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, the correctness proof of Wand's type reconstruction has never been machine checked. We give here a brief description of our attempt at machine-certified proof of correctness of Wand's algorithm. The correctness is essentially given in terms of completeness and soundness with respect to the Hindley-Milner type system. Unlike other works, we do not axiomatize MGUs.

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 formalizations of the correctness of Alg. W [6, 2, 8], we know of no previous formalizations of any constraint-based algorithms. This is the first attempt at machine-checked proof of correctness 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].

We have adopted the following conventions in this paper: atomic types (of the form $\mathsf{Tvar}\ x$) are denoted by α, β, α' etc.; compound types by τ, τ', τ_1 etc.; substitutions by $\sigma, \sigma', \sigma_1$ etc.

We consider the language of pure untyped lambda terms given by the following grammar:

$$\Lambda ::= \quad x \mid MN \quad \mid \lambda x.M$$
$$\text{where } x \in \mathit{Var} \text{ and } M, N \in \Lambda.$$

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

$$\tau ::= \quad \mathsf{Tvar}\ x \mid \quad \tau_1 \rightarrow \tau_2$$
$$\text{where } x \in \mathbb{N} \text{ and } \tau_1, \tau_2 \in \tau.$$

And the constraints are given by the following grammar:

$$\mathbb{C} ::= \tau \overset{e}{=} \tau$$
$$\text{where } \tau_1, \tau_2 \in \tau.$$

A *type environment* is a list of pairs of type $\mathit{Var} \times \tau$. A *substitution* is a finite function from $\mathbb{N}$ to types. Application of a substitution to a type is defined as:

$$\sigma\ (\mathsf{Tvar}\ (n)) \quad \overset{def}{=} \quad \mathit{if}\ \langle n, \tau \rangle\ \in\ \sigma\ \mathit{then}\ \tau\ \mathit{else}\ \mathsf{Tvar}(n)$$
$$\sigma\ (\tau_1 \rightarrow \tau_2) \quad \overset{def}{=} \quad \sigma(\tau_1) \rightarrow \sigma(\tau_2)$$

Thus, if a variable x is not in the domain of the substitution, the substitution application lifts that variable to $\mathsf{Tvar}(n)$. Application of a substitution to a constraint and type environment are defined similarly:

$$\sigma(\tau_1 \overset{e}{=} \tau_2) \quad \overset{def}{=} \quad \sigma(\tau_1) \overset{e}{=} \sigma(\tau_2)$$
$$\sigma((x, \tau) :: \Gamma) \quad \overset{def}{=} \quad (x, \sigma(\tau)) :: \sigma(\Gamma)$$

Two type terms τ_1 and τ_2 are *unifiable* if there exists a substitution σ such that $\sigma(\tau_1) = \sigma(\tau_2)$. In such a case, σ is called a *unifier*. More formally, we denote solvability of a constraint by $\models$ (read "solves"). We write $\sigma \models (\tau_1 \overset{e}{=} \tau_2)$, if $\sigma(\tau_1) = \sigma(\tau_2)$. We extend the solvability notion to a list of constraints and we write $\sigma \models \mathbb{C}$ if and only if for every $c \in \mathbb{C}$, $\sigma \models c$. A unifier σ is the *most general unifier* (MGU) if there is a substitution σ' such that for any other unifier σ'', $\sigma \circ \sigma' \approx \sigma''$, where substitution composition ($\circ$) is defined as:

$$\sigma \circ \sigma' \overset{def}{=} \lambda\tau.\sigma'(\sigma(\tau))$$

and squiggle extensionality($\approx$) is defined as:

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

One of the most important issue 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 as 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. The Coq 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

The correctness is essentially given in term of completeness and soundness with respect to the Hindley-Milner type system. But first we describe the rules in the HM type system.

