A Generalization of Hybrid Let-Polymorphic Type Inference Algorithms∗

Oukseh Lee and Kwangkeun Yi
{cookcu;kwang}@ropas.kaist.ac.kr

Research On Program Analysis System (ROPAS)†
Korea Advanced Institute of Science and Technology (KAIST)

Abstract
We present a generalized let-polymorphic type inference algorithm, prove that any of its instances is sound and complete with respect to the Hindley/Milner let-polymorphic type system, and find a condition on two instance algorithms so that one algorithm should find type errors earlier than the other.

By instantiating the generalized algorithm with different parameters, we can achieve not only the two opposite algorithms (the bottom-up standard algorithm $\mathcal{W}$ and the top-down algorithm $\mathcal{M}$) but also other hybrid algorithms which are used in real compilers. Such instances’ soundness and completeness follow automatically, and their relative earliness in detecting type-errors are determined by checking a simple condition. The set of instances of the generalized algorithm is a superset of those used in the two most popular ML compilers: SML/NJ and OCaml.

1 Introduction

1.1 This Work
When we implement the Hindley/Milner let-polymorphic type system [14] in realistic compilers, its two opposite algorithms ($\mathcal{W}$ [5, 14] and $\mathcal{M}$ [9]) are hardly appealing. In order to generate helpful type-error messages we need to balance between their two opposite behaviors in type-checking: the bottom-up algorithm $\mathcal{W}$ is context-insensitive, finding type errors too late, while the top-down algorithm $\mathcal{M}$ is as much context-sensitive as possible, finding type errors too early. Because of these behaviors, the Standard ML of New Jersey (SML/NJ) [18] and Objective Caml (OCaml) [11] compilers use some combinations of the two algorithms.

As suggested by existing works [2, 3, 6, 7, 13, 17, 22], there exists some room for various type-checking strategies. In order to systematically explore other type-checking algorithms, as well as to justify the existing hybrid ones, we need a framework (1) for integrating the two opposite algorithms into one algorithm; (2) for assuring that such an integrated algorithm is still sound and complete; and (3) for measuring, if possible, how any two hybrid algorithms differ in type-checking.

Within the format of recursive function with unification, we present a generalized let-polymorphic type inference algorithm, prove that *any* of its instances is sound and complete with respect to the Hindley/Milner let-polymorphic type system, and find a condition on two instance algorithms so that one algorithm should find type errors earlier than the other. By instantiating the generalized algorithm with different parameters, we can achieve not only the two opposite algorithms ($\mathcal{W}$ and $\mathcal{M}$) but also other various hybrid algorithms that avoid their extremities in type-checking. The set of hybrid algorithms that come from the generalized algorithm is a superset of the existing hybrid algorithms in SML/NJ and OCaml. Within this algorithmic framework, compiler

∗This work is supported by Creative Research Initiatives of the Korean Ministry of Science and Technology.
†Web: http://ropas.kaist.ac.kr.
developers can freely experiment with various combinations without the burden of proving their correctness every time.

### 1.2 Notation

We use the same conventional notation as used in [9]. Vector $\alpha$ is a shorthand for $\{\alpha_1, \cdots, \alpha_n\}$, and $\forall \alpha.\tau$ is for $\forall \alpha_1 \cdots \alpha_n.\tau$. Equality of type schemes is up to renaming of bound variables.

For a type scheme $\sigma = \forall \alpha.\tau$, the set $ftv(\sigma)$ of free type variables in $\sigma$ is $ftv(\tau) \setminus \alpha$, where $ftv(\tau)$ is the set of type variables in type $\tau$. For a type environment $\Gamma$, $ftv(\Gamma) = \bigcup_{x \in dom(\Gamma)} ftv(\Gamma(x))$.

