On Extending Wand's Type Reconstruction Algorithm to Handle Polymorphic Let*

Sunil Kothari and James L. Caldwell

Department of Computer Science
University of Wyoming
Laramie, WY 82071-3315, USA
{skothari,jlc}@cs.uwyo.edu

Abstract. We have extended Wand's type reconstruction algorithm to polymorphic let 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 environment[1] Γ such that a judgment $\Gamma \vdash M : \tau$ is valid ?

Type reconstruction is a popular feature in modern functional programming languages. Underlying any type reconstruction algorithm is a set of rules encoding 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, DM82, LY98] have been proposed to implement the HM type system. Many of these algorithms are characterized by 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 [Hee05, PR05, Wan87]. This separation leads to better error messages [Hee05] 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 ab-

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

[1] We assume the initial type environment is empty since we are dealing with closed terms.

straction of the various substitution-based algorithms[2] 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 DEXPTIME[3]-complete and PSPACE-hard problem [PJ89, Mai89] in the level of nested lets. Moreover, the literature on constraint-based type reconstruction is sparse and uneven. For example, Pierce's book [Pie02] has no references on how to handle ML-Let construct in constraint-based algorithms, HM(X) [SOW97, PR05] requires specialized knowledge, whereas Aiken and Wimmers [AW93] use subtyping constraints. Furthermore, none of the literature, in our view, describes it as a *direct* extension to well known Wand's algorithm [Wan87].

The Helium compiler [HLI03] is known for giving good quality error messages and a very simple constraint representation is used for handling the let construct[4]. This paper describes an approach where 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 are:

1. A new algorithm extending Wand's algorithm [Wan87] to include polymorphic let.
2. New soundness and completeness proofs for Wand's system and the extended system using a novel desugaring of polymorphic lets.

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 Wand's algorithm and states soundness and completeness theorems. Section 5 describes the changes needed for the extension. Section 6 gives an overview of the correctness proofs. Section 7 summarizes our current work.

2 Literature Review

We review some of the constraint-based algorithms in their handling of the let construct. A detailed survey of substitution-based algorithms is available in [Kot07]. **Wand** [Wan87] looked at the type inference problem as a type-erasure: whether it is decidable that a term of the untyped lambda calculus is the image under type-erasing of a term of the simply typed lambda calculus, and presented a type reconstruction algorithm. This was the first successful attempt at separating constraint generation from constraint solving phase. However, extending the algorithm to handle the let construct remained a future work[5]. **Heeren**

[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.

[5] Personal communication from Mitchell Wand.

[Hee05] suggested three constraint representations to handle the let construct; each equally expressive but differing in constraint solving.

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

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

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 type of the expression is $\forall \tau_1, \tau_2.\tau_1 \equiv \tau_2 \Rightarrow \tau_1 \rightarrow \tau_2$. In many ways, this approach is similar to HM(X)[SOW97], Pottier and Rémy's account of ML type inference [PR05], and Jones's qualified types [Jon95].

Approach 2. Type scheme variables as placeholders: The type constraint language is extended to take into account generalization and instantiation of type schemes by means of type scheme variables. The constraint language is given by the following grammar:

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

Our approach uses a slightly modified version of the above constraint representation. We know of no published account of the constraint solving phase for this representation, although Heeren does mention in his thesis that a special substitution that maps type scheme variables to type schemes is needed for this representation. Our algorithm does not require any such substitution.

Approach 3. Using implicit instance constraints: This approach merges the generalization and instantiation constraints in the above approach. The constraint language is given by the following grammar:

$$\mathcal{C} ::= \ \tau_1 \equiv \tau_2 \mid \tau_1 \leq_M \tau_2$$

This representation is described, in detail, in Heeren's thesis [Hee05].

3 Preliminaries

In this paper, we assume familiarity with the notion of types, functional programming and the lambda calculus. The terms considered here are pure untyped lambda terms given by the following grammar:

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

We follow the usual conventions for lambda calculus: arrow types associate to the right, function applications associate to the left and application binds more tightly than abstraction. 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 a function type. We call any expression formed by following the above grammar as a *type expression*. We follow the convention that α, β denote type variables, whereas τ denotes a type expression. A *type environment*, denoted by Γ, maps type variables to type expressions. The set of *free type variables* of a type expression τ is denoted by $\mathrm{FTV}(\tau)$ and those of a type environment Γ is denoted by $\mathrm{FTV}(\Gamma)$. A term and its type is related by an *assertion*, denoted by $\Gamma \vdash M : \tau$, 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 $\Gamma \backslash x$. A *derivation* of an assertion $\Gamma \vdash M : \tau$ is a finite tree of assertions,

where the root is $\Gamma \vdash M : \tau$. Every interior node in the tree is related to its parent by an instance of one of the non-axiom type rules and every leaf node is an instance of an axiom type rule. In this paper, we consider three different type systems: the Hindley-Milner type system illustrated in Fig. 1, Wand's system illustrated in Fig. 2, and the extended Wand system illustrated later in Section 5. We denote a judgment in a particular system by a subscript. For example, $\Gamma \vdash_{HM} M : \tau$ is a HM judgment.

