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.

Abstract Syntax

Expr	e	::=	`()`	constant
		$\mid$	x	variable
		$\mid$	$\lambda x.e$	function
		$\mid$	$e\ e$	application
		$\mid$	`let` x`=`e `in` e	
		$\mid$	`fix` $f\ \lambda x.e$	
Type	τ	::=	ι	constant type
		$\mid$	α	type variable
		$\mid$	$\tau \to \tau$	function type
TypeScheme	σ	::=	$\tau \mid \forall\vec{\alpha}.\sigma$	
TypeEnv	Γ	$\in$	$Var \xrightarrow{\text{fin}} TypeScheme$	type environment

$$\text{(CON)}\quad \Gamma \vdash \texttt{()} : \iota$$

$$\text{(VAR)}\quad \frac{\Gamma(x) \succ \tau}{\Gamma \vdash x\colon \tau}$$

$$\text{(FN)}\quad \frac{\Gamma + x\colon \tau_1 \vdash e : \tau_2}{\Gamma \vdash \lambda x.e : \tau_1 \to \tau_2}$$

$$\text{(APP)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \to \tau_2 \quad \Gamma \vdash e_2 : \tau_1}{\Gamma \vdash e_1\ e_2 : \tau_2}$$

$$\text{(LET)}\quad \frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma + x\colon Clos_\Gamma(\tau_1) \vdash e_2 : \tau_2}{\Gamma \vdash \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2 : \tau_2}$$

$$\text{(FIX)}\quad \frac{\Gamma + f\colon \tau \vdash \lambda x.e : \tau}{\Gamma \vdash \texttt{fix}\ f\ \lambda x.e : \tau}$$

Figure 1: Language and its let-polymorphic type system

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 $\vec{\alpha}$ 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 $ftv(\sigma)$ of free type variables in σ is $ftv(\tau) \setminus \vec{\alpha}$, where $ftv(\tau)$ is the set of type variables in type τ. For a type environment Γ, $ftv(\Gamma) = \bigcup_{x \in dom(\Gamma)} ftv(\Gamma(x))$. 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 $supp(S)$ is $\{\alpha \mid S\alpha \neq \alpha\}$, and the set $itv(S)$ of involved type variables is $\{\alpha | \beta \in supp(S), \alpha \in \{\beta\} \cup ftv(S\beta)\}$. 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 $\vec{\beta} \cap (itv(S) \cup ftv(\sigma)) = \emptyset$. 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 $\{R(S\alpha)/\alpha \mid \alpha \in supp(S)\} \cup \{R\alpha/\alpha \mid \alpha \in supp(R) \setminus supp(S)\}$. 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 $\forall\vec{\alpha}.\tau' \succ \tau$ means that there exists a substitution S such that $S\tau' = \tau$ and $supp(S) \subseteq \vec{\alpha}$. We write $\Gamma + x\colon\sigma$ to mean $\{y \mapsto \sigma' \mid x \neq y, y \mapsto \sigma' \in \Gamma\} \cup \{x \mapsto \sigma\}$. $Clos_\Gamma(\tau)$ is the same as Gen(Γ,τ) in [5], i.e., $\forall\vec{\alpha}.\tau$, where $\vec{\alpha} = ftv(\tau) \setminus ftv(\Gamma)$.

