Type Inference for Partial Types is Decidable

Patrick M. O'Keefe

ICAD, Inc.
201 Broadway
Cambridge, MA 02139
pmo@icad.com

Mitchell Wand*

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA
wand@corwin.ccs.northeastern.edu

Abstract

The type inference problem for partial types, introduced by Thatte [15], is the problem of deducing types under a subtype relation with a largest element Ω and closed under the usual antimonotonic rule for function types. We show that this problem is decidable by reducing it to a satisfiability problem for type expressions over this partial order and giving an algorithm for the satisfiability problem. The satisfiability problem is harder than the one conventionally given because comparable types may have radically different shapes.

1 Introduction

Statically-typed languages are desirable for many reasons, but they are often more restrictive than dynamically-typed languages. In particular, it is desirable to allow strongly-typed languages to have "holes" in the type structure, so that portions of the program that are not fully understood may be written using dynamic typing. There have been several proposals for creating such holes, such as [5, 15, 17]. Typically, one gives the result of such an untyped computation a special type, **untyped**. Such a value can be passed as an ordinary value, but is not manipulable except by a polymorphic procedure, such as `print` [17]. Thatte [15] called this *partial* type inference.

The addition of a type **untyped** allows several different kinds of flexibility. It allows portions of a program to escape the scrutiny of the type-checker [17]; it allows for heterogeneous lists and persistent data [15]; and it can also be used to facilitate binding-time analysis or analysis of type errors [5]. It also serves as a basis for dealing with the "don't care" types for records in [14].

Here we consider the problem of type inference for a language with a type **untyped**. In [15], Thatte proposed a type system for dealing with this problem. His idea was to treat this as a subtyping problem, with the inequalities between types generated by

*Work supported by the National Science Foundation and DARPA under grants CCR-9002253 and CCR-9014603.

$t \leq \Omega$ for all types t; we follow Thatte by using Ω to denote the type **untyped**. By well-known reductions [18, 16], the type inference problem for untyped lambda-terms reduces to determining the satisfiability of inequalities over this system. Thatte presented the system and showed how to determine the satisfiability of single inequalities. Here we show how to solve general sets of inequalities, thus solving the partial type inference problem.

2 The Formal System

2.1 Types

The set of types is defined by the following grammar:

$$\langle \text{type} \rangle ::= \Omega \mid \langle \text{type} \rangle \rightarrow \langle \text{type} \rangle$$

Note that types have no variables. It is easy to extend these results to allow additional base types [16].

The inclusions between types are defined by the rules:

$$t \leq \Omega$$

$$\frac{s' \leq s \quad t \leq t'}{s \rightarrow t \leq s' \rightarrow t'}$$

where s, t, etc., range over types. We will refer to the second rule as the *congruence* rule for arrow types.

Thus, typical inclusions are $\Omega \rightarrow \Omega \leq \Omega$, $\Omega \rightarrow \Omega \leq (\Omega \rightarrow \Omega) \rightarrow \Omega$, etc.

While our coercions are not atomic in the sense of [13], they have the property that for any t and u there is at most one proof of $t \leq u$, and this proof follows the structure of u. This makes it easier to reason about these inequalities. For example, we do not include the reflexive or transitive rules, because these are admissible:

Theorem 1 *The relation $\leq$ is a partial order.*

Proof: It is easy to show that for all types t, $t \leq t$ is provable from these rules. The base case is $\Omega \leq \Omega$, and the rest follows by induction on the size of t.

For the transitive property, assume $t \leq t'$ and $t' \leq t''$ are provable. We need to show that $t \leq t''$ is provable as well. We proceed by induction on the structure of t''. If $t'' = \Omega$, then $t \leq t''$. If $t' = \Omega$, then $t'' = \Omega$, so $t \leq t''$. Otherwise all of t, t', and t'' are arrow types, and both proofs must have the congruence rule as their last step. Then the result follows by induction on the components.