A substitution $\{\tau_i/\alpha_i \mid 1 \leq i \leq n\}$ substitutes type $\tau_i$ for type variable $\alpha_i$. We write $\{\tau/\alpha\}$ as a shorthand for a substitution $\{\tau_i/\alpha_i \mid 1 \leq i \leq n\}$, where $\alpha$ and $\tau$ have the same length $n$ and $R\alpha$ for $\{R\alpha_1, \cdots, R\alpha_n\}$. For a substitution $S$, the support $supp(S)$ is $\{\alpha \mid S\alpha \neq \alpha\}$, and the set $itv(S)$ of involved type variables is $\{\alpha, \beta \in supp(S), \alpha \in \{\beta\} \cup ftv(S)\}$. For a substitution $S$ and a type $\tau$, $S\tau$ is the type resulting from applying every substitution component $\tau_i/\alpha_i$ in $S$ to $\tau$. Hence, $\{\}\tau = \tau$.

For a substitution $S$ and a type scheme $\sigma$, $S\sigma = \forall \beta. S(\beta/\alpha)\tau$, where $\beta \cap (itv(S) \cup ftv(\sigma)) = \emptyset$. For a substitution $S$ and a type environment $\Gamma$, $S\Gamma = \{x \mapsto S\alpha \mid x \mapsto \sigma \in \Gamma\}$. The composition of substitutions $S$ and $R$ is written as $RS$, which is $\{R(\sigma) \mid \sigma \in supp(S)\} \cup \{R\alpha \mid \alpha \in supp(R) \setminus supp(S)\}$. Two substitutions $S$ and $R$ are equal if and only if $S\alpha = R\alpha$ for every $\alpha \in supp(S) \cup supp(R)$. 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 $\forall \alpha.\tau \times \tau$ means that there exists a substitution $S$ such that $S\tau = \tau$ and $supp(S) \subseteq \alpha$. We write $\Gamma + x: \sigma$ to mean $\{y \mapsto \sigma' \mid x \neq y, y \mapsto \sigma' \in \Gamma\} \cup \{x \mapsto \sigma\}$.

$Clos_\tau(\tau)$ is the same as $Gen(\Gamma, \tau)$ in [5], i.e., $\forall \alpha.\tau$, where $\alpha = ftv(\tau) \setminus ftv(\Gamma)$. 

Figure 1: Language and its let-polymorphic type system
of two sub-expressions independently and checks later by unification whether those types conflict.

The source language and its Hindley/Milner style let-polymorphic type system are shown in Fig-

its type must satisfy the type constraint that the algorithm has carried to that point.

Example 1 As an example to show the difference between \( W \) and \( M \), consider an application expression

Expression type that the

\( W \): TypEnv \( \times \) Expr \( \rightarrow \) Subst \( \times \) Type

\( W(\Gamma, O) \) = (id, i)

\( W(\Gamma, x) \) = (id, \{ \bar{\beta} / \bar{\alpha} \} \tau \) where \( \Gamma(x) = \forall \bar{\alpha}. \tau \), new \( \bar{\beta} \)

\( W(\Gamma, \lambda x.e) \) = let \( (S_1, \tau_1) = W(\Gamma + x: \beta, e) \), new \( \beta \)
in \( (S_1, S_1\tau_1) \)

\( W(\Gamma, e_1 e_2) \) = let \( (S_1, \tau_1) = W(\Gamma, e_1) \)
\( (S_2, \tau_2) = W(S_1\Gamma, \tau_2) \)
\( S_3 = U(S_2\tau_1, S_2\tau_2 \rightarrow \beta), \) new \( \beta \)
in \( (S_3S_2S_1, S_3\beta) \)

\( W(\Gamma, \text{let } x = e_1 \text{ in } e_2) \) = let \( (S_1, \tau_1) = W(\Gamma, e_1) \)
\( (S_2, \tau_2) = W(S_1\Gamma + x: \beta, S_1\tau_1, e, \tau_2) \)
in \( (S_2S_1, \tau_2) \)

\( W(\Gamma, \text{fix } f \lambda x.e) \) = let \( (S_1, \tau_1) = W(\Gamma + f: \beta, \lambda x.e) \), new \( \beta \)
\( S_2 = U(S_1\beta, \tau_1) \)
in \( (S_2S_1, S_2\tau_1) \)

\( M \): TypEnv \( \times \) Expr \( \times \) Type \( \rightarrow \) Subst

\( M(\Gamma, O, \rho) \) = \( U(\rho, i) \)

\( M(\Gamma, x, \rho) \) = \( U(\rho, \{ \bar{\beta} / \bar{\alpha} \} \tau) \) where \( \Gamma(x) = \forall \bar{\alpha}. \tau \), new \( \bar{\beta} \)

