Type Inference Algorithms

February 19, 2009

S. Saeidi Mobarakeh (0559938)

Abstract

In this paper we are going to describe the Wand's type inference algorithm and we'll try to extend this algorithm with the notion of polymorphic let. By means of a type system, which we're going to extend with some constraint language, we are able to extend the algorithm's first phase (generation of equations) with let-polymorphism. The second phase of the algorithm (solving of the generated equations) needs some modifications to be done on standard unification algorithms because the new generated equation constructs can not directly be fed in to the standard unification algorithm. The correctness of the first phase of the algorithm is been proved by extending the Wand's soundness and completeness prove. Finally a simple example is given to clarify the idea behind the algorithm.

1 Short Historical Overview

The notion of principal type schemes is an old idea. In this section some of the historical developments in this area are reviewed. Note that this area is still under development and there are several researches ongoing about this topic.
The notion of principal type schemes originates at least from Curry's work [Curry-Fey58](probably in the 1930's). Based on Curry's idea, Hindley has introduced an algorithm in order to compute the principal types for combinatory logic in 1969 [Hindley69]. An important extension to computation of the principal type was the notion of let-polymorphism which is taken in to account in Milner's work for the Meta language in Edinburgh LCF system in 1987 and he has proven the soundness of this algorithm. [Milner78].
After the introduction of Milner's algorithm, which was empowered with Let-polymorphism, one question was left open, namely whether the type assignment algorithm finds the most general type possible for every expression and declaration. This question has been answered in 1982 by Damas-Milner [Damas-Milner82], which has been proved sound and complete in Damas's PhD thesis.
Alan Mycroft has extended Milner's algorithm in 1984 with recursive types [Mycroft84].

In 1987 Mitchell Wand has introduced another algorithm to infer the principal types. Wand's approach, which is called constraint-based type inference in literature [Heeren05], is different in the sense that it has separated the type inference process into two parts: namely the algorithm for generating the type equations and the algorithm for solving them [Hemerik08].
Moreover, separation provides abstraction from the various unification algorithms i.e. for solving the generated equations from Wand's algorithm, any unification algorithm can be used, for instance

Robinson's unification algorithm [Robinson65].

2 Introduction

There are several Type Inference Algorithms discussed in the literature. Most of these algorithms can be broadly classified into two distinct categories:

1. Substitution/Unification based algorithms.

2. Equation/Constraint based algorithms.

In the first category the process of generating and solving the equations are interleaved with each other while the second category separates these two. In the first category errors due to miss-typing are detected faster. Some of the advantages of the second approach is that a larger set of constraints available to reason about errors, and as the result, more descriptive error messages can be generated in cases where the constraint set is unsatisfiable. Moreover, the separation of implementation and specification in this category of algorithms, provides a clean separation of concerns and one can abstract from the various unification algorithms.[Heeren05]

In this paper we're going to define the principal type schemes and other related notions formally. Furthermore the required building blocks of the type inference algorithms are taken into account. The main focus is Wand's Type Inference Algorithm because of its considerable simplification. One of the shortcomings of this algorithm is its limited application domain i.e. simply typed λ-calculus. We are going to extend this algorithm with the notion of let-polymorphism which enables us to use a language with more complex terms qua structure and provides more freedom. To this end, one of the three constraint representations i.e. Type scheme variables as placeholders, suggested by Bastian Heeren [Heeren05] in his PhD thesis is being used.

The rest of this paper is organized as follows: Problem statement is outlined in section 3. Section 4 is devoted to Preliminaries needed to describe the algorithms. Section 5 contains the original Wand's algorithm with proof of termination. Section 6 is devoted to the extended Wand's algorithm with let polymorphism and the correctness proof of this extension using the invariants introduced by Wand. Finally section 7 contains some concluding remarks.

3 Problem Statement

The type inference problem can be stated as follows: Assume M is a closed term of the simple untyped lambda calculus which contains type variables. We aim to find the possible type substitutions (possible typings) for M. These possible typings are called type schemes -or as Curry called it: the "functional character" of the term. The type substitution can be defined as a mapping from type variables to types. In other words, Types are type schemes without variables[Hindley69].

Among these possible typings, our ultimate goal is to find the most general type scheme i.e. all possible types for M are substitution instances of the most general type. This most general type is called the principal type scheme of M.
As an illustration consider the following closed term in $\lambda \rightarrow$:

$$M = \lambda x.\lambda y.x$$

Some possible typings for this term are :

- $\alpha \rightarrow \beta \rightarrow \alpha$
- $nat \rightarrow bool \rightarrow nat$
- $list(nat) \rightarrow bool \rightarrow list(nat)$

As one may observe the two latter typings are instances of the first one. The fist typing is the most general typing which one can find for term M and hence it is the principal type scheme of term M.

It is decidable whether a term of the simple un-typed lambda calculus is typable [Wand87]. In the following we will describe some algorithms that, given the term M, computes a type if M is typable in $\lambda \rightarrow$, and "fail" otherwise.

4 Formal Preliminaries

Underlying each type inference algorithm is a set of rules encoding a type system. The type system which is used in Wand's algorithm is Un-typed $\lambda \rightarrow$ à *la Curry* type system. In this section we describe the syntax and type deduction rules of the simply typed $\lambda \rightarrow$ à *la Curry*.