$\mathcal{W}$: *TypEnv* $\times$ *Expr* $\to$ *Subst* $\times$ *Type*

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

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

$$
\begin{array}{lcl}
\mathcal{M}(\Gamma,\ (),\ \rho)
& = & \mathcal{U}(\rho,\ \iota) \\
\mathcal{M}(\Gamma,\ x,\ \rho)
& = & \mathcal{U}(\rho,\ \{\vec{\beta}/\vec{\alpha}\}\tau) \text{ where } \Gamma(x) = \forall\vec{\alpha}.\tau,\ \text{new } \vec{\beta} \\
\mathcal{M}(\Gamma,\ \lambda x.e,\ \rho)
& = & \text{let} \quad S_1 = \mathcal{U}(\rho,\ \beta_1 \to \beta_2),\ \text{new } \beta_1,\ \beta_2 \\
& & \qquad\ S_2 = \mathcal{M}(S_1\Gamma + x\colon S_1\beta_1,\ e,\ S_1\beta_2) \\
& & \text{in} \quad S_2S_1 \\
\mathcal{M}(\Gamma,\ e_1\ e_2,\ \rho)
& = & \text{let} \quad S_1 = \mathcal{M}(\Gamma,\ e_1,\ \beta \to \rho),\ \text{new } \beta \\
& & \qquad\ S_2 = \mathcal{M}(S_1\Gamma,\ e_2,\ S_1\beta) \\
& & \text{in} \quad S_2S_1 \\
\mathcal{M}(\Gamma, \texttt{let}\ x\texttt{=}e_1\ \texttt{in}\ e_2, \rho) = & & \\
& & \text{let} \quad S_1 = \mathcal{M}(\Gamma,\ e_1,\ \beta),\ \text{new } \beta \\
& & \qquad\ S_2 = \mathcal{M}(S_1\Gamma + x\colon \mathit{Clos}_{S_1\Gamma}(S_1\beta),\ e_2,\ S_1\rho) \\
& & \text{in} \quad S_2S_1 \\
\mathcal{M}(\Gamma,\ \texttt{fix}\ f\ \lambda x.e,\ \rho)
& = & \mathcal{M}(\Gamma + f\colon \rho,\ \lambda x.e,\ \rho)
\end{array}
$$

Figure 2: Algorithm $\mathcal{W}$ and $\mathcal{M}$. 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 $\mathcal{W}(\Gamma, e)$ (respectively, $\mathcal{M}(\Gamma, e, \rho)$) satisfies $\mathit{New} \cap \mathit{ftv}(\Gamma) = \emptyset$ (respectively, $\mathit{New} \cap (\mathit{ftv}(\Gamma) \cup \mathit{ftv}(\rho)) = \emptyset$.)

1.3 Algorithms $\mathcal{W}$ and $\mathcal{M}$

The source language and its Hindley/Milner style let-polymorphic type system are shown in Figure 1 and its two opposite algorithms ($\mathcal{W}$ and $\mathcal{M}$) are shown in Figure 2.

Algorithm $\mathcal{W}$ 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 $\mathcal{M}$ 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 `int`, the type constraint for e_1 is $\alpha \to$ `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.

Example 1 As an example to show the difference between $\mathcal{W}$ and $\mathcal{M}$, consider an application expression

$\mathcal{W}$ fails at the application expression *after* having successfully type-checked the two sub-expressions, while $\mathcal{M}$ 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 $\mathcal{G}$

2.1 Overview

Our generalized algorithm is based on the top-down, context-sensitive algorithm $\mathcal{M}$. Key observation is that we can vary the type-checking strategy by changing two factors in $\mathcal{M}$: the information amount of the type constraints and the places of the unification. Algorithm $\mathcal{M}$ carries as much information as possible at its type constraints and applies a unification at every value (constant, variable, and lambda) expression. Algorithm $\mathcal{W}$, 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

```
(IsOne 2):bool
```

where `IsOne` has type `int` → `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 $\beta \rightarrow$ `bool`, which is the strongest expectation. After its success, we type-check `2` with the function's domain type `int` as its constraint. ($\mathcal{M}$)
- We type-check `IsOne` with a weaker constraint, $\beta_1 \rightarrow \beta_2$ with β_1 and β_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` → `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`. ($\mathcal{W}$) □

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

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

$\mathcal{G}(\Gamma, (), \rho) = \mathcal{U}(\rho, \iota)$ ($\mathcal{G}$.1)