$$\frac{}{\Gamma \vdash_{HM} x : \tau} \text{ where } x : \tau \in \Gamma \qquad \text{(HM-Var)}$$

$$\frac{\Gamma \backslash x \cup \{x : \tau_1\} \vdash_{HM} M : \tau_0}{\Gamma \vdash_{HM} \lambda x.M \ : \ \tau_1 \to \tau_0} \qquad \text{(HM-Abs)}$$

$$\frac{\Gamma \vdash_{HM} M : \tau_1 \to \tau \qquad \Gamma \vdash_{HM} N : \tau_1}{\Gamma \vdash_{HM} MN : \tau} \qquad \text{(HM-App)}$$

Fig. 1. Hindley-Milner type system

$$\frac{}{\Gamma, \{\alpha \stackrel{e}{=} \tau\} \vdash_W x : \alpha} \text{ where } x : \tau \in \Gamma \text{ and } \alpha \text{ is fresh} \qquad \text{(W-Var)}$$

$$\frac{(\Gamma \backslash x) \cup \{x : \alpha\}, E \vdash_W M : \beta}{\Gamma, E \cup \{\tau \stackrel{e}{=} \alpha \to \beta\} \vdash_W \lambda x.M : \tau} \text{ where } \alpha, \beta \text{ are fresh} \qquad \text{(W-Abs)}$$

$$\frac{\Gamma, E_1 \vdash_W M : \alpha \to \tau \qquad \Gamma, E_2 \vdash_W N : \alpha}{\Gamma, E_1 \cup E_2 \vdash_W MN : \tau} \text{ where } \alpha \text{ is fresh} \qquad \text{(W-App)}$$

Fig. 2. Wand's type system

A *substitution* is a mapping from type variables to type expressions. Let ρ_1, ρ_2 be two substitutions then *substitution composition* $\rho_1 \circ \rho_2$ is defined extensionally as $\forall \alpha.(\rho_1 \circ \rho_2)\alpha \stackrel{\text{def}}{=} \rho_1(\rho_2 \alpha)$, where α is a type variable. Substitution composition is associative but non-commutative. A substitution ρ is *idempotent* if $\rho \circ \rho = \rho$. We treat all the substitutions as idempotent substitutions. A type τ' is a substitution instance of a type τ if and only if $\tau' = \rho\tau$, for some substitution ρ. 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, referred to as *e-constraints*, and are denoted by $\tau_1 \stackrel{e}{=} \tau_2$, where τ_1, τ_2 are type expressions. An e-constraint $\tau_1 \stackrel{e}{=} \tau_2$ is *solvable* if there exists a substitution ρ such that $\rho\tau_1 = \rho\tau_2$. More formally, we denote solvability of a constraint by $\models$ (read solve). We write $\rho \models \tau_1 \stackrel{e}{=} \tau_2$, if $\rho\tau_1 = \rho\tau_2$. A type judgment in Wand's system is given as $\Gamma, E \vdash_W M : \alpha$, where E denotes a constraint set. Sometimes we elide the constraint component of the judgment to simplify the presentation. In that case, we denote a judgment by $\Gamma \vdash_W M : \tau$, where E is implicit and so there is a substitution generated by solving E such that $\rho\alpha = \tau$.

4 Wand's Algorithm

Next we briefly discuss Wand's algorithm [Wan87] and state the soundness and completeness theorems. Central to Wand's algorithm is the notion of *action table*, which takes as input a type system and a term and generates equational constraints. For untyped lambda calculus, the type system is shown in Fig. 2. Let G denote a set of assertions (also called goals) and E a set of equational constraints. Then the algorithm sketch is given as:

Input. A term M_0 of Λ.
Initialization. Set $E = \emptyset$ and $G = \{(\Gamma_0, M_0, \alpha_0)\}$, where α_0 is a fresh type variable and Γ_0 is an empty environment.
Loop Step. If $G = \emptyset$ then return E else choose a sub-goal (Γ, M, τ) from G, delete the sub-goal from G and add to E and G new verification conditions and subgoals generated by the action table[6].
Solve Constraints. Unify constraints in E.