4.1 Un-typed $\lambda \rightarrow$ *à la Curry*

The syntax definition of so-called first order λ calculus in Curry style is as follows:

$$\Lambda ::= Var|(\Lambda\Lambda)|(\lambda Var.\Lambda)$$

These terms have monomorphic (simple types). Let $TVar = \{\alpha, \beta, \gamma... \}$ be an infinite set of type variables. Simple Types in $\lambda \rightarrow$ with typical type expression elements : σ, τ, ρ, ... are introduced as follows:

$$Typ ::= TVar \mid Typ \rightarrow Typ$$

The corresponding typing rules of $\lambda \rightarrow$ à *la Curry* which define the semantics of this system are defined as follows:

$$\{VAR\} \quad \frac{(x{:}\tau) \in \Gamma}{\Gamma \vdash x{:}\tau}$$

$$\{\rightarrow E\} \quad \frac{\Gamma \vdash M{:}\,\sigma \rightarrow \tau \quad \Gamma \vdash N{:}\sigma}{\Gamma \vdash M\ N{:}\tau}$$

$$\{\rightarrow I\} \quad \frac{\Gamma \cup x{:}\sigma \vdash M{:}\tau}{\Gamma \vdash \lambda x.M{:}\,\sigma \rightarrow \tau}$$

These set of rules are directly used in Wand's algorithm as underlying type system (see section 5).

4.2 Definitions and Theorems

Having the definition of a type system and the corresponding typing rules, we need to define a couple of notions which are required in order to proceed with describing the type inference algorithm. Some of these definitions are specifically needed for Wand's algorithm, while some of them are generally also applicable to other type inference algorithms.

`Definition.0: Substitution`
A type substitution (or just substitution) is a map S from type variable to types. As a function, we write it after the type, so σ S denotes the result of carrying out the substitution S on σ. Note that S, T,... are used as typical substitution symbols through the rest of this paper.

`Definition.1: Principal Type Scheme`
Let M be a closed λ term; σ is called a *principal type scheme* for M iff:

1. $\vdash M : \sigma$.

2. $(\forall \tau. \vdash M : \tau.(\exists S.\tau = \sigma S)$).

`Definition.2: Unifier`
Let S be a *Substitution*. S is a *unifier* of σ and τ iff:
$\sigma S \equiv \tau S$.

`Definition.3: Most General Unifier`
S is called a *most general unifier* of σ and τ iff:

1. $\sigma S \equiv \tau S$.

2. $(\forall T.\sigma T \equiv \tau T.(\exists R.T \equiv S; R))$.

Where $\sigma(S; T)$ obtained by $(\sigma S)T$.

The following definitions are required ingredients for Wand's algorithm.

Definition.2: Principal Pair
Let M be a λ term (which may have free variables). The pair $< \Gamma, \sigma >$ is a *principal pair* of M iff:

1. $\Gamma \vdash M : \sigma$.

2. $(\forall \Gamma', \tau . \Gamma' \vdash M : \tau . (\exists S.\ \Gamma' = \Gamma S \wedge \tau = \sigma S)\ )$.

Definition.3: Type Assumption (Context)
A *type assumption* is a partial function $\Gamma : V \rightarrow Typ$ with finite domain.

Definition.4: Type Assertion
A *type assertion* is a triple (Γ, M, τ), where Γ is a type assumption, M is a lambda term, and τ is a type. Moreover the domain of Γ is exactly the free variables of M. This triple is called goal in Wand's algorithm.

Definition.5: Solving Condition
Let S be a substitution.

- if $S \vDash \sigma = \tau iff \sigma S = \tau S$
- $S \vDash (\Gamma, M, \tau)$ iff $\Gamma S \vdash M : \tau S$.
- if E is a set of equations $S \vDash E$ (read S solves E) iff $(\forall e \in E . S \vDash e)$.
- if G is a set of subgoals $S \vDash G$ (read S solves G) iff $(\forall g \in G . S \vDash g)$.

We say substitution $S \vDash (E, G)$ (S solves (E, G)) iff $S \vDash E$ and $S \vDash G$.

Type Inference Algorithm

1. Given a closed term M in $\lambda \rightarrow$ à la Curry, it is decidable whether there exists a type τ such that $\emptyset \vdash M : \tau$.

2. If there exists any such τ, then there also exists a *principal type scheme* for M.

4.3 λ_2 with Weakly Polymorphic Types in *à* la Curry

The extension of terms with let polymorphism yields an extended version of the previously introduced system. It is so-called the second order λ calculus in Curry style. Note that type checking and typability in λ_2 Curry is undecidable [Barendsen05]. However the weak version of polymorphism (let polymorphism) is decidable. The syntactical definition of terms in λ_2 with Weakly Polymorphic Types in *à la Curry* form is given as follows:

$$\Lambda^+ ::= Var | (\Lambda^+ \Lambda^+) | (\lambda Var.\Lambda^+) | let\ Var = \Lambda^+\ in\ \Lambda^+$$

The polymorphic types in λ_2 with Weak Polymorphic Types are defined as follows:

$$\begin{array}{rcll} Typ_w & ::= & Typ & \text{(monomorphic type)} \\ & | & \forall\vec{\alpha}.Typ_w & \text{(polymorphic type scheme)} \end{array}$$

- $Type\ Scheme\ Variables$ typically $\mathcal{S}, \mathcal{T}, ...$

- $\forall\vec{\alpha}.\mathcal{T}$ where $\vec{\alpha}$ denotes the sequence of type variables $(\alpha_1, \alpha_2, ...\alpha_n)$ such that $\alpha_i \in FTV(\mathcal{T})$; where $FTV(\mathcal{T})$ is a set of free type variables in type scheme $\mathcal{T}$.

- $Free\ Type\ Variables$ of a $Type\ Expression\ \tau$ are indicated as $FTV(\tau)$.

