Type Reconstruction Algorithms - A Survey

Sunil Kothari
Department of Computer Science
University of Wyoming

June 26, 2007

Abstract

Most type reconstruction algorithms can be broadly classified into two distinct categories: unification and substitution based and constraint based. This report is a survey of some of the popular type reconstruction algorithms in the above two categories to promote better understanding of these algorithms. We have implemented the above algorithms for a language based on pure lambda calculus extended extended with poylorphic let construct on some non-trivial examples.

1 Introduction

Types, in programming languages world, help to detect compile time errors and serve as useful tool for debugging and program optimization. Typing i.e. giving types to terms, is a prominent feature in applicative languages where function application is the central construct. In practice, however, assigning type to each and every expression in a functional program (or for that matter even object oriented program) is practically impossible.

Languages where a variable can have one and only one type is called a *monomorphic* language. For example, programming languages like Pascal. In contrast, *polymorphic languages*, like ML, have variable and values that can have more than one type. Consider the higher-order function `map` used in ML. A map function can be applied to list of integers and also to a list of booleans. We want `map` function to have polymorphic type. But despite the multitude of types an expression can have there is a canonical type that is the most general of these types. This type is called *principal type*. Thus, map has the following principal type;

$$\forall\alpha\forall\beta.(\alpha \to \beta) \to (\alpha list \to \beta list)$$

Type inference/reconstruction systems do precisely this. They are able to infer types the principal type of an expression from the source program. Functional languages have powerful type systems that can infer most of the types that an expression can have. One of the most popular type systems is Hindley-Milner type system, first mentioned in [Mil78] by Milner but discovered at the same time by Hindley [Hin69]. The general type reconstruction problem can be formulated as:

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

We focus on various algorithms used in type reconstruction in Hindley Milner type systems. Specifically, we look at substitution and unification based algorithms

- Algorithm $\mathcal{W}$
- Algorithm $\mathcal{J}$

- Algorithm $\mathcal{M}$

and we also look at a constraint based approach to type reconstruction.

- Wand's Algorithm

The main contributions of this paper are

- To bring more clarity into the understanding of the various type reconstruction algorithms.
- To implement some of the well-known algorithms on some non-trivial examples.
- To bring out the difference between substitution-based and constraint-based approaches.

The rest of this article is organized as follows. Section 2 gives an overview of the language syntax and related concepts. Section 3 does a brief literature survey on the various extensions and modifications to the prominent algorithms and describes the type inference algorithms. Section 4 discusses some examples for type inference. Finally Section 5 concludes with future extensions to this paper.

2 Preliminaries

In this discussion, we assume some familiarity with the notion of types and functional programming especially the lambda calculus. For an overview of type systems in programming languages readers can refer to Cardelli's paper [Car97] or the Pierce's book [Pie02].

Throughout our discussion we follow the usual conventions: arrow types associate to the right, function applications associates to the left and application binds more tightly than abstraction. We call our term language Core-ML - pure untyped lambda calculus extended with Milner's let. The syntax for Core-ML is shown in Figure 1.

$\Lambda ::= \quad x$	(Variables)
$\mid MN$	(Application)
$\mid \lambda x.M$	(Abstraction)
$\mid$ **let** x = M **in** N	(Milner-Let)

where M, N are Λ terms and x is a variable.

Figure 1: **Core-ML: Term Language syntax**

An interesting aspect of this language is that lambda bound variables are monomorphic whereas let bound variables are polymorphic. For example,

1. λ m → **let** f = $\lambda x.x$
2. **in let** y = f True
3. (f 3) + m

Here, f has a polymorphic type. It has a *bool* → *bool* when its applied to *true* (line 2) but has type *int* → *int* when its applied to 3 (line 3). Now consider the type of m it has always got to be *int*. In fact, the type of the expression is `int → int`.

Fig. 2 shows the type language for Core-ML. In Core-ML, types are either type variables or function types or type schemes.

Terms and types are related by assertions, $\Gamma \vdash t : \tau$, where t is a term and τ is a type and Γ is an environment that assigns types to free variables of t. A *derivation* of an assertion $\Gamma \vdash t : \tau$ is a finite sequence of assertions ending with $\Gamma \vdash t : \tau$. Every assertion in that sequence is either an instance of axiom or is derivable from the type rules. The type rules for Core-ML is shown in figure 3.

$$
\begin{array}{lll}
\tau ::= & \alpha & \text{(Type Variable)} \\
 & \mid \tau_1 \rightarrow \tau_2 & \text{(Function type)} \\
 & \mid \forall\vec{\alpha}.\tau & \text{(Type Scheme)}
\end{array}
$$

where $\vec{\alpha}$ denotes a sequence of
type variables $\alpha_1, ..., \alpha_k$ such that each $\alpha_i \in \text{FV}(\tau)$

