polymorphism [Pierce and Turner 1998].

Our formalization enables us to see clearly that algorithm $\mathcal{M}$ can unobtrusively adapt the existing techniques of generating informative type error messages, which were developed with Algorithm $\mathcal{W}$ in mind. The techniques of Wand [1986], Beaven and Stansifer [1993], Duggan and Bent [1996], and Rideau and Théry [1997] essentially record the history of the type instantiations (unifications) in the resulting substitutions. Because the type instantiation in $\mathcal{M}$ is via the same standard unification algorithm, these approaches can be directly used in $\mathcal{M}$. For the same reason, inferring the types of free variables [Bernstein and Stark 1995], by which the programmer can probe the types of some puzzling points in his/her program, is also straightforward to implement inside $\mathcal{M}$, by making such free variables all distinct and initially bound to fresh type variables. The idea in the destructive implementation of Algorithm $\mathcal{W}$ [Cardelli 1987], in which type variables are destructively updated during unification and free variables are remembered for quickly computing the type closure, can also be used in a practical implementation of algorithm $\mathcal{M}$.

APPENDIX

A. PROOF OF LEMMA 3

Let $\vec{\alpha} = \mathit{ftv}(\tau)\backslash \mathit{ftv}(\Gamma)$, and let $\vec{\beta}$ be type variables such that $\vec{\beta} \cap (\mathit{itv}(S) \cup \mathit{ftv}(\tau)) = \emptyset$ and $|\vec{\alpha}| = |\vec{\beta}|$. Then

$$S\,\mathit{Clos}_\Gamma(\tau) = S\forall\vec{\alpha}.\tau = \forall\vec{\beta}.S\{\vec{\beta}/\vec{\alpha}\}\tau.$$

Let $R = \{S\vec{\alpha}/\vec{\beta}\}$. Then $S\tau = RS\{\vec{\beta}/\vec{\alpha}\}\tau$ and $\mathit{supp}(R) \subseteq \vec{\beta}$; thus, the first condition of generic instance is satisfied. The second condition holds as follows

$$\begin{aligned}
\mathit{ftv}(S\,\mathit{Clos}_\Gamma(\tau)) &= \bigcup_{\gamma\in \mathit{ftv}(\tau)\cap \mathit{ftv}(\Gamma)} \mathit{ftv}(S\gamma)\\
&\subseteq \left(\bigcup_{\gamma\in \mathit{ftv}(\tau)} \mathit{ftv}(S\gamma)\right) \cap \left(\bigcup_{\gamma\in \mathit{ftv}(\Gamma)} \mathit{ftv}(S\gamma)\right)\\
&= \mathit{ftv}(S\tau) \cap \mathit{ftv}(S\Gamma)\\
&= \mathit{ftv}(\mathit{Clos}_{S\Gamma}(S\tau)). \quad \square
\end{aligned}$$

B. PROOF OF LEMMA 6

The proof uses Lemma 12.

LEMMA 12 [MILNER 1978]. *Let R and S be substitutions and τ be a type. Then*

—$\mathit{itv}(RS) \subseteq \mathit{itv}(R) \cup \mathit{itv}(S)$ *and*

—$\mathit{ftv}(S\tau) \subseteq \mathit{ftv}(\tau) \cup \mathit{itv}(S)$.

The proof is by the structural induction on e.

—**case** (): By Lemma 5, $\mathit{itv}(\mathcal{U}(\rho,\iota)) \subseteq \mathit{ftv}(\rho) \cup \mathit{ftv}(\iota) \subseteq \mathit{ftv}(\rho)$.

—**case** x:

$$\begin{aligned} itv(\mathcal{U}(\rho,[\beta_i/\alpha_i]\tau)) &\subseteq ftv(\rho)\cup ftv(\{\vec{\beta}/\vec{\alpha}\}\tau) && \text{by Lemma 5}\\ &\subseteq ftv(\rho)\cup (ftv(\tau)\setminus\vec{\alpha})\cup\vec{\beta}\\ &= ftv(\rho)\cup ftv(\forall\vec{\alpha}.\tau)\cup\vec{\beta}\\ &= ftv(\rho)\cup ftv(\Gamma(x))\cup\vec{\beta}\\ &\subseteq ftv(\rho)\cup ftv(\Gamma)\cup New. \end{aligned}$$

—**case** $\lambda x.e$: By Lemma 5, $itv(S_1)\subseteq ftv(\rho)\cup ftv(\beta_1\to\beta_2)$. By induction,

$$\begin{aligned} itv(S_2) &\subseteq ftv(S_1\Gamma + x{:}\,S_1\beta_1)\cup ftv(S_1\beta_2)\cup New_1\\ &\subseteq itv(S_1)\cup ftv(\Gamma)\cup New_1\cup\{\beta_1,\beta_2\} && \text{by Lemma 12.} \end{aligned}$$

Hence, $itv(S_2S_1)\subseteq itv(S_1)\cup itv(S_2)\subseteq ftv(\Gamma)\cup ftv(\rho)\cup New_1\cup\{\beta_1,\beta_2\}$.