\( M(\Gamma, \lambda x.e, \rho) \) = let \( S_1 = U(\rho, \beta_1 \rightarrow \beta_2) \), new \( \beta_1, \beta_2 \)
\( S_2 = M(S_1\Gamma + x: S_1\beta_1, e, S_1\beta_2) \)
in \( S_2S_1 \)

\( M(\Gamma, e_1 e_2, \rho) \) = let \( S_1 = M(\Gamma, e_1, \beta \rightarrow \rho) \)
\( S_2 = M(S_1\Gamma, e_2, S_1\beta) \)
in \( S_2S_1 \)

\( M(\Gamma, \text{let } x = e_1 \text{ in } e_2, \rho) \) = let \( S_1 = M(\Gamma, e_1, \beta) \), new \( \beta \)
\( S_2 = M(S_1\Gamma + x: S_1\beta, e_2, S_1\rho) \)
in \( S_2S_1 \)

\( M(\Gamma, \text{fix } f \lambda x.e, \rho) \) = \( M(\Gamma + f: \rho, \lambda x.e, \rho) \)

Figure 2: Algorithm \( W \) and \( M \). Note that every new type variable is distinct from each other, and the set New of new type variables introduced at each recursive call to \( W(\Gamma, e) \) (respectively, \( M(\Gamma, e, \rho) \)) satisfies \( \text{New} \cap \text{ftv}(\Gamma) = \emptyset \) (respectively, \( \text{New} \cap (\text{ftv}(\Gamma) \cup \text{ftv}(\rho)) = \emptyset \)).

1.3 Algorithms \( W \) and \( M \)

The source language and its Hindley/Milner style let-polymorphic type system are shown in Figure

1 and its two opposite algorithms (\( W \) and \( M \)) are shown in Figure 2.

Algorithm \( W \) is context-insensitive. It fails only at an application expression. It infers types of
two sub-expressions independently and checks later by unification whether those types conflict.

Because of this, an erroneous expression is often successfully type-checked (context-insensitively)
long before its consequence collides. On the other hand, algorithm \( M \) is as much context-sensitive
as possible. It carries a type constraint (or an expected type) implied by the context of an
expression down to its sub-or-sibling expressions. It fails when the current expression’s type
cannot satisfy the carried type constraint. For example, for an application expression “\( e_1 e_2 \)” with
a type constraint, say of \( \text{int} \), the type constraint for \( e_1 \) is \( \alpha \rightarrow \text{int} \) and the constraint for \( e_2 \) is the

 type that the \( \alpha \) 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.

Example 1 As an example to show the difference between \( W \) and \( M \), consider an application

expression
W fails at the application expression after having successfully type-checked the two sub-expressions, while M fails at the left expression 1 because its type int conflicts with a function type expected from the context (an application).

2 The Generalized Algorithm \( G \)

2.1 Overview

Our generalized algorithm is based on the top-down, context-sensitive algorithm M. Key observation is that we can vary the type-checking strategy by changing two factors in M: the information amount of the type constraints and the places of the unification. Algorithm M carries as much information as possible at its type constraints and applies a unification at every value (constant, variable, and lambda) expression. Algorithm W, on the other hand, carries no information at its type constraints and applies a unification at every application expression. By tuning the two factors, other type-checking strategies are also possible:

Example 2 Consider an application expression

\[(\text{IsOne } 2):\text{bool}\]

where IsOne has type int \(\rightarrow\) bool. As we impose less and less constraints in type-checking sub-expressions yet apply more and more checks later, we achieve the following type-checking variations:

- We type-check IsOne with constraint \(\beta \rightarrow \text{bool}\), which is the strongest expectation. After its success, we type-check 2 with the function’s domain type int as its constraint. (M)

- We type-check IsOne with a weaker constraint, \(\beta_1 \rightarrow \beta_2\) with \(\beta_1\) and \(\beta_2\) being new type variables. The constraint enforces that IsOne’s type be just a function, whatever its domain and range types are. After its success, we check whether the function’s range type is bool. Then we type-check 2 with the function’s domain type int as its constraint.