Figure 2: **Core-ML: Type Language syntax**

$$\frac{}{\Gamma \cup \{x:\tau\} \vdash x:\tau} \text{ (Axiom-Var)}$$

$$\frac{\Gamma_x \cup \{x:\tau_1\} \vdash M:\tau_2}{\Gamma \vdash \lambda x.M : \tau_1 \rightarrow \tau_2} \text{ (Rule-Abs)}$$

$$\frac{\Gamma \vdash M:\tau_1 \rightarrow \tau_2 \qquad \Gamma \vdash N:\tau_1}{\Gamma \vdash MN:\tau_2} \text{ (Rule-App)}$$

$$\frac{\Gamma \vdash M:\tau_1 \qquad \Gamma,\ x:\tau_1 \vdash N:\tau_2}{\Gamma \vdash \textbf{let}\ x = M\ \textbf{in}\ N : B} \text{ (Rule-Let)}$$

$$\frac{\Gamma \vdash M:\tau}{\Gamma \vdash M:\forall\vec{\alpha}.\tau} \text{ (Rule-Gen)}$$

where $\alpha \notin FV(\Gamma) \wedge \alpha \in FV(\tau)$

$$\frac{\Gamma \vdash M:\forall\vec{\alpha}.\tau}{\Gamma \vdash M:\tau[\alpha_1 := \tau_1, ..., \alpha_n := \tau_n]} \text{ (Rule-Inst)}$$

Figure 3: **Core-ML: Typing rules**

2.1 Substitution and related concepts

A *type scheme* $\sigma = \forall\vec{\alpha}.\tau$ is a type where zero or more type variables are universally quantified. A *type environment*, denoted by Γ, maps variables to the corresponding type schemes. A *substitution* S is a mapping from type variables to types. We denote type environment where x is not in the domain of Γ by Γ_x. Let S_1, S_2 be two substitutions then the composition of substitution $S_1 \circ S_2$ is given as the result of applying substitution S_2 to S_1 and adding all the terms in S_2 that have not been substituted for in S_1. For example, if $S_1 = \{z \mapsto g(x,y)\}$ and $S_2 = \{x \mapsto A, y \mapsto B, w \mapsto C, z \mapsto D\}$ then $S_1 \circ S_2 = \{z \mapsto g(A,B), x \mapsto A, y \mapsto B, w \mapsto C\}$. Substitution composition is associative but non-commutative. For example, consider substitution S_1, S_2, S_3. Then $(S_1 \circ S_2) \circ S_3 = S_1 \circ (S_2 \circ S_3)$ but $S_1 \circ S_2 \neq S_2 \circ S_1$. A type τ' is a *substitution instance* of a type τ if and only if $\tau' = S\tau$. We assume that substitutions are *idempotent i.e.* for a substitution S and a type τ S (S τ) = S τ. *Substitution application* to a type environment Γ, denoted by $S\Gamma$, is simply S applied to each variable type pair. Two type terms t and t' are *unifiable* if there exists a substitution S such that $S(t) = S(t')$. In such a case S is called a *unifier*. A unifier S is the *most general unifier* if there is a substitution T such that for any other unifier W, $TS = W$.

2.2 Free and bound type variables

The set of *free type variables* of a type τ is denoted by FV(τ). We can extend our definition of free type variables for a type scheme, $\forall\vec{\alpha}.\tau$, as $\text{FV}(\forall\vec{\alpha}.\tau) = FV(\tau) - \vec{\alpha}$. *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. More formally,

$$generalize(\Gamma,\tau)\overset{\text{def}}{=}\forall\vec{\alpha}.\tau \text{ where } \vec{\alpha}=FV(\tau)-FV(\Gamma).$$

On the other hand, *instantiation* of a type scheme involves replacing the quantified variables by fresh type variables.

$$instantiate(\forall\vec{\alpha}.\tau)\overset{\text{def}}{=}\tau[\alpha_1 := \beta_1,\ldots,\alpha_n := \beta_n] \text{ where } \beta_1,\ldots\beta_n \text{ are fresh variables.}$$

2.3 Unification Algorithm

The unification algorithm is based on the Robinson's algorithm [Rob65] which is based on the resolution principle. Let C denote a constraint store. Let ρ be an initial list of substitutions (empty in our case). We will follow the conventions that α,β represent monomorphic types and τ,σ are meta variables and range over the set of types. The unification algorithm *unify* takes a constraint store and an initial substitution and returns a list of substitutions. The algorithm *unify* is formally defined in Figure 4. An excellent overview of unification is given by Knight [Kni89].