Other cases can be similarly proven. □

C. PROOF OF LEMMA 7

We prove that $(RS)\!\restriction_A\alpha = R\!\restriction_A S\alpha$ for all α.

—**case** $\alpha\in A$: $(RS)\!\restriction_A\alpha=\alpha$, and because $itv(S)\cap A=\emptyset$, $R\!\restriction_A S\alpha = R\!\restriction_A\alpha=\alpha$.

—**case** $\alpha\notin A$: $(RS)\!\restriction_A\alpha = RS\alpha$. When $\alpha\in supp(S)$, $R\!\restriction_A S\alpha = RS\alpha$ because $itv(S)\cap A=\emptyset$. When $\alpha\notin supp(S)$, $R\!\restriction_A S\alpha = R\alpha = RS\alpha$. □

D. PROOF OF LEMMA 8

By the definition, $\Gamma(x)\succ\Gamma'(x)$ for $\forall x\in dom(\Gamma)$. This implies $ftv(\Gamma(x))\subseteq ftv(\Gamma'(x))$ for $\forall x\in dom(\Gamma)$, that is, $ftv(\Gamma)\subseteq ftv(\Gamma')$. Therefore,

$$ftv(Clos_\Gamma(\tau)) = ftv(\tau)\cap ftv(\Gamma)\subseteq ftv(\tau)\cap ftv(\Gamma') = ftv(Clos_{\Gamma'}(\tau)). \quad \square$$

ACKNOWLEDGMENTS

We thank anonymous referees and the associate editor for their valuable suggestions and corrections that substantially improved our presentation. We thank David MacQueen for his encouragement on an early draft of this article, and Laurent Théry, Xavier Leroy, and Pierre Weis for sharing their experiences of the two type inference algorithms. We thank Hyunjun Eo, Hyunsok Oh, and Sukyoung Ryu for their comments during this work.

REFERENCES

Beaven, M. and Stansifer, R. 1993. Explaining type errors in polymorphic languages. *ACM Lett. Program. Lang. Syst. 2*, 17–30.

Bernstein, K. L. and Stark, E. W. 1995. Debugging type errors (full version). Tech. Rep., State University of New York at Stony Brook.

Cardelli, L. 1987. Basic polymorphic typechecking. *Sci. Comput. Program. 8,* 2 (Apr.).

Damas, L. and Milner, R. 1982. Principal type-scheme for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 207–212.

Duggan, D. and Bent, F. 1996. Explaining type inference. *Sci. Comput. Program. 27,* 1 (July), 37–83.

Johnson, G. F. and Walz, J. A. 1986. A maximal-flow approach to anomaly isolation in unification-based incremental type inference. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages.* ACM, New York, 44–57.

LEROY, X. 1993. The caml light system, release 0.6. Institut National de Recherche en Informatique et en Automatique.

LEROY, X. 1995. The caml light system, release 0.7. Institut National de Recherche en Informatique et en Automatique.

MACQUEEN, D. B. AND APPEL, A. W. 1993. Standard ML of New Jersey. Tech. Memo., AT&T Bell Labs.

MILNER, R. 1978. A theory of type polymorphism in programming. *J. Comput. Syst. Sci. 17*, 348–375.

PIERCE, B. C. AND TURNER, D. N. 1998. Local type inference. In *Proceedings of the 25th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York.

RIDEAU, L. AND THÉRY, L. 1997. Interactive programming environment for ML. Tech. Rep. 3139, Institut National de Recherche en Informatique et en Automatique. March.

ROBINSON, J. A. 1965. A machine-oriented logic based on the resolution principle. *J. ACM 12,* 1 (Jan.), 23–41.

WAND, M. 1986. Finding the source of type errors. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*. ACM, New York, 38–43.

Received July 1997; revised March 1998; accepted July 1998