$\mathcal{G}(\Gamma, x, \rho) = \mathcal{U}(\rho, \{\vec{\beta}/\vec{\alpha}\}\tau),\ \text{new } \vec{\beta},\ \Gamma(x) = \forall\vec{\alpha}.\tau$ ($\mathcal{G}$.2)

$\mathcal{G}(\Gamma, \lambda x.e, \rho) =$

  let $S_1 = \mathcal{U}(\beta_1 \to \beta_2, \theta),\ \text{new } \beta_1,\ \text{new } \beta_2,\quad (1)\ \theta \geq \rho$ ($\mathcal{G}$.3)

  $S_2 = \mathcal{G}(S_1\Gamma + x\colon S_1\beta_1, e, S_1\beta_2)$ ($\mathcal{G}$.4)

  $S_3 = \mathcal{U}(S_2S_1\theta, S_2S_1\rho)$ ($\mathcal{G}$.5)

  in $S_3S_2S_1$

$\mathcal{G}(\Gamma, e_1\ e_2, \rho) =$

  let $S_1 = \mathcal{G}(\Gamma, e_1, \theta_1),\ \text{new } \beta,\quad (2)\ \theta_1 \geq \beta \to \rho$ ($\mathcal{G}$.6)

  $S_2 = \mathcal{U}(S_1\theta_1, \theta_2),\quad (3)\ \theta_2 \geq S_1(\beta \to \rho)$ ($\mathcal{G}$.7)

  $S_3 = \mathcal{G}(S_2S_1\Gamma, e_2, \theta_3),\quad (4)\ \theta_3 \geq S_2S_1\beta$ ($\mathcal{G}$.8)

  $S_4 = \mathcal{U}(S_3S_2S_1\theta_1, S_3S_2S_1(\beta \to \rho))$ ($\mathcal{G}$.9)

  $S_5 = \mathcal{U}(S_4S_3\theta_3, S_4S_3S_2S_1\beta)$ ($\mathcal{G}$.10)

  in $S_5S_4S_3S_2S_1$

$\mathcal{G}(\Gamma, \texttt{let } x\texttt{=}e_1 \texttt{ in } e_2, \rho) =$

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

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

  $S_3 = \mathcal{U}(S_2\theta, S_2S_1\rho)$ ($\mathcal{G}$.13)

  in $S_3S_2S_1$

$\mathcal{G}(\Gamma, \texttt{fix } f\ \lambda x.e, \rho) =$

  let $S_1 = \mathcal{G}(\Gamma + f\colon \theta_1, \lambda x.e, \theta_2),\quad (6)\ \theta_1 \wedge \theta_2 \geq \rho$ ($\mathcal{G}$.14)

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

  in S_2S_1

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

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 ρ. It type-checks e_1 with a type constraint that can be less restraining than the strongest possible constraint $\beta \to \rho$. 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 β 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 type-checking each sub-expression. The unifications check whether the types from the loosened constraints agree with what the strongest constraint $\beta \to \rho$ implies.

2.2 Algorithm Definition

The generalized algorithm $\mathcal{G}$ is shown in Figure 3. As in $\mathcal{M}$, 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 ρ is loosened into θ at each recursive call. This less restraining type constraint is the one that can be instantiated to ρ by a substitution that ranges over the type variables in only θ:

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

$\theta_1 \wedge \theta_2 \geq \rho$ if and only if there exists a substitution G such that $G\theta_1 = \rho$ and $G\theta_2 = \rho$ and $supp(G) = (ftv(\theta_1) \cup ftv(\theta_2)) \setminus ftv(\rho)$.

For the variable case ($\mathcal{G}.2$), the variable's type $\Gamma(x)$ must satisfy the current type constraint ρ: $\mathcal{U}(\rho, \{\vec{\beta}/\vec{\alpha}\}\tau)$. Similarly for the constant case ($\mathcal{G}.1$).