$unify$: `Constraint Store` $\rightarrow$ `Substitution List` $\rightarrow$ `Substitution List`

$$
\begin{array}{lcl}
unify\ (\alpha\overset{c}{=}\beta\cup C)\ \rho & = & \text{if } \alpha=\beta \text{ then } unify\ C\ \rho\\
unify\ (\alpha\overset{c}{=}\tau\cup C)\ \rho & = & \text{if } \alpha \text{ occurs in } \tau \text{ then raise Failure}\\
unify\ (\alpha\overset{c}{=}\tau\cup C)\ \rho & = & unify\ (C[\alpha:=\tau])(\rho[\alpha:=\tau]\cup\ \{\alpha\mapsto\ \tau\})\\
unify\ (\tau\overset{c}{=}\alpha\cup C)\ \rho & = & unify\ (\alpha\overset{c}{=}\tau\cup C)\ \rho\\
unify\ (\tau_1\rightarrow\tau_2\overset{c}{=}\tau_3\rightarrow\tau_4\cup C)\ \rho & = & unify\ (\tau_1\overset{c}{=}\tau_3,\tau_2\overset{c}{=}\tau_4\cup C)\ \rho\\
unify\ \emptyset\ \rho & = & \rho
\end{array}
$$

Figure 4: **Unification Algorithm**

2.4 Type inference versus type reconstruction

Type Inference is the problem of finding a type for a term within a given type system if any such type exists whereas type reconstruction is a much harder problem than type inference. A type reconstruction program takes an untyped term M as input, and finds an environment Γ and a type-annotated version M' of M, and a type τ such that τ is a type for M' with respect to environment Γ. For simplicity, we do not distinguish between type inference and type reconstruction in the rest of this paper.

2.5 Decidability of type inference

The introduction of polymorphism (Milner's Let) in a language makes the type inference much harder. P. Kanellakis et al.[PJ89] have shown that ML type reconstruction algorithm has the worst case complexity DEXPTIME and PSPACE-hard but on an average the algorithm runs in polynomial time. Later Mairson [Mai89] showed that in fact it is DEXPTIME-complete in the size of the nested LET statements. Figure 5 shows a classic example when type inference is much harder. A thorough discussion for type inference is by Tiuryn [Tiu90].

3 Type Reconstruction Algorithms

3.1 Substitution and Unification Based Approaches

A distinctive feature of all substitution and unification based algorithms is their complexity which makes it difficult to reason about formally. Yet they have been used in popular functional language implementations. In the next few sections, we describe, in detail, Algorithms $\mathcal{W}$, $\mathcal{J}$, and $\mathcal{M}$.

```
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 (λ z. z)
```

Figure 5: **A program which makes type inference hard**

3.1.1 Algorithm $\mathcal{W}$

Hindley [Hin69] first mentioned type schemes in the context of types for terms of combinator logic. He noticed that resolution [Rob65] was appropriate for unification of two type terms. The language used in his analysis was based on pure lambda calculus. Milner [Mil78] had implemented a type reconstruction algorithm in the context of LCF [GMM+78] metalanguage. He extended Hindley's result to include local declaration in the form of LET constructs and gave the first thorough account of not one but two type reconstruction algorithms, namely, Algorithm $\mathcal{W}$ and Algorithm $\mathcal{J}$. The paper also gave a first denotational proof of soundness. Damas and Milner [DM82] were the first to give the soundness result for Algorithm W. Later more extensions and variants of Algorithm $\mathcal{W}$ followed. Coppo [Cop80] describes a system which can give a most general type to self application i.e $\lambda x.xx$ by means of sequence of types i.e. a variable having two types in different occurrences of the same binding. Coppo also mentions that type reconstruction by Algorithm $\mathcal{W}$ is undecidable for a type scheme with sequence of types.

Cardelli [Car87] gives a detailed account of the algorithmic and constraint-based approach to type reconstruction and also supplies the code for type reconstruction but unfortunately the code is in Modula 2 - a language not widely used now. Laufer's PhD thesis [Lau92] and Mycroft [Myc84] have tried to give a much better account (including corrected typos) of Algorithm W. Mycroft extended Algorithm $\mathcal{W}$ to deal with more general type schemes associated with mutual recursive definitions. Nelson [Nel95] has used a variant of Algorithm $\mathcal{W}$ to show a sound and complete type reconstruction system for first order dependent types.

