

Wand’s Algorithm Extended For the Polymorphic ML-Let^{*}

This article is an unabridged version of the paper: On Extending Wand’s Algorithm to Handle Polymorphic Let

Sunil Kothari and James L. Caldwell

Department of Computer Science
University of Wyoming
Laramie, USA
{skothari, jlc}@cs.uwo.edu

Abstract. This paper details an extension to Wand’s algorithm to handle the polymorphic let (a.k.a. ML-Let) construct. We have extended Wand’s algorithm by extending the constraint language and by using a multi-phase unification algorithm in the constraint solving phase. We show the correctness of our approach by extending the Wand’s soundness and completeness results. We have validated our approach against other popular type reconstruction algorithms by implementing OCaml prototypes and running them on non-trivial examples.

1 Introduction

The general type reconstruction problem can be formulated as:

Given a well-formed term M without any types, does there exist a type τ and a type environment¹ Γ such that we can form a judgment $\Gamma \vdash M : \tau$?

Type reconstruction is a popular feature in many modern functional programming languages. Underlying any type reconstruction algorithm is a set of rules called a type system. One of the most widely used type systems is the Hindley-Milner (HM) type system, first mentioned in [Mil78] by Milner, but discovered independently by Hindley [Hin69]. Various type reconstruction algorithms [Mil78, LY98, DM82, Wan87] have been proposed to implement the HM type system. In fact, there is a very elegant exposition on ML type inference by Pottier and Rémy [PR05]. Many of the algorithms are characterized by an intermittent constraint generation and constraint solving. But over the years, focus has shifted to algorithms having a clear separation of constraint generation and constraint solving phases. This separation leads to better error messages

^{*} This material is based upon work supported by the National Science Foundation under Grant No. NSF CNS-0613919.

¹ In our case, the initial environment is empty since we are dealing with closed terms.

[Hee05, PR05] when the constraint set is unsatisfiable (since a larger set of constraints is available to reason about the error). Moreover the separation provides a clean abstraction of the various substitution-based algorithms² since most well known algorithms are specific instances of various constraint solving strategies.

The type inference involving polymorphic let construct is a non-trivial problem, in fact, in the worst case, it is a DEXPTIME³-complete and PSPACE-hard problem [PJ89, Mai89] in the levels of nested lets. But on average the type inference problem is solvable in polynomial time. Moreover the literature on constraint-based type reconstruction is sparse and uneven; they do not seem to be a *natural* extension of Wand’s algorithm [Wan87]. Even the popular book [Pie02] on types has no references on how to handle ML-Let construct in constraint-based algorithms. Whatever literature there is requires specialized knowledge [SOW97] or is too specialized to a particular constraint representation [AW93]. Also, it is questionable whether these approaches help in generating good quality error messages when the type inference fails.

On the other hand, Helium compiler [HLI03] is known for giving good quality error messages and a very simple constraint representation⁴ is used for handling the let construct. But there have been no published account of such an approach so far. This paper describes an approach where the Helium’s constraint representation is used to handle let polymorphism. Our approach builds upon Wand’s algorithm and our proofs rely on the soundness and completeness of Wand’s algorithm. We have validated our approach with some of the known type reconstruction algorithms [Kot07]. In summary, our contributions in this paper are:

1. A new *modular* extension to Wand’s algorithm to handle polymorphic let.
2. New soundness and completeness proofs for Wand’s algorithm.
3. Soundness and completeness proofs of the extension with a novel desugaring of polymorphic lets.
4. Ocaml implementation of Wand’s algorithm and the extension.

The rest of this paper is organized as follows: Section 2 reviews the previous methods for inferring the type of the let construct. Section 3 introduces the concepts and terminologies needed for this paper. Section 4 gives an overview of the Wand’s algorithm and details soundness and correctness results. Section 5 describes the extension to Wand’s algorithm for handling the polymorphic let construct. Section 7 summarizes our current work and mentions further work.

We use the following type-preserving transformation [Mai89] to make all the let-bindings let free:

$$\frac{}{\mathbf{let } x = (\mathbf{let } y = M \mathbf{ in } N) \mathbf{ in } P \implies \mathbf{let } y = M \mathbf{ in } (\mathbf{let } x = N \mathbf{ in } P)}$$

² Throughout this paper we term *substitution-based algorithms* as those algorithms which intermix constraint generation and constraint solving whereas algorithms with a clear separation are termed *constraint-based algorithms*.

³ DTIME($2^{n^{O(1)}}$)

⁴ Personal Communication from Bastiaan Heeren.

In other words, in a term like **let** $x = M$ **in** N the sub-term M can be made let-free without changing the type of an expression. This is not a restriction on our algorithm (see Example 2) but is intended to make proofs simpler. We restrict the notion of monomorphism⁵ to just one occurrence of a let-bound variable in the let-body rather than the more general notion of having just one form. This is a more restricted notion but does not affect any of the results in this paper. For a polymorphic-let term **let** $x = M$ **in** N , we use N as a context and denote one hole (one occurrence of a polymorphic let variable) in the context by $[]$. Thus, context with a hole filled by a let-bound variable is represented by the term $N[x]$.

2 Literature Review

We review both constraint-based and substitution-based algorithms in their handling of the let construct. A survey of substitution-based algorithms is available in [Kot07]. **Wand** [Wan87] looked at a language based on pure untyped lambda calculus. However, extension to the algorithm to handle let construct remained a future work⁶. **Aiken and Wimmers** [AW93] hint at how inclusion constraints (of the form $X \subseteq Y$) over type expressions can be used for handling the let construct. They introduce a type scheme $\forall\alpha.\tau$ where S involving universal quantification of a type expression τ and its associated constraints S . However, it doesn't lend itself as an extension to Wand's algorithm. **Liang** [Lia97] handled the let construct by giving a new algorithm named W_π where the *gen* operation is not used as it leads to declarative formulation of type inference. This approach again is a substitution-based algorithm. **Lee and Yi** [LY98] mention a substitution-based, top-down algorithm very similar to Algorithm W but gives better error messages when the unification fails. They handle the let construct by using the *gen* operator. **Shao and Appel** [SA93] uses an operation called *polyunify*, which merges the two assumption environments generated from the expressions e_1, e_2 in **let** $x = e_1$ **in** e_2 and followed by matching with the type environment. Again the algorithm presented in their work is substitution-based. **Milner** [Mil78] handles the let construct by giving two algorithms, namely Algorithm W and Algorithm J, but both these algorithms are substitution-based algorithms where constraints are solved as soon as possible. There is no clear separation between constraint solving and constraint generation phases. **Heeren** [Hee05] suggests three constraint representations to handle the let construct. Specifically, he addresses the third constraint representation (given below) in his thesis. Note that each representation is equally expressive. We summarize the three representations:

Approach 1. Qualification of type constraints: Types schemas contain a constraint component as part of its type as shown by the following grammar:

⁵ In a let-term **let** $x = M$ **in** N , x is monomorphic if the cardinality of the set $FV(N)$ with respect to x (denoted by $|FV(N)|_x$) is 1 or less.

⁶ Personal communication from Mitchell Wand.

$$\sigma ::= \forall \vec{\alpha}. \sigma \mid C \Rightarrow \tau$$

For example, an expression $\lambda x.x$ can be assigned a type $\tau_1 \rightarrow \tau_2$ under the constraint $\tau_1 \equiv \tau_2$. So the overall type for the above expression is given as $\forall \tau_1, \tau_2. \tau_1 \equiv \tau_2 \Rightarrow \tau_1 \rightarrow \tau_2$. In many ways, this approach is similar to HM(X)[SOW97], qualified types [?], and Pottier and Rémy’s account of type inference . Clearly this approach is simple as we deal only with equality constraints. But to deal with polymorphism, local definitions are in-lined at each usage and thus leads to poor error messages if the constraints generated are unsatisfiable.

Approach 2. Introducing type scheme variables: The type constraint language is extended to take into account generalization and instantiation of type schemes. The constraint language is modified to:

$$\mathcal{C} ::= \tau_1 \equiv \tau_2 \mid \sigma := GEN(\Gamma, \tau) \mid \tau := INST(\sigma)$$

In fact our approach uses the above constraint language and follows closely the constraint generation phase as described by Heeren [Hee05] but the constraint solving phase is quite different. There is no published account of the constraint solving phase for this representation.

Approach 3. Collecting implicit instance constraints: This approach suggests merges the generalization and instantiation constraints in the above approach. The constraint language now becomes:

$$\mathcal{C} ::= \tau_1 \equiv \tau_2 \text{ (equality constraint)} \mid \tau_1 \leq_M \tau_2 \text{ (implicit instance constraint)}$$

The semantics of implicit instance constraint is as follows: the type τ_1 is obtained by generalizing the free variables of type τ_2 excluding the type variable that are free in \mathcal{M} . The advantage of this approach is that there are no type scheme variables in the constraints or in the type environment. The solving of implicit instance constraints is order-dependent. The order given by the set of active variables.

Muller [Mül94] gives a constraint-based algorithm for handling the let construct. Here the constraints are implemented in a relational calculus, ρ_{deep} with higher-order abstractions (for polymorphic types) and 1st-order constraints (for monomorphic type). **Sulzmann et. al.** [SOW97] introduced HM(X) framework parametrized by the constraint domain X. For the HM type systems, the parameter X is instantiated to standard Herbrand constraint system. Type schemes are treated as constrained types of the form $\forall \alpha. C \Rightarrow \sigma$, where C is a constraint in X that restricts the types that can be substituted for α . Monomorphic type τ is treated as $\forall \alpha. true \Rightarrow \tau$ where $\alpha \notin FTV(\tau)$. The typing judgments are of the form $\psi, C, \Gamma \vdash^w e : \tau$. Additionally, a normalization step is performed to convert constraints in term constraint system to get a constraint C in cylindrical constraint system X. **Rémy and Pottier** [PR05] describe a less abstract approach to type inference but their approach is again based on constrained types much like HM(X). **Choppella** [Cho05] presents an algorithm W^E that relies on a limited separation of type equations from their solutions. Specifically, type generation is continued until a let expression is encountered in which case the type equations are solved before proceeding further.

3 Preliminaries

We assume familiarity with the notion of types and functional programming and lambda calculus. For an overview of type systems in programming languages readers can refer to Cardelli's paper [Car97] and the Pierce's book(s) [Pie02, PR05]. Next we describe the terms and types language considered in this section.

The terms considered here are pure lambda terms given by the following grammar:

$$A ::= x \mid MN \mid \lambda x.M$$

In this paper we follow the usual conventions for lambda calculus: arrow types associate to the right, function applications associate to the left, and applications bind more tightly than abstractions. The set of *free variables* of a term is defined as:

$$\begin{aligned} FV(x) &= \{x\} \\ FV(\lambda x.M) &= FV(M) - \{x\} \\ FV(M N) &= FV(M) \cup FV(N) \end{aligned}$$

The set of *bound variables* of a term is defined as:

$$\begin{aligned} BV(x) &= \{x\} \\ BV(\lambda x.M) &= BV(M) \cup \{x\} \\ BV(M N) &= BV(M) \cup BV(N) \end{aligned}$$

Capture avoiding substitution, denoted by $M[x:=N]$, is given as:

$$\begin{aligned} x[y:=M] &= x \\ x[x:=M] &= M \\ (M L)[x:=N] &= (M[x:=N] L[x:=N]) \\ (\lambda x. M)[x:=N] &= \lambda x. M \\ (\lambda y. M)[x:=N] &= (\lambda y. M[x:=N]) \quad \text{if } y \notin FV(N) \\ (\lambda y. M)[x:=N] &= (\lambda z. (M[y:=z])[z:=N]) \quad \text{if } y \in FV(N) \text{ and } z \text{ is new} \end{aligned}$$

The types for untyped lambda terms is given by the following grammar:

$$\tau ::= \alpha \mid \tau_1 \rightarrow \tau_2$$

The type is either a type variable or an arrow type. We follow the convention that α, β denotes a type variable whereas τ denotes a type expression that is derived from the above grammar and \rightarrow associates to the right. For example, $\alpha \rightarrow \beta \rightarrow \alpha$ should be read as $\alpha \rightarrow (\beta \rightarrow \alpha)$. A *type environment*, denoted by Γ , maps type variables to type expressions. The set of *free type variables* (FTV) of a type expression τ is denoted by $FTV(\tau)$. Terms and types are related by assertions, $\Gamma \vdash M : \tau$, where M is a term, τ is a type, and Γ is a type environment. We denote type environment where x is not in the domain of Γ by $\Gamma \setminus x$. A *derivation* of an assertion $\Gamma \vdash t : \tau$ is a finite tree of assertions, where the root is $\Gamma \vdash t : \tau$. Every assertion in that tree is related to its parent by an instance of the type rules. In this paper, we consider three different type systems: Hindley-Milner type system illustrated in Fig. 1, Wand's system illustrated in Fig. 2 and Wand's system extended with polymorphic-let (described in Sec. 5). We denote a derivation by Δ_x where Δ denotes a derivation and the subscript x denotes the type system. For example, Δ_W, Δ_{W+} , and Δ_{HM} denote a derivation

in Wand's system, extended Wand's system and the Hindley-Milner type system respectively. Similarly, every judgment also gets a subscript to qualify it as a judgment in a particular type system. For example, a judgment in Hindley-Milner type system is denoted by $\Gamma \vdash_{HM} t : \tau$.

$$\begin{array}{c}
\frac{}{\Gamma \vdash_{HM} x : \tau} \text{ where } x : \tau \in \Gamma \quad (HM\text{-Var}) \\
\frac{\Gamma \setminus x \cup \{x : \tau_1\} \vdash_{HM} M : \tau_0}{\Gamma \vdash_{HM} \lambda x. M : \tau_1 \rightarrow \tau_0} \quad (HM\text{-Abs}) \\
\frac{\Gamma \vdash_{HM} M : \tau_1 \rightarrow \tau \quad \Gamma \vdash_{HM} N : \tau_1}{\Gamma \vdash_{HM} MN : \tau} \quad (HM\text{-App})
\end{array}$$

Fig. 1. Syntax Directed Hindley-Milner type system

$$\begin{array}{c}
\frac{}{\Gamma, \{\alpha \stackrel{e}{=} \tau\} \vdash_W x : \alpha} \text{ where } x : \tau \in \Gamma \text{ and } \alpha \text{ is fresh} \quad (W\text{-Var}) \\
\frac{(\Gamma \setminus x) \cup \{x : \alpha\}, E \vdash_W M : \beta}{\Gamma, E \cup \{\tau \stackrel{e}{=} \alpha \rightarrow \beta\} \vdash_W \lambda x. M : \tau} \text{ where } \alpha, \beta \text{ are fresh} \quad (W\text{-Abs}) \\
\frac{\Gamma, E_1 \vdash_W M : \alpha \rightarrow \tau \quad \Gamma, E_2 \vdash_W N : \alpha}{\Gamma, E_1 \cup E_2 \vdash_W MN : \tau} \text{ where } \alpha \text{ is fresh} \quad (W\text{-App})
\end{array}$$