- We type-check IsOne with no constraint. After its success, we check whether the result type is a function type to bool. Then we type-check 2 with the function’s domain type int as its constraint. (OCaml’s type inference algorithm)

- We type-check IsOne with no constraint. After its success, we check whether the result type is just a function type, whatever its domain and range types are. Then we type-check 2 with the function’s domain type int as its constraint. After its success, we check whether the function’s range type is bool.

- We type-check IsOne with no constraint. After its success, we check, as before, whether the result type is just a function type. Then we type-check 2, but with no constraint. After its success, we check whether the function’s type is int \(\rightarrow\) bool.

- We type-check IsOne with no constraint. After its success, we don’t check anything but continue type-checking the second expression 2 with no constraint. After its success, we check everything at once: we check whether IsOne’s type is a function type from int to bool. (W)

Every type-checking variation in the above example exposes a common property: it loosens the type constraints for sub-expressions then checks afterward whether the results from loosened constraints agree with the contexts implied from the original, unloosened constraints.

Our generalized algorithm is one that allows, wherever possible, the loosening of the type constraints and yet makes sure that posterior unifications compensate for the loosening effects. The places for loosening the constraints are right before recursive calls for type-checking sub-expressions. The places for posterior unifications that compensate for the loosened constraints are after the successful returns from the recursive-calls. Some unifications may only partially...
compensate for the loosened constraints. Thus, before the original call returns there must be final unification(s) that completes the compensations. For example, consider type-checking application expression $e_1 e_2$ with initial constraint $\rho$. It type-checks $e_1$ with a type constraint that can be less restraining than the strongest possible constraint $\beta \rightarrow \rho$. Right after its return, it applies a unification that can compensate, not necessarily completely, for the loosened constraint. It then type-checks the argument expression $e_2$ with a type constraint that can be less restraining than the type that the $\beta$ became. After its success, there exists no more sub-expressions to type-check, hence it’s time to finalize the compensation for the loosened constraints at the two recursive calls. This is done by two unifications: each one compensates for the loosened constraint used in a recursive call to $G(\Gamma, e, \rho)$.

2.2 Algorithm Definition

The generalized algorithm $G$ is shown in Figure 3. As in $M$, it returns a substitution from three components: an expression, a type environment, and a type constraint. The inferred type of the expression is the result from applying the final substitution to the type constraint of the expression. The type constraints are just types.

By the phrases of the form $\theta \geq \rho$ marked (1) to (6) in the algorithm, the strongest type constraint $\rho$ is loosened into $\theta$ at each recursive call. This less restraininng type constraint is the one that can be instantiated to $\rho$ by a substitution that ranges over the type variables in only $\theta$.

Definition 1 ($\theta \geq \rho$) Type $\theta$ is more general (less restraining) than type $\rho$, written $\theta \geq \rho$, if and only if there exists a substitution $G$ such that $G\theta = \rho$ and $\text{supp}(G) = \text{ftv}(\theta) \setminus \text{ftv}(\rho)$. We write

Figure 3: Generalized type inference algorithm $G$. All the type variables in $\text{ftv}(\theta) \setminus \text{ftv}(\rho)$ (respectively, $(\text{ftv}(\theta_1) \cup \text{ftv}(\theta_2)) \setminus \text{ftv}(\rho)$) for each $\theta \geq \rho$ (respectively, $\theta_1 \land \theta_2 \geq \rho$) are new, every new type variable is distinct from each other, and the set $\text{New}$ of new type variables introduced at each recursive call to $G(\Gamma, e, \rho)$ satisfies $\text{New} \cap (\text{ftv}(\Gamma) \cup \text{ftv}(\rho)) = \emptyset$. 