Another extension of Algorithm $\mathcal{W}$ has been for type inference based tools. Lackwit [OJ97] uses a variant of Algorithm $\mathcal{W}$ (not clear which one of the two proposed by Milner is used) to capture value flow. Algorithm $\mathcal{W}$ has a full machine checked proof of correctness and completeness [NN99] using Isabelle/HOL.

Given an initial environment Γ, and a term t, if Algorithm $\mathcal{W}$ succeeds it finds a substitution S and a type τ such that $S\Gamma \vdash t : \tau$. Formally, Algorithm $\mathcal{W}$ can be seen as a kind of mapping
`Type Environment` × `Expression` → `Substitution` × `Type`
but informally we can define the Algorithm $\mathcal{W}$ by induction on the structure of t such that $\mathcal{W}(\Gamma, t) = (S, \tau)$:

1. If t is just a variable, x, and bound to a type scheme $\forall\alpha_1 ... \alpha_n.\tau'$ then
   $S = id$ and $\tau = \tau'[\alpha_i := \beta_i]$ (where β_i is a new (fresh) type variable for all i).

2. if t is an application, MN, then
   let $\mathcal{W}(\Gamma, M) = (S_1, \tau_1)$
   and $\mathcal{W}(S_1\Gamma, N) = (S_2, \tau_2)$
   and β be a new type variable
   and let V = $unify\ (S_2\tau_1, \tau_2 \rightarrow \beta)$
   in $S = VS_2S_1$ and $\tau = V\beta$

3. if t is an abstraction, $\lambda x.M$, then
   let β be a new type variable and
   $\mathcal{W}(\Gamma_x \cup \{x : \beta\}, M) = (S_1, \tau_1)$
   in $S = S_1$ and $\tau = S_1\beta \rightarrow \tau_1$

4. if t is a let expression, **let** x = M **in** N, then
   let $\mathcal{W}(\Gamma, M) = (S_1, \tau_1)$

and $\mathcal{W}((\Gamma) :: [x := gen(S\Gamma, \tau)], M) = (S_2, \tau_2)$
where $gen(\Gamma, \tau) = \forall(FV(\tau)/FV(\Gamma)).\tau$
in $(S_2 S_1, \tau_1)$

3.1.2 Algorithm $\mathcal{J}$

Algorithm $\mathcal{J}$ was first mentioned by [Mil78] and is a simpler version of Algorithm $\mathcal{W}$. Surprisingly even though both, Algorithm $\mathcal{W}$ and Algorithm $\mathcal{J}$, are mentioned in the same paper, the former is the more popular one although there have been implementation of Algorithm $\mathcal{J}$ in Haskell [Jon99]. It differs from $\mathcal{W}$ in two ways. Firstly, substitutions are composed but applied only when it is essential. Secondly, it returns not the whole type assignment but only the type assigned to that term. Figure 6 informally defines Algorithm $\mathcal{J}$ by induction on the structure of t such that $\mathcal{J}(\Gamma, t, S) = (S, \tau)$

1. If t is just a variable x and bound to a type scheme $\forall\alpha_1...\alpha_n.\tau' \in \Gamma$
   then $S = id$ and $\tau = \tau'[\alpha_i = \beta_i]$
   (where β_i is a new (fresh) type variable).
   in (S, τ)
2. if t is an application MN then
   let $J(\Gamma, M, S) = (S, \tau_1)$
   and $J(\Gamma, N, S) = (S, \tau_2$
   and β be a new type variable
   and S $= unify\ (S\tau_1, S(\tau_2 \to \beta)) \circ S$
   in (S, β)
3. if t is an abstraction $\lambda x.M$ then
   let β be a new type variable and
   $J(\Gamma_x \cup \{x : \beta\}, M, S) = (S, \tau_1)$
   in $(S, \beta \to \tau_1)$
4. if t is a let expression **let** $x = M$**in** N then
   let $(\tau_1, S) = J(\Gamma, M, S)$ and $\sigma = J(gen(S\Gamma, S\tau_1))$
   where $gen(\Gamma, \tau) = \forall(FV(\tau)/FV(\Gamma)).\tau$
   in $J(\Gamma_x \cup \{x : \sigma\}, N, S)$

Figure 6: **Algorithm $\mathcal{J}$ for Core-ML**

3.1.3 Algorithm $\mathcal{M}$

This algorithm was first implemented in CamlLight [Ler93] (a lighter version of OCaml). A formal description of the algorithm and proof of its soundness and completeness was given by [LY98]. Lee and Yi [LY98] mention a type inference algorithm, named Algorithm $\mathcal{M}$, that has been used in early ML compilers [Ler93]. The algorithm is top-down (context-sensitive). They claim that it stops earlier than Algorithm $\mathcal{W}$ if the program is ill-typed. Algorithm $\mathcal{M}$ can be seen as a mapping:

$$Type\ Environment \times Expression \times Substitution \to Type$$

Algorithm $\mathcal{M}$ is informally defined as:

Let Γ be a type environment. Let τ be the type of the input expression.