For the asymmetry property, assume that $t \leq t'$ and $t' \leq t$. If either t or t' is Ω, then the other must be Ω also. If neither is Ω, then both proofs must have the congruence rule as their last step, and the result follows by induction. □

2.2 Programs

Programs are ordinary untyped lambda-terms with typed constants. Polymorphic **let** can be treated by using the equivalence **let** $x = M$ **in** N as $N[M/x]$. We use the usual typing rules plus the subsumption rule

$$\frac{A \vdash M : t \quad t \leq t'}{A \vdash M : t'}$$

The details are routine.

This system types terms which are not typable in the simply-typed lambda-calculus without coercions. For example, consider $\lambda f.(fK(fI))$, where K and I are the usual combinators. This is not typable in the ordinary calculus, since K and I have different types, but it is typable under partial typing: assign f the type $\Omega \to \Omega \to \Omega$. both the K and I can be coerced to type Ω, and the result (fI), of type $\Omega \to \Omega$, can be coerced to Ω to form the second argument of the first f. Therefore the entire term has type $(\Omega \to \Omega \to \Omega) \to \Omega$.

Similarly, some self-application is possible: $(\lambda x.xx)$ has type $(\Omega \to t) \to t$ for all t, since the final x can be coerced to Ω.

However, not all terms are typable in this system. For example:

Proposition 1 *$(\lambda x.xx)(\lambda x.xx)$ is not typable in this system.*

Proof: The types of $(\lambda x.xx)$ are those types which are bounded below by a type of the form $(t_1 \to t_2) \to t_2$ where $t_1 \to t_2 \leq t_1$. In order to type $(\lambda x.xx)(\lambda x.xx)$, we need to find two such types, one for each occurrence of $(\lambda x.xx)$, such that the type of the second coerces to the argument type of the first. By a standard argument, we need consider only the case in which the types are actually of the lower-bound types; all the coercion can be incorporated into the coercion of the argument. That is, we need to find two types of the form $(t_1 \to t_2) \to t_2$ and $(u_1 \to u_2) \to u_2$, where $(u_1 \to u_2) \to u_2 \leq (t_1 \to t_2)$. After splitting the last equation, we are left with four inequalities to solve:

$$\begin{aligned} &t_1 \to t_2 \leq t_1 \\ &u_1 \to u_2 \leq u_1 \\ &t_1 \leq u_1 \to u_2 \\ &u_2 \leq t_2 \end{aligned}$$

From these a short deduction leads to $u_1 \to u_2 \leq u_1 \leq u_1 \to u_2$. Since $\leq$ is a partial order, we have $u_1 = u_1 \to u_2$, which has no solution in our system. □

We will use this as a running example in the paper. It can be shown that every term typable in this system is strongly normalizing; we hope to present this result elsewhere.

2.3 Semantics

We can give semantics for these rules in the fashion of [9]: the rules are obviously sound for either the simple or the F-semantics, interpreting Ω as the entire set. A more interesting semantics is given via PERs: Let D be a model of the untyped lambda calculus, and define a type to be any partial equivalence relation (that is, a symmetric, transitive, but not necessarily reflexive relation) on D, with type inclusion given by set-theoretic inclusion between the PERs. Let Ω denote the PER $D \times D$. Then the rules for inclusion and type inference are sound. This model has the property that elements of type Ω are indistinguishable; this nicely mimics the idea that elements of type Ω are not manipulable by any ordinary functions. The completeness of these rules (in the sense of [9]) is considered in [13].

3 The Problem

We begin by introducing type expressions, which are defined by

$$\langle \text{type exp} \rangle ::= \langle \text{type variable} \rangle \mid \Omega \mid \langle \text{type exp} \rangle \rightarrow \langle \text{type exp} \rangle$$

A *constraint* is a judgement of the form $s \leq t$, where s and t are type expressions. We say that a set E of constraints is *satisfiable* iff there is some substitution σ, mapping type variables to types, such that for each constraint $s \leq t$ in the set, $s\sigma \leq t\sigma$ is provable. We write σ_v for the value of σ on the variable v, and we write $E\sigma$ for the effect of applying σ to some composite object (type, constraint, set of constraints) E.