For the lambda expression case $\lambda x.e$ 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:

$$S_1 = \mathcal{U}(\beta_1 \to \beta_2, \theta), \text{ new } \beta_1, \beta_2, \quad (1)\ \theta \geq \rho. \tag{$\mathcal{G}.3$}$$

Then we use the resulting range type $S_1\beta_2$ as the constraint in type-checking the function's body expression:

$$S_2 = \mathcal{G}(S_1\Gamma + x\colon S_1\beta_1, e, S_1\beta_2). \tag{$\mathcal{G}.4$}$$

For example, if we choose the θ to be a new type variable, then the unification ($\mathcal{G}.3$) 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:

$$S_3 = \mathcal{U}(S_2S_1\theta, S_2S_1\rho). \tag{$\mathcal{G}.5$}$$

Consider type-checking application expression $e_1\ e_2$ with type constraint ρ. First we decide on the type constraint for the function expression e_1. It can be any type that is less restraining than the most informative constraint $\beta \to \rho$ with β being a new type variable:

$$S_1 = \mathcal{G}(\Gamma, e, \theta_1), \text{ new } \beta, \quad (2)\ \theta_1 \geq \beta \to \rho. \tag{$\mathcal{G}.6$}$$

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 θ_1. This reparation can be varied by how much we want to expect for the type of e_1. We can check the result type against the strongest constraint $\beta \to \rho$ or we can check against nothing. This varied degree of reparation is achieved by choosing yet another less restraining type θ_2 than $S_1(\beta \to \rho)$ and by unifying it with the type that θ_1 becomes:

$$S_2 = \mathcal{U}(S_1\theta_1, \theta_2), \quad (3)\ \theta_2 \geq S_1(\beta \to \rho). \tag{$\mathcal{G}.7$}$$

Next we decide on the type constraint to pass for type-checking the argument expression e_2. It can be any type that is less constraining than the type that β becomes. Hence the next recursive call is:

$$S_3 = \mathcal{G}(S_2S_1\Gamma, e_2, \theta_3), \quad (4)\ \theta_3 \geq S_2S_1\beta. \tag{$\mathcal{G}.8$}$$

The finalizing compensation for passing the less restraining type constraints to the two recursive calls are done by checking whether the first loosened constraint θ_1 can agree with the type that the original type $\beta \to \rho$ becomes:

$$S_4 = \mathcal{U}(S_3S_2S_1\theta_1, S_3S_2S_1(\beta \to \rho)) \tag{$\mathcal{G}.9$}$$

and by checking whether the other loosened constraint θ_3 for the argument expression can agree with what the original type β becomes:

$$S_5 = \mathcal{U}(S_4S_3\theta_3, S_4S_3S_2S_1\beta). \tag{$\mathcal{G}.10$}$$

We don't have to check for θ_2 because of its unification with θ_1 at line ($\mathcal{G}$.7).

Consider inferring the type of let-expression `let` x=e_1 `in` e_2 with type constraint ρ. Because there is no context information about the type of the first expression e_1, there is no room for varying its type constraint:

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

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 ρ:

$$S_2 = \mathcal{G}(S_1\Gamma + x\colon Clos_{S_1\Gamma}(S_1\beta), e_2, \theta), \quad (5)\ \theta \geq S_1\rho. \tag{$\mathcal{G}$.12}$$

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

$$S_3 = \mathcal{U}(S_2\theta, S_2S_1\rho). \tag{$\mathcal{G}$.13}$$

The case for recursive function `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 ρ:

$$S_1 = \mathcal{G}(\Gamma + f\colon \theta_1, \lambda x.e, \theta_2), \quad (6)\ \theta_1 \wedge \theta_2 \geq \rho. \tag{$\mathcal{G}$.14}$$