1. If t is just a variable x and bound to a type scheme $\forall\alpha_1...\alpha_n.\tau'$ in Γ then
   $S = id$ and $\tau = \tau'[\alpha_i = \beta_i]$
   (where β_i is a new (fresh) type variable)
   in $unify(\tau, \tau')$.

2. if t is an application, MN, then
   let $S_1 = \mathcal{M}(\Gamma, M, \beta \rightarrow \tau)$
   and β be a new type variable
   and $S_2 = \mathcal{M}(S_1\Gamma, N, S_1\beta)$
   in S_2S_1.

3. if t is an abstraction, $\lambda x.M$, then
   let β_1, β_2 be two new type variables and
   let $S_1 = unify(\tau, \beta_1 \rightarrow \beta_2)$ and
   let $S_2 = \mathcal{M}(S_1\Gamma_x \cup \{x : S_1\beta_1\}, M, S_2\beta_2)$
   in S_2S_1.

4. if t is a let expression, **let** $x = M$ **in** N, then
   let $S_1 = \mathcal{M}(\Gamma, M, \beta)$
   (β is a new type variable)
   and $\sigma = \mathcal{M}(generalize((S_1\Gamma), S_1(\beta)), N, S_1\tau)$
   in S_2S_1.

3.2 Constraint Based Approach

The main idea behind constraint based approach to type reconstruction is to separate constraint generation from constraint solving. Algorithm $\mathcal{J}$, $\mathcal{W}$ and $\mathcal{M}$ correspond to different constraint solving strategies. We describe a classic constraint based algorithm and point out the various extensions and variations of the given algorithm.

3.2.1 Wand's Algorithm

Wand [Wan87] looked at the type inference problem as 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. This was the first successful attempt at looking at type inference as solving constraints. However, Milner's Let is not a part of their language syntax and therefore it is much simpler than the algorithms mentioned so far. Wand's algorithm can be easily extended to handle polymorphic let construct see [KC07] for details.

We formally define the algorithm below. Let G denote a set of goals. And E a set of equations. Then the main steps of the algorithm are:

Input. A term M_0 of Λ.

Initialization. Set $E = \emptyset$ and $G = \{(A_0, M_0, t_0)\}$.

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

This can be visualized as shown in Figure 7. The action table serves as a black box effectively separating the constraint solving from constraint generation. For Core-ML, the action table has the following semantics:

- **Case** (Γ, x, τ)**.** Generate the equation $\tau = \gamma$(x).
- **Case** (Γ, MN, τ)**.** Generate subgoals (Γ,M,$\tau_1 \rightarrow \tau$) and (Γ,N,τ_1).

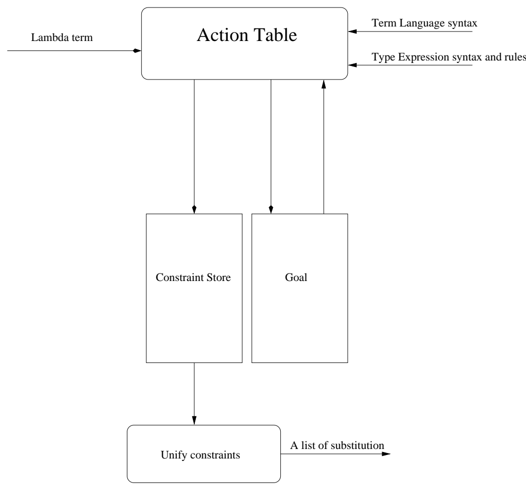

Figure 7: An overview of Wand's algorithm

- **Case $(\Gamma, \lambda x.M, \tau)$.** Generate equation $\tau = \tau_1 \rightarrow \tau_2$ and subgoal $(\Gamma_x \cup \{x : \tau_1\}, M, \tau_2)$.