It is well-known that the type inference problem reduces to the problem of satisfying a set of constraints between type expressions. For type inference without subtyping, this reduction is folkloric (e.g. [3, 2, 18]), and is implicit in [8, 11]. For the case of subtyping, the reduction is given in detail in [13].

The solution of the satisfiability problem depends on the details of the definition of the ordering $\leq$. In the case where the ordering is generated by the congruence rule alone, a solution can be obtained by observing that comparable types must have the same shape [12, 13]; see also [4].

The satisfiability problem for the system considered here is harder, because it no longer has the same-shape property. Indeed, in constrast to ordinary unification, one may have satisfiable constraints $v \leq t$ where v occurs in t. For example, $v \leq v \rightarrow \Omega$ is satisfiable, since $\Omega \rightarrow \Omega \leq (\Omega \rightarrow \Omega) \rightarrow \Omega$. However, there appears to be no simple rule characterizing which such occurrences are solvable.

4 The Algorithm

The algorithm has three main steps. The first step puts the constraints in a standard form, by introducing a new variable for every interior node of a tree in the original constraints. The second step calculates lower and upper bounds for each variable. The third step generates a satisfying assignment while propagating necessary information and doing a check for circularity. The steps are divided into substeps.

Step 1: Convert every constraint $t \leq t'$ to constraints of the form $v_1 \rightarrow v_2 \leq w$ or $w \leq v_1 \rightarrow v_2$ or $\Omega \leq v$ or $v \leq w$, where the v and w's are type variables. To do this we proceed as follows:

Step 1a: First, if any constraint has one side that is a term $t_1 \rightarrow t_2$ where either t_1 or t_2 is not a variable, replace each non-variable t_i by a variable v_i and add two new constraints $v_i \leq t_i$ and $t_i \leq v_i$. This transformation preserves satisfiability and always decreases the number of interior nodes in the set of constraints.

Step 1b: When this step is done, we are left with constraints where each side is either a variable v, the constant Ω, or a type expression $v_1 \rightarrow v_2$. These can be cleaned up as follows:

- Replace $v \rightarrow w \leq v' \rightarrow w'$ by $v' \leq v$ and $w \leq w'$.
- Replace $t \leq \Omega$ by nothing. Such a constraint is always provable.
- If any constraint is of the form $\Omega \leq v_1 \rightarrow v_2$, then terminate and report failure, as this constraint is not satisfiable.

Note that there is no substitution during this phase of the algorithm; there is an implicit substitution in step 1a, but these substitutions are bounded by the size of the problem.

Step 2: Let A denote the set of arrows $v \rightarrow w$ in the set of constraints at the end of the Step 1. For each variable appearing in the constraints, we will keep track of the those arrows in A which are lower and upper bounds for that variable. We will also do this for Ω. More precisely, let v be a variable or Ω. Define L_v and U_v to be the smallest subsets of A such that:

1. If $v \leq v_1 \rightarrow v_2$ is among the constraints, then $(v_1 \rightarrow v_2) \in U_v$.
2. If $v_1 \rightarrow v_2 \leq v$ is among the constraints, then $(v_1 \rightarrow v_2) \in L_v$.
3. If $v \leq w$ is among the constraints, then $U_w \subseteq U_v$ and $L_v \subseteq L_w$.
4. If $v_1 \rightarrow v_2 \in L_w$ and $w_1 \rightarrow w_2 \in U_w$, then

$$\begin{array}{l} U_{v_1} \subseteq U_{w_1} \\ L_{w_1} \subseteq L_{v_1} \\ U_{w_2} \subseteq U_{v_2} \\ L_{v_2} \subseteq L_{w_2} \end{array}$$

These sets can be built by a simple closure operation. Since each L_v and U_v is a subset of the finite set A, this closure process must terminate in finite time. Furthermore, it is clear that for any satisfying assignment σ, $(L_v)\sigma$ and $(U_v)\sigma$ will be sets of lower and upper bounds for v. Therefore, if at the end of this step U_Ω is non-empty, then terminate and report failure, since Ω is not less than or equal to any arrow type.