Fig. 2. Wand's type system

A *substitution* is a mapping from a type variables to type expressions. For a substitution ρ , the *domain* (dom) is the set of variables given by $\{x \mid \rho x \neq x\}$ and the *range* is the set $\bigcup_{x \in dom(\rho)} \{\rho x\}$. Often, we consider the mappings as finite. In that case, a substitution maps type variables to themselves, if the type variables are not in the domain of the substitution. A substitution ρ is *idempotent* if $\rho \circ \rho = \rho$. An *empty* substitution is denoted by Id . Idempotency can also be characterized as the following condition: $dom(\rho) \cap FV(range(\rho)) = \emptyset$. Let ρ_1, ρ_2 be two substitutions then *composition of substitution*, $\rho_1 \circ \rho_2$, is given extensionally as:

$$\forall \alpha. (\rho_1 \circ \rho_2)(\alpha) = \rho_1(\rho_2(\alpha))$$

Substitution composition is associative but non-commutative. For example, consider substitutions ρ_1, ρ_2, ρ_3 then $(\rho_1 \circ \rho_2) \circ \rho_3 = \rho_1 \circ (\rho_2 \circ \rho_3)$ but $\rho_1 \circ \rho_2 \neq \rho_2 \circ \rho_1$. A type τ' is a *substitution instance* of a type τ if $\tau' = \rho\tau$. Substitution application to a type environment Γ , denoted by $\rho\Gamma$, is defined as:

$$\{x : \rho\tau \mid x : \tau \in \Gamma\}$$

In Wand’s system, constraints plays a central role. Specifically, Wand’s algorithm generates equality constraints between type expressions of the form $\tau_1 \stackrel{e}{=} \tau_2$. We call such a constraint an *e-constraint* and denote it by $\stackrel{e}{=}$. Two type expressions τ_1 and τ_2 are *unifiable* if there exists a substitution ρ such that $\rho\tau_1 = \rho\tau_2$. In such a case ρ is called a *unifier*. More formally, we denote solvability of a constraint by \models (read solve). We write $\rho \models \tau_1 \stackrel{e}{=} \tau_2$, if $\rho\tau_1 = \rho\tau_2$. We can generalize solvability to a set of constraints. Let e denote a particular e-constraint of the above form then we write $\rho \models E$ if and only if for every $e \in E$, $\rho \models e$. A type judgment in Wand’s system is given as $\Gamma, E \vdash M : \alpha$, where E denotes a constraint set. Sometimes we ignore the constraint component of the judgment to simplify the proofs. In that case, we denote a judgment by $\Gamma \vdash M : \tau$ wherein we have assumed that there is some substitution ρ , generated by solving E , such that $\rho\alpha = \tau$. Although Wand doesn’t cite any particular constraint solving algorithm, we include here (see Appendix A) a description of a unification algorithm based on the Robinson’s algorithm [Rob65]. This makes the paper complete and also gives clarity to our work.

4 Wand’s Algorithm

We discuss Wand’s algorithm and prove algorithm’s soundness and completeness with respect to Hindley-Milner type system.

4.1 Sketch of Wand’s Algorithm

Let G denote a set of goals and E a set of constraints given by the grammar \mathcal{C} above. Then we can sketch the algorithm as:

Input. A term M_0 of Λ .

Initialization. Set $E = \emptyset$ and $G = \{(T_0, M_0, \tau_0)\}$, where τ_0 is a fresh variable.

Loop Step. If $G = \emptyset$ then return E else choose a subgoal (T, M, τ) from G , delete the subgoal from G and add to E and G new verification conditions and subgoals generated by the action table given below.

Unify constraints. Unify constraints in E using the algorithm described in Appendix A.

The action table serves as a black box effectively separating the constraint solving from constraint generation. For untyped lambda calculus, the action table has the following semantics:

Case (Γ, x, τ) . Let $\tau_1 = \Gamma(x)$. Add $\tau \stackrel{e}{=} \tau_1$ to E .

Case (Γ, MN, τ) . Let α be a fresh type variable. Generate subgoals $(\Gamma, M, \alpha \rightarrow \tau)$ and (Γ, N, α) .

Case $(\Gamma, \lambda x.M, \tau)$. Let α_1 and α_2 be two fresh type variables. Generate equation $\tau \stackrel{e}{=} \alpha_1 \rightarrow \alpha_2$ and subgoal $(\Gamma_x \cup \{x : \alpha_1\}, M, \alpha_2)$.

We can now describe the soundness and completeness results as stated by Wand but before that we introduce some additional terminologies. If (Γ, M, τ) is a type expression assertion, we write $\rho \models (\Gamma, M, \tau)$ if and only if $\rho\Gamma \vdash M : \rho\tau$, and if G is a set of type-expression assertions, we write $\rho \models G$ if and only if $\rho \models g$ for each $g \in G$. Finally, we say ρ solves (E, G) if and only if $\rho \models E$ and $\rho \models G$.

(Soundness) $\forall \rho. \rho \models (E, G) \Rightarrow \rho\Gamma_0 \vdash M_0 : \rho\tau_0$

(Completeness) $\Gamma \vdash M_0 : \tau \Rightarrow (\exists \rho. \rho \models (E, G) \wedge \Gamma = \rho\Gamma_0 \wedge \tau = \rho\tau_0)$

The proof of the soundness invariant was never published by Wand whereas the proof of completeness invariant is proved by induction on the number of action steps. We have reformulated completeness and soundness to make it more abstract (by eliminating the goal set). This also help us state and prove the correctness of the extension in terms of the correctness of the Wand's algorithm.

Theorem 1 (Soundness and Completeness). *Wand's algorithm is sound and complete with respect to the Hindley-Milner type system.*

Proof. The proof is given by Theorem 2 (soundness) and Theorem 3 (completeness). □

Before we state and prove the soundness result we state some auxiliary lemmas.

Lemma 1. *If $\rho \models E_1 \cup E_2$, then $\rho \models E_1$ and $\rho \models E_2$.*

Lemma 2. *For any type environment Γ , substitution ρ , and $x \in \text{dom}(\Gamma)$, $(\rho\Gamma) \setminus x = \rho(\Gamma \setminus x)$.*

Proof. By induction on the structure of Γ .

Base case. Let $\Gamma = \emptyset$. We must show $(\rho\emptyset) \setminus x = \rho(\emptyset \setminus x)$. But that follows trivially.

Induction case. Assume $\Gamma' = \Gamma \cup \{y : \tau\}$ (for some y, τ) then we must show $(\rho\Gamma') \setminus x = \rho(\Gamma' \setminus x)$. We have two cases:

Case $y \neq x$. Then $(\rho\Gamma') \setminus x = \rho(\Gamma \cup \{y : \tau\}) \setminus x$. But, by Lemma 2, that is $(\rho\Gamma \cup \{y : \rho\tau\}) \setminus x = (\rho\Gamma \setminus x) \cup (\{y : \rho\tau\} \setminus x)$. By induction hypothesis, this becomes $\rho(\Gamma \setminus x) \cup (\{y : \rho\tau\} \setminus x)$. But this is same as $\rho((\Gamma \setminus x) \cup (\{y : \tau\} \setminus x))$ and that is nothing else but $\rho((\Gamma \cup \{y : \tau\}) \setminus x) = \rho(\Gamma' \setminus x)$ as desired.

Case $y = x$. Trivial. □

Theorem 2 (Soundness). *If there is a derivation of $\Gamma_0, E \vdash_W M_0 : \tau_0$ generating constraint set E , then, for any ρ such that $\rho \models E$, $\rho\Gamma_0 \vdash_{HM} M_0 : \rho\tau_0$ is derivable.*

Proof. Proof by induction on the structure of derivation. Let Δ_W be a derivation of $\Gamma_0, E \vdash_W M_0 : \tau_0$. We have three cases based on the typing rule used in the last derivation:

Case W-Var. Then M_0 is x for some x . Then there is a derivation of $\Gamma_0, E \vdash_W x : \tau_0$, where $E = \{\tau \stackrel{e}{=} \tau_0\}$ such that $x:\tau \in \Gamma_0$ for some τ . Let $\rho \models E$. Then $\rho = \{\tau_0 \mapsto \tau\}$. Since $x:\tau_0 \in \Gamma_0$ so $x : \rho\tau_0 \in \rho\Gamma_0$ and therefore there is a derivation of $\rho\Gamma_0 \vdash_{HM} x : \rho\tau_0$.

Case W-Abs. Then M_0 is of the form $\lambda x.M$ for some x, M and there is a derivation of $\Gamma_0, E \vdash_W \lambda x.M : \tau_0$ given by:

$$\frac{\frac{\vdots}{(\Gamma_0 \setminus x) \cup \{x : \beta\}, E' \vdash_W M : \tau_1}}{\Gamma_0, E' \cup \{\tau_0 \stackrel{e}{=} \beta \rightarrow \tau_1\} \vdash_W \lambda x.M : \tau_0}$$

Let $E = E' \cup \{\tau_0 \stackrel{e}{=} \beta \rightarrow \tau_1\}$. So, for some τ_1 and β , $(\tau_0 \stackrel{e}{=} (\beta \rightarrow \tau_1)) \in E$. Assume $\rho \models E$ but then, by Lemma 1, $\rho \models E'$ and $\rho \models \tau_0 \stackrel{e}{=} (\beta \rightarrow \tau_1)$. Then, by the induction hypothesis, there is a HM derivation of $\rho((\Gamma_0 \setminus x) \cup \{x : \beta\}) \vdash_{HM} M : \rho\tau_1$. But this is same as $(\rho(\Gamma_0 \setminus x) \cup \{x : \rho\beta\}) \vdash_{HM} M : \rho\tau_1$. By Lemma 2, $(\rho\Gamma_0 \setminus x) \cup \{x : \rho\beta\} \vdash_{HM} M : \rho\tau_1$ is derivable. Then we can construct the following derivation:

$$\frac{\frac{\vdots}{((\rho\Gamma_0) \setminus x) \cup \{x : \rho\beta\} \vdash_{HM} M : \rho\tau_1}}{\rho\Gamma_0 \vdash_{HM} \lambda x.M : \rho\beta \rightarrow \rho\tau_1}$$

But we already know $\rho \models \tau_0 \stackrel{e}{=} \beta \rightarrow \tau_1$ i.e. $\rho\tau_0 = \rho\beta \rightarrow \rho\tau_1$. Thus, $\rho\Gamma_0 \vdash_{HM} \lambda x.M : \rho\tau_0$ is derivable.

Case W-App. Then M_0 is of the form MN for some M and N . Then there is a derivation of $\Gamma_0, E \vdash_W MN : \tau_0$ for some E and is:

$$\frac{\frac{\frac{\vdots}{\Gamma_0, E_1 \vdash_W M : \alpha \rightarrow \tau_0}}{\Gamma_0, E_1 \cup E_2 \vdash_W MN : \tau_0} \quad \frac{\frac{\vdots}{\Gamma_0, E_2 \vdash_W N : \alpha}}{\Gamma_0, E_1 \cup E_2 \vdash_W MN : \tau_0} \quad (\alpha \text{ is fresh})$$

We must show that if, for any ρ such that $\rho \models E_1 \cup E_2$, then $\rho\Gamma_0 \vdash_{HM} MN : \rho\tau_0$ is derivable. Choose arbitrary ρ such that $\rho \models E_1 \cup E_2$. But then, by Lemma 1, $\rho \models E_1$ and $\rho \models E_2$. So, by the inductive hypothesis, both $\rho\Gamma_0 \vdash_{HM} M : \rho(\alpha \rightarrow \tau_0)$, and $\rho\Gamma_0 \vdash_{HM} N : \rho\alpha$ are derivable. But then, since $\rho(\alpha \rightarrow \tau_0) = \rho\alpha \rightarrow \rho\tau_0$, so

$$\frac{\frac{\frac{\vdots}{\rho\Gamma_0 \vdash_{HM} M : \tau_1 \rightarrow \rho\tau_0}}{\rho\Gamma_0 \vdash_{HM} MN : \rho\tau_0} \quad \frac{\frac{\vdots}{\rho\Gamma_0 \vdash_{HM} N : \tau_1}}{\rho\Gamma_0 \vdash_{HM} MN : \rho\tau_0} \quad (\text{for some } \tau_1 = \rho\alpha)$$

is a valid derivation in Hindley-Milner type system. □

Lemma 3. For any two type environments Γ_1 and Γ_2 , and for some substitution ρ , if $\rho\Gamma_1 = \Gamma_2$ and $FTV(\tau) \subseteq \text{dom}(\Gamma_2)$ then $\rho\tau = \tau$.

Lemma 4. If Δ_W proves $\Gamma, E \vdash_W MN : \tau$ and $\{\alpha_i\} \in FTV(\Delta) - \{FTV(\tau), FTV(\Gamma)\}$ where $\{\alpha_i\} \cap \{\beta_i\} = \emptyset$ then $\Delta[\alpha_i := \beta_i]$ is also a derivation of $\Gamma, E \vdash_W MN : \tau$.

Next, we prove the completeness result. The result is formally stated in the following theorem:

Theorem 3. (Completeness) *If Δ_{HM} is a derivation of $\Gamma \vdash_{HM} M_0 : \tau$, then for any ρ, τ_0 , and Γ_0 such that $dom(\rho) = (FTV(\Gamma_0) \cup \{\tau_0\})$, $\rho\Gamma_0 = \Gamma$, and $\rho\tau_0 = \tau$ then there exists a derivation Δ_w of $\Gamma_0, E \vdash_W M_0 : \tau_0$, and there exists a substitution ρ' such that $\rho \subseteq \rho'$ and $\rho' \models E$.*

Proof. The proof proceeds by induction on the last step used in the derivation Δ_{HM} . Assume Δ_{HM} is the derivation of $\Gamma \vdash_{HM} M_0 : \tau$. We have three cases:

Case HM-Var. Then M_0 is a variable x , for some x . Choose arbitrary ρ, τ_0, Γ_0 such that $dom(\rho) = FTV(\Gamma_0) \cup \{\tau_0\}$, $\rho\Gamma_0 = \Gamma$ and $\rho\tau_0 = \tau$. Then we must show: $\Gamma_0, E \vdash_W x : \tau_0$ is derivable and there exists a substitution ρ' such

that $\rho \subseteq \rho'$ and $\rho' \models E$. Then Δ_{HM} is $\frac{}{\Gamma \vdash_{HM} x : \tau}$ where $x : \tau \in \Gamma$.