The corresponding type deduction rules of λ_2 with Weak Polymorphic Types in *à la Curry* are defined as follows:

$$\{VAR'\} \quad \frac{(x{:}\mathcal{T}){\in}\Gamma}{\Gamma{\vdash}x{:}\mathcal{T}}$$

$$\{{\rightarrow}\ E\} \quad \frac{\Gamma{\vdash}M{:}\sigma{\rightarrow}\tau\ \Gamma{\vdash}N{:}\sigma}{\Gamma{\vdash}M\ N{:}\tau}$$

$$\{{\rightarrow}\ I\} \quad \frac{\Gamma \cup x{:}\sigma{\vdash}M{:}\tau}{\Gamma{\vdash}\lambda x.M{:}\sigma{\rightarrow}\tau}$$

$$\{LET\} \quad \frac{\Gamma \vdash M{:}\mathcal{T} \quad \Gamma \cup x{:}\mathcal{T} \vdash N{:}\tau}{\Gamma \vdash let\ x{=}M\ in\ N{:}\tau}$$

$$\{\forall \rightarrow E\} \quad \frac{\Gamma \vdash M{:}\forall\vec{\alpha}.\mathcal{T}}{\Gamma \vdash M{:}\mathcal{T}[\vec{\alpha}{:=}\vec{\tau}]}$$

$$\{\forall \rightarrow I\} \quad \frac{\Gamma \vdash M{:}\mathcal{T}}{\Gamma \vdash M{:}\forall\vec{\alpha}.\mathcal{T}} \qquad \text{where } \alpha_i \in FTV(\mathcal{T}) \wedge \alpha_i \notin FTV(\Gamma)$$

In order to be able to use this set of rules in the extended version of Wand's algorithm, some issues have to be solved. These issues are listed below:

1. generalize types to type schemes and instantiate these type schemes.

2. restriction on the order of type inference preceding.

3. we refer to these restrictions as $Type\ Constraints$.

   - infer the polymorphic type of the body.

- proceed with typing the program.

There are three suggestions in Heeren's thesis to deal with these problems. These solutions are:

- **Alternative.1** : qualification of equality constraints.
- **Alternative.2** : introducing type scheme variables.
- **Alternative.3** : collecting implicit instance constraints.

Among these three alternatives, we are going to take a look at the second one more closely and use it to extend the Wand's algorithm accordingly. One might be interested in reading the two other alternatives in Heeren's thesis chapter.4 [Heeren05].

4.4 Introducing Type Scheme Variables

The main problem when taking Let-polymorphism into account is the order of which the type equations are inferred. Type Schemes represent the Polymorphic Types and these Polymorphic Types are in turn the result of solving a part of the collected Type Constraints which are not known beforehand. To solve this problem, one approach suggested by Bastian Heeren [Heeren05] is to use Type Scheme variables as place-holders for Polymorphic Types which are not known yet, but will become available at some time during constraint solving. The resulted $TypeConstraint$ language is defined as follows:

$$\begin{array}{rcll} \mathcal{E} & ::= & \tau \equiv \sigma & (equality\ constraint) \\ & | & \mathcal{T} := \textsc{Gen}(\Gamma, \tau) & (generalization\ constraint) \\ & | & \tau := \textsc{Inst}(\mathcal{T}) & (instantiation\ constraint) \end{array}$$

- $\textsc{Gen}(\Gamma, \tau) = \forall\vec{\alpha}.\mathcal{T}$; where $\vec{\alpha} = FTV(\mathcal{T}) - FTV(\Gamma)$.
- $\textsc{Inst}(\forall\vec{\alpha}.\mathcal{T}) = \mathcal{T}[\vec{\alpha} := \vec{\beta}]$; where $\vec{\beta}$ consists of fresh type variables.

The Generalization Constraint produces a Type Scheme by generalizing a type τ with respect to the environment Γ and assigns the result to a Type Scheme variable.
The Instantiation Constraint on the other hand instantiates the Type Scheme variable with a fresh type variable like τ. A special case is however when the to be instantiated Type Scheme is a simple type. In this case $\tau := \textsc{Inst}(\sigma)$ is equal to $\tau \equiv \sigma$.
It is important to realize that Type Variables will never be mapped to Type Schemes.
From now on, we'll use shorthand notations i.e. *e-constraints* stands for *equality constraint*.

4.5 Syntax Directed Rules with Constraints

Having a definition of Type Constraint, we are now ready to introduce the syntax directed set of rules which we are going to apply directly in the extended version of Wand's algorithm.
The Type Judgment in new type system is as follows:

$$\Gamma, \mathcal{E} \vdash M : \tau$$
(where $\mathcal{E}$ a constraint set with typical elements E_i)

The corresponding syntax directed and constraint driven set of Type Deduction Rules in this new Type System are as follows:

$$\{W - VAR'\} \quad \frac{(x{:}\mathcal{T}) \in \Gamma}{\Gamma, \{\tau{:=}\textsc{Inst}(\mathcal{T})\} \vdash x{:}\tau}$$

$$\{W - \rightarrow E\} \quad \frac{\Gamma, E_1 \vdash M{:}\sigma \rightarrow \tau \quad \Gamma, E_2 \vdash N{:}\sigma}{\Gamma, E_1 \cup E_2 \vdash M\ N{:}\tau}$$

$$\{W - \rightarrow I\} \quad \frac{\Gamma \cup \{x{:}\sigma\}, E \vdash M{:}\rho}{\Gamma, E \cup \{\tau \equiv \sigma \rightarrow \rho\} \vdash \lambda x.M{:}\tau}$$

$$\{W - LET\} \quad \frac{\Gamma, E_1 \vdash M{:}\sigma \quad \Gamma \cup \{x{:}\mathcal{T}\}, E_2 \vdash N{:}\tau}{\Gamma, E_1 \cup \{\mathcal{T}{:=}\textsc{Gen}(\Gamma, \sigma)\} \cup E_2 \vdash let\ x{=}M\ in\ N{:}\tau}$$