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

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

2.3 Instances

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

- $\mathcal{W}$ is an instance of $\mathcal{G}$ where every θ is a new type variable.
- $\mathcal{M}$ is an instance of $\mathcal{G}$ where every θ is not loosened: for each case $\theta \geq \rho$ in $\mathcal{G}$, we choose ρ for θ.
- The OCaml's type inference algorithm is an instance of $\mathcal{G}$ where the θ at (2) (line ($\mathcal{G}$.6)) is a new type variable and other θ's are not loosened.
- The SML/NJ's type inference algorithm is an instance of $\mathcal{G}$ where the θ at (1) (line ($\mathcal{G}$.3)) is ρ if the lambda is a recursive function, otherwise, a new type variable, the θ_1 and θ_2 at (6) (line ($\mathcal{G}$.14)) are the same new type variable, and other θ's are new type variables.
- Other variations than the existing algorithms are also possible from $\mathcal{G}$. For example, consider an instance of $\mathcal{G}$ where the θ at ($\mathcal{G}$.6) is a new function type ($\beta_1 \to \beta_2$ for new variables β_1 and β_2) and other θ's are their most restraining constraints. Let's call this instance algorithm $\mathcal{H}$.

The θ's used in the five instances are summarized in Figure 4.

2.4 Every Instance is Sound and Complete

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

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

	(1) θ	(2) θ_1	(3) θ_2	(4) θ_3	(5) θ	(6) θ_1, θ_2
$\mathcal{W}$	β_1	β_1	β_2	β_3	β_1	β_1, β_2
SML/NJ's	β_1 or ρ	β_1	β_2	β_3	β_1	β_1, β_1
OCaml's	ρ	β_1	$S_1(\beta \to \rho)$	$S_2S_1\beta$	$S_1\rho$	ρ, ρ
$\mathcal{H}$	ρ	$\beta \to \beta_2$	$S_1(\beta \to \rho)$	$S_2S_1\beta$	$S_1\rho$	ρ, ρ
$\mathcal{M}$	ρ	$\beta \to \rho$	$S_1(\beta \to \rho)$	$S_2S_1\beta$	$S_1\rho$	ρ, ρ

Figure 4: Five instances of algorithm $\mathcal{G}$. β_i's are new type variables introduced in the θ's.

Theorem 2 (Completeness) *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 $\mathcal{G}(\Gamma, e, \rho)$ succeeds with S and there exists a substitution R such that $P|_{New} = (RS)|_{New}$ 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 τ that satisfies a type constraint ρ (i.e., $\exists P.\tau = P\rho$), then algorithm $\mathcal{G}$ for the expression with the constraint ρ succeeds with substitution S such that the result type $S\rho$ subsumes τ (i.e., the principality, $\exists R.\tau = R(S\rho)$).

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 loosenings $\theta_i \geq \rho_i$ in A and $\theta'_i \geq \rho'_i$ in A' for the same input, if ρ_i is more restraining than ρ'_i then so is θ_i than θ'_i.

Definition 2 ($A \sqsubseteq A'$) Let A and A' be two instances of $\mathcal{G}$. A is more restraining than A', written $A \sqsubseteq A'$, if and only if for each pair of corresponding loosenings $\theta_i \geq \rho_i$ 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 $\theta_i = (R|_{supp(P)} \cup P)\theta'_i$ for a substitution P with $supp(P) \subseteq ftv(\theta'_i) \setminus ftv(\rho'_i)$.

Lemma 1 *$\mathcal{M} \sqsubseteq \mathcal{H} \sqsubseteq$ OCaml's $\sqsubseteq$ SML/NJ's $\sqsubseteq \mathcal{W}$.*

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 *d*ownward or *u*pward movement of the stack pointer when the inference algorithm is recursively called or returned. Note that the call strings 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 $A \sqsubseteq A'$, Γ_0 be a type environment, e_0 be an expression, and ρ_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 $R\Gamma' \succ \Gamma$ 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 Γ be a type environment, e be an expression and ρ be a type.*

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

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 $\mathcal{G}$'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 $\mathcal{G}$ includes the two opposite algorithms ($\mathcal{W}$ and $\mathcal{M}$) 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 $\mathcal{G}(\Gamma, e, \rho)$. We can loosen not only the type constraint ρ but also the type environment Γ. Note that algorithm $\mathcal{G}$ 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 $\mathcal{G}$, 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 $\mathcal{G}$ 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

[1] Alexander Aiken and Edward L. Wimmers. Type inclusion constraints and type inference. In *Proceedings of Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.