Here's an example
$\{(\emptyset, \lambda x.\lambda y.\lambda z.xz(yz), \tau_0)\}; \{\}$
$\{((x : \tau_1), \lambda y.\lambda z.xz(yz), \tau_2)\}; \{\tau_0 = \tau_1 \rightarrow \tau_2\}$
$\{((x : \tau_1, y : \tau_3), \lambda z.xz(yz), \tau_4)\}; \{\tau_2 = \tau_3 \rightarrow \tau_4\}$
$\{((x : \tau_1, y : \tau_3, z : \tau_5), xz(yz), \tau_6)\}; \{\tau_4 = \tau_5 \rightarrow \tau_6\}$
$\{(\underline{((x : \tau_1, z : \tau_5), xz, \tau_7 \rightarrow t_6)}, ((y : \tau_3, z : \tau_5), yz, \tau_7))\}$
$\{(\underline{(x : \tau_1), x, \tau_8 \rightarrow (\tau_7 \rightarrow \tau_6))}, ((z : \tau_5), z, \tau_8), \underline{((y : \tau_3, z : \tau_5), yz, \tau_7)}\}$
$\{(((z : \tau_5), z, \tau_8), \underline{((y : \tau_3, z : \tau_5), yz, \tau_7))}\}; \{\tau_1 = \tau_8 \rightarrow \tau_7 \rightarrow \tau_6\}$
$\{((y : \tau_3, z : \tau_5), yz, \tau_7)\}; \{\tau_8 = \tau_5\}$
$\{((y : \tau_3), y, \tau_9 \rightarrow \tau_7), ((z : \tau_5), z, \tau_9)\}$
$\{((z : \tau_5), z, \tau_9)\}; \{\tau_9 \rightarrow \tau_7 = \tau_3\}$
$\emptyset; \{\tau_9 = \tau_5\}$
The constraint store has the following constraints when the algorithm terminates:
$\tau_0 = \tau_1 \rightarrow \tau_2$
$\tau_2 = \tau_3 \rightarrow \tau_4$
$\tau_4 = \tau_5 \rightarrow \tau_9$
$\tau_1 = \tau_8 \rightarrow \tau_7 \rightarrow \tau_9$
$\tau_8 = \tau_5$
$\tau_9 \rightarrow \tau_7 = \tau_3$
$\tau_9 = \tau_5$
After unifying the above constraints,
$\tau_0 = (\tau_5 \rightarrow \tau_7 \rightarrow \tau_6) \rightarrow (\tau_5 \rightarrow \tau_7) \rightarrow (\tau_5 \rightarrow \tau_6)$

3.2.2 HM(X) Framework

Sulzmann et. al [SOW97] came up with a framework, called HM(X), where constraint solving becomes a parameter to the type inference algorithm. An overview can be found of this framework and other constraint based approaches can be found in [Pot05].

A remarkable amount of work has focused on providing better error messages when a type inference algorithm fails. Wand [Wan86], McAdam [Mca00]. Heeren et al. [HHS02] derive a constraint graph from the syntax traversal and they claim that by varying how this graph is traversed and simplified they can simulate Algorithm $\mathcal{W}$ and Algorithm $\mathcal{M}$ as deterministic instances of their method.

4 Some examples

We have implemented Algorithm $\mathcal{W}, \mathcal{J}$ and Wand's algorithm. All the algorithms were tested on the examples given below. The proofs of the typings are also given to show the algorithm is sound.

4.1 Example 1 - Let Polymorphism

Here's an example taken from Damas and Milner [DM82].

let $i \;=\; \lambda\, x.\, x$ in $i\, i$

The correct type for this is:

$\forall\, 'a.\, 'a \;\rightarrow\; 'a$

The type derivation for the above type is shown in Figure 8

$$\dfrac{\dfrac{\dfrac{\dfrac{}{x : \alpha \;\vdash\; \rightarrow\; x : \alpha}\;\text{(Axiom-Var)}}{\vdash \lambda x.x : \alpha \;\rightarrow\; \alpha}\;\text{(Rule-Abs)}}{\vdash \lambda x.x : \forall\alpha.\alpha \;\rightarrow\; \alpha}\;\text{(Rule-Gen)} \qquad \dfrac{\dfrac{\dfrac{}{i : \forall\alpha.(\alpha \rightarrow \alpha) \vdash i : \forall\alpha.(\alpha \rightarrow \alpha)}\;\text{(Axiom-Var)}}{i : \forall\alpha.(\alpha \rightarrow \alpha) \vdash i : (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)}\;\text{(Rule-Inst)} \quad \dfrac{\dfrac{}{i : \forall\alpha.(\alpha \rightarrow \alpha) \vdash i : \forall\alpha.\alpha \rightarrow \alpha}\;\text{(Axiom-Var)}}{i : \forall\alpha.(\alpha \rightarrow \alpha) \vdash i : \alpha \rightarrow \alpha}\;\text{(Rule-Inst)}}{i : \forall\alpha.(\alpha \rightarrow \alpha) \vdash (ii) : \alpha \rightarrow \alpha}\;\text{(Rule-App)}}{\vdash let\ i = \lambda x.x\ in\ (ii) : \alpha \rightarrow \alpha}\;\text{(Rule-Let)}$$