One can observe that this syntax directed type system is equivalent to the one of section 4.3.
$\{\forall \rightarrow I\}$ and $\{LET\}$ rules are combined to create the $\{W - LET\}$ rule of this system and in the same manner the $\{\forall \rightarrow E\}$ and $\{VAR'\}$ rules are combined to create the $\{W - VAR'\}$ rule.

5 Wand's Algorithm

Wand's Type Inference algorithm can be divided into two main fases,

1. generation of a set of typing equations.

2. solving these equations.

In order to accomplish these two fases, the following steps are done:

Let M_0 be a term with $FV(M) = \{x_1, ..., x_n\}$.

1. Introduce type variables $\alpha_0, ..., \alpha_n$

2. Define context $\Gamma \triangleq \{x_1 : \alpha_1, ..., x_n : \alpha_n\}$

3. Compute a set of equations E which satisfies:

   - $Q_1 : (\forall T.(T \vDash E) \Rightarrow (\Gamma_0 T \vdash M_0 : \alpha_0 T))$
     soundness : Each solution of set of typing equations E yield a typing for M.

   - $Q_2 : (\forall \Gamma', \tau'.(\Gamma' \vdash M_0 : \tau') \Rightarrow (\exists T.\ T \vDash E \wedge \Gamma' = \Gamma T \wedge \tau' = \alpha_0 T))$
     completeness: For each typing of M, there exists a solution of the set E.

4. Compute a most general unifier S of E.

5. The pair $< \Gamma_0 S, \alpha_0 S >$ is a principal pair for M_0.

Steps 1 up to 3 are done in the first phase of the algorithm. For computing steps 4 and 5, any unification algorithm i.e. Robinson's unification algorithm can be used (see section 5.2).

5.1 Fase.1: Generation of Equations

The Pseudo-code of this phase is given as follows:

WAND (M : Λ): $\mathbb{P}(T \times T)$
begin
$E, G := \emptyset, \{(\Gamma_0, M_0, \alpha_0)\}$;
`do` $G \neq \emptyset \rightarrow$
`let` $g \in G$; $G := G \setminus \{g\}$;
$\Delta E := EQ(g)$; $\Delta G := SG(g)$;

$E, G := E \cup \Delta E, G \cup \Delta G;$
od;
return E;
end.

We refer to this as a skeleton for an algorithm because it can be completed in different ways using different action tables for processing the subgoals in the loop step. This skeleton will be used in the extended version of Wand's algorithm for instance with the extended action table according to the added "let" construct.

Regardless of this choice, running this algorithm keeps two following equations invariant relating the possible completions of the derivation with the possible typings of M_0:

- $soundness: \ (\forall T : T \vDash (E, G) \Rightarrow \Gamma_0 T \vdash M_0 : \alpha_0 T).$
  soundness : Any solution for (E,G) generates only correct typings for (Γ_0, M_0, α_0)
- $completeness: \ (\forall \Gamma', \tau'.\Gamma' \vdash M_0 : \tau' \Rightarrow (\exists T.T \vDash (E, G) \wedge \Gamma' = \Gamma_0 T \wedge \tau' = \alpha_0 T).$
  completeness : Every typing of M_0 is generated by some solution for (E, G).

Note that invariant is satisfied by initialization i.e. from the invariants and $G = \emptyset$, post conditions Q_1 and Q_2 can be inferred.

Moreover, the algorithm terminates with the following function on the size of the set of goals containing the subgoals:
Assume the set SG of subgoals consists of terms M_1, M_2, ..., M_n, where the size of a term is defined according to its structure as follows:

- |x| = 0
- |λ $x.M$| = 1 + |M|
- |M N| = 1+ |M| + |N|

Using this size function, $s(SG)$= $|M_1| + |M_2| + ... + |M_n|$ decreases in each loop iteration □.

The proof of Invariance according to the action table below can be found in Wand's paper [Wand87]. The internal steps of this algorithm depend on the given term. According to the so-called action table, this given term is analyzed and the set of subgoals $SG(g)$ and the set of equations $EQ(g)$ is being modified accordingly.

During each iteration of the loop, one term is picked up from the set of goals and it's being analyzed according to its syntax. The action table defined below determines which changes should than be made in the set of equations and the set of subgoals respectively. This process will terminate when the set of goals is empty. One can observe that these rules are result of the syntax directed type system

introduced in section 4.1.

`Action Table:`

g	$SG(g)$	$EQ(g)$
(Γ, x, τ)	$\emptyset$	$\{\tau = \Gamma x\}$
$(\Gamma, (\lambda x.M), \tau)$	$\{((\Gamma, x : \alpha_1), M, \alpha_2)\}$	$\{\tau = \alpha_1 \rightarrow \alpha_2\}$
$(\Gamma, M\ N, \tau)$	$\{(\Gamma, M, \alpha \rightarrow \tau), (\Gamma, N, \alpha)\}$	$\emptyset$

Note that this table can easily be extended for other language constructs. Note also that the α's are fresh variables.