But then we can construct Δ_W as $\frac{}{\rho\Gamma_0, \{\alpha \stackrel{e}{=} \tau_0\} \vdash_W x : \tau_0}$ and is a

derivation if $x : \alpha \in \rho\Gamma_0$ for some α . Let $\alpha = \rho\tau_0$ then since we know $\rho\Gamma_0 = \Gamma$, $\rho\tau_0 = \tau$ and $x : \tau \in \Gamma$, so $x : \rho\tau_0 \in \rho\Gamma_0$. Then $x : \alpha \in \rho\Gamma_0$. And $E = \{\tau_0 \stackrel{e}{=} \tau_\rho tau_0\}$. Then, let $\rho' = \rho$ and we need to show $\rho \models E$ i.e. $\rho(\rho\tau) = \rho\tau$. But that follows since ρ is idempotent. Thus $\rho \subseteq \rho'$ and $\rho' \models E$ as was to be shown.

Case HM-App. Then M_0 is of the form LN for some L and N . Then the derivation Δ_{HM} is of the following form:

$$\frac{\frac{\vdots}{\Gamma \vdash_{HM} L : \tau_j \rightarrow \tau} \quad \frac{\vdots}{\Gamma \vdash_{HM} N : \tau_j}}{\Gamma \vdash_{HM} LN : \tau} \text{ for some } \tau_j$$

Choose arbitrary ρ, β, Γ_0 such that $dom(\rho) = FTV(\Gamma_0) \cup \{\beta\}$, $\rho\Gamma_0 = \Gamma$ and $\rho\beta = \tau$. Then we must show $\Gamma_0, E_1 \cup E_2 \vdash_W LN : \beta$ is derivable, for some E_1, E_2 , and there exists a substitution ρ' such that $\rho \subseteq \rho'$ and $\rho' \models (E_1 \cup E_2)$. Since $dom(\rho) = FTV(\Gamma_0) \cup \{\beta\}$ so $\alpha \notin dom(\rho)$. Since $\Gamma \vdash_{HM} L : \tau_j \rightarrow \tau$ is derivable so, by the induction hypothesis, if we can show $(\rho \cup \{\alpha \mapsto \tau_j\})\Gamma_0 = \Gamma$ and $(\rho \cup \{\alpha \mapsto \tau_j\})(\alpha \rightarrow \beta) = \tau_j \rightarrow \tau$, then we can assume $\Gamma_0, E_1 \vdash_w L : \alpha \rightarrow \beta$ is derivable and there exists a ρ'_L such that $\rho_L \subseteq \rho'_L$ and $\rho'_L \models E_1$. Consider $(\rho \cup \{\alpha \mapsto \tau_j\})\Gamma_0$. But since α is fresh, so $\alpha \notin FTV(\Gamma_0)$ and therefore $(\rho \cup \{\alpha \mapsto \tau_j\})\Gamma_0 = \rho\Gamma_0$. Next, consider $\rho \cup \{\alpha \mapsto \tau_j\}(\alpha \rightarrow \beta)$. This is same as $\rho(\tau_j \rightarrow \beta) = \rho\tau_j \rightarrow \rho\beta$. By Lemma 3, $\rho\tau_j = \tau_j$, and since $\rho\beta = \tau$, therefore $\rho \cup \{\alpha \mapsto \tau_j\}(\alpha \rightarrow \beta) = \tau_j \rightarrow \tau$. So we can assume $\Gamma_0, E_1 \vdash_w L : \alpha \rightarrow \beta$ is derivable and there exists a ρ'_L such that $\rho_L \subseteq \rho'_L$ and $\rho'_L \models E_1$.

Similarly, since $\Gamma \vdash_{HM} N : \tau_j$ is derivable, so by the induction hypothesis, let Γ_0 be Γ_0 chosen earlier and τ_0 be α , and $\rho_L = \rho \cup \{\alpha \mapsto \tau_j\}$ so must show $\rho_N\Gamma_0 = \Gamma$ and $\rho_N\alpha = \tau_j$. But $\rho_N\Gamma_0 = \rho \cup \{\alpha \mapsto \tau_j\}(\Gamma_0)$ and since

α is fresh so we get $\rho\Gamma_0 = \Gamma$ which holds by our assumption. And $\rho_N\alpha = (\rho \cup \{\alpha \mapsto \tau_j\})\alpha = \rho\tau_j$ which by Lemma 3 is nothing else but τ_j . So we can assume $\Gamma_0, E_2 \vdash_w N : \alpha$ is derivable and there exists a ρ'_N such that $\rho_N \subseteq \rho'_N$ and $\rho'_N \models E_2$. Since $\Gamma_0, E_1 \vdash_w L : \alpha \rightarrow \beta$ is derivable and $\Gamma_0, E_2 \vdash_w N : \alpha$ is derivable, so $\Gamma_0, E_1 \cup E_2 \vdash_w LN : \beta$ is derivable and is as follows:

$$\frac{\frac{\vdots}{\Gamma_0, E_1 \vdash_w L : \alpha \rightarrow \beta} \quad \frac{\vdots}{\Gamma_0, E_2 \vdash_w N : \alpha}}{\Gamma_0, E_1 \cup E_2 \vdash_w LN : \beta} \text{ where } \tau_0 = \beta \text{ for some } \alpha$$

All that remains to show is $\exists \rho'. \rho \subseteq \rho'$ and $\rho' \models E_1 \cup E_2$. Let ρ' be $\rho'_L \cup \rho'_N$ then we must show that $\rho \subseteq \rho'_L \cup \rho'_N$ and $\rho'_L \cup \rho'_N \models E_1 \cup E_2$. Since $\rho_L \subseteq \rho'_L$ and $\rho_L = \rho \cup \{\alpha \mapsto \tau_j\}$ therefore $\rho \subseteq \rho'_L \cup \rho'_N$. We must show $\rho'_L \cup \rho'_N \models E_1 \cup E_2$ i.e. $(\rho'_L \cup \rho'_N \models E_1) \wedge (\rho'_L \cup \rho'_N \models E_2)$. But that follows from the induction hypothesis and Lemma 4.

Case HM-Abs. Then M_0 is of the form $\lambda x.M$ for some x and M . The derivation Δ_{HM} is of the following form:

$$\frac{\frac{\vdots}{(\Gamma \setminus x) \cup \{x : \tau_j\} \vdash_{HM} M : \tau_k}}{\Gamma \vdash_{HM} \lambda x.M : \tau_j \rightarrow \tau_k} \text{ where } \tau = \tau_j \rightarrow \tau_j \text{ for some } \tau_j, \tau_k$$

Choose arbitrary ρ, τ_0, Γ_0 such that $\rho\Gamma_0 = \Gamma$ and $\rho\tau_0 = \tau_j \rightarrow \tau_k$. Then we must show $\Gamma_0, E_1 \cup \{\tau_0 \stackrel{e}{=} \beta \rightarrow \gamma\} \vdash_w \lambda x.M : \tau_0$ is derivable and there exists a substitution ρ' such that $\rho \subseteq \rho'$ and $\rho' \models E_1 \cup \{\tau_0 \stackrel{e}{=} \beta \rightarrow \gamma\}$. Since there is a Hindley-Milner derivation of $(\Gamma \setminus x) \cup \{x : \tau_j\} \vdash_{HM} M : \tau_k$, by the induction hypothesis, we can assume if $\rho_M((\Gamma_0 \setminus x) \cup \{x : \beta\}) = (\Gamma \setminus x) \cup \{x : \tau_j\}$ and $\rho_M\gamma = \tau_k$ then $(\Gamma_0 \setminus x) \cup \{x : \beta\}, E_1 \vdash_w M : \gamma$ is derivable and $\exists \rho'. \rho_M \subseteq \rho' \wedge \rho' \models E_1$.

By Lemma 2, $\rho_M((\Gamma_0 \setminus x) \cup \{x : \beta\}) = (\rho_M\Gamma_0 \setminus x) \cup \{x : \rho_M\beta\}$. But since $\rho_M = \rho \cup \{\beta \mapsto \tau_j, \gamma \mapsto \tau_k\}$ so we have $(\rho \cup \{\beta \mapsto \tau_j, \gamma \mapsto \tau_k\})\Gamma_0 \setminus x \cup (\{x : \rho \cup \{\beta \mapsto \tau_j, \gamma \mapsto \tau_k\}\beta\})$. But since β, γ are fresh, so we have $\rho\Gamma_0 \setminus x \cup \{x : \tau_j\}$. And since $\rho\Gamma_0 = \Gamma$ so $\rho_M((\Gamma_0 \setminus x) \cup \{x : \beta\}) = (\Gamma \setminus x) \cup \{x : \tau_j\}$. Also, $\rho_M\gamma = (\rho \cup \{\beta \mapsto \tau_j, \gamma \mapsto \tau_k\})\gamma = \rho\tau_k$. But since all the variables in E_1 are fresh, so $\rho\tau_k = \tau_k$. Now we can assume there exists a derivation of $(\Gamma_0 \setminus x) \cup \{x : \beta\}, E_1 \vdash_w M : \gamma$ given as follows:

$$\frac{\frac{\vdots}{(\Gamma_0 \setminus x) \cup \{x : \beta\}, E_1 \vdash_w M : \gamma}}{\Gamma_0, E_1 \cup \{\tau_0 \stackrel{e}{=} \beta \rightarrow \gamma\} \vdash_w \lambda x.M : \tau_0}$$

and there exists a ρ'_M such that $\rho_M \subseteq \rho'_M$ and $\rho'_M \models E_1$. We must show there $\exists \rho'. \rho \subseteq \rho' \wedge \rho' \models E_1 \cup \{\tau_0 \stackrel{e}{=} \beta \rightarrow \gamma\}$. Let $\rho' = \rho'_M$. Consider the first goal but since $\rho_M \subseteq \rho'_M$ and $\rho_M = \rho \cup \{\beta \mapsto \tau_j, \gamma \mapsto \tau_k\}$ so $\rho \subseteq \rho'_M$ as desired. Also,

we already know that $\rho'_M \models E_1$ so we must show $\rho'_M \models \{\tau_0 \stackrel{e}{=} \beta \rightarrow \gamma\}$ i.e. we must show $\rho'_M \tau_0 = \rho'_M(\beta \rightarrow \gamma)$. Consider $\rho'_M \tau_0$, since $\rho \tau_0 = \tau_j \rightarrow \tau_k$ and $\rho \subseteq \rho_M \subseteq \rho'_M$ so by Lemma 5, $\rho'_M \tau_0 = \tau_j \rightarrow \tau_k$. Now consider $\rho'_M(\beta \rightarrow \gamma)$ but since $\rho_M \subseteq \rho'_M$ so $\rho'_M(\beta \rightarrow \gamma) = \rho_M \beta \rightarrow \rho_M \gamma$ but since ρ_M maps β to τ_j and γ to τ_k so $\rho'_M \models \{\tau_0 \stackrel{e}{=} \beta \rightarrow \gamma\}$. Thus $\rho'_M \models E_1 \cup \{\tau_0 = \beta \rightarrow \gamma\}$. \square

Lemma 5. *If ρ_1 and ρ_2 are two substitutions such that $\rho_1 \subseteq \rho_2$, then for any α such that $\alpha \in \text{dom}(\rho_1) \cap \text{dom}(\rho_2)$, $\rho_2 \alpha = \rho_1 \alpha$.*

5 Extended Wand's Algorithm

In this section we illustrate the extended algorithm and the extended proofs of completeness and soundness. But before that we enrich our term language and type language with additional constructs to handle the polymorphic let. The constraints are now enriched with two other forms of constraints and the type rules are extended to handle the let construct. We call the extended language Core-ML [MH88]. The terms in Core-ML are given by the following grammar:

$$\text{Core-ML} ::= x \mid MN \mid \lambda x.M \mid \mathbf{let} \ x = M \ \mathbf{in} \ N$$

The let expression $\mathbf{let} \ x = M \ \mathbf{in} \ N$ is not merely a syntactic sugar for $(\lambda x.N)M$. For instance, the term $\mathbf{let} \ i = \lambda x.x \ \mathbf{in} \ i \ i$ is typable but the desugared term $(\lambda i.i \ i)(\lambda x.x)$ is not typable. This is true for both Haskell and ML type reconstruction algorithms.

The type syntax is extended to include the polymorphic types given by type schemes and type scheme variables as shown below:

$$\sigma ::= \alpha^* \mid \forall \vec{\alpha}.\tau$$

A *type scheme*, denoted by $\forall \vec{\alpha}.\tau$, is a type where zero or more type variables are universally quantified. We denote a type scheme variable by annotating a type variable with a “*”. For example, α^* is a type scheme variable. We can extend our definition of free type variables for a type scheme, $\forall \vec{\alpha}.\tau$, as $\text{FTV}(\forall \vec{\alpha}.\tau) = \text{FTV}(\tau) - \vec{\alpha}$. *Generalizing* a type τ with respect to a type environment Γ entails quantifying over the free variables of τ that are not free in Γ : $\text{gen}(\Gamma, \tau) \stackrel{\text{def}}{=} \forall \vec{\alpha}.\tau$ where $\vec{\alpha} = \text{FTV}(\tau) - \text{FTV}(\Gamma)$. On the other hand, *instantiation* of a type scheme involves replacing the quantified variables by fresh type variables: $\text{inst}(\forall \vec{\alpha}.\tau) \stackrel{\text{def}}{=} \tau[\alpha_1 := \beta_1, \dots, \alpha_n := \beta_n]$ where β_1, \dots, β_n are fresh variables.

To accommodate the let construct, we extend the constraint language \mathcal{C} to include two other kinds of constraints as shown below:

$$\mathcal{C} ::= \tau_1 \stackrel{e}{=} \tau_2 \quad | \quad \alpha \stackrel{s}{=}_{\Gamma} \alpha^* \quad | \quad \alpha^* \stackrel{i}{=} \tau$$

(e-constraint) (s-constraint) (i-constraint)

A *s-constraint* between a type expression and a type scheme variable, $\tau \stackrel{s}{=}_{\Gamma} \alpha^*$, expresses the fact that α^* denotes a type scheme obtained by generalizing a type denoted by τ with respect to the environment Γ . The *i-constraint* between a type

scheme variable and a type expression, α and τ respectively, expresses the fact that τ is constrained to be the instantiated value of the type scheme denoted by α^* .