From now on, we discard the constraints and work entirely with the the bound sets L_v and U_v.

For our example of $(\lambda x.xx)(\lambda x.xx)$, Step 1 is trivial, since the constraints are already of the desired form. For Step 2, the first two closure conditions give:

	L	U
t_1	$t_1 \rightarrow t_2$	$u_1 \rightarrow u_2$
u_1	$u_1 \rightarrow u_2$	

All the other sets are empty. Iterating condition 4 twice gives:

	L	U
t_1	$t_1 \to t_2$ $u_2 \to u_2$	$u_1 \to u_2$
u_1	$u_1 \to u_2$	$u_1 \to u_2$

which satisfies all the closure conditions.

Step 3: In this section of the algorithm, we develop the satisfying assignment σ. As we do this, we will generate a new variable for each interior node of the assignment, and for each such variable v we will generate sets of bounds L_v and U_v; these bounds will also be subsets of A.

Step 3a: Choose a variable v. If there are no more variables to process, return the substitution σ. If U_v is empty, then set $\sigma_v = \Omega$ and go to step 3a.

Step 3b: (Occurs Check). Consider the set $\{ w \mid v \text{ occurs in } \sigma_w \}$. This is the set of variables above v in the tree described by σ. If, for any such w, $L_w = L_v$ and $U_w = U_v$, then terminate and report failure.

Step 3c: If v passes the occurs check, proceed as follows: Since v is bounded above by an arrow, it must also be an arrow. Generate two new variables v_1 and v_2, and set $\sigma = \sigma[v := v_1 \to v_2]$.

We must next define the bound sets for v_1 and v_2. We do this by looking at the bound sets for v. For example, if $(a \to b) \in U_v$, then we must have $\sigma_a \leq \sigma_{v_1}$ for any satisfying assignment σ. Therefore $L_a \subseteq L_{v_1}$ for each $(a \to b) \in U_v$. Therefore we set $L_{v_1} = \bigcup\{ L_a \mid (a \to b) \in U_v \}$. We proceed similarly for each of the four sets:

$$\begin{aligned} L_{v_1} &= \bigcup\{ L_a \mid (a \to b) \in U_v \} \\ U_{v_1} &= \bigcup\{ U_a \mid (a \to b) \in L_v \} \\ L_{v_2} &= \bigcup\{ L_b \mid (a \to b) \in L_v \} \\ U_{v_2} &= \bigcup\{ U_b \mid (a \to b) \in U_v \} \end{aligned}$$

Then go to step 3a.

This completes the description of the algorithm.

For our example, we can assign Ω to any variable except t_1 or u_1. If we select u_1, then we assign $\sigma(u_1) = u_{11} \to u_{12}$. Calculating the bound sets for u_{11} and u_{12} gives

	L	U
t_1	$t_1 \to t_2$ $u_2 \to u_2$	$u_1 \to u_2$
u_1	$u_1 \to u_2$	$u_1 \to u_2$
u_{11}	$u_1 \to u_2$	$u_1 \to u_2$
u_{12}	$\emptyset$	$\emptyset$

When we select u_{11} for expansion, we discover that its bounds sets are the same as those of u_1, so the occurs check fails and the algorithm reports that the term is untypable, as desired.

5 Proof of the Main Theorem

We must show two things: if the algorithm reports failure, then there is no satisfying solution; and if the algorithm reports success, then all the original constraints are satisfied.

Lemma 1 *For any satisfying assignment σ, $L_v\sigma$ and $U_v\sigma$ will be sets of lower and upper bounds for v.*

Proof: Easy. □

Lemma 2 *If the algorithm reports failure, then there is no satisfying solution.*

Proof: The algorithm can report failure in two ways: either by noting that U_Ω is nonempty, or by failure of the occurs check. If U_Ω is nonempty, then the constraints imply that Ω is less than some arrow type, which is impossible. So the constraints are unsatisfiable.