Figure 8: **Type derivation for Example 1**

4.2 Example 2 - Compose operator

The familiar *compose* operator is defined as:

$\lambda\, f.\, \lambda\, g.\, \lambda\, x.\; f(g(x))$

The correct type for the above function is correctly reconstructed as:

$\forall\, 'a.\, \forall\, 'b. \forall\, 'c.\, ('c \;\rightarrow\; 'b) \;\rightarrow\; (('a \;\rightarrow\; 'c) \;\rightarrow\; ('a \;\rightarrow\; 'b))$

The type derivation for the above type is shown in Figure 9.

$$\dfrac{\dfrac{\dfrac{\dfrac{\dfrac{}{f : c \rightarrow b, g : a \rightarrow c, x : a \vdash f : c \rightarrow b}\;\text{(Axiom-Var)} \qquad \dfrac{\dfrac{}{f : c \rightarrow b, g : a \rightarrow c, x : a \vdash g : a \rightarrow c}\;\text{(Axiom-Var)} \quad \dfrac{}{f : c \rightarrow b, g : a \rightarrow c, x : a \vdash x : a}\;\text{(Axiom-Var)}}{f : c \rightarrow b, g : a \rightarrow c, x : a \vdash g(x) : c}\;\text{(Rule-App)}}{f : c \rightarrow b, g : a \rightarrow c, x : a \vdash f(g(x)) : b}\;\text{(Rule-App)}}{f : c \rightarrow b, g : a \rightarrow c \vdash \lambda x.f(g(x)) : (a \rightarrow b)}\;\text{(Rule-Abs)}}{f : c \rightarrow b \vdash \lambda g.\lambda x.f(g(x)) : (a \rightarrow c) \rightarrow (a \rightarrow b)}\;\text{(Rule-Abs)}}{\vdash \lambda f.\lambda g.\lambda x.f(g(x)) : (c \rightarrow b) \rightarrow ((a \rightarrow c) \rightarrow (a \rightarrow b))}\;\text{(Rule-Abs)}$$

Figure 9: **Type derivation for `compose` function**

4.3 Example 3 - S operator

Consider the S operator defined as:

$\lambda\, f.\, \lambda\, g.\, \lambda\, x.\; (fx)(gx)$

Again, the type for the above term is correctly reconstructed as:

$\forall\, 'a. \forall\, 'b. \forall\, 'c.(('a \;\rightarrow\; ('b \;\rightarrow\; 'c)) \;\rightarrow\; (('a \;\rightarrow\; 'b) \;\rightarrow\; ('a \;\rightarrow\; 'c)))$

And he derivation for the above type is shown in Figure 10

$$\dfrac{}{f :'a \to ('b \to' c), x :'a \vdash f :'a \to ('b \to' c)}\text{(Axiom-Var)} \quad \dfrac{}{x :'a \vdash x :'a}\text{(Axiom-Var)}$$

$$f :'a \to ('b \to' c), g :'a \to' b, x :'a \vdash f(x) :'b \to' c \quad \text{(Rule-App)}$$

$$\dfrac{}{g :'a \to' b, x :'a \vdash g :'a \to' b}\text{(Axiom-Var)} \quad \dfrac{}{x :'a \vdash x :'a}\text{(Axiom-}$$

$$f :'a \to ('b \to' c), g :'a \to' b, x :'a \vdash g(x) :'b \quad \text{(Rule-A}$$

$$f :'a \to ('b \to' c), g :'a \to' b, x :'a \vdash (f(x)g(x)) :'c \quad \text{(Rule-App)}$$

$$f :'a \to ('b \to' c), g :'a \to' b \vdash \lambda x.(f(x)g(x)) : ('a \to' c) \quad \text{(Rule-Abs)}$$

$$f :'a \to ('b \to' c) \vdash \lambda g.\lambda x.(f(x)g(x)) : ('a \to' b) \to ('a \to' c) \quad \text{(Rule-Abs)}$$

$$\vdash \lambda f.\lambda g.\lambda x.(f(x)g(x)) : ('a \to ('b \to' c)) \to (('a \to' b) \to ('a \to' c)) \quad \text{(Rule-Abs)}$$

Figure 10: **Type derivation for S operator**

4.4 Example 4 - Y operator

Here's an example where type reconstruction fails because of the failure in unifying terms: The Y operator is defined as:
$\lambda\ f.\ (\lambda\ x.(f\ (xx)))(\lambda\ x.(f\ (xx))))$

The algorithm fails because it is unable to unify an arrow type $'a \to' b$ with $'a$ due to the failure of the occurs check. Interestingly, Mario [Cop80] avoids this problem by using sequence of types.