The type rules for untyped lambda terms are now extended with two rules to handle the let construct. The new type system is called *extended Wand's system* and a judgment in the extended system is denoted by the subscript W^+ .

$$\text{(W-Var-i)} \frac{}{\Gamma, \{\alpha^* \stackrel{i}{=} \tau\} \vdash_{W^+} x : \alpha^*} \text{ where } x : \tau \in \Gamma$$

$$\text{(W-Let)} \frac{\Gamma, E_1 \vdash_{W^+} M : \alpha_1 \quad (\Gamma \setminus x) \cup \{x : \alpha_2^*\}, E_2 \vdash_{W^+} N : \tau}{\Gamma, E_1 \cup E_2 \cup \{\alpha_1 \stackrel{s}{=} \alpha_2^*\} \vdash_{W^+} \mathbf{let } x = M \mathbf{ in } N : \tau} \text{ where } \alpha_1, \alpha_2^* \text{ are fresh}$$

5.1 Extended Algorithm sketch

Apart from extending the type rules, we also had to extend the notion of satisfiability and substitution application to a constraint. First, we describe some notations used in the description of satisfiability. We use the notation E_{α^*} to denote a set of i-constraints related⁷ to a s-constraint $\tau_0 \stackrel{s}{=} \alpha^*$. From this point onwards, we think of a s-constraint and related i-constraint as a pair $(\tau_0 \stackrel{s}{=} \alpha^*, E_{\alpha^*})$; the first component being the s-constraint and the second component being the list of related i-constraint(s). We use the symbol \leq to express the notion of an instance. Specifically, $\tau \leq \sigma$ expresses the fact that τ is an instance of σ in the sense that τ is obtained by instantiating all the bound variables of σ . This notion can then be used to express the satisfiability of *i&s* constraints. We say ρ satisfies $(\tau_0 \stackrel{s}{=} \alpha^*, E_{\alpha^*})$ if $\forall (\alpha^* \stackrel{i}{=} \tau_1) \in E_{\alpha^*}. \rho \tau_1 \leq \rho(\text{gen}(\Gamma, \tau_0))$. Substitution application to a pair of s-constraint and related i-constraints is defined as: $\rho(\tau_0 \stackrel{s}{=} \alpha^*, E_{\alpha^*}) \stackrel{\text{def}}{=} (\rho \tau_0 \stackrel{s}{=} \alpha^*, \{\rho \tau \stackrel{i}{=} \alpha^* \mid (\tau \stackrel{i}{=} \alpha^*) \in E_{\alpha^*}\})$.

Next, we sketch the algorithm for the extended language. Let G denote a set of goals and E a list⁸ given by the grammar \mathcal{C} above.

Input. A term M_0 of Λ .

Initialization. Set $E = \emptyset$ and $G = \{(\Gamma_0, M_0, t_0)\}$.

Loop Step. If $G = \emptyset$ then return E else choose a subgoal (Γ, M, t) from G , delete the subgoal from G and add to E and G new verification conditions and subgoals generated by the action table given below.

Unify constraints. Unify constraints in E using the multi-phase unification algorithm described below.

⁷ A s-constraint is *related* to an i-constraint if they share the same type scheme variable.

⁸ This is needed to preserve the order of s-constraints.

The behavior of action table for Core-ML is same as that for untyped lambda calculus except for the variable (a slight modification) and the let case as shown below:

- Case** (Γ, x, τ_0) . If x is bound to a type scheme variable α^* in Γ , *i.e.* $x : \alpha^* \in \Gamma$, then add $\alpha^* \stackrel{i}{=} \tau_0$ to E else add $\tau_0 \stackrel{e}{=} \tau_1$ (where $x : \tau_1 \in \Gamma$) to E .
- Case** (Γ, MN, τ_0) . Let α be a fresh type variable. Generate subgoals $(\Gamma, M, \alpha \rightarrow \tau_0)$ and (Γ, N, α) , and add to G .
- Case** $(\Gamma, \lambda x.M, \tau_0)$. Let α and β be two fresh type variables. Generate equation $\tau_0 \stackrel{e}{=} \alpha \rightarrow \beta$ and sub-goal $((\Gamma \setminus x) \cup \{x : \alpha\}, M, \beta)$, and add to E & G respectively.
- Case** $(\Gamma, \text{let } x = M \text{ in } N, \tau_0)$. Let α_1, α_2^* be fresh type variables. Append⁹ $E_l @ [\alpha_1 \stackrel{s}{=} \Gamma \alpha_2^*] @ E_r$ to list E , where E_l, E_r are obtained by recursively calling the extended algorithm on (Γ, M, α_1) and $((\Gamma \setminus x) \cup \{x : \alpha_2^*\}, N, \tau_0)$ respectively.

The next few paragraphs highlight the constraint solving phase. This phase consists of two distinct unification phases: Phase I and Phase II. We first give an informal description of both the phases and follow it with a formal description. In the first phase, e-constraints are unified. Note that if there are no i&s-constraints, *i.e.* the term is a pure lambda term, then our Phase I mirrors the constraint solving phase for Wand's algorithm. In the second phase, a s-constraint and related i-constraints are chosen and transformed to e-constraints and unified using the Phase I unification. Let $E = E_e @ E_{i\&s}$ be the constraint list obtained from the constraint generation phase, where E_e denotes a list containing e-constraints, $E_{i\&s}$ denotes a list containing i&s-constraints. The constraint solving algorithm, *SOLVE*, integrates the two unification phases as follows:

$$\begin{aligned} \text{SOLVE}(E) = & \\ & \rho := Id \\ & \text{let } \rho_1 = \text{unify}_1(E_e, \rho) \text{ in} \\ & \rho_1 \circ (\text{unify}_2 E_{i\&s}) \end{aligned}$$

The first phase, unify_1 , is defined as:

$$\begin{aligned} \text{unify}_1(E, \rho) = & \\ \text{unify}_1((\alpha \stackrel{e}{=} \beta) :: E) \rho & = \text{if } \alpha = \beta \text{ then } \text{unify}_1 E \rho \\ \text{unify}_1((\alpha \stackrel{e}{=} \tau) :: E) \rho & = \text{if } \alpha \text{ occurs in } \tau \text{ then raise Failure} \\ \text{unify}_1((\alpha \stackrel{e}{=} \tau) :: E) \rho & = \text{unify}_1(E[\alpha := \tau])(\rho \circ [(\alpha \mapsto \tau)]) \\ \text{unify}_1((\tau \stackrel{e}{=} \alpha) :: E) \rho & = \text{unify}_1((\alpha \stackrel{e}{=} \tau) :: E) \rho \\ \text{unify}_1((\tau_1 \rightarrow \tau_2 \stackrel{e}{=} \tau_3 \rightarrow \tau_4) :: E) \rho & = \text{unify}_1((\tau_1 \stackrel{e}{=} \tau_3) :: ((\tau_2 \stackrel{e}{=} \tau_4) :: E)) \rho \end{aligned}$$

We assume that s-constraint and corresponding i-constraints are grouped together while preserving the order of s-constraints. We use the notation $[\tau_0]_{\alpha^*}$ to denote the set of i-constraints of the form $\alpha^* \stackrel{i}{=} \tau_1$ for a s-constraint of the form $\tau_0 \stackrel{s}{=} \Gamma \alpha^*$. unify_2 is defined as:

⁹ This will ensure that we solve the leftmost innermost let first.

$$\begin{aligned}
unify_2 (E'@E) &= \text{let } \rho_1 = unify_1 E'' \\
&\quad \text{in } \rho_1 \circ unify_2 (\rho_1 E) \\
&\quad \text{where } E'' = \{inst(gen(\Gamma, \tau_1)) \stackrel{e}{=} \tau_2 \mid (\alpha^* \stackrel{i}{=} \tau_2) \in E_{\alpha^*}\} \\
&\quad \text{and } E' = [(\tau_1 \stackrel{s}{=} \Gamma \alpha^*), E_{\alpha^*}] \\
unify_2 [] &= Id
\end{aligned}$$

Phase I gets rid of all the e-constraints. Phase II gets rid of s-constraints and i-constraints to form a resulting substitution. The entire algorithm is summarized in Fig. 3. The curve in the figure delineates our extension with the original Wand's algorithm.

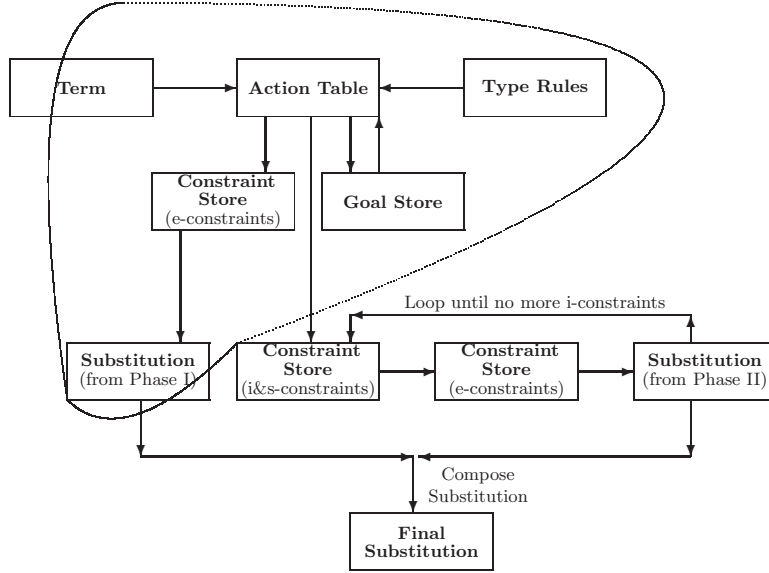


Fig. 3. An overview of the extended algorithm; original Wand's algorithm is shown by the curved line.

Finally we analyze the running time of the algorithm. It is known that the unification algorithm can be made to run in linear time [PW76], [BS01] in the size

of the input if the inputs and outputs are coded as dags; it becomes exponential if the inputs and outputs are coded as strings since a dag of size n can encode a string of $2^{O(n)}$. The typability problem for pure lambda terms is PTIME-complete [Mit96], [Tys88]. But when the *let* operator is added to the language the typability problem becomes PSPACE-hard [PJ89] and DEXPTIME-complete [KTU90]. Therefore, the complexity the first phase of our algorithm is essentially the complexity of this algorithm and hence is double exponential in the size of the input in the worst case. The second phase of the algorithm involves solving i&s constraints and since they are linear in the size of the input and for a given s-constraint there can be only $O(n)$ i-constraints. Therefore, the total time required by the algorithm is double exponential in the size of the input in the worst case.

6 Extension's Correctness

In this section, we prove the various lemmas and theorems needed to prove the correctness of our extension. But first we define the two reductions, denoted by \rightarrow_{let} and \rightarrow_{β} respectively, as:

$$\begin{array}{ll} (beta) & (\lambda x.N)M \rightarrow_{\beta} N[x := M] \\ (let) & \mathbf{let } x = M \mathbf{ in } N \rightarrow_{let} (\lambda x.N)M \text{ if } x \text{ is monomorphic.} \end{array}$$

We need two functions for the correctness proof. The function *morph* changes a polymorphic let construct into a monomorphic let and is defined as:

$$\begin{array}{ll} morph(x) & = x \\ morph(\lambda x.M) & = \lambda x. morph(M) \\ morph(M N) & = morph(M) morph(N) \\ morph(\mathbf{let } x = M \mathbf{ in } N[x]) & = \mathbf{let } N_1 = morph(N) \mathbf{ in} \\ & \quad \mathbf{let } x = M \mathbf{ in } morph(N_1[x]) \quad \text{if } |FV(N)|_x \leq 1 \\ morph(\mathbf{let } x = M \mathbf{ in } N[x]) & = \mathbf{let } N_1 = morph(N) \mathbf{ in} \\ & \quad \mathbf{let } x_1 = M \mathbf{ in } morph(\mathbf{let } x = M \mathbf{ in } N_1[x_1]) \quad \text{if } |FV(N)|_x > 1 \end{array}$$

We also need another function *transform* to transform monomorphic lets to lambda expressions:

$$\begin{array}{ll} transform(x) & = x \\ transform(\lambda x. M) & = \lambda x. transform(M) \\ transform(M N) & = transform(M) transform(N) \\ transform(\mathbf{let } x = M \mathbf{ in } N) & = (\lambda x. transform(N))transform(M) \text{ if } |FV(N)|_x \leq 1. \\ transform(\mathbf{let } x = M \mathbf{ in } N) & = \mathbf{let } x = M \mathbf{ in } transform(N) \quad \text{if } |FV(N)|_x > 1. \end{array}$$

Note that in our CiE paper [KC08] the transformation is given by a single function $ptol$, and is defined as:

$$\begin{aligned}
ptol(x) &= x \\
ptol(\lambda x.M) &= \lambda x. ptol(M) \\
ptol(MN) &= (ptol(M) ptol(N)) \\
ptol(\mathbf{let} x = M \mathbf{in} N[x]) &= \mathbf{let} N_1 = ptol(N) \mathbf{in} \quad \text{if } |FV(N)|_x \leq 1 \\
&\quad (\lambda x. N_1[x])M \\
ptol(\mathbf{let} x = M \mathbf{in} N[x]) &= \mathbf{let} N_1 = ptol(N) \mathbf{in} \quad \text{if } |FV(N)|_x > 1 \\
&\quad ptol(\mathbf{let} x_1 = M \mathbf{in} \mathbf{let} x = M \mathbf{in} N_1[x_1]) \\
&\quad \text{where } x_1 \text{ is a fresh variable.}
\end{aligned}$$

The three functions are related by the following relation:

$$ptol = transform \circ morph$$

The transformation of a let-term to a pure lambda term is a type and value preserving transformation. We illustrate this characteristic of our transformation with the help of an example. Consider the term $\mathbf{let} y = \lambda x.x \mathbf{in} y y$, taken from Milner [DM82]. This term is typable and has the principal type scheme $\forall \alpha. \alpha \rightarrow \alpha$. If we desugar a let-term $\mathbf{let} x = M \mathbf{in} N$ as $(\lambda x.N)M$ then desugaring of the term $\mathbf{let} y = \lambda x.x \mathbf{in} y y$ results in $(\lambda y.y y)(\lambda x.x)$, which is not typable¹⁰. However, our transformation, although similar to the desugaring process mentioned above, preserves typability. We illustrate this in a step-by-step process. First we use the *morph* function and then follow it with an application of *transform* function.