5.2 Fase.2: Unification

In order to solve the resulting equations, a unification algorithm such as Robinson's unification algorithm can be used. The pseudo code of Robinson's algorithm is given below. Let R be a ranked alphabet; we write F^m for a symbol F with rank m. Let T be the set of terms over R and Var, with typical elements τ_i and σ_i. Let E_0 be a finite subset of T x T.

UNIFICATION() : $Substitution$;

var $E : \mathbb{P}(T \times T)$; $S : Substitution$;
$E, S := E_0, Id$;
do $E \neq \emptyset \rightarrow$
  let $e \in E$; $E := E \setminus \{e\}$;
  if $e :: (F^m(\tau_1, ..., \tau_m), G^n(\sigma_1, ..., \sigma_n)) \rightarrow$
    if $F^m \not\equiv G^n \rightarrow$ ***fail***
    [] $F^m \equiv G^n \rightarrow E := E \cup \{(\tau_1, \sigma_1), ..., (\tau_m, \sigma_n)\}$
    fi
  [] $e :: (\alpha, \tau) \rightarrow$
    if $\tau \equiv \alpha \rightarrow$ ***skip***
    [] $\tau \not\equiv \alpha \wedge \alpha \in FV(\tau) \rightarrow$ ***fail***
    [] $\tau \not\equiv \alpha \wedge \alpha \notin FV(\tau) \rightarrow E := E[\alpha := \tau]$; $S := (S; [\alpha := \tau])$
    fi
  fi
od

6 Wand's Algorithm Extended with Let Polymorphism

The main skeleton of the algorithm remains the same with a small difference. The result of the extended algorithm is a list of equations, unlike the result of Wand's algorithm which was a set of equations. The previously generated equations can not be solved with a standard unification algorithm any more. In the following we describe how both phases of the algorithm should change in order to infer the principal type for a term with let polymorphism construct.

6.1 Fase.1: Generation of Equations

In order to extend Wand's algorithm, we use the type system introduced in section 4.5. The action table will change in two places. The rule for variable now involves more; it contains the instantiation of type scheme variables with a fresh type variable. Furthermore, the rule for let is added.

g	$SG(g)$	$EQ(g)$
(Γ, x, τ)	$\emptyset$	$\{\tau \equiv \text{INST}(\Gamma x)\}$
$(\Gamma, (\lambda x.M), \tau)$	$\{(\Gamma \cup x : \alpha_1), M, \alpha_2\}$	$\{\tau \equiv \alpha_1 \rightarrow \alpha_2\}$
$(\Gamma, M\ N, \tau)$	$\{(\Gamma, M, \alpha \rightarrow \tau), (\Gamma, N, \alpha)\}$	$\emptyset$
$(\Gamma, let\ x = M\ in\ N, \tau)$	$\{(\Gamma, M, \alpha_1), ((\Gamma \cup x : \mathcal{T}), N, \tau)\}$	E^*

In this table $E^* := E_M@[\mathcal{T}_1 := \text{GEN}(\Gamma, \alpha_1)]@E_N$ where E_M is obtained by recursively calling the extended algorithm on (Γ, M, α_1) and E_N is obtained by recursively calling the extended algorithm on $((\Gamma \cup x : \mathcal{T}_1), N, \tau)$ respectively. This ordering results in solving the left most inner most occurrence of "let" first. Note that α_i's are fresh type variables and $\mathcal{T}_i$'s are fresh type scheme variables.

Note that the recursive call of the algorithm on M and N is finite because the size of the terms in these recursions decrease with each call. Furthermore, per let construction in the given term we'll get only one GEN() operator which doesn't have a blow up effect on the whole.

6.1.1 Extension's Correctness Proof

The interesting question here is whether this algorithm is correct or not. Note that the initialization arguments hold for the extension as well i.e. we didn't change the structure of algorithm skeleton.
The remaining proof obligations are to show that this algorithm preserves the Wand's algorithm invariants and that it terminates.

`Proposition : Termination`
With this action table, the algorithm always terminates.

Proof:

Assume the set SG of subgoals consists of terms M_1, M_2, ..., M_n, where the size of a term is defined according to its structure as follows:

- |x| = 0

- |λ $x.M$| = 1 + |M|

- |M N| = 1+ |M| + |N|

- | $let\ x = M\ in\ N$ | = 1+ |M| + |N|

using this size function, the $s(SG)$= $|M_1| + |M_2| + ... + |M_n|$ decreases in each loop iteration □.

Proposition : Invariance

Each action in the action table preserves the invariant of the algorithm.
The invariants are given as below :

- $soundness: \ (\forall T : T \vDash (E, G) \Rightarrow \Gamma_0 T \vdash M_0 : \alpha_0 T)$.

- $completeness: \ (\forall \Gamma^{'}, \tau^{'}.\Gamma^{'} \vdash M_0 : \tau^{'} \Rightarrow (\exists T.T \vDash (E, G) \wedge \Gamma^{'} = \Gamma_0 T \wedge \tau^{'} = \alpha_0 T)$.

Proof:
We split the proof into two parts. The soundness proof which states that the algorithm perseveres the first invariant and the completeness proof which states that the algorithm perseveres the second invariant.

Soundness proof:

Assuming that the soundness invariant holds before the action is executed, we have to show that the clause also holds after execution of that action. We know that the following holds:

$$(\forall T : T \vDash (E, G) \Rightarrow \Gamma_0 T \vdash M_0 : \alpha_0 T)$$

We have to show that:

$$(\forall T : T \vDash (E', G') \Rightarrow \Gamma_0 T \vdash M_0 : \alpha_0 T)$$

In total we have four cases to prove the soundness for them, however because two of these cases are left unchanged and are proven correct in Wand's paper, we are only going to prove the soundness for the two other cases which are modified/added by means of our extension. We divide the soundness proof into two cases.