We can now describe the soundness and completeness results as stated by Wand, but before that we introduce some terminology. We denote a goal by a 3-tuple, *i.e.* (Γ, M, τ), and we write $\rho \models g$ *i.e.* $\rho \models (\Gamma, M, \tau)$ if and only if $\rho\Gamma \vdash_{HM} M : \rho\tau$. We write $\rho \models G$ if and only if $\forall g \in G.\ \rho \models g$. Similarly, if E is a set of e-constraints, we write $\rho \models E$ if and only if $\forall e \in E.\ \rho \models e$. Finally, we say ρ solves (E, G), where (E,G) result from applying Wand's algorithm, if and only if $\rho \models E$ and $\rho \models G$.

(Soundness) $\forall \rho.\ \rho \models (E, G) \Rightarrow \rho\Gamma_0 \vdash_{HM} M_0 : \rho\tau_0$
(Completeness) $\Gamma \vdash_{HM} M_0 : \tau \Rightarrow (\exists \rho.\ \rho \models (E, G) \wedge \Gamma = \rho\Gamma_0 \wedge \tau = \rho\tau_0)$

We have reformulated Wand's statement of completeness and soundness and made them more abstract (by eliminating the goal set). The reformulated theorems are used later in the proofs of soundness and completeness of the extended proof system. Our statement of the soundness and completeness are given below:

Theorem 1 (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.*

Theorem 2 (Completeness). *If there is a derivation of $\Gamma \vdash_{HM} M_0 : \tau$, then for any 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 there exists a derivation of $\Gamma_0,\ E \vdash_W M_0 : \tau_0$, and there exists a substitution ρ' such that $\rho \subseteq \rho'$ and $\rho' \models E$.*

Both these theorems are proved in [KC07].

5 Extended Wand's Algorithm

In this section we describe the changes needed to incorporate the let construct. First, we enrich our term and type language. We call the extended language *Core-ML* [MH88], and it is defined as:

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

[6] See Section 5 for action table behavior for untyped lambda calculus.

The let expression **let** $x = M$ **in** N is not merely a syntactic sugar for $(\lambda x.N)M$. For instance, the term **let** $i = \lambda x.x$ **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. To handle polymorphic types introduced by let-expressions, the type syntax is extended with type scheme variable and type scheme as shown below:

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

A *type scheme*, denoted by $\forall\overrightarrow{\alpha}.\tau$, 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 "*". *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. Thus $gen(\Gamma,\tau) \stackrel{\text{def}}{=} \forall\overrightarrow{\alpha}.\tau$, where $\overrightarrow{\alpha} = FTV(\tau) - FTV(\Gamma)$. On the other hand, *instantiation* of a type scheme involves replacing the quantified variables by fresh type variables and is given by $inst(\forall\overrightarrow{\alpha}.\tau) \stackrel{\text{def}}{=} \tau[\alpha_1 := \beta_1, \ldots, \alpha_n := \beta_n]$, where $\beta_1, \ldots, \beta_n$ are fresh type variables.

Next, we extend the constraint language $\mathcal{C}$ to include two other kinds of constraints as shown below:

$$\mathcal{C} ::= \underbrace{\tau \stackrel{\text{e}}{=} \tau}_{\text{(e-constraint)}} \mid \underbrace{\tau \stackrel{\text{s}}{=}_{\Gamma} \alpha^*}_{\text{(s-constraint)}} \mid \underbrace{\alpha^* \stackrel{\text{i}}{=} \tau}_{\text{(i-constraint)}}$$

A *s-constraint* $\tau \stackrel{\text{s}}{=}_{\Gamma} \alpha^*$ expresses the fact that α^* denotes a type scheme obtained by generalizing a type τ with respect to the environment Γ. An *i-constraint* $\alpha^* \stackrel{\text{i}}{=} \tau$ expresses the fact that τ is constrained to be an instantiated value of the type scheme denoted by α^*. Wand's type system is now extended with two rules to account for i&s-constraints. The new type system is called *extended Wand's system* and a judgment in the extended system is denoted by the subscript W^+.