The morphed term, *i.e.* the term with monomorphic lets, is given as:

$$\begin{aligned}
&\mathbf{let} y_1 = \lambda x.x \mathbf{in} \\
&\quad \mathbf{let} y = \lambda x.x \mathbf{in} \\
&\quad \quad y y_1
\end{aligned}$$

The let-free term, obtained by transforming the monomorphic terms to pure lambda terms, is typable¹¹. Here is a computation, starting from the term above showing that this term is the same as $yy [y := (\lambda x.x)]$ (as it should be.)

$$\begin{aligned}
&\rightarrow_{let} (\lambda y_1. (\mathbf{let} y = \lambda x.x \mathbf{in} y y_1))(\lambda x.x) \\
&\rightarrow_{let} (\lambda y_1. (\lambda y.y y_1)(\lambda x.x))(\lambda x.x) \\
&\rightarrow_{\beta} ((\lambda y.y (\lambda x.x))(\lambda x.x)) \\
&\rightarrow_{\beta} (\lambda x.x)(\lambda x.x) \\
&= yy [y := (\lambda x.x)]
\end{aligned}$$

Now we can consider a more complicated example taken from [Mai89] involving multiple nested lets.

¹⁰ An implementation of Alg. W fails with a unification error

¹¹ We ran a prototype implementation of Alg. W [Mil78] on the term and the proof itself is easy to find.

```

let x1 = λy.λz.(z y) y in
  let x2 = λz.x1 (x1 z) in
    let x3 = λz.x2 (x2 z) in
      let x4 = λz.x3 (x3 z) in
        x4 (λz.z)

```

Application of the *morph* function will change the above term to:

```

let x11 = λy.λz.(z y) y in
  let x12 = λy.λz.(z y) y in
    let x13 = λy.λz.(z y) y in
      let x14 = λy.λz.(z y) y in
        let x15 = λy.λz.(z y) y in
          let x16 = λy.λz.(z y) y in
            let x17 = λy.λz.(z y) y in
              let x1 = λy.λz.(z y) y in
                let x21 = λz.x11 (x12 z) in
                  let x22 = λz.x13 (x14 z) in
                    let x23 = λz.x15 (x16 z) in
                      let x2 = λz.x17 (x1 z) in
                        let x31 = λz.x21 (x22 z) in
                          let x3 = λz.x23 (x2 z) in
                            let x4 = λz.x31 (x3 z) in
                              x4 (λz.z)

```

Finally, applying *transform* changes the above term to:

```

((λx11.((λx12.((λx13.((λx14.((λx15.((λx16.((λx17.((λx1.((λx21.((λ
x22.((λx23.((λx2.((λx31.((λx3.((λx4.(x4 (λz .z))) (λ z.(x31 (x3 z))))))
(λz.(x23 (x2 z)))))) (λz.(x21 (x22 z)))))) (λz.(x17 (x1 z)))))) (λz.(x15
(x16 z)))))) (λz.(x13 (x14 z)))))) (λz.(x11 (x12 z)))))) (λy.(λz.((z
y) y)))) (λy.(λz.((z y) y)))) (λy.(λz.((z y) y)))) (λy.(λz.((z
y) y)))) (λy.(λz.((z y) y)))) (λy.(λz.((z y) y)))) (λy.(λz.((z
y) y)))) (λy.(λz.((z y) y))))

```

We state and prove several lemmas and theorems for the correctness. We need one additional concept, *i.e.* a *context*. A context C is an expression with one sub-expression replaced by a hole denoted by \square . The expression $C[e]$ denotes an expression resulting from placing an expression e in the hole of C . The context is specified by the following grammar:

$$C ::= \square \mid C e \mid e C \mid \mathbf{let} x = C \mathbf{in} N \mid \mathbf{let} x = M \mathbf{in} C \mid \lambda x.C$$

We can use the above notion to make explicit an occurrence of a polymorphic-let bound variable. Given a polymorphic let term $\mathbf{let} x = M \mathbf{in} N$ we denote a context by the term N and one hole in the context by $N[x]$, meaning one occurrence of the polymorphic let variable. So, for a polymorphic-let term $\mathbf{let} x = M \mathbf{in} N \equiv \mathbf{let} x = M \mathbf{in} N[x]$.

The following theorem is adapted to our paper and is helpful in the proofs of soundness and completeness.

Theorem 4 (Subject Reduction). *If $\Gamma \vdash M_0 : \tau$ is derivable and $M_0 \longrightarrow M'_0$ then $\Gamma \vdash M'_0 : \tau$ is derivable, where \longrightarrow is either a β or a let reduction.*

Proof. By case analysis on the reduction $M_0 \longrightarrow M'_0$. Assume $\Gamma \vdash M : \tau$ is derivable and assume $M_0 \longrightarrow M'_0$. We have two cases:

Case $M_0 \equiv (\lambda x.N)M$. Then we know $(\lambda x.N)M \Rightarrow_{\beta} N[x := M]$. Since $\Gamma \vdash (\lambda x.N)M : \tau$ is derivable, therefore, $\Gamma \vdash \lambda x.N : \tau' \rightarrow \tau$ and $\Gamma \vdash M : \tau'$ is derivable for some τ' . From the former, we have $(\Gamma \setminus x) \cup \{x : \tau'\} \vdash N : \tau$ is derivable. Then by Lemma 6, $\Gamma \vdash N[x := M] : \tau$ is derivable.

Case $M_0 \equiv \mathbf{let} \ x = M \ \mathbf{in} \ N$. Note that this reduction holds only for $W+$. We assume x is monomorphic in N and so $\mathbf{let} \ x = M \ \mathbf{in} \ N \Rightarrow_{\mathbf{let}} (\lambda x.N)M$. Since $\Gamma \vdash \mathbf{let} \ x = M \ \mathbf{in} \ N : \tau$ is derivable, therefore, there is a derivation of $\Gamma \vdash M : \tau'$ for some τ' and there is a derivation of $\Gamma' \vdash N : \tau$ for some Γ' . Since x is monomorphic, therefore either x occurs only once in N or does not occur at all. So $\Gamma' = (\Gamma \setminus x) \cup \{x : \tau'\}$. Then $(\Gamma \setminus x) \cup \{x : \tau'\} \vdash N : \tau$ is derivable. But then $\Gamma \vdash (\lambda x.N) : \tau' \rightarrow \tau$ is derivable. And so $\Gamma \vdash (\lambda x.N)M : \tau$ is derivable. □

Lemma 6 (Preservation of Types Under Substitution). *If $\Gamma \cup \{x : \tau'\} \vdash N : \tau$ is derivable and $x \notin \text{dom}(\Gamma)$ and $\Gamma \vdash M : \tau'$, then $\Gamma \vdash N[x := M] : \tau$ is derivable.*

Proof. See Pierce's book (page 106) for proof.

The following theorem allows replacing a sub-expression of a typable expression with another sub-expression of the same type, without changing the type of the expression.

Theorem 5 (Replacement). *If*

1. Δ is a derivation concluding $\Gamma \vdash C[t] : \tau$,
2. Δ_1 is a sub-derivation of Δ concluding $\Gamma' \vdash t : \tau'$,
3. Δ_1 occurs in Δ in the position corresponding to the hole ($[]$) in C , and
4. $\Gamma' \vdash t' : \tau'$ is derivable.

Then $\Gamma \vdash C[t'] : \tau$. Note that this holds for untyped lambda calculus and Core-ML terms.

Proof. Proof by induction on the height of the derivation tree. Refer [HS86] page 181 and [WF94] for proof. □

Theorem 6. *The extended algorithm is sound and complete with respect to Wand's system.*

Proof. The proof involves two separate proofs of soundness and completeness as given by Theorem 7 and Theorem 8. □

Lemma 7 (Polymorphic-Let elimination for a let term). *Let t be the term $\mathbf{let } x = M \mathbf{ in } N[x]$ for some x, M , and N and let t' be the term $\mathbf{let } x_1 = M \mathbf{ in } (\mathbf{let } x = M \mathbf{ in } N[x_1])$ (where x_1 is a fresh variable and N' is obtained from N by replacing one occurrence of x in N by x_1). Then if $\Gamma \vdash_{W^+} t : \tau$ is derivable then $\Gamma \vdash_{W^+} t' : \tau$ is also derivable.*

Proof. Assume $\Gamma \vdash_W t : \tau$ is derivable. Since $\mathbf{let } x = M \mathbf{ in } N[x] \xrightarrow{\text{let}} N[x][x := M] = N[M][x := M]$. And $\mathbf{let } x_1 = M \mathbf{ in } (\mathbf{let } x = M \mathbf{ in } N[x_1]) \xrightarrow{\text{let}} (\mathbf{let } x = M \mathbf{ in } N[x_1])[x_1 := M] = (\mathbf{let } x = M \mathbf{ in } N[M]) \xrightarrow{\text{let}} N[M][x := M]$. Then by the subject reduction theorem $\Gamma \vdash_{W^+} t' : \tau$ is derivable.

Lemma 8 (Morph preserves typability). *Let t be a Core-ML term such that if $\Gamma \vdash_{W^+} t : \tau$ is derivable, then $\Gamma \vdash_{W^+} \text{morph}(t) : \tau$ is derivable.*

Proof. By complete induction on the number of occurrences of polymorphic-let variables in t (denote this number as $|t|$). Choose arbitrary $t \in \text{Core-ML}$ and assume $\Gamma \vdash_{W^+} t : \tau$ is derivable. Our induction hypothesis says $\forall t' : \Lambda. |t'| < |t| \Rightarrow (\Gamma \vdash_{W^+} t' : \tau \Rightarrow \Gamma \vdash_{W^+} \text{morph}(t') : \tau)$. We have two cases:

Case $|t| = 0$. Then $\text{morph}(t) = t$ so the Lemma holds.

Case $|t| \neq 0$. Then t is of the form $C[\mathbf{let } x = M \mathbf{ in } N[x]] : \tau$, where C is some context and the let is the uppermost leftmost occurrence of a polymorphic let. By the Theorem 5 and Lemma 7 we know $\Gamma \vdash_{W^+} t_1 : \tau$, where $t_1 = C[\mathbf{let } x_1 = M \mathbf{ in } \mathbf{let } x = M \mathbf{ in } N[x_1]]$. But $|t_1| < |t|$ so the induction hypothesis gives us that $\Gamma \vdash_{W^+} \text{morph}(t_1) : \tau$ is derivable. But, by Lemma 9 and Lemma 10, $\text{morph}(t_1) =_\alpha \text{morph}(t)$ and so $\Gamma \vdash_{W^+} \text{morph}(t) : \tau$ is derivable.

Lemma 9 (Alpha equivalence preserves derivation). *Let M and M' be Core-ML term such that $M =_\alpha M'$, then $\Gamma \vdash_{W^+} M : \tau$ is derivable if and only if $\Gamma \vdash_{W^+} M' : \tau$ is derivable.*

Lemma 10 (Morph Equivalence). *Given a term M_0 such that $M_0 = C[\mathbf{let } x = M \mathbf{ in } N[x]]$, where C is a context where the let is uppermost leftmost such occurrence in M_0 then $\text{morph}(C[\mathbf{let } x = M \mathbf{ in } N[x]]) = \text{morph}(C[\mathbf{let } x_1 = M \mathbf{ in } \mathbf{let } x = M \mathbf{ in } N[x_1]])$.*

Lemma 11 (Transform preserves typing). *Given a term M without polymorphic lets then $\Gamma \vdash_{W^+} M : \tau$ is derivable if and only if $\Gamma \vdash_{W^+} \text{transform}(M) : \tau$ is derivable.*

Corollary 1 (Transform-Morph Preserves Typing). *Let M be a Core-ML term such that if $\Gamma \vdash_{W^+} \text{morph}(M) : \tau$ is derivable, then $\Gamma \vdash_{W^+} \text{transform}(\text{morph}(M)) : \tau$ is also derivable.*

Proof. Follows directly from Lemma 8 and Lemma 11.

Corollary 2. *Let M be a Core-ML term such that if $\Gamma \vdash_{W^+} M : \tau$ is derivable, then $\Gamma \vdash_{W^+} \text{transform}(\text{morph}(M)) : \tau$ is also derivable.*

Proof. Follows directly from Lemma 8, Lemma 11 and Corollary 1.

Lemma 12 (Lambda Terms Equivalence). *If M is a Core-ML term then $\Gamma \vdash_{W^+} \text{transform}(\text{morph}(M)) : \tau$ is derivable if and only if $\Gamma \vdash_W \text{transform}(\text{morph}(M)) : \tau$ is derivable.*

Theorem 7 (Soundness). *Let M be a term in Core-ML. Then if $\Gamma \vdash_{W^+} M : \tau$ is derivable then $\Gamma \vdash_{HM} \text{transform}(\text{morph}(M)) : \tau$ is also derivable.*

Proof. Assume $\Gamma \vdash_{W^+} M : \tau$ is derivable then, by Corollary 2, $\Gamma \vdash_{W^+} \text{transform}(\text{morph}(M)) : \tau$ is derivable. Then by lemma 12 $\Gamma \vdash_w \text{transform}(\text{morph}(M)) : \tau$ is derivable. But then we have already proved the soundness of Wand's system to Hindley-Milner type system. So the extended system is sound w.r.t. HM type system. \square

We now show that the extended algorithm is complete with respect to the Hindley-Milner system but before that we need another Lemma.

Lemma 13. *Let M_0 be a Core-ML term such that if $\Gamma_0 \vdash_{W^+} \text{morph}(M_0) : \tau_0$ is derivable then $\Gamma_0 \vdash_{W^+} M_0 : \tau_0$ is also derivable.*

Proof. The proof is by induction on the number of calls to *morph* (denote this number as $|M_0|$). Choose arbitrary M_0 and assume $\Gamma_0 \vdash_{W^+} \text{morph}(M_0) : \tau_0$ is derivable. Our induction hypothesis is $\forall M_1. \forall \Gamma_1 \forall \tau : A. |M_1| < |M_0| \Rightarrow (\Gamma_1 \vdash_{W^+} \text{morph}(M_1) : \tau_1 \Rightarrow \Gamma_1 \vdash_{W^+} M_1 : \tau_1)$. We have four cases:

M_0 **is x .** Then $\Gamma \vdash_{W^+} \text{morph}(x) : \tau_0$ is derivable. Since $\text{morph}(x) = x$ and so the lemma holds.

M_0 **is $\lambda x.M$ for some x and M .** Then $\Gamma \vdash_{W^+} \text{morph}(\lambda x.M) : \tau_0$ is derivable. But since $\text{morph}(\lambda x.M) = \lambda x.\text{morph}(M)$, so by Lemma 9, $\Gamma \vdash_{W^+} \lambda x.\text{morph}(M) : \tau_0$ is derivable. Then by the induction hypothesis, $\Gamma' \vdash_{W^+} M : \tau_1$ is derivable and therefore $\Gamma \vdash_{W^+} \lambda x.M : \tau_2 \rightarrow \tau_1$ is derivable.