Case.1 : $g :: (\Gamma, x, \tau)$

(E', G') = $(E \ \cup \ \{\tau \ \equiv \ \textsc{Inst}(\Gamma x)\}, G \backslash \{(\Gamma, x, \tau)\})$.

Assume $T \vDash (E', G')$ for a certain substitution T. Then $T \vDash E$.

$T \vDash \tau \equiv \textsc{Inst}(\Gamma x)$ implies $\tau\ T \equiv \textsc{Inst}(\Gamma x)T$.

From the type inference rule $(W - VAR')$ we obtain $(\Gamma\ , \tau := \textsc{Inst}(\mathcal{T}))T \vdash x : \tau\ T$ which is equivalent with $(\Gamma T\ , \tau T := \textsc{Inst}(\mathcal{T})T) \vdash x : \tau\ T$ and this is equivalent to $T \vDash (\Gamma, x, \tau)$.

We also know that $T \vDash G\backslash\{(\Gamma, x, \tau)\}$ so we get $T \vDash G$. From $T \vDash (E, G)$ we obtain by invariant that $\Gamma_0 T \vdash M_0 : \alpha_0 T\ \square$.

Case.2 : $g :: (\Gamma, let\ x = M\ in\ N, \tau)$

$(E', G') = ((E \cup (E_M @ [\mathcal{T}_1 := \textsc{Gen}(\Gamma, \alpha_1)] @ E_N)), (G\backslash\{(\Gamma, let\ x = M\ in\ N, \tau) \cup \{(\Gamma, M, \alpha_1), ((\Gamma \cup x : \mathcal{T}), N, \tau)\}))$.

Assume $T \vDash (E', G')$ for a certain substitution T. Then $T \vDash E$

$T \vDash (E_M @ [\mathcal{T}_1 := \textsc{Gen}(\Gamma, \alpha_1)] @ E_N)$ implies $(E_M T @ ([\mathcal{T}_1 := \textsc{Gen}(\Gamma, \alpha_1)])T @ E_N T)$.

From $T \vDash (\Gamma, M, \alpha_1)$ we obtain $\Gamma T \vdash M : \alpha_1 T$.
From $T \vDash ((\Gamma \cup x : \mathcal{T}), N, \tau)$ we obtain $(\Gamma \cup x : \mathcal{T})T \vdash N : \tau T$ which is equal to $(\Gamma T \cup x : (\mathcal{T})T) \vdash N : \tau T$.

By the type inference rule $(W - LET)\ \Gamma T, E_1 T \cup (\{\mathcal{T} := \textsc{Gen}(\Gamma, \sigma)\})T \cup E_2 T \vdash (let\ x = M\ in\ N) : \tau T$.
This is equivalent with $T \vDash (\Gamma, let\ x = M\ in\ N, \tau)$.

Because $T \vDash G\backslash\{(\Gamma, let\ x = M\ in\ N, \tau)$ we get $T \vDash G$.
From $T \vDash (E, G)$ we obtain by invariant that $\Gamma_0 T \vdash M_0 : \alpha_0 T\ \square$.

Completeness proof:

Assuming that the completeness invariant holds before the action is executed, we have to show that the clause also holds after execution of that action. Therefore we assume $\Gamma^{'} \vdash M_0 : \tau^{'}$ for certain type assumption $\Gamma^{'}$ and type τ. By the invariant we know that :

$$(\exists T.T \vDash (E, G) \wedge \Gamma^{'} = \Gamma_0 T \wedge \tau^{'} = \alpha_0 T).$$

and we have to show that:

$$(\exists T'.T' \vDash (E', G') \wedge \Gamma^{'} = \Gamma_0 T' \wedge \tau^{'} = \alpha_0 T').$$

where (E', G') is the state of the algorithm after the action step.
Choose a goal $g \in G$. Then we know that there is a substitution say T, such that $T \vDash E$,

$T \vDash G\backslash\{g\}$ and $T \vDash g$.

In total we have four cases to prove the completeness for them, however because two of these cases are left unchanged and are proven correct in Wand's paper, we are only going to prove the correctness for the two other cases which are modified/added by means of our extension. We split the completeness proof into two corresponding cases:

Case.1 : $g :: (\Gamma, x, \tau)$

$(E', G') = (E \cup \{\tau \equiv \text{INST}(\Gamma x)\}, G\backslash\{(\Gamma, x, \tau)\})$.

Then by invariant $T \vDash (\Gamma, x, \tau)$ and thus $\Gamma T \vdash x : \tau T$.
By the type inference rule for variables $(W - VAR')\ \tau\ T \equiv \text{INST}(\Gamma x)T$.
Hence $T \vDash \tau \equiv \text{INST}(\Gamma x)$. Define $T' = T$. Then $T' \vDash (E', G')\ \square$.

Case.2 : $g :: (\Gamma, let\ x = M\ in\ N, \tau)$

$(E', G') = ((E \cup (E_M @ [\mathcal{T}_1 := \text{GEN}(\Gamma, \alpha_1)] @ E_N)), (G\backslash\{(\Gamma, let\ x = M\ in\ N, \tau) \cup \{(\Gamma, M, \alpha_1), ((\Gamma \cup x : \mathcal{T}), N, \tau)\}))$.