\[
G : \text{TypEnv} \times \text{Expr} \times \text{Type} \rightarrow \text{Subst}
\]

\[
G(\Gamma, \emptyset, \rho) = \text{U}(\rho, \iota) \quad (G.1)
\]

\[
G(\Gamma, x, \rho) = \text{U}(\rho, \{\beta/\alpha\} \tau), \text{ new } \alpha, \Gamma(x) = \forall \alpha \tau \quad (G.2)
\]

\[
G(\Gamma, \lambda x.e, \rho) = \\
\text{let } S_1 = \text{U}(\beta_1 \rightarrow \beta_2, \theta), \text{ new } \beta_1, \beta_2, \quad (1) \theta \geq \rho \quad (G.3)
\]

\[
S_2 = G(S_1 \Gamma + x : S_1 \beta_1, e, S_1 \beta_2) \quad (G.4)
\]

\[
S_3 = \text{U}(S_2 S_1 \Theta, S_2 S_1 \rho) \quad (G.5)
\]

\[
\text{in } S_2 S_1 S_2 S_1
\]

\[
G(\Gamma, e_1 e_2, \rho) = \\
\text{let } S_1 = G(\Gamma, e_1, \Theta), \text{ new } \beta, \quad (2) \theta_1 \geq \beta \rightarrow \rho \quad (G.6)
\]

\[
S_2 = \text{U}(S_1\Theta, \theta_2), \quad (3) \theta_2 \geq S_2(\beta \rightarrow \rho) \quad (G.7)
\]

\[
S_3 = G(S_2 S_1 \Gamma, e_2, \Theta), \quad (4) \theta_3 \geq S_2 S_1 \beta \quad (G.8)
\]

\[
S_4 = \text{U}(S_3 S_2 S_1 \Theta, S_3 S_2 S_1 (\beta \rightarrow \rho)) \quad (G.9)
\]

\[
S_5 = \text{U}(S_3 S_2 S_1 \Theta, S_3 S_2 S_1 \beta) \quad (G.10)
\]

\[
\text{in } S_2 S_1 S_2 S_1
\]

\[
G(\Gamma, \text{let } x := e_1 \text{ in } e_2, \rho) = \\
\text{let } S_1 = G(\Gamma, e_1, \Theta), \text{ new } \beta, \quad (6) \theta_1 \land \theta_2 \geq \rho \quad (G.14)
\]

\[
S_2 = \text{U}(S_1 \Theta, S_1 \theta_2, S_1 \rho) \quad (G.15)
\]

\[
\text{in } S_2 S_1
\]
θ₁ ∧ θ₂ ≥ ρ if and only if there exists a substitution G such that Gθ₁ = ρ and Gθ₂ = ρ and supp(G) = (ftv(θ₁) ∪ ftv(θ₂)) \ ftv(ρ).

For the variable case (G.2), the variable’s type Γ(x) must satisfy the current type constraint ρ: U(ρ, {β/α})τ. Similarly for the constant case (G.1).

For the lambda expression case λx.e with type constraint ρ, we first decide on the type constraint for the function’s body expression e. It can be any type that is less restraining than the range type of ρ. We choose such a type by loosening ρ first, then picking up its range component by unification:

\[ S₁ = U(\beta₁ → \beta₂, \theta), \text{ new } \beta₁, \beta₂, \quad (1) \, \theta ≥ ρ. \]  

Then we use the resulting range type S₁β₂ as the constraint in type-checking the function’s body expression:

\[ S₂ = G(S₁Γ + x: S₁β₁, e, S₁β₂). \]  

For example, if we choose the θ to be a new type variable, then the unification (G.3) has no effect, hence e’s type is inferred without any constraint. The other extreme is to choose θ to be the ρ. Then e’s type is inferred with ρ’s range type, if ρ is a function type.

After returning from the recursive call to e, we have to make up for passing less restraining type constraint. This last step is done by checking whether the loosened constraint θ can agree with the type that its original ρ becomes:

\[ S₃ = U(S₂S₁θ, S₂S₁ρ). \]  

Consider type-checking application expression e₁ e₂ with type constraint ρ. First we decide on the type constraint for the function expression e₁. It can be any type that is less restraining than the most informative constraint β → ρ with β being a new type variable:

\[ S₁ = G(Γ, e, θ₁), \text{ new } \beta, \quad (2) \, θ₁ ≥ β → ρ. \]  

After the success of this recursive call and before we continue by type-checking the argument expression, we can make up, not necessarily completely, for passing less restraining type constraint θ₁. This reparation can be varied by how much we want to expect for the type of e₁. We can check the result type against the strongest constraint β → ρ or we can check against nothing. This varied degree of reparation is achieved by choosing yet another less restraining type θ₂ than S₁(β → ρ) and by unifying it with the type that θ₁ becomes:

\[ S₂ = U(S₁θ₁, θ₂), \quad (3) \, θ₂ ≥ S₁(β → ρ). \]  