M_0 **is MN for some M, N .** Assume there is a derivation of $\Gamma \vdash_{W^+} \text{morph}(MN) : \tau_0$. But $\text{morph}(MN) = \text{morph}(M)\text{morph}(N)$. So $\Gamma \vdash_w \text{morph}(M)\text{morph}(N) : \tau_0$ is derivable. By the induction hypothesis, $\Gamma \vdash_{W^+} M : \tau_1 \rightarrow \tau_0$ and $\Gamma \vdash_{W^+} N : \tau_1$ are derivable and therefore $\Gamma \vdash_{W^+} MN : \tau_0$ is derivable as desired.

M_0 **is $\text{let } x = M \text{ in } N$, for some M, N .** Assume there is a derivation of $\Gamma \vdash_w \text{morph}(\text{let } x = M \text{ in } N)$. We have two cases:

Case x is monomorphic. Then $\text{morph}(\text{let } x = M; \text{ in } N) = \text{let } x = M \text{ in } \text{morph}(N)$ and so by Lemma 9 $\Gamma \vdash_{W^+} \text{let } x = M \text{ in } \text{morph}(N) : \tau$ is derivable. That means $\Gamma_1 \vdash_W M : \tau_1$ and $\Gamma_2 \vdash_W \text{morph}(M) : \tau_2$ is derivable for some $\Gamma_1, \Gamma_2, \tau_1, \tau_2$. By the induction hypothesis $\Gamma_2 \vdash_W N : \tau_2$ is derivable and so we can construct a derivation of $\Gamma \vdash_{W^+} \text{let } x = M \text{ in } N : \tau_0$ is derivable.

Case x is polymorphic. Since there is a derivation of $\Gamma \vdash_{W^+} \text{morph}(\text{let } x = M \text{ in } N[x]) : \tau_0$ and since $\text{morph}(\text{let } x = M \text{ in } N[x]) = \text{let } x_1 = M \text{ in } \text{morph}(\text{let } x = M \text{ in } N[x_1])$. So, by Lemma 9, $\Gamma \vdash_{W^+} \text{let } x_1 =$

M in $\text{morph}(\text{let } x = M \text{ in } N[x_1])$ is derivable. Then by the induction hypothesis there exists some Γ_1, τ_1 such that $\Gamma_1 \vdash_{W^+} \text{let } x = M \text{ in } N[x_1] : \tau_1$ is derivable. And so we can construct a derivation of $\Gamma \vdash_{W^+} \text{let } x_1 = M \text{ in let } x = M \text{ in } N[x_1] : \tau_0$ but since $\text{let } x = M \text{ in } N[x] =_{\alpha} \text{let } x_1 = M \text{ let } x = M \text{ in } N[x]$ therefore, by Lemma 9, $\Gamma \vdash_{W^+} \text{let } x = M \text{ in } N[x] : \tau_0$ is derivable. \square

Theorem 8 (Completeness). *Let M be a Core-ML term then if $\Gamma \vdash_{HM} \text{transform}(\text{morph}(M)) : \tau$ is derivable then $\Gamma \vdash_{W^+} M : \tau$ is derivable.*

Proof. Assume $\Gamma \vdash_{HM} \text{transform}(\text{morph}(M)) : \tau$ is derivable. Assume $\Gamma \vdash_{W^+} \text{transform}(\text{morph}(M)) : \tau$ is derivable. By Lemma 12 $\Gamma \vdash_{W^+} \text{transform}(\text{morph}(M)) : \tau$ is derivable. Then, by Lemma 11 $\Gamma \vdash_{W^+} \text{morph}(M) : \tau$ is derivable. But then $\Gamma \vdash_{W^+} M : \tau$ is derivable by Lemma 13. \square

Now we describe the soundness of the algorithm solve. We denote a list E as \vec{E} and a projection that projects s-constraints as $|\vec{E}|_s$ and use a predicate $\text{ordered} : \text{List} \rightarrow \text{Prop}$ to test whether a constraint list is in the proper order. Next we define the ordering on the s-constraints.

Lemma 14 (Ordering). *Let $|\vec{E}_1|_s, |\vec{E}_2|_s$ be two constraint lists with only s-constraints then let $e_1 \in \vec{E}_1$ and $e_2 \in \vec{E}_2$ then $e_1 \leq e_2 \Leftrightarrow \text{inn}(e_1) \leq \text{inn}(e_2)$ where $\text{inn} : \text{Constraint} \rightarrow \text{int}$ denotes the in-order numbering of the constraints in the derivation tree.*

Lemma 15 (Projection). *Let \vec{E}_1, \vec{E}_2 be two constraint lists then $\vec{E}_1 \leq \vec{E}_2$ if and only if $|\vec{E}_1|_s \leq |\vec{E}_2|_s$.*

Lemma 16 (Append preserves order). *Let \vec{E}_1, \vec{E}_2 be two constraint lists then $\text{ordered}(\vec{E}_1) \wedge \text{ordered}(\vec{E}_2) \wedge (\text{last}(\vec{E}_1) \leq \text{first}(\vec{E}_2))$ then $\text{ordered}(\vec{E}_1 @ \vec{E}_2)$.*

Lemma 17 (Soundness and Completeness of constraint generation). *For any Core-ML term M_0 and for any type environment Γ $\text{Wand}^+(\Gamma, M_0, \tau_0) = \vec{E}$ if and only if there is a derivation of $\Gamma, \text{unorder}(\vec{E}) \vdash_{W^+} M_0 : \tau_0$.*

Proof. (\Rightarrow) By induction on the structure of the term M_0 . Choose an arbitrary Core-ML term M_0 and a type environment Γ . We have the following cases:

Case $M_0 = x$, for some variable x . Then either x is not bound in Γ in which case the algorithm fails and therefore there is no derivation. In the case x is bound to some type in the type environment we have two cases.

x **is bound to a type variable.** Then $\Gamma = \Gamma_0 \cup \{x : \tau_1\}$ for some Γ_0 and some τ_1 . So $\vec{E} = [\tau_0 \stackrel{e}{=} \tau_1]$ and clearly there is a derivation of $\Gamma, \{\tau_0 \stackrel{e}{=} \tau_1\} \vdash_{W^+} x : \tau_0$.

x **is bound to a type scheme variable.** Then $\Gamma = \Gamma_0 \cup \{x : \tau_1^*\}$ for some Γ_0 and some τ_1^* . So $\vec{E} = [\tau_0 \stackrel{i}{=} \tau_1^*]$ and clearly there is a derivation of $\Gamma, \{\tau_0 \stackrel{i}{=} \tau_1^*\} \vdash_{W^+} x : \tau$.

Case $M_0 = \lambda x.M$, **for some** x **and** M . Then $Wand^+(\Gamma, \lambda x.M, \tau_0) = [\tau_0 \stackrel{e}{=} \alpha_0 \rightarrow \alpha_1] :: Wand^+(\Gamma \setminus x \cup \{x : \alpha_0\}, M, \alpha_1)$, where α_0, α_1 are fresh variables. Then, by the inductive hypothesis, there is a derivation of $\Gamma, E_1 \vdash_{W^+} M : \alpha_1$ and so we can construct a derivation of $\Gamma, \{\tau_0 \stackrel{e}{=} \alpha_0 \rightarrow \alpha_1\} \vdash_{W^+} \lambda x.M : \tau_0$.

Case $M_0 = MN$, **for some** M **and** N . Then $Wand^+(\Gamma, MN, \tau_0) = Wand^+(\Gamma, M, \alpha_0 \rightarrow \tau_0) @ Wand^+(\Gamma, N, \alpha_0)$, where α_0 is a fresh variable. Then, by the inductive hypothesis, there is a derivation of $\Gamma, E_1 \vdash M : \alpha_0 \rightarrow \tau_0$ and there is a derivation of $\Gamma, E_2 \vdash N : \alpha_0$, and so we can construct a derivation of $\Gamma, E_1 \cup E_2 \vdash MN : \tau_0$.

Case $M_0 = \text{let } x = M \text{ in } N$, **for some** M **and** N . Then $Wand^+(\Gamma, \text{let } x = M \text{ in } N, \tau_0) = Wand^+(\Gamma, M, \alpha_0) @ [(\alpha_0 \stackrel{s}{=} \Gamma \alpha_1^*)] @ Wand^+(\Gamma \setminus x \cup \{x : \alpha_1^*\}, N, \tau_0)$, where α_0, α_1^* are fresh type and type scheme variables respectively. Then, by the inductive hypothesis, there is a derivation of $\Gamma, E_1 \vdash_{W^+} M : \alpha_0$ and there is a derivation of $(\Gamma \setminus x) \cup \{x : \alpha_1^*\}, E_2 \vdash_{W^+} N : \alpha_0$, for some E_1, E_2 , and so we can construct a derivation of $\Gamma, \{\alpha_0 \stackrel{s}{=} \Gamma \alpha_1^*\} \cup E_1 \cup E_2 \vdash_{W^+} \text{let } x = M \text{ in } N : \tau_0$.

(\Leftarrow) By induction on the structure of derivation. Choose an arbitrary M_0, Γ . Assume there is a derivation of $\Gamma, E \vdash_{W^+} M_0 : \tau_0$. We have the following cases based on the last rule used in the derivation:

Rule W-Var. Then $M_0 = x, E = \{\tau_1 \stackrel{e}{=} \tau_0\}$ and $x : \tau_1 \in \Gamma$. Then $Wand^+(\Gamma, x, \tau_0) = \overrightarrow{(\tau_1 \stackrel{e}{=} \tau_0)}$ as was to be shown.

Rule W-Var-i. Then $M_0 = x, E = E' \cup \{\tau_1 \stackrel{e}{=} \tau_0\}$, for some E' , and $x : \alpha_0^* \in \Gamma$. Then $Wand^+(\Gamma, x, \tau_0) = E' \cup \{\alpha_0^* \stackrel{i}{=} \tau_1\}$.

Rule W-Abs. Then $M_0 = \lambda x.M$ and $E = E' \cup \{\tau_0 \stackrel{e}{=} \alpha_0 \rightarrow \alpha_2\}$ for some E' . Since $\Gamma, E \vdash_{W^+} \lambda x.M : \tau_0$ is derivable so is $\Gamma \setminus x \cup \{x : \alpha_0\}, E' \vdash_{W^+} x : \alpha_1$. Therefore, by the induction hypothesis, $Wand^+(\Gamma \setminus x \cup \{x : \alpha_0\}, M, \alpha_1) = \overrightarrow{E'}$. Therefore $Wand^+(\Gamma, \lambda x.M, \tau_0) = \overrightarrow{(\tau_0 \stackrel{e}{=} \alpha_0 \rightarrow \alpha_2)} @ Wand^+(\Gamma \setminus x \cup \{x : \alpha_0\}, M, \alpha_1) = \overrightarrow{(\tau_0 \stackrel{e}{=} \alpha_0 \rightarrow \alpha_2)} @ \overrightarrow{E'} = \{(\tau_0 \stackrel{e}{=} \alpha_0 \rightarrow \alpha_2)\} \cup E' = \overrightarrow{E}$.

Rule W-App. Then $M_0 = MN$. Since $\Gamma, E \vdash_{W^+} MN : \tau_0$ is derivable so is $\Gamma, E' \vdash_{W^+} M : \alpha_0 \rightarrow \tau_0$ and $\Gamma, E'' \vdash_{W^+} N : \alpha_0$, for some E', E'' . Therefore $E = E' \cup E''$. By the induction hypothesis, $Wand^+(\Gamma, M, \alpha_0 \rightarrow \tau_0) = \overrightarrow{E'}$ and $Wand^+(\Gamma, N, \alpha_0) = \overrightarrow{E''}$. Therefore $Wand^+(\Gamma, MN, \tau_0) = Wand^+(\Gamma, M, \alpha_0 \rightarrow \tau_0) @ Wand^+(\Gamma, N, \alpha_0) = \overrightarrow{E'} @ \overrightarrow{E''} = \overrightarrow{E' \cup E''} = \overrightarrow{E}$ as was to be shown.

Rule W-Let. Then $M_0 = \text{let } x = M \text{ in } N$. Since $\Gamma, E \vdash_{W^+} \text{let } x = M \text{ in } N : \tau_0$ is derivable so is $\Gamma, E' \vdash_{W^+} M : \alpha_0$ and $(\Gamma \setminus x) \cup \{x : \alpha_1^*\}, E'' \vdash_{W^+} N : \tau_0$, for some E', E'' . Therefore $E = \{\alpha_0 \stackrel{s}{=} \Gamma \alpha_1^*\} \cup E' \cup E''$. By the induction hypothesis, $Wand^+(\Gamma, M, \alpha_0) = \overrightarrow{E'}$ and $Wand^+(\Gamma \setminus x \cup \{x : \alpha_1^*\}, N, \tau_0) = \overrightarrow{E''}$. Therefore $Wand^+(\Gamma, \text{let } x = M \text{ in } N, \tau_0) = Wand^+(\Gamma, M, \alpha_0 \rightarrow \tau_0) @ (\alpha_0 \stackrel{s}{=} \Gamma \alpha_1^*) @ Wand^+(\Gamma, N, \alpha_0) = \overrightarrow{E'} @ (\alpha_0 \stackrel{s}{=} \Gamma \alpha_1^*) @ \overrightarrow{E''} = \overrightarrow{E}$ as was to be shown.

□

Lemma 18 (Constraint solving is correct). *Let M_0 be a Core-ML term and let \vec{E}_0 be a list of constraints, Γ_0 be a type environment and let τ_0 be some type variable then if $Wand^+(\Gamma_0, M_0, \tau_0) = \vec{E}_0$ then $solve(\vec{E}_0)\Gamma_0 \vdash_{HM} M_0 : solve(\vec{E}_0)(\tau_0)$ is derivable.*

Proof. By induction on the number of polymorphic lets in M_0 (denote this number by $|M_0|$). Assume induction hypothesis holds *i.e.* $\forall M_1. |M_1| < |M_0| \Rightarrow \forall \vec{E}_1. \forall \Gamma_1. \forall \tau_1. Wand^+(\Gamma_1, M_1, \tau_1) = \vec{E}_1 \Rightarrow solve(\vec{E}_1)(\Gamma_1) \vdash_{HM} M_1 : solve(\vec{E}_1)(\tau_1)$. Choose arbitrary $\vec{E}_0, \Gamma_0, \tau_0$ and assume $Wand^+(\Gamma_0, M_0, \tau_0) = \vec{E}_0$ and we must show $solve(\vec{E}_0)(\Gamma_0) \vdash_{HM} M_0 : solve(\vec{E}_0)(\tau_0)$. We have two cases:

Case $|M_0| = 0$. Then $solve(\vec{E}_0)(\Gamma_0) \vdash_{HM} M_0 : solve(\vec{E}_0)(\tau_0)$ holds by Lemma 11.

