

# Toward a machine-certified correctness proof of Wand’s type reconstruction algorithm

Sunil Kothari  
 Department of Computer Science  
 University of Wyoming  
 Laramie, USA  
 skothari@uwyo.edu

James L. Caldwell  
 Department of Computer Science  
 University of Wyoming  
 Laramie, USA  
 jlc@cs.uwyo.edu

## ABSTRACT

Although there are machine-certified proofs of correctness of Alg. W and Alg. J, we know of no machine-checked correctness proof of Wand’s type reconstruction algorithm. We give here a brief description of our progress on a machine-certified proof of correctness of Wand’s algorithm. Correctness is given in terms of completeness and soundness with respect to the Hindley-Milner type system. Also, we have verified the MGU axioms using the Coq’s finite map library.

## 1. INTRODUCTION

Type reconstruction algorithms can be broadly categorized into two categories: *substitution-based* and *constraint-based*. This categorization is based on whether the algorithms generate and solve constraints intermittently (substitution-based) or separately (constraint-based). There is now a trend toward constraint-based algorithms/frameworks [7, 3, 5, 9].

Although there are various machine-checked correctness proofs of Alg. W [6, 2, 8], we know of no previous machine-checked correctness proof of any constraint-based algorithms. We are working on a machine-checked correctness proof of Wand’s algorithm [9] in Coq [1]. Our current work is a step toward machine-certified proof of correctness of our extension to Wand’s algorithm to polymorphic let [5], which is a variant of the one presented in [7, 3].

We have adopted the following conventions in this paper: atomic types (of the form  $\text{Tvar } x$ ) are denoted by  $\alpha, \beta, \alpha'$  etc.; compound types by  $\tau, \tau', \tau_1$  etc.; substitutions by  $\sigma, \sigma', \sigma_1$  etc.

We consider the language of pure untyped lambda terms.

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

where  $x \in \text{Var}$  and  $M, N \in \Lambda$ .

The types for the terms of the above language is given by the following grammar:

$$\tau ::= \text{Tvar } x \mid \tau_1 \rightarrow \tau_2$$

where  $x \in \mathbb{N}$  and  $\tau_1, \tau_2 \in \tau$ .

Constraints are of the form  $\tau_1 \stackrel{\text{def}}{=} \tau_2$ , where  $\tau_1, \tau_2 \in \tau$ . A *type environment* is a list of pairs of type  $\text{Var} \times \tau$ . A *substitution* is a finite function from  $\mathbb{N}$  to types. Substitution application to a type is defined as:

$$\begin{aligned} \sigma(\text{Tvar } (n)) &\stackrel{\text{def}}{=} \text{if } \langle n, \tau \rangle \in \sigma \text{ then } \tau \text{ else } \text{Tvar}(n) \\ \sigma(\tau_1 \rightarrow \tau_2) &\stackrel{\text{def}}{=} \sigma(\tau_1) \rightarrow \sigma(\tau_2) \end{aligned}$$

Substitution application to a constraint and type environment are defined similarly:

$$\begin{aligned} \sigma(\tau_1 \stackrel{\text{def}}{=} \tau_2) &\stackrel{\text{def}}{=} \sigma(\tau_1) \stackrel{\text{def}}{=} \sigma(\tau_2) \\ \sigma(\langle x, \tau \rangle :: \Gamma) &\stackrel{\text{def}}{=} \langle x, \sigma(\tau) \rangle :: \sigma(\Gamma) \end{aligned}$$

We write  $\sigma \models (\tau_1 \stackrel{\text{def}}{=} \tau_2)$ , if  $\sigma(\tau_1) = \sigma(\tau_2)$ , i.e.  $\sigma$  is a *unifier* of  $\tau_1$  and  $\tau_2$ . We extend this notion to a set of constraints and we write  $\sigma \models \mathbb{C}$  if and only if for every  $c \in \mathbb{C}$ ,  $\sigma \models c$ . A unifier  $\sigma$  is the *most general unifier* (MGU) if there is a substitution  $\sigma'$  such that for any other unifier  $\sigma''$ ,  $\sigma \circ \sigma' \approx \sigma''$ , where substitution composition ( $\circ$ ) is defined as:

$$\sigma \circ \sigma'(\tau) \stackrel{\text{def}}{=} \sigma'(\sigma(\tau))$$

and extensionality on substitutions ( $\approx$ ) is defined as:

$$\sigma \approx \sigma' \stackrel{\text{def}}{=} \forall \alpha. \sigma(\alpha) = \sigma'(\alpha)$$

One of the most important issues in machine checked correctness proofs of the type reconstruction algorithms is the representation used for substitutions and most general unifiers. To a large extent, this representation determines the kind of reasoning needed for substitutions. The type reconstruction verification literature has substitutions represented as normal functions, list of pairs, and a set of pairs. We represent substitution as finite functions and use the Coq finite map library *Coq.FSets.FMapInterface*, which provides an axiomatic presentation of finite maps and a number of supporting implementations. However, the finite map library (ver. 8.1.pl3) that we used does not provide an induction principle and forward reasoning is often needed for reasoning about some simple lemmas. Despite these limitations, the library is powerful and expressive.