[2] Mike Beaven and Ryan Stansifer. Explaining type errors in polymorphic languages. *ACM Letters on Programming Languages and Systems*, 2:17–30, March-December 1993.

[3] Karen L. Bernstein and Eugene W. Stark. Debugging type errors (full version). Technical report, State University of New York at Stony Brook, 1995.

[4] Kenta Cho and Kazunori Ueda. Diagnosing non-well-moded concurrent logic programs. In *Joint International Conference on Logic Programming*, pages 215–229. MIT Press, 1996.

[5] Luis Damas and Robin Milner. Principal type-scheme for functional programs. In *Proceedings of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 207–212, New York, 1982. ACM Press.

[6] Dominic Duggan. Correct type explanation. In *Proceedings of Workshop on ML*, pages 49–58, 1998.

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

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

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

[10] Oukseh Lee and Kwangkeun Yi. A generalized let-polymorphic type inference algorithm. Technical Memorandum ROPAS-2000-5, Research On Program Analysis System, National Creative Research Center, Korea Advanced Institute of Science and Technology, March 2000.

[11] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. The objective caml system release 2.04. Institut National de Recherche en Informatique et en Automatique, November 1999. `http://caml.inria.fr`.

[12] Bruce J. McAdam. On the unification of substitutions in type inference. In Kevin Hammond, Anthony J. T. Davie, and Chris Clack, editors, *Proceedings of The International Workshop on Implementation of Fuctional Languages*, volume 1595 of *Lecture Notes in Computer Science*, pages 139–154. Springer-Verlag, September 1998.

[13] Bruce J. McAdam. Generalising techniques for type debugging. In *Proceedings of 1st Scottish Functional Programming Workshop*, 1999.

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

[15] Martin Odersky, Martin Sulzmann, and Martin Wehr. Type inference with constrained types. *Theory and Practice of Object Systems*, 5(1), 1999.

[16] Didier Rémy. Extending ML type system with a sorted equational theory. Research Report 1766, Institut National de Recherche en Informatique et Automatisme, Rocquencourt, BP 105, 78 153 Le Chesnay Cedex, France, 1992.

[17] Laurence Rideau and Laurent Théry. Interactive programming environment for ML. Technical Report 3139, Institut National de Recherche en Informatique et en Automatique, March 1997.

[18] The Standard ML of New Jersey, release 110.0.6. Bell Labs, Lucent Technologies, November 1999. `http://cm.bell-labs.com/cm/cs/what/smlnj`.

[19] Martin Sulzmann. A general type inference framework for Hindley/Milner style systems. Technical Report TR2000/15, The University of Melbourne, Department of Computer Science and Software Engineering, July 2000.

[20] Martin Sulzmann, Martin Müller, and Christoph Zenger. Hindley/Milner style type systems in constraint form. Technical Report ACRC-99-009, University of South Australia, School of Computer and Information Science, July 1999.

[21] Satish R. Thatte. Type inference with partial types. *Theoretical Computer Science*, 124(1):127–148, February 1994.

[22] Mitchell Wand. Finding the source of type errors. In *Proceedings of the 13th Annual ACM Symposium on Principles of Programming Languages*, pages 38–43, New York, 1986. ACM Press.

[23] Jun Yang. Explaining type errors by finding the sources of type conflicts. In *Proceedings of 1st Scottish Functional Programming Workshop*, pages 387–401, August 1999.