5 Implementation Details and Future work

The entire code has been implemented in OCaml and is available online at `http://www.cs.uwyo.edu/~skothari/research/typeinference/HM.html`. We would like to test our algorithm on more complicated examples. As of now, we haven't included many of the interesting but complicated language features like

- Recursive definitions
- Recursive types
- GADT (Generalized Algebraic Data Types)

References

[Car87] Luca Cardelli. Basic polymorphic typechecking. *Science of computer programming*, 8(Issue 2):147–172, April 1987.

[Car97] Luca Cardelli. *Type Systems*, chapter 103. CRC Press, Boca Raton, FL, 1997.

[Cop80] Mario Coppo. An extended polymorphic type system for applicative languages. In Piotr Dembinski, editor, *MFCS'80*, volume 88 of *Lecture Notes in Computer Science*, pages 194–204. Springer-Verlag, 1980.

[DM82] L. Damas and R. Milner. Principal type schemes for functional languages. *Proceedings of the 9th ACM Symposium on Principles of Programming Languages*, pages 157–168, 1982.

[GMM+78] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadsworth. A metalanguage for interactive proof in lcf. In *Proceedings of the 5th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, pages 119–130. ACM Press, 1978.

[HHS02] Bastiaan Heeren, Jurriaan Hage, and Doaitse Swierstra. Generalizing hindley-milner type inference algorithms. Technical Report UU-CS-2002-031, Institute of Information and Computing Science, University Utrecht, Netherlands, July 2002. Technical Report.

[Hin69] J. R. Hindley. The principal type-scheme of an object in combinatory logic. *Trans. American Math. Soc*, 146:29–60, 1969.

[Jon99] Mark P. Jones. Typing haskell in haskell. In *Proceedings of the 1999 Haskell Workshop*, October 1999.

[KC07] Sunil Kothari and James L. Caldwell. On extending wand's algorithm to handle polymorphic let. Technical report, University of Wyoming, 2007.

[Kni89] Kevin Knight. Unification: A multidiscplinary survey. *ACM Computing Surveys*, 21:93–124, 1989.

[Lau92] Konstantin Laufer. *Polymorphic type inference and abstract data types*. PhD thesis, New York University, 1992.

[Ler93] Xavier Leroy. *The caml light system, release 0.6*. Institut National de Recherche en Informatique et en Automatique, 1993.

[LY98] Oukseh Lee and Kwangkeun Yi. Proofs about a folklore let-polymorphic type inference algorithm. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 20(4):707–723, 1998.

[Mai89] Harry 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.

[Mca00] Bruce J. Mcadam. Generalising techniques for type debugging. In *SFP '99: Selected papers from the 1st Scottish Functional Programming Workshop (SFP99)*, pages 50–58. Intellect Books, 2000.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of computer and system sciences*, pages 348–375, 1978.

[Myc84] Alan Mycroft. Polymorphic type schemes and recursive definitions. *Lecture Notes in Computer Science*, 167:217–228, 1984.

[Nel95] Neal Nelson. *Type Inference and Reconstruction for First Order Dependent Types*. PhD thesis, Oregon Graduate Institute of Science and Technology, 1995.

[NN99] Wolfgang Naraschewski and Tobias Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *Journal of Automated Reasoning*, 23(3-4):299–318, 1999.

[OJ97] Robert O'Callahan and Daniel Jackson. Lackwit: a program understanding tool based on type inference. In *Proceedings of the 19th international conference on Software engineering*, pages 338–348, 1997.

[Pie02] Benjamin 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.

[Pot05] Francois Pottier. A modern eye on type inference. 2005. Available at.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM (JACM)*, 12:23–41, 1965.

[SOW97] Martin Sulzmann, Martin Odersky, and Martin Wehr. Type inference with constrained types. In *Fourth International Workshop on Foundations of Object-Oriented Programming (FOOL 4)*, 1997.

[Tiu90] Jerzy Tiuryn. Type inference problems: A survey. In Banska Bystrica, editor, *Mathematical Foundations of Computer Science*, volume 452 of *Lecture Notes in Computer Science*, pages 105–120. Springer-Verlag, 1990.

[Wan86] Mitchell Wand. Finding the source of type errors. In *Conference Record of the Thirteenth Annual Symposium on Principles of Programming Languages*, pages 38–43. ACM, 1986.

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