By the invariant $T \vDash (\Gamma, let\ x = M\ in\ N, \tau)$ and thus $\Gamma T \vdash (let\ x = M\ in\ N) : \tau T$.
By the type inference rule for $(W - LET)$ there must be a σ such that $\Gamma T, E_1 \vdash M : \sigma$ and there must be a fresh type scheme variable $\mathcal{T} := \text{GEN}(\Gamma, \sigma)$ such that $\Gamma T \cup \{x : \mathcal{T}\}, E_2 \vdash N : \tau T$.

So let α_1 be a fresh type variable not in the domain of Γ and let $\mathcal{T}_1 := \text{GEN}(\Gamma, \alpha_1)$ be a fresh type scheme variable.
Let T' be defined by $T' = T[\alpha_1 \leftarrow \sigma][\mathcal{T}_1 \leftarrow \mathcal{T}]$.
Then $T' \vDash (\Gamma, M, \alpha_1)$ and $T' \vDash (\Gamma \cup x : \mathcal{T}), N, \tau)$ and $T' \vDash (E_M @ [\mathcal{T}_1 := \text{GEN}(\Gamma, \alpha_1)] @ E_N)$.

From $T \vDash G\backslash g$ we get $T' \vDash G\backslash g$ (α_1 is fresh).
From $T \vDash E$ we get $T' \vDash E$ (α_1 is fresh).
So we have $T' \vDash (E', G')\ \square$.

6.2 Fase.2: Unification of Extended Algorithm

The Unification algorithm receives as input a list of equations and it provides the *Most General Unifier*.
Let R be a ranked alphabet; we write F^m for a symbol F with rank m. Let T be the set of terms over R and Var, with typical elements τ_i and σ_i. Let E be an input set of equations resulted by the first phase of the algorithm.

EW-UNIFICATION(***var*** $E : [e_0, e_1, ..., e_n]) : Substitution;$

var $S : Substitution;$
$S := Id;$
do $E \neq \emptyset \rightarrow$
  let $e_0 \in E; E := E \setminus \{e_0\};$

  if $e_0 :: \mathcal{T} \equiv \text{GEN}(\Gamma, \tau) \rightarrow$
    $E := E[\mathcal{T} \equiv \forall\vec{\alpha}.\tau];$

  [] $e_0 :: \tau \equiv \text{INST}(\mathcal{T}_i) \rightarrow$
    $E := E[\mathcal{T}_i \equiv \vec{\beta}];$
    $\triangleright\ where\ \vec{\beta}\ fresh$

  [] $e_0 :: \tau \equiv \text{INST}(\sigma) \rightarrow$
    $E := E[\tau \equiv \sigma];$

  [] $e_0 :: (F^m(\tau_1, ..., \tau_m) \equiv G^n(\sigma_1, ..., \sigma_n)) \rightarrow$
    if $F^m \not\equiv G^n \rightarrow$ ***fail***
    [] $F^m \equiv G^n \rightarrow E := E \cup \{(\tau_1, \sigma_1), ..., (\tau_m, \sigma_n)\}$
    fi

  [] $e_0 :: (\tau \equiv \sigma) \rightarrow$
    if $\tau \equiv \sigma \rightarrow$ ***skip***
    [] $\tau \not\equiv \sigma \wedge \sigma \in FTV(\tau) \rightarrow$ ***fail***
    [] $\tau \not\equiv \sigma \wedge \sigma \notin FTV(\tau) \rightarrow E := E[\sigma := \tau]; S := (S; [\sigma := \tau])$
    fi
  fi
result $:= S;$
od

As one may observe, this algorithm is linear in the number of elements of the list E of equations.
Since the original algorithm is proved to be correct and the extensions follows the structure of the original algorithm we are not going to give a correctness prove of this algorithm.

6.3 Simple Example

Given $M := let\ x\ =\ \lambda y.y\ in\ xx$, find the Principal Type Scheme of M using the extended Wand's algorithm.

G	E
$(\emptyset, let\ x\ =\ \lambda y.y\ in\ xx, \tau)$	
$\{(\emptyset, \lambda y.y, \alpha_1), (x : \mathcal{T}_1, xx, \tau)\}$ $\triangleright$ choose i.e. the second one! ($\rightarrow$ E rule applicable)	$\mathcal{T}_1 \equiv \text{GEN}(\emptyset, \alpha_1)$
$\{(\emptyset, \lambda y.y, \alpha_1), (x : \mathcal{T}_1, x, \alpha_2 \rightarrow \tau), (x : \mathcal{T}_1, x, \alpha_2)\}$ $\triangleright$ the second one(VAR' rule applicable).	
$\{(\emptyset, \lambda y.y, \alpha_1)(x : \mathcal{T}_1, x, \alpha_2)\}$ $\triangleright$ the first one($\rightarrow$ I rule applicable).	$\alpha_2 \rightarrow \tau \equiv \text{INST}(\mathcal{T}_1)$
$\{(y : \alpha_3, y, \alpha_4), (x : \mathcal{T}_1, x, \alpha_2)\}$ $\triangleright$ the first one(VAR' rule applicable).	$\alpha_1 \equiv \alpha_3 \rightarrow \alpha_4$
$\{(y : \alpha_3, y, \alpha_4)\}$	$\alpha_2 \equiv \text{INST}(\mathcal{T}_1)$
$\{\}$	$\alpha_4 \equiv \text{INST}(\alpha_3)$

List of generated equations:
[
$\alpha_1 \equiv \alpha_3 \rightarrow \alpha_4$,
$\alpha_4 \equiv \text{INST}(\alpha_3)$,
$\mathcal{T}_1 \equiv \text{GEN}(\emptyset, \alpha_1)$,
$\alpha_2 \rightarrow \tau \equiv \text{INST}(\mathcal{T}_1)$,
$\alpha_2 \equiv \text{INST}(\mathcal{T}_1)$
]