Case $|M_0| \neq 0$. Then let M_1 be the term obtained from M_0 by the transformation which results in eliminating one polymorphic let occurrence. Since $|M_1| < |M_0|$ so we can use the induction hypothesis for M_1 . Let $\Gamma_1 = \Gamma_0$ and $\tau_1 = \tau_0$ and let $\vec{E}_1 = Wand^+(\Gamma_0, M_1, \tau_0)$ so we know $solve(\vec{E}_1)(\Gamma_0) \vdash_{HM} M_1 : solve(\vec{E}_1)(\tau_0)$. Now, by Lemma 19, we have $solve(\vec{E}_0)(\Gamma_0) = solve(\vec{E}_1)(\Gamma_0)$ and $solve(\vec{E}_0)(\tau_0) = solve(\vec{E}_1)(\tau_0)$ so we know $solve(\vec{E}_0)(\Gamma_0) \vdash_{HM} M_0 : solve(\vec{E}_0)(\tau_0)$ holds.

□

Lemma 19 (Constraint Substitution Lemma). *Let $M_0 \stackrel{\text{def}}{=} \mathbf{let} x = M \mathbf{ in} N[x]$ and let $M_1 \stackrel{\text{def}}{=} \mathbf{let} x_1 = M \mathbf{ in} \mathbf{let} x = M \mathbf{ in} N[x_1]$ then if $\vec{E}_0 = Wand^+(\Gamma, M_0, \tau)$ and $\vec{E}_1 = Wand^+(\Gamma, M_1, \tau)$ then $solve(\vec{E}_0)(\Gamma) = solve(\vec{E}_1)(\Gamma)$ and $solve(\vec{E}_0)(\tau) = solve(\vec{E}_1)(\tau)$.*

Theorem 9 (Termination). *Both constraint generation ($Wand^+$) and constraint solving ($solve$) algorithms terminate.*

Proof. The constraint generation phase terminates because after each step of action table the term gets smaller. The constraint solving phase terminates if we can show that both Phase I and Phase II of our algorithm terminates. First, consider Phase I algorithm. It terminates because all e-constraints involving function types on both left side and right side of $\stackrel{e}{=}$ are converted to smaller constraints. Next, consider the Phase II of our algorithm. It terminates because we are dealing with smaller constraint set after each step and since there are a finite number of $s\&i$ constraints and so the entire Phase II algorithm terminates.

Theorem 10 (Correctness of the extended algorithm). *The extended algorithm is a sound and complete implementation of HM type rules and returns the principal type of M under Γ .*

Proof. Note that we are looking at total correctness since we have already showed that the algorithm terminates (Theorem 9). We have shown that the extended type system is sound and complete with respect to HM type system (Theorem 7, Theorem 8). We have also shown the correctness of constraint generation and constraint solving algorithms. Therefore the extended algorithm returns the principal type. The overall idea is shown in Fig. 4.

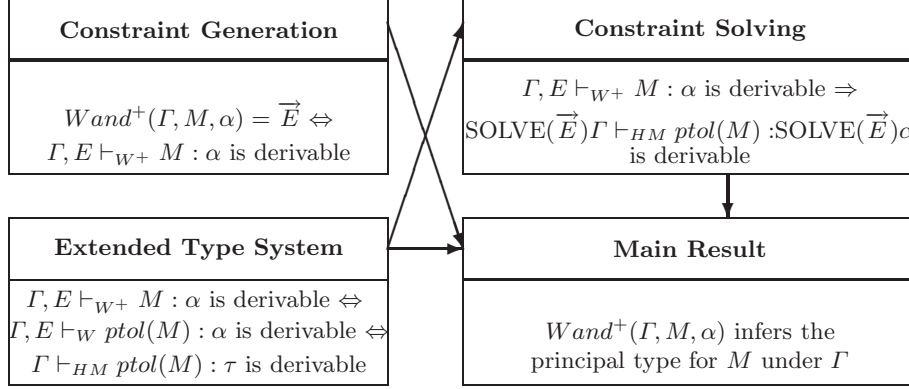


Fig. 4. Correctness proof overview

6.1 Examples

We now elaborate our algorithm on some non-trivial examples.

Example 1. Let us use the above algorithm to infer the type of the following:

`let i = λx. x in (i i)`

We start with an empty environment \emptyset .

The derivation tree is as follows:

$$\frac{\frac{x : \alpha_3 \vdash x : \alpha_4 \quad \alpha_3 \stackrel{e}{=} \alpha_4}{\emptyset \vdash \lambda x. x : \alpha_1} \quad \alpha_1 \stackrel{e}{=} \alpha_3 \rightarrow \alpha_4}{\emptyset \vdash \text{let } i = \lambda x. x \text{ in } i i : \alpha_0} \quad \frac{\frac{i : \alpha_2^* \vdash i : \alpha_5 \rightarrow \alpha_0 \quad \alpha_2^* \stackrel{i}{=} \alpha_5 \rightarrow \alpha_0}{i : \alpha_2^* \vdash i i : \alpha_0} \quad \alpha_1 \stackrel{e}{=} \alpha_2^*}{\alpha_1 \stackrel{e}{=} \alpha_2^*}$$

The constraints generated by the above algorithm are:

$[(\alpha_1 \stackrel{s}{=} \alpha_2^*); (\alpha_1 \stackrel{e}{=} \alpha_3 \rightarrow \alpha_4); (\alpha_3 \stackrel{e}{=} \alpha_4); (\alpha_2^* \stackrel{i}{=} \alpha_5 \rightarrow \alpha_0); (\alpha_2^* \stackrel{i}{=} \alpha_5)]$.

After the first phase of unification, the equality constraints resolve into the substitution ρ specified as:

$\rho = \{(\alpha_1 \mapsto (\alpha_4 \rightarrow \alpha_4)), (\alpha_3 \mapsto \alpha_4)\}$

The above substitution is then applied to the constraint store which modifies the constraints as:

$$\begin{aligned}
\alpha_4 &\rightarrow \alpha_4 \stackrel{s}{=} \emptyset \alpha_2^* \\
\alpha_2^* &\stackrel{i}{=} \alpha_5 \rightarrow \alpha_0 \\
\alpha_2^* &\stackrel{i}{=} \alpha_5
\end{aligned}$$

Since there are no more e-constraints so $empty_e$ is satisfied. At this point, Phase I algorithm terminates and the second phase of unification starts and a s-constraint, $\alpha_4 \rightarrow \alpha_4 \stackrel{s}{=} \emptyset \alpha_2^*$, is chosen to be unified (using Phase I unification) with its corresponding i-constraints. This results in the following constraint set :

$$\begin{aligned}
\alpha_6 &\rightarrow \alpha_6 \stackrel{e}{=} \alpha_5 \rightarrow \alpha_0 \\
\alpha_7 &\rightarrow \alpha_7 \stackrel{e}{=} \alpha_5, \text{ where } \alpha_6 \text{ and } \alpha_7 \text{ are fresh type variables.}
\end{aligned}$$

These constraints are unified and results in the substitution list ρ_1
 $\rho_1 = \{\alpha_6 \mapsto (\alpha_7 \rightarrow \alpha_7), \alpha_5 \mapsto (\alpha_7 \rightarrow \alpha_7), \alpha_0 \mapsto (\alpha_7 \rightarrow \alpha_7)\}$

Since there are no more s-constraints, we compose the substitutions, ρ and ρ_1 resulting in the final substitution:

$$\rho \circ \rho_1 = \{(\alpha_1 \mapsto (\alpha_4 \rightarrow \alpha_4)), (\alpha_3 \mapsto \alpha_4)(\alpha_6 \mapsto (\alpha_7 \rightarrow \alpha_7)), (\alpha_5 \mapsto (\alpha_7 \rightarrow \alpha_7)), (\alpha_0 \mapsto (\alpha_7 \rightarrow \alpha_7))\}$$

Applying this substitution to the original type variable α_0 gives the final type as:

$$\alpha \rightarrow \alpha.$$

Since the initial type environment Γ was empty so the principal type is:
 $\forall \alpha. \alpha \rightarrow \alpha.$

Example 2. Yet another example taken from [Hee05] demonstrates our approach and involves a lambda abstraction with nested lets in the body of the lambda.

```

λx. let f = let g = x
      in λa. g
    in let h = f in
      h x

```

The derivation for the above term is as follows:

$$\begin{array}{c}
\frac{\frac{x : \alpha_1 \vdash x : \alpha_8 \quad \alpha_1 \stackrel{e}{=} \alpha_8}{x : \alpha_1 \vdash \text{let } g = x \text{ in } \lambda a. g : \alpha_3} \quad \frac{\frac{x : \alpha_1, g : \alpha_9^*, a : \alpha_{10} \vdash g : \alpha_{11} \quad \alpha_9^* \stackrel{i}{=} \alpha_{11}}{x : \alpha_1, g : \alpha_9^* \vdash \lambda a. g : \alpha_3} \quad \alpha_3 \stackrel{e}{=} \alpha_{10} \rightarrow \alpha_{11}}{\alpha_8 \stackrel{e}{=} \{x : \alpha_1\} \alpha_9^*}}{x : \alpha_1 \vdash \text{let } g = x \text{ in } \lambda a. g : \alpha_3} \quad \alpha_8 \stackrel{e}{=} \{x : \alpha_1\} \alpha_9^* \\
\text{A} \\
\frac{\frac{x : \alpha_1, f : \alpha_4^* \vdash f : \alpha_5 \quad \alpha_4^* \stackrel{i}{=} \alpha_5}{x : \alpha_1, f : \alpha_4^* \vdash \text{let } h = f \text{ in } h x : \alpha_2} \quad \frac{\frac{f : \alpha_4^*, h : \alpha_6^* \vdash h : \alpha_7 \rightarrow \alpha_2 \quad \alpha_6^* \stackrel{i}{=} \alpha_7 \rightarrow \alpha_2}{x : \alpha_1, f : \alpha_4^*, h : \alpha_6^* \vdash h x : \alpha_2} \quad x : \alpha_1, \vdash x : \alpha_7 \quad \alpha_1 \stackrel{e}{=} \alpha_7}}{x : \alpha_1, f : \alpha_4^* \vdash \text{let } h = f \text{ in } h x : \alpha_2} \quad \alpha_5 \stackrel{e}{=} \{x : \alpha_1, f : \alpha_4^*\} \alpha_6^* \\
\text{B}
\end{array}$$

$$\frac{\frac{}{A} \quad \frac{}{B}}{x : \alpha_1 \vdash \text{let } f = \text{let } g = x \text{ in } \lambda a. g \text{ in let } h = f \text{ in } h x : \alpha_2} \quad \alpha_3 \stackrel{s}{=} \{x:\alpha_1\} \alpha_4^*}{\emptyset \vdash \lambda x. \text{let } f = \text{let } g = x \text{ in } \lambda a. g \text{ in let } h = f \text{ in } h x : \alpha_0} \quad \alpha_0 \stackrel{e}{=} \alpha_1 \rightarrow \alpha_2$$

We get the following constraints from the derivation:

$$\begin{aligned} \alpha_1 &\stackrel{e}{=} \alpha_8 \\ \alpha_9^* &\stackrel{i}{=} \alpha_{11} \\ \alpha_3 &\stackrel{e}{=} \alpha_{10} \rightarrow \alpha_{11} \\ \alpha_4^* &\stackrel{i}{=} \alpha_5 \\ \alpha_6^* &\stackrel{i}{=} \alpha_7 \rightarrow \alpha_2 \\ \alpha_1 &\stackrel{e}{=} \alpha_7 \\ \alpha_0 &\stackrel{e}{=} \alpha_1 \rightarrow \alpha_2 \\ \alpha_5 &\stackrel{s}{=} \{\alpha_1\} \alpha_6^* \\ \alpha_3 &\stackrel{s}{=} \{\alpha_1\} \alpha_4^* \\ \alpha_8 &\stackrel{s}{=} \{\alpha_1\} \alpha_9^* \end{aligned}$$

Phase I unification terminates when all the e-constraints are unified. At the end of Phase I we get the following constraint store:

$$\begin{aligned} \alpha_9^* &\stackrel{i}{=} \alpha_{11} \\ \alpha_4^* &\stackrel{i}{=} \alpha_5 \\ \alpha_6^* &\stackrel{i}{=} \alpha_7 \rightarrow \alpha_2 \\ \alpha_5 &\stackrel{s}{=} \{\alpha_7\} \alpha_6^* \\ \alpha_{10} &\rightarrow \alpha_{11} \stackrel{s}{=} \{\alpha_7\} \alpha_4 \\ \alpha_7 &\stackrel{s}{=} \{\alpha_7\} \alpha_9^* \end{aligned}$$

a substitution given by:

$$\begin{aligned} \alpha_1 &\mapsto \alpha_7 \\ \alpha_3 &\mapsto (\alpha_{10} \rightarrow \alpha_{11}) \\ \alpha_0 &\mapsto (\alpha_7 \rightarrow \alpha_2) \\ \alpha_8 &\mapsto \alpha_7 \end{aligned}$$

The next phase of unification starts and in the first iteration all constraints $\alpha_7 \stackrel{s}{=} \{\alpha_7\} \alpha_9^*$ is chosen and the corresponding i-constraints are unified are transformed to following e-constraints:

$$\alpha_7 \stackrel{e}{=} \alpha_{11}$$

This generates the following substitution

$$\alpha_7 \mapsto \alpha_{11}$$

Apply this substitution to the remaining constraint store we get

$$\begin{aligned} \alpha_4^* &\stackrel{i}{=} \alpha_5 \\ \alpha_6^* &\stackrel{i}{=} \alpha_{11} \rightarrow \alpha_2 \\ \alpha_5 &\stackrel{s}{=} \{\alpha_{11}\} \alpha_6^* \end{aligned}$$

$$\alpha_{10} \rightarrow \alpha_{11} \stackrel{s}{=} \{\alpha_{11}\}\alpha_4^*$$