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

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

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 related[7] to a s-constraint $\tau_0 \stackrel{\text{s}}{=}_{\Gamma} \alpha^*$. From this point onwards, we think of a s-constraint and related i-constraint as a pair $(\tau_0 \stackrel{\text{s}}{=}_{\Gamma} \alpha^*, E_{\alpha^*})$; 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

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

ρ satisfies $(\tau_0 \stackrel{s}{=}_\Gamma \alpha^*, E_{\alpha^*})$ if $\forall(\alpha^* \stackrel{i}{=} \tau_1) \in E_{\alpha^*}.\ \rho\,\tau_1 \leq \rho(gen(\Gamma, \tau_0))$. Substitution application to a pair of s-constraint and related i-constraints is defined as: $\rho(\tau_0 \stackrel{s}{=}_\Gamma \alpha^*, E_{\alpha^*}) \stackrel{\text{def}}{=} (\rho\tau_0 \stackrel{s}{=}_{\rho\Gamma} \alpha^*, \{\rho\tau \stackrel{i}{=} \alpha^* \mid (\tau \stackrel{i}{=} \alpha^*) \in E_{\alpha^*}\})$.

Next, we sketch the constraint generation phase[8] for Core-ML. The algorithm sketch remains the same except that E now is a list[9] of equational constraints instead of a set. 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 (Γ, M, $\alpha \to \tau_0$) 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 $\tau_0 \stackrel{e}{=} \alpha \to \beta$ and sub-goal $((\Gamma \backslash x) \cup \{x : \alpha\}, M, \beta)$, and add to E & G respectively.

Case $(\Gamma$,**let** $x = M$ **in** $N, \tau_0)$**.** Let α_1, α_2^* be fresh type variables. Append[10] $E_l @ [\alpha_1 \stackrel{s}{=}_\Gamma \alpha_2^*] @ E_r$ to list E, where E_l, E_r are obtained by recursively calling the extended algorithm on (Γ, M, α_1) and $((\Gamma \backslash x) \cup \{x : \alpha_2^*\}, N, \tau_0)$ 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:

$$
\begin{array}{l}
SOLVE(E) = \\
\qquad \text{let } \rho_1 = unify_1 E_e \text{ in} \\
\qquad\quad \rho_1 \circ (unify_2\ \rho_1(E_{i\&s}))
\end{array}
$$

The first phase, $unify_1$ is defined as:

$$
\begin{array}{lll}
unify_1(E) = & & \\
\quad unify_1\ ((\alpha \stackrel{e}{=} \beta) :: E) & = & \text{if } \alpha = \beta \text{ then } unify_1\ E \\
\quad unify_1\ ((\alpha \stackrel{e}{=} \tau) :: E) & = & \text{if } \alpha \text{ occurs in } \tau \text{ then raise Failure} \\
\quad unify_1\ ((\alpha \stackrel{e}{=} \tau) :: E) & = & (\alpha \mapsto \tau) \circ unify_1\ (E[\alpha := \tau]) \\
\quad unify_1\ ((\tau \stackrel{e}{=} \alpha) :: E) & = & unify_1\ ((\alpha \stackrel{e}{=} \tau) :: E) \\
\quad unify_1\ ((\tau_1 \to \tau_2 \stackrel{e}{=} \tau_3 \to \tau_4) :: E) & = & unify_1\ ((\tau_1 \stackrel{e}{=} \tau_3) :: ((\tau_2 \stackrel{e}{=} \tau_4) :: E))
\end{array}
$$

[8] We denote this constraint generation phase by $Wand^+$ in Fig. 3.

[9] This is needed to preserve the order of s-constraints.

[10] This will ensure that we solve the leftmost innermost let first.

The second phase, $unify_2$, is defined as:

$$
\begin{array}{lll}
unify_2\ (E'@E) = & \text{let } \rho_1 = unify_1 E'' & \\
& \text{in } \rho_1 \circ unify_2\ (\rho_1 E) & \\
& \text{where } E'' = \{inst(gen(\Gamma, \tau_1)) \stackrel{\text{e}}{=} \tau_2 \mid (\alpha^* \stackrel{\text{i}}{=} \tau_2) \in E_{\alpha^*}\} & \\
& \text{and } E' = [(\tau_0 \stackrel{\text{s}}{=}_{\Gamma} \alpha^*), E_{\alpha^*}] & \\
unify_2\ [\,] & = Id &
\end{array}
$$

The first phase of our constraint solving algorithm is very similar to the algorithm proposed by Martelli and Montanari [MM82], which is known to be exponential[11] in the size of the input in the worst case [BS01]. Therefore, the first phase of the unification is linear while our implementation, which does not use the efficient DAG representation, is exponential in the size of the input in the worst case. The second phase of the algorithm involves calling Phase I algorithm as many times as the number of s-constraints, *i.e.*, the number of let-bound variables and there can be only $O(n)$ of those. Therefore, in the worst case, the total time required by the algorithm is $O(2^n)$, where n is the size of the term.