## 2. CORRECTNESS PROOF OVERVIEW

Correctness is given in term of completeness and soundness with respect to the Hindley-Milner type system given below in Table 1. Judgments are denoted by  $\Gamma \triangleright M : \tau$  and can be read as “in the type environment  $\Gamma$ ,  $M$  has type  $\tau$ ”. We write  $\vdash \Gamma \triangleright M : \tau$  to denote that judgment  $\Gamma \triangleright M : \tau$  has a derivation in the Hindley Milner type system.

$\frac{\langle x, \tau \rangle \in \Gamma \text{ is the leftmost binding}}{\Gamma \triangleright x : \tau}$	(HM-Var)
$\frac{\langle x, \tau \rangle :: \Gamma \triangleright M : \tau'}{\Gamma \triangleright \lambda x.M : \tau \rightarrow \tau'}$	(HM-Abs)
$\frac{\Gamma \triangleright M : \tau' \rightarrow \tau \quad \Gamma \triangleright N : \tau'}{\Gamma \triangleright MN : \tau}$	(HM-App)

Table 1: Modified Hindley-Milner type system

$\frac{\text{search\_type\_env}(x, \Gamma) = \text{Some } \tau}{\text{Wand}(\Gamma, x, n_0) = (\text{Some } \{\text{Tvar}(n_0) \stackrel{e}{=} \tau\}, n_0 + 1)}$	(W-Var)
$\frac{\text{Wand}((x : \text{Tvar}(n_0 + 1)) :: \Gamma), M, n_0 + 2) = (\text{Some } \mathbb{C}, n_1)}{\text{Wand}(\Gamma, \lambda x. M, n_0) = (\text{Some } \{\text{Tvar}(n_0) \stackrel{e}{=} \text{Tvar}(n_0 + 1) \rightarrow \text{Tvar}(n_0 + 2)\} \cup \mathbb{C}, n_1)}$	(W-Abs)
$\frac{\text{Wand}(\Gamma, M, n_0 + 1) = (\text{Some } \mathbb{C}', n_1) \quad \text{Wand}(\Gamma, N, n_1) = (\text{Some } \mathbb{C}'', n_2)}{\text{Wand}(\Gamma, MN, n_0) = (\text{Some } \{\text{Tvar}(n_0 + 1) \stackrel{e}{=} \text{Tvar}(n_1) \rightarrow \text{Tvar}(n_0)\} \cup \mathbb{C}' \cup \mathbb{C}'', n_2)}$	(W-App)

**Table 2: Rule-based description of Wand’s algorithm**

Our experience shows that the above presentation of type system is easier to reason than the standard representation of Hindley-Milner type systems, where the existing binding of  $x$  is removed from the type environment in the rule HM-Abs. By considering the leftmost binding in the HM-Var rule, we get the same effect as the traditional Hindley-Milner type system. In hindsight, we might have used finite maps to represent type environments as well.

Wand’s original description of the algorithm does not easily lend itself to formalization. Our efforts to formalize it lead to a number of changes in its presentation. The modified version of Wand’s algorithm is shown in Table 2 and the changes are described below:

- i) Failure has to be made explicit. We use Coq’s `option` type to represent failure. For example, in the rule W-Var, if the function `search_typ_env` is unable to find a binding, it returns a `None`, otherwise it returns `Some  $\tau$` , where  $\tau$  is the binding of  $x$  in  $\Gamma$ . This failure to find a binding is reflected in the constraint generation too. All calls to `Wand` now result in either `Some  $\mathbb{C}$`  or `None`, meaning that constraint generation might fail, unlike Wand’s algorithm.
- ii) Freshness is now explicit - a freshness counter is threaded through the entire algorithm to keep track of the fresh type variables introduced so far. The freshness counter also serves as the initial type of a term - by lifting the counter to a type by applying the constructor `Tvar`, unlike Wand’s original description where a type is passed as an argument.
- iii) In W-App rule, an additional constraint is added to the constraints generated by the recursive calls, unlike Wand’s original description. This corresponds to a strengthened induction hypothesis.