We use the syntax-directed formulation of the Hindley Milner type system mentioned in Table 1. 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. Though not shown here but in the rule HM-Var, we consider the most recent binding in the environment as the binding for the identifier.

$$\frac{\mathsf{search\_type\_env}(x,\Gamma) = \tau}{\mathsf{Wand}(\Gamma, x, n_0) = (\{\mathsf{Tvar}(n_0) \overset{e}{=} \tau\}, n_0 + 1)}$$

$$\frac{\mathsf{Wand}(((x : \mathsf{Tvar}(n_0+1)) :: \Gamma), M, n_0+2) = (\mathbb{C}, n_1)}{\mathsf{Wand}(\Gamma, \lambda x.M, n_0) = (\{\mathsf{Tvar}(n_0) \overset{e}{=} \mathsf{Tvar}(n_0+1) \to \mathsf{Tvar}(n_0+2)\} \cup \mathbb{C}, n_1)}$$

$$\frac{\mathsf{Wand}(\Gamma, M, n_0+1) = (\mathbb{C}', n_1) \qquad \mathsf{Wand}(\Gamma, N, n_1) = (\mathbb{C}'', n_2)}{\mathsf{Wand}(\Gamma, MN, n_0) = (\{\mathsf{Tvar}(n_0+1) \overset{e}{=} \mathsf{Tvar}(n_0) \to \mathsf{Tvar}(n_1)\} \cup \mathbb{C}' \cup \mathbb{C}'', n_2)}$$

Table 2: Wand's algorithm description by cases

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

$$\frac{\{x : \tau\} :: \Gamma \ \triangleright \ M : \tau'}{\Gamma \ \triangleright \ \lambda x.M \ : \ \tau \to \tau'} \qquad \text{(HM-Abs)}$$

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

Table 1: Modified Hindley-Milner type system

Wand's original description of the algorithm does not easily lend itself to formalization (induction hypothesis is not quite right). We had to make many changes to the original description. The modified version of Wand's algorithm is shown in Table 2. The above description ensures that 1) Starting with an empty type environment, the algorithm always makes the least assumption about the type of an identifier. But if the type environment is not empty then those assumptions are not discarded. 2) Freshness is now explicit - a freshness counter is threaded through the entire algorithm to keep track of the variables introduced so far. Note that the application case results in a constraint apart from the constraints generated by the recursive calls, unlike Wand's original description.

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 (w.r.t. the Hindley Milner type system as shown in Table 1) are stated rather differently:

THEOREM 1 (SOUNDNESS). $\forall\Gamma, \forall M, \forall\sigma, \forall n\ n', \forall\mathbb{C}.$ $\mathsf{Wand}(\Gamma, M, n) = (\mathsf{Some}\ \mathbb{C},\ n') \ \wedge\ \mathsf{unify}\ \mathbb{C} = \mathsf{Some}\ \sigma \Rightarrow$ $\vdash \sigma(\Gamma) \triangleright_{HM} M : \sigma(\tau)$

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). $\forall\Gamma', \forall M, \forall\tau.$ $\vdash \Gamma' \triangleright_{HM} M : \tau \Rightarrow \forall\Gamma, \forall n, \forall\tau_1, (\exists\sigma.\ \sigma(\Gamma) = \Gamma') \wedge$ $\mathsf{fresh\_env}\ n\ \Gamma \Rightarrow \forall\mathbb{C}, \forall n', \mathsf{Wand}(\Gamma, M, n) = (\mathsf{Some}\ \mathbb{C}, n') \Rightarrow$ $\exists\sigma'.\mathsf{Some}\ \sigma' = \mathsf{unify}\ \mathbb{C} \Rightarrow \exists\sigma''.\sigma''(\sigma'(\mathsf{Tvar}\ n)) = \tau \ \wedge$ $\sigma''(\sigma'(\Gamma)) = \Gamma'$

A little note about the notations used in the two statements above: Coq provides an *option* type (also available in OCaml as a standard data type) to allow for failure.

```
Inductive option (A : Set) : Set := Some (_ : A) | None.
```

We use the option None to indicate failure and in the result Some(σ), σ is the resulting substitution. Similarly, Wand's algorithm may fail if the search_typ_env is unable to find a binding. In Wand's original description, the failure aspect of the unification or the constraint generation is left implicit.

The unify used in both the theorems above refers to the first-order unification algorithm. Existing literature on machine checked proofs of correctness of type reconstruction algorithm 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 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 remains incomplete. We believe the proofs of MGU axioms will come handy. As of now, the types do not require binders. Therefore, binding has been less of an issue. This will change when we do the correctness proof of an extension of Wand's type reconstruction algorithm. Other important steps in the correctness proof of our extension are a formalization of the replacement lemma [10] and verifying that the *ptol* transformation, a polymorphic let desugaring that preserves type and value, is correct.

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, Universitiet Utrecht, 2005.

[4] S. Kothari and J. Caldwell. A machine checked model of MGU axioms: applications of finite maps and functional induction. 2009. Submitted to 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.