Next we decide on the type constraint to pass for type-checking the argument expression e₂. It can be any type that is less constraining than the type that θ becomes. Hence the next recursive call is:

\[ S₃ = G(S₂S₁Γ, e₂, θ₃), \quad (4) \, θ₃ ≥ S₂S₁β. \]  

The finalizing compensation for passing the less restraining type constraints to the two recursive calls are done by checking whether the first loosened constraint θ₁ can agree with the type that the original type β → ρ becomes:

\[ S₄ = U(S₃S₂S₁θ₁, S₃S₂S₁(β → ρ)) \]  

and by checking whether the other loosened constraint θ₃ for the argument expression can agree with what the original type β becomes:

\[ S₅ = U(S₄S₃θ₃, S₄S₃S₂S₁β). \]
We don’t have to check for $\theta_2$ because of its unification with $\theta_1$ at line (G.7).

Consider inferring the type of let-expression $\text{let } x = e_1 \text{ in } e_2$ with type constraint $\rho$. Because there is no context information about the type of the first expression $e_1$, there is no room for varying its type constraint:

$$S_1 = G(\Gamma, e_1, \beta), \text{ new } \beta. \quad (G.11)$$

Next we decide on the type constraint for the body expression $e_2$. It can be any type that is less restraining than the given constraint $\rho$:

$$S_2 = G(S_1 \Gamma + x : \mathsf{Clos}_{S_1}(S_1 \beta), e_2, \theta), \quad (5) \theta \geq S_1 \rho. \quad (G.12)$$

Finally, we have to check whether the loosened constraint agrees with the type that the original constraint becomes:

$$S_3 = \mathcal{U}(S_2 \theta, S_2 S_1 \rho). \quad (G.13)$$

The case for recursive function $\text{fix } f \lambda x. e$ is similar. We decide on what is expected for the type of $\lambda x. e$ and what is carried for the type of $f$. Both can be less restraining than $\rho$:

$$S_1 = G(\Gamma + f : \theta_1, \lambda x. e, \theta_2), \quad (6) \theta_1 \land \theta_2 \geq \rho. \quad (G.14)$$

Then we check whether the loosened type agrees with the type that the original constraint becomes:

$$S_2 = \mathcal{U}(S_1 \theta_1, S_1 \theta_2, S_1 \rho). \quad (G.15)$$

2.3 Instances

By determining the loosened constraints $\theta$’s in $G$, we obtain various type-inference algorithms, including the standard algorithm $W$, the top-down algorithm $M$, and the combinations of the two algorithms used in the SML/NJ [18] and OCaml [11] compiler systems.

- $W$ is an instance of $G$ where every $\theta$ is a new type variable.
- $M$ is an instance of $G$ where every $\theta$ is not loosened: for each case $\theta \geq \rho$ in $G$, we choose $\rho$ for $\theta$.
- The OCaml’s type inference algorithm is an instance of $G$ where the $\theta$ at (2) (line (G.6)) is a new type variable and other $\theta$’s are not loosened.
- The SML/NJ’s type inference algorithm is an instance of $G$ where the $\theta$ at (1) (line (G.3)) is $\rho$ if the lambda is a recursive function, otherwise, a new type variable, the $\theta_1$ and $\theta_2$ at (6) (line (G.14)) are the same new type variable, and other $\theta$’s are new type variables.
- Other variations than the existing algorithms are also possible from $G$. For example, consider an instance of $G$ where the $\theta$ at (G.6) is a new function type ($\beta_1 \to \beta_2$ for new variables $\beta_1$ and $\beta_2$) and other $\theta$’s are their most restraining constraints. Let’s call this instance algorithm $H$.

The $\theta$’s used in the five instances are summarized in Figure 4.

2.4 Every Instance is Sound and Complete

Every instance of $G$ is sound and complete with respect to the Hindley/Milner let-polymorphic type system.