Unification : After 1^{st} loop iteration:
[
$\alpha_4 \equiv \text{INST}(\alpha_3)$,
$\mathcal{T}_1 \equiv \text{GEN}(\emptyset, \alpha_3 \rightarrow \alpha_4)$,
$\alpha_2 \rightarrow \tau \equiv \text{INST}(\mathcal{T}_1)$,
$\alpha_2 \equiv \text{INST}(\mathcal{T}_1)$
]

Unification : After 2^d loop iteration:
[
$\alpha_4 \equiv \alpha_3)$,
$\mathcal{T}_1 \equiv \text{GEN}(\emptyset, \alpha_3 \rightarrow \alpha_4)$,
$\alpha_2 \rightarrow \tau \equiv \text{INST}(\mathcal{T}_1)$,
$\alpha_2 \equiv \text{INST}(\mathcal{T}_1)$
]

Unification : After 3^d loop iteration:
[
$\mathcal{T}_1 \equiv \text{GEN}(\emptyset, \alpha_4 \rightarrow \alpha_4)$,

$\alpha_2 \to \tau \equiv \text{INST}(\mathcal{T}_1)$,
$\alpha_2 \equiv \text{INST}(\mathcal{T}_1)$
]

Unification : After 4^{th} **loop iteration:**
[
$\mathcal{T}_1 \equiv \forall\vec{\alpha}.\alpha_4 \to \alpha_4)$,
$\alpha_2 \to \tau \equiv \text{INST}(\mathcal{T}_1)$,
$\alpha_2 \equiv \text{INST}(\mathcal{T}_1)$
]

Unification : After 5^{th} **loop iteration:**
[
$\alpha_2 \to \tau \equiv \text{INST}(\alpha_5 \to \alpha_5)$,
$\alpha_2 \equiv \text{INST}(\alpha_6 \to \alpha_6)$
]

Unification : After 6^{th} **loop iteration:**
[
$\alpha_2 \to \tau \equiv \alpha_7 \to \alpha_7$,
$\alpha_2 \equiv \text{INST}(\alpha_6 \to \alpha_6)$
]

Unification : After 7^{th} **loop iteration:**
[
$\alpha_2 \to \tau \equiv \alpha_7 \to \alpha_7$,
$\alpha_2 \equiv \alpha_8 \to \alpha_8$
]

Unification : After 8^{th} **loop iteration:**
[
$\alpha_8 \to \alpha_8 \to \tau \equiv \alpha_7 \to \alpha_7$,
]

Principal type: Principal type of

$$let\ x\ =\ \lambda y.y\ in\ xx \text{ is } \tau := \alpha \to \alpha.$$

7 Concluding Remarks

The extension of Wand's algorithm with polymorphic let is possible. This is done by extending the underlying type system with constraints and enforce an order during the equation gathering from the body and the program part of let construct and more important the nested let constructs. Generating equations using this algorithm leads to a list of equations which

can't be unified with a standard unification algorithm. Hence a unification algorithm is given to solve these equations.
Correctness of Wand's algorithm is rather simple to prove due to its modularity and it can be extended for other language constructs such as the cartesian product.

References

[Curry-Fey58] H.B. Curry & R. Feys. *Combinatory Logic*. Vol.1. North Holland, Amsterdam,1958.

[Pierce02] Benjamin C. Pierce. *Types and Programming Languages*. ISBN: 0-262-16209-1, Ch. 22. pp. 317-338, 2002.

[Hindley69] J.R. Hindley. *The principal type-scheme of an object in combinatory logic*. Transactions of the American Mathematical Society 146 pp. 29-60, 1969.

[Milner78] Robin Milner. *A theory of polymorphism in programming*. JCSS 17,3, pp. 348-375, 1978.

[Damas-Milner82] Damas-Milner. *Principal type-schemes for functional program*. ACM O-89791-065-6/82/OOl/0207, 1982.

[Mycroft84] Alan Mycroft. *type schemes and recursive definitions*. Lecture Notes in Computer Science: Proc. 6_{th} intl. symp. on programming, vol. 167, Springer-Verlag, 1984.

[Wand87] Mitchell Wand. *A Simple Algorithm and Proof for Type Inference*. Fundamenta Infomaticae ,10: pp.115-122, 1987.

[Robinson65] J.Robinson. *; A Machine-Oriented Logic Based on the Resolution Principle*. Journal of the ACM (JACM), Volume 12, Issue 1, pp.23 - 41, 1965.

[Geuvers08] Herman Geuvers. *Introduction to type theory*. Lecture Notes Alpha Lernet Summer School Piriapolis, Uruguay, 02-2008.

[Hemerik08] Kees Hemerik. *Principal Type Schemes*. Notes on an un-published book, 06-03-2008.

[Cardone-Hindley06] Felice Cardone & J. Roger Hindley. *History of Lambda-calculus and Combinatory Logic*. Swansea University Mathematics Department Research Report No. MRRS-05-06, pp.42, 2006.

[Heeren05] Bastiaan Johannes Heeren *Top Quality Type Error Messages*. ISBN: 90-393-4005-6, Research School IPA, University of Utrecht, Ch.4, pp.35-57, 2005.

[Barendsen05] Erik Barendsen *Introduction to type theory*. selected from: Basic Course Type Theory, Dutch Graduate School in Logic, June 1996, 2005.