6 Extension Correctness

For the correctness result, we need a function to transform polymorphic lets to pure lambda terms since Wand's type system has no notion of let. We call this function *ptol*. Note that *ptol* is a type and value preserving transformation [KC07]. We introduce two additional notations before formally describing *ptol*. First, we use $N[x]$ to denote one hole (denoted by [ ]) in a context (denoted by N) filled with the let-bound variable (x in this case). Second, we use the notation $|FV(N)|_x$ to denote the count[12] of free occurrence of x in N. Notice that we transform only the body of the let since we assume the let-binding is let-free. This assumption is strictly not necessary since there is a type and value preserving transformation mentioned by Mairson[Mai89], which can make a let-binding let-free. We use the assumption to simplify our proofs but its certainly not a restriction on our algorithm. With the above notations, we describe *ptol* below:

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

[11] Linear if the input and output are represented as directed acyclic graph (DAG) [PW76].

[12] We use the *conservative* notion of occurrence of a let-bound variable rather than the actual types to differentiate between a monomorphic and polymorphic let.

The correctness is given by the proof sketch in Fig. 3. The most complicated proof was showing that the extended type system was sound and complete with respect to the Hindley-Milner type system via Wand's type system. The detailed arguments involved in the correctness, the proofs of various theorems and lemmas, and a detailed discussion on the novel desugaring of polymorphic lets to pure lambda terms can be found in [KC07].

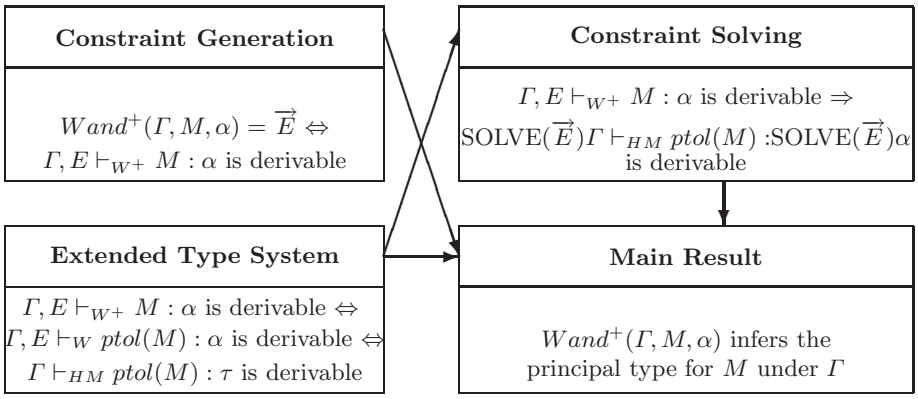

Fig. 3. Correctness proof overview

7 Conclusions and Future Work

The extension of Wand's algorithm is a non-trivial extension and requires careful handling of constraints. Our algorithm is a *direct* extension of Wand's algorithm and our soundness and completeness proofs rely on completeness and soundness of Wand's algorithm. The main idea behind our algorithm is to preserve the order of generated s-constraints (as described in the action table for Core-ML) and use a multi-phase unification, while preserving this order, in the constraint solving phase. We have validated our approach by running the examples mentioned in this paper on Alg. W [Mil78, DM82], Alg. M [LY98] and Alg. J [Mil78]. An implementation of our algorithm and other popular type reconstruction algorithms in OCaml is available online at `http://www.cs.uwyo.edu/~skothari`. We are working on a formalization of the proofs in CoQ.

Acknowledgments

Thanks to Bastiaan Heeren for a detailed response to author's query regarding the constraint representations mentioned in his thesis. We also want to thank anonymous referees for their detailed comments and suggestions (on an earlier draft of this paper), which greatly improved the presentation of this paper.

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.

[DM82] L. Damas and R. Milner. Principal type-schemes for functional programs. In *POPL '82*, pages 207–212, New York, 1982. ACM Press.

[Hee05] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universitiet 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.

[Jon95] M. P. Jones. *Qualified types: theory and practice*. Cambridge University Press, New York, NY, USA, 1995.

[KC07] S. Kothari and J. L. Caldwell. Wand's Algorithm Extended for the Polymorphic ML-Let. Technical report, University of Wyoming, 2007.

[Kot07] S. Kothari. Type Reconstruction Algorithms: A Survey. Technical report, University of Wyoming, 2007.

[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.

[MM82] A. Martelli and U. Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[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.

[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.

[Wan87] M. Wand. A simple algorithm and proof for type inference. *Fundamenta Informaticae*, 10:115–122, 1987.