**Theorem 1 (Soundness)** Let $e$ be an expression, $\Gamma$ be a type environment, and $\rho$ be a type. If $G(\Gamma, e, \rho)$ succeeds with $S$, then $\Gamma \vdash e : S\rho$. 


\[ \begin{array}{cccccc}
\text{W} & \{1\} & \{2\} & \{3\} & \{4\} & \{5\} & \{6\} \\
\theta & \beta_1 & \beta_1 & \beta_2 & \beta_3 & \beta_1 & \beta_1 \beta_2 \\
\text{SML/NJ's} & \beta_1 \text{ or } \rho & \beta_1 & \beta_1 & \beta_2 & \beta_3 & \beta_1 & \beta_1 \beta_2 \\
\text{OCaml's} & \rho & \beta_1 & S_1(\beta \rightarrow \rho) & S_2 S_2 \beta & S_1 \rho & \rho & \rho \\
\mathcal{H} & \rho & \beta \rightarrow \beta_2 & S_1(\beta \rightarrow \rho) & S_2 S_2 \beta & S_1 \rho & \rho & \rho \\
\mathcal{M} & \rho & \beta \rightarrow \beta_2 & S_1(\beta \rightarrow \rho) & S_2 S_2 \beta & S_1 \rho & \rho & \rho \\
\end{array} \]

Figure 4: Five instances of algorithm \( \mathcal{G} \). \( \beta_i \)'s are new type variables introduced in the \( \theta_i \)'s.

**Theorem 2 (Completeness)** Let \( e \) be an expression, and let \( \Gamma \) be a type environment. If there exist a type \( \rho \) and a substitution \( P \) such that \( PT \vdash e : P\rho \), then \( \mathcal{G}(\Gamma, e, \rho) \) succeeds with \( S \) and there exists a substitution \( \tau \) such that \( P_{\downarrow \text{New}} = (RS)_{\downarrow \text{New}} \) where New is the set of new type variables used by \( \mathcal{G}(\Gamma, e, \rho) \).

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

**2.5 More Restraining Instance Stops Earlier**

The information amount in the type constraints determines how early the algorithm detects type errors. Carrying less informative (restraining) constraints during type-checking sub-expressions makes it more probable that the algorithm successfully infers their types with being less sensitive to the context, hence delays detecting type errors as such.

We say that an instance \( A \) of \( \mathcal{G} \) is more restraining than another instance \( A' \) whenever \( A \) always passes more restraining constraints than \( A' \). The “always” means that the loosening operations preserve the restraining order between the original constraints: for each pair of corresponding loosening \( \theta_i \geq \rho_i \) in \( A \) and \( \theta_i' \geq \rho_i' \) in \( A' \) for the same input, if \( \rho_i \) is more restraining than \( \rho_i' \) then so is \( \theta_i \) than \( \theta_i' \).

**Definition 2** \( (A \subseteq A') \) Let \( A \) and \( A' \) be two instances of \( \mathcal{G} \). \( A \) is more restraining than \( A' \), written \( A \subseteq A' \), if and only if for each pair of corresponding loosening \( \theta_i \geq \rho_i \) during \( A(\Gamma, e, \rho) \) and \( \theta_i' \geq \rho_i' \) during \( A'(\Gamma, e, \rho) \), if \( \rho_i = R\rho'_i \) for a substitution \( R \) then \( \theta_i = (R_{\text{supp}(\rho)} \cup P)\theta'_i \) for a substitution \( P \) with \( \text{supp}(P) \subseteq \text{ftv}(\theta'_i) \setminus \text{ftv}(\rho'_i) \).

**Lemma 1** \( \mathcal{M} \subseteq \mathcal{H} \subseteq \text{OCaml's} \subseteq \text{SML/NJ's} \subseteq \text{W} \).

The time of detecting type errors can be formalized by the notion of call string [9]. The call string of \( \mathcal{G}(\Gamma, e, \rho) \) (written \( [\mathcal{G}(\Gamma, e, \rho)] \)) is constructed by starting with the empty call string and appending a tuple \( (\Gamma_1, e_1, \rho_1)^d \) (respectively, \( (\Gamma_1, e_1, \rho_1)^u \)) whenever \( \mathcal{G}(\Gamma_1, e_1, \rho_1) \) is called (respectively, returned). The \( d \) or \( u \) superscript indicates the downward or upward movement of the stack pointer when the inference algorithm is recursively called or returned. Note that the callstrings of every instance algorithm of \( \mathcal{G} \) are always finite, because at most one call to the algorithm occurs for each sub-expression of the program, and that the order of visiting sub-expressions of the input program in every instance algorithm’s call string is the same.