With the changes above, Wand algorithm will always return a principal type, if the term is typable and the initial type environment is empty. The modified description helps us to account for both principal as well as non-principal Hindley-Milner derivations in the completeness theorem (mentioned below).

In Wand’s original paper, the correctness of his algorithm is stated as an invariant preservation in all steps of the algorithm. Our soundness and completeness theorem are stated rather differently:

**THEOREM 1 (SOUNDNESS).**

$$\begin{aligned} & \forall \Gamma, \forall M, \forall \sigma, \forall n, \forall n', \forall \mathbb{C}. \\ & \text{Wand}(\Gamma, M, n) = (\text{Some } \mathbb{C}, n') \wedge \text{unify } \mathbb{C} = \text{Some } \sigma \\ & \Rightarrow \vdash \sigma(\Gamma) \triangleright_{HM} M : \sigma(\tau) \end{aligned}$$

The completeness theorem is more involved, and also involves a notion of freshness of type variables (with respect to the type environment):

**THEOREM 2 (COMPLETENESS).**

$$\begin{aligned} & \forall \Gamma', \forall M, \forall \tau. \\ & \vdash \Gamma' \triangleright_{HM} M : \tau \\ & \Rightarrow \forall \Gamma, \forall n. (\exists \sigma. \sigma(\Gamma) = \Gamma') \wedge \text{fresh\_env } n \Gamma \\ & \Rightarrow \forall \mathbb{C}, \forall n'. \text{Wand}(\Gamma, M, n) = (\text{Some } \mathbb{C}, n') \wedge \\ & \quad \exists \sigma'. \text{unify } \mathbb{C} = \text{Some } \sigma' \\ & \Rightarrow \exists \sigma''. (\sigma' \circ \sigma'')(\text{Tvar}(n)) = \tau \wedge \\ & \quad (\sigma' \circ \sigma'')(\Gamma) = \Gamma' \end{aligned}$$

The `unify` used in both the theorems above refers to the first-order unification algorithm. Existing literature on machine-certified correctness proofs of type reconstruction algorithms have axiomatized the behavior of unification algorithm as a set of four axioms. We have generalized the standard presentation of those axioms to specify the MGU of a list of equational constraints and we have formally verified that the unification algorithm does satisfies those axioms [4].

### 3. CURRENT STATUS AND FUTURE WORK

The entire exercise has currently exceeded 8000 lines of Coq specification and tactics. So far we have proved the soundness. Interestingly, the concept of freshness is not needed in the soundness. The completeness proof turns out to be much more complicated to reason about. We are sure about the proof argument, but it is still a work-in-progress. We believe the proofs of MGU axioms will come in handy. As of now, the types do not require binders. Therefore, binding has not been an issue. This will change when we do the correctness proof of an extension of Wand’s type reconstruction algorithm, since polymorphic-let introduces a universally quantified type constructor to the language of types. Other important steps in the correctness proof of our extension are a formalization of the replacement lemma [10] and verification of the *ptol* transformation [5], a type and value preserving desugaring of polymorphic let.

### 4. REFERENCES

- [1] T. Coq development team. *The Coq proof assistant reference manual*. INRIA, LogiCal Project, 2007. Version 8.1.3.

- [2] C. Dubois and V. M. Morain. Certification of a type inference tool for ML: Damas–milner within Coq. *J. Autom. Reason.*, 23(3):319–346, 1999.
- [3] B. Heeren. *Top Quality Type Error Messages*. PhD thesis, Universiteit Utrecht, 2005.
- [4] S. Kothari and J. Caldwell. A machine checked model of MGU axioms: applications of finite maps and functional induction. 2009. To be presented at UNIF’09.
- [5] S. Kothari and J. L. Caldwell. On Extending Wand’s Type Reconstruction Algorithm to Handle Polymorphic Let. *Local Proceedings of the Fourth Conference on Computability in Europe*, 15(5):795–825, June 2008.
- [6] W. Naraschewski and T. Nipkow. Type Inference Verified: Algorithm W in Isabelle/HOL. *J. Autom. Reason.*, 23(3):299–318, 1999.
- [7] F. Pottier and D. Rémy. The essence of ML type inference. In B. C. Pierce, editor, *Advanced Topics in Types and Programming Languages*, chapter 10, pages 389–489. MIT Press, 2005.
- [8] C. Urban and T. Nipkow. *From Semantics to Computer Science*, chapter Nominal verification of algorithm W. Cambridge University Press, Not yet published 2009.
- [9] M. Wand. A Simple Algorithm and Proof for Type Inference. *Fundamenta Informaticae*, 10:115–122, 1987.
- [10] A. K. Wright and M. Felleisen. A Syntactic Approach to Type Soundness. *Information and Computation*, 115(1):38–94, 1994.