Next, in the second iteration $\alpha_{10} \rightarrow \alpha_{11} \stackrel{s}{=} \{\alpha_{11}\}\alpha_4$ is chosen and the corresponding i-constraints are transformed to:

$$\alpha_{12} \rightarrow \alpha_{11} \stackrel{e}{=} \alpha_5$$

Phase I unification of the above constraint store generates the following substitution:

$$\alpha_5 \mapsto (\alpha_{12} \rightarrow \alpha_{11})$$

Apply this substitution to the remaining constraint store we get

$$\begin{aligned} \alpha_6^* &\stackrel{i}{=} \alpha_{11} \rightarrow \alpha_2 \\ \alpha_{12} \rightarrow \alpha_{11} &\stackrel{s}{=} \{\alpha_{11}\}\alpha_6^* \end{aligned}$$

This constraint can be solved similarly to get the following constraint store

$$\alpha_{12} \rightarrow \alpha_{11} \stackrel{e}{=} \alpha_{11} \rightarrow \alpha_2$$

Phase I unification of the above constraint store generates the following substitution:

$$\begin{aligned} \alpha_{12} &\mapsto \alpha_2 \\ \alpha_{11} &\mapsto \alpha_2 \end{aligned}$$

This generates the substitution $\{\alpha_7 \mapsto \alpha_{11}\} \circ \{\alpha_5 \mapsto (\alpha_2 \rightarrow \alpha_2), \alpha_{12} \mapsto \alpha_2, \alpha_{11} \mapsto \alpha_2\}$, which is nothing else but $\{\alpha_7 \mapsto \alpha_2, \alpha_5 \mapsto (\alpha_2 \rightarrow \alpha_2), \alpha_{12} \mapsto \alpha_2, \alpha_{11} \mapsto \alpha_2\}$. Finally, all the substitutions generated in the second phase are composed with substitution generated in the first phase to get the final substitution as:

$$\begin{aligned} \{\alpha_1 \mapsto \alpha_2, \alpha_3 \mapsto (\alpha_{10} \rightarrow \alpha_2), \alpha_0 \mapsto (\alpha_2 \rightarrow \alpha_2), \\ \alpha_8 \mapsto \alpha_2, \alpha_7 \mapsto \alpha_2, \alpha_5 \mapsto (\alpha_2 \rightarrow \alpha_2), \alpha_{12} \mapsto \alpha_2, \alpha_{11} \mapsto \alpha_2\} \end{aligned}$$

Applying this substitution to α_0 , the initial type of the expression, we get $\alpha \rightarrow \alpha$, which is the final type of the expression. Since the initial type environment was empty the principal type of the term is $\forall \alpha. \alpha \rightarrow \alpha$.

Example 3. We have verified the inferred type of the following example taken from Mairson [Mai89] involving nested lets.

```

let x1 = λy.λz.(z y) y in
  let x2 = λz.x1 (x1 z) in
    let x3 = λz.x2 (x2 z) in
      let x4 = λz.x3 (x3 z) in
        x4 (λz.z)

```

See Appendix B for the typing derivation of the above expression. The inferred type for the above term is given below:

```

All 'a.All 'b.All 'c.All 'd.All 'e.All 'f.All 'g.All 'h.All 'i.(((((((((((
(((((((('a -> 'a) -> (('a -> 'a) -> 'b)) -> 'b) -> (((('a -> 'a) -> (('a ->
'a) -> 'b)) -> 'b) -> 'c)) -> 'c) -> ((((((('a -> 'a) -> (('a -> 'a) -> 'b))

```


-> 'a) -> 'b)) -> 'b) -> 'c)) -> 'c) -> 'd)) -> 'd) -> 'e)) -> 'e) -> 'f)) -> 'f)
-> 'g)) -> 'g) -> 'h)) -> 'h) -> 'i)) -> 'i)

As a final note we want to make some comments regarding our approach. Phase II of our multi-phase unification is deterministic - the order being the in-order traversal of derivation tree modulo e-constraints. Recall that Phase I of our algorithm is non-deterministic. The order of solving s-constraints becomes important in constraint-based algorithms because we want to generalize over a type such that it undergoes no more changes. If we generalize too early it will be overly general and if we generalize too late it would be overly specific. We generate constraints in a top-down fashion whereas Heeren generates constraints in a bottom-up fashion. We chose top-down approach because it integrates nicely with the action table of Wand's algorithm. Heeren suggests that this representation requires a special substitution that maps type scheme variables to inferred type schemes but our algorithm doesn't require such a substitution. We do not need this substitution because in our approach type scheme variables are mere placeholders to relate a s-constraint with corresponding i-constraints. Furthermore, Heeren suggests (and we agree) that there is an order in which the constraints should be solved. But the order in our algorithm comes into picture in only the second phase of unification, which involves *i&s*-constraint only.

7 Conclusions and future work

We have extended Wand's algorithm to handle the polymorphic let construct. This is certainly not the first attempt at handling the let construct but what is unique is that our algorithm is a *natural* extension of Wand's algorithm and our correctness proof is an extension of Wand's proof (presented in this paper). The main idea behind our approach is to have a multi-phase unification in the constraint solving phase. By having two phases, we ensure that the generated type schemes generalize over the right type. We have validated our approach by running the examples mentioned in this paper on Alg. W, Alg. M, and Alg. J. An implementation of the above algorithm in OCaml is available online at <http://www.cs.uwyo.edu/~skothari>. We plan to use to use proof assistants to prove this approach formally.

References

- [AW93] A. Aiken and E. L. Wimmers. Type inclusion constraints and type inference. In *Functional Programming Languages and Computer Architecture*, pages 31–41, 1993.
- [BS01] F. Baader and W. Snyder. Unification theory. In A. Robinson and A. Voronkov, editors, *Handbook of Automated Reasoning*, volume I, chapter 8, pages 445–532. Elsevier Science, 2001.
- [Car97] L. Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

- [Cho05] Venkatesh Choppella. Polymorphic Type Reconstruction Using Type Equations. In *Implementation of Functional Languages*, volume 3145, pages 53–68, 2005.
- [DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 207–212, New York, NY, USA, 1982. ACM Press.
- [Hee05] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, 2005.
- [Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Math. Soc*, 146:29–60, 1969.
- [HLI03] B. Heeren, D. Leijen, and A. IJzendoorn. Helium, for learning haskell. In *Haskell '03: Proceedings of the 2003 ACM SIGPLAN workshop on Haskell*, pages 62–71, New York, NY, USA, 2003. ACM Press.
- [HS86] J. R. Hindley and J. P. Seldin. *Introduction to Combinators and λ -Calculus*. Cambridge University Press, 1986.
- [KC08] S. Kothari and J.L. Caldwell. On Extending Wand’s Type Reconstruction Algorithm to Handle Polymorphic Let. In Arnold Beckmann, Costas Dimitracopoulos, and Benedikt Lwe, editor, *CiE 2008: Abstracts and extended abstracts of unpublished papers*, 2008.
- [Kni89] K. Knight. Unification: A multidisciplinary survey. *ACM Computing Surveys*, 21:93–124, 1989.
- [Kot07] S. Kothari. Type Reconstruction Algorithms: A Survey. Technical report, University of Wyoming, 2007.
- [KTU90] A. J. Kfoury, J. Tiuryn, and P. Urzyczyn. ML typability is dextptime-complete. In *CAAP '90: Proceedings of the fifteenth colloquium on CAAP'90*, pages 206–220, New York, NY, USA, 1990. Springer-Verlag New York, Inc.
- [Lia97] C. Liang. Let-polymorphism and eager type schemes. In *TAPSOFT*, pages 490–501, 1997.
- [LY98] O. Lee and K. Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):707–723, 1998.
- [Mai89] H. G. Mairson. Deciding ML typability is complete for deterministic exponential time. In *Proc. of the 16th ACM Sym. Principles of Programming Languages*, pages 382–401, 1989.
- [MH88] J. C. Mitchell and R. Harper. The essence of ML. In *POPL '88: Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 28–46, 1988.
- [Mil78] R. Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, pages 348–375, 1978.
- [Mit96] J. C. Mitchell. *Foundations for Programming Languages*, chapter 11. The MIT Press, 1996.
- [Mül94] M. Müller. A constraint-based recast of ML-polymorphism (extended abstract). In Denis Lugiez, editor, *International Workshop on Unification*, 1994.
- [Pie02] B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
- [PJ89] P.C.Kanellakis and J.C.Mitchell. Polymorphic unification and ML typing. In *6th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 105–115. ACM Press, 1989.

- [PR05] F. Pottier and D. Rémy. The essence of ML type inference. In Benjamin C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- [PW76] M. S. Paterson and M. N. Wegman. Linear unification. In *STOC '76: Proceedings of the eighth annual ACM symposium on Theory of computing*, pages 181–186, New York, NY, USA, 1976. ACM.
- [Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12:23–41, 1965.
- [SA93] Z. Shao and A. W. Appel. Smartest recompilation. In *Conference Record of the Twentieth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 439–450, Charleston, South Carolina, 1993.
- [SOW97] M. Sulzmann, M. Odersky, and M. Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.
- [Tys88] J. Tyszkiewicz. Complexity of type inference in finitely typed lambda calculus. Master's thesis, University of Warsaw, 1988.
- [Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.
- [WF94] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.

A Unification algorithm

Let E denote a constraint store and let ρ denote a substitution. We will follow the convention that α, β represent monomorphic types and τ represents a meta variable and ranges over the set of types. The unification algorithm *unify* takes as input a constraint store and returns a substitution. For an excellent overview of unification, interested readers should consult [Kni89], [BS01]. Note that this algorithm always returns an idempotent substitution. The algorithm is described below:

$$\text{unify}_1 : \text{ConstraintStore} \rightarrow \text{Substitution}$$

$$\begin{aligned} \text{unify} (\alpha \stackrel{e}{=} \beta) \cup E &= \text{if } \alpha = \beta \text{ then } \text{unify } E \\ \text{unify} (\alpha \stackrel{e}{=} \tau) \cup E &= \text{if } \alpha \text{ occurs in } \tau \text{ then raise Failure} \\ \text{unify} (\alpha \stackrel{e}{=} \tau) \cup E &= \text{unify} (E[\alpha := \tau]) \circ \{\alpha \mapsto \tau\} \\ \text{unify} (\tau \stackrel{e}{=} \alpha) \cup E &= \text{unify} (\alpha \stackrel{e}{=} \tau) \cup E \\ \text{unify} (\tau_1 \rightarrow \tau_2 \stackrel{e}{=} \tau_3 \rightarrow \tau_4) \cup E &= \text{unify} (\tau_1 \stackrel{e}{=} \tau_3, \tau_2 \stackrel{e}{=} \tau_4 \cup E) \\ \text{unify } \emptyset &= \text{Id} \end{aligned}$$

B Example 3 - Derivation

$$\begin{array}{c} \frac{\frac{\frac{y : \alpha_3, z : \alpha_5 \vdash z : \alpha_8 \rightarrow (\alpha_7 \rightarrow \alpha_6) \quad \alpha_5 \stackrel{e}{=} \alpha_8 \rightarrow (\alpha_7 \rightarrow \alpha_6)}{y : \alpha_3, z : \alpha_5 \vdash (z y) : \alpha_7 \rightarrow \alpha_6} \quad \frac{\frac{y : \alpha_3, z : \alpha_5 \vdash y : \alpha_8 \quad \alpha_3 \stackrel{e}{=} \alpha_8}{y : \alpha_3, z : \alpha_5 \vdash (z y) y : \alpha_6} \quad \frac{y : \alpha_3, z : \alpha_5 \vdash y : \alpha_7 \quad \alpha_3 \stackrel{e}{=} \alpha_7}{y : \alpha_3 \vdash \lambda z. (z y) y : \alpha_4 \quad \alpha_4 \stackrel{e}{=} \alpha_5 \rightarrow \alpha_6}}{\emptyset \vdash \lambda y. \lambda z. (z y) y : \alpha_1 \quad \alpha_1 \stackrel{e}{=} \alpha_3 \rightarrow \alpha_4} \text{A}}{\frac{x1 : \alpha_2^* \vdash x1 : \alpha_{12} \rightarrow \alpha_{11} \quad \alpha_2^* \stackrel{i}{=} \alpha_{12} \rightarrow \alpha_{11} \quad \frac{x1 : \alpha_2^* \vdash x1 : \alpha_{13} \rightarrow \alpha_{12} \quad \alpha_2^* \stackrel{i}{=} \alpha_{13} \rightarrow \alpha_{12} \quad z : \alpha_{10} \vdash z : \alpha_{13} \quad \alpha_{10} \stackrel{e}{=} \alpha_{13}}{x1 : \alpha_2^*, z : \alpha_{10} \vdash (x1 z) : \alpha_{12}}}{x1 : \alpha_2^*, z : \alpha_{10} \vdash x1 (x1 z) : \alpha_{11}} \text{B}}{\frac{x2 : \alpha_{10}^* \vdash x2 : \alpha_{18} \rightarrow \alpha_{17} \quad \alpha_{10}^* \stackrel{i}{=} \alpha_{18} \rightarrow \alpha_{17} \quad \frac{x2 : \alpha_{10}^* \vdash x2 : \alpha_{19} \rightarrow \alpha_{18} \quad \alpha_{10}^* \stackrel{i}{=} \alpha_{19} \rightarrow \alpha_{18} \quad z : \alpha_{16} \vdash z : \alpha_{19} \quad \alpha_{16} \stackrel{e}{=} \alpha_{19}}{x2 : \alpha_{10}^*, z : \alpha_{16} \vdash (x2 z) : \alpha_{18}}}{x2 : \alpha_{10}^*, z : \alpha_{16} \vdash x2 (x2 z) : \alpha_{17}} \text{C}}{\frac{x3 : \alpha_{15}^* \vdash x3 : \alpha_{23} \rightarrow \alpha_{22} \quad \alpha_{15}^* \stackrel{i}{=} \alpha_{23} \rightarrow \alpha_{22} \quad \frac{x3 : \alpha_{15}^* \vdash x3 : \alpha_{24} \rightarrow \alpha_{23} \quad \alpha_{15}^* \stackrel{i}{=} \alpha_{24} \rightarrow \alpha_{23} \quad z : \alpha_{21} \vdash z : \alpha_{24} \quad \alpha_{21} \stackrel{e}{=} \alpha_{24}}{x1 : \alpha_2^*, x2 : \alpha_{10}^*, x3 : \alpha_{15}^*, z : \alpha_{21} \vdash (x3 z) : \alpha_{23}}}{x1 : \alpha_2^*, x2 : \alpha_{10}^*, x3 : \alpha_{15}^* \vdash \lambda z. x3 (x3 z) : \alpha_{20} \quad \alpha_{20} \stackrel{e}{=} \alpha_{21} \rightarrow \alpha_{22}} \text{D}} \end{array}$$