For two instance algorithms \( A \) and \( A' \) of \( \mathcal{G} \), if \( A \) is more restraining than \( A' \) then \( A \) stops earlier than \( A' \) if the input program is ill-typed.

**Theorem 3** Let \( A \) and \( A' \) be instances of \( \mathcal{G} \) such that \( A \subseteq A' \), \( \Gamma_0 \) be a type environment, \( e_0 \) be an expression, and \( \rho_0 \) be a type. If \( [A(\Gamma_0, e_0, \rho_0)] \) has \( (\Gamma, e, \rho)^d/u \), then \( [A'(\Gamma_0, e_0, \rho_0)] \) has \( (\Gamma', e, \rho')^d/u \) and there exists a substitution \( R \) such that \( R\Gamma' \geq \Gamma \) and \( R\rho' = \rho \).

Because the order of visiting sub-expressions during the execution of the two instance algorithms are the same, the above theorem implies that if \( A \) is more restraining than \( A' \) then the length (the
number of tuples) \( |\{A(\Gamma_0, e_0, \rho_0)\}| \) of \( A \)'s call string is shorter than or equal to that \( |\{A'(\Gamma_0, e_0, \rho_0)\}| \) of \( A' \)'s call string, i.e., \( A \) stops earlier than \( A' \).

By Lemma 1 and Theorem 3, the following order holds:

**Corollary 1** Let \( \Gamma \) be a type environment, \( e \) be an expression and \( \rho \) be a type.

\[
|\{M(\Gamma, e, \rho)\}| \leq |\{H(\Gamma, e, \rho)\}| \leq |\{OCaml's(\Gamma, e, \rho)\}| \leq |\{SML/NJ's(\Gamma, e, \rho)\}| \leq |\{W(\Gamma, e, \rho)\}|
\]

where \( s \) is the number of tuples in call string \( s \).

Note that all the proofs of the theorems and the lemma are presented in our technical report [10].

### 3 Discussion

We presented a generalized let-polymorphic type inference algorithm, from which, by changing its degree of context-sensitivity, various hybrid algorithms can be instantiated. We proved that any of \( G \)'s instances is sound and complete with respect to the Hindley/Milner let-polymorphic type system, and showed a condition on two instance algorithms so that one algorithm should find type errors earlier than the other. The set of instances of \( G \) includes the two opposite algorithms (\( W \) and \( M \)) and is a superset of those hybrid algorithms used in the SML/NJ [18] and OCaml [11].

Note that the earliness condition cannot be an ultimate criterion to judge the algorithm's goodness in detecting the cause of type-errors. For any algorithm there exists an ill-typed program that falsifies its type-error message. The earliness condition can just be a criterion by which compiler developers can achieve different type-checking strategies.

It is possible to further generalize \( G(\Gamma, e, \rho) \). We can loosen not only the type constraint \( \rho \) but also the type environment \( \Gamma \). Note that algorithm \( G \) passes as much informative type environment as possible to sub-or-sibling expressions; it accumulates all substitutions in type environment at its recursive calls. Contrary to this top-down strategy for type environment, a bottom-up approach [3] uses completely loosened type environments when it checks sub-or-sibling expressions. In between these two opposites, some hybrid strategies are also possible [12, 23]. These variations can be formalized, similarly to \( G \), by type-environment loosening and posterior unification.

In general settings [1, 4, 8, 15, 16, 19, 20, 21] where we view type inference algorithms consist of two separate stages - deriving constraints and solving them - the parameters in our generalized algorithm \( G \) can be considered a way to control when to solve the constraints within the Hindley/Milner type system. We delay the constraint-solving by passing loosened constraints to recursive calls, and then solve the delayed constraints by applying posterior unifications. Sulzmann's general framework [19] in constraint forms is not algorithmic to be directly used in implementation.

### References