Next, consider the occurs check. Let w be a variable such that v occurs in σ_w. For every variable z along the path from w to v, $U_z \neq \emptyset$, since otherwise the path would terminate at z.

The path from w to v is determined entirely by L_w and U_w, since each bounds pair (L_z, U_z) along the path is determined by the preceding bounds pair. Since $L_v = L_w$ and $U_v = U_w$, if the occurs check were omitted, then the algorithm would loop by duplicating this path indefinitely. Every node between w and v must be an arrow node, and this path would be replicated infinitely.

Now, recall that if a type has an upper bound which is an arrow type, then it must be an arrow type itself. In the algorithm, a variable is assigned an arrow type only if it has an arrow upper bound. Hence, if the algorithm assigns an arrow type to a variable, that variable must be an arrow type in any satisfying assignment. Therefore, if there is a path from w to v in the generated assignment σ, there must be a path from w to v in any satisfying assignment.

If the upper and lower bounds for v are the same as those for w, then the algorithm is guaranteed (in the absence of the occurs check) to loop by duplicating this path infinitely. But since every satisfying assignment must have an arrow type at every node along this path, we conclude that every satisfying assignment must have an infinite repetition of this path. Hence there is no (finite) satisfying assignment. □

We need the following lemma:

Lemma 3 *If $L_v \subseteq L_w$ and $U_w \subseteq U_v$, then $\sigma_v \leq \sigma_w$.*

Proof: Define a relation $\sqsubseteq$ on variables by $v \sqsubseteq w$ iff $L_v \subseteq L_w$ and $U_w \subseteq U_v$. Then the lemma can be rephrased as $v \sqsubseteq w \Rightarrow \sigma_v \leq \sigma_w$. We proceed by induction on the size of σ_v and σ_w.

If $\sigma_w = \Omega$, then the conclusion holds trivially. If $\sigma_v = \Omega$, then we must have had $U_v = \emptyset$. But $v \sqsubseteq w$ implies $U_w \subseteq U_v$, so $U_w = \emptyset$ and $\sigma_v = \Omega$ as well.

The remaining case is that both σ_v and σ_w are compound, that is,

$$\sigma_v = \sigma_{v_1} \rightarrow \sigma_{v_2}$$
$$\sigma_w = \sigma_{w_1} \rightarrow \sigma_{w_2}$$

However, in this case, if we check the set-theoretic arithmetic in Step 3c, we discover that $v \sqsubseteq w$ implies that $w_1 \sqsubseteq v_1$ and $v_2 \sqsubseteq w_2$. For example, since $v \sqsubseteq w$, we have $L_v \subseteq L_w$, so

$$U_{v_1} = \bigcup\{ U_c \mid (c \rightarrow d) \in L_v \} \subseteq \bigcup\{ U_c \mid (c \rightarrow d) \in L_w \} = U_{w_1}$$

The other inclusions follow similarly.

Hence, by induction, $\sigma_{w_1} \leq \sigma_{v_1}$ and $\sigma_{v_2} \leq \sigma_{w_2}$. Therefore $\sigma_v \leq \sigma_w$. □

Theorem 2 *Given a set of constraints E, the algorithm always terminates, and it returns a substitution σ iff E is satisfiable.*

Proof: The algorithm always terminates, since the occurs check prevents any branch of the solution from being longer than $2^{|A|+1}$. We have already shown that if the algorithm reports failure, then the constraints are unsatisfiable. We can now complete the proof of the theorem by showing that σ satisfies each of the constraints at the end of step 3.

If the constraint is of the form $v \leq w$, Step 2 guarantees that $L_v \subseteq L_w$ and $U_w \subseteq U_v$. Hence σ satisfies the constraint.

If the constraint is of the form $\Omega \leq v$, then U_v is empty (otherwise we would have failed), and the algorithm assigns $\sigma_v = \Omega$, which satisfies the constraint.

If the constraint is of the form $v_1 \rightarrow v_2 \leq w$, then there are two cases: either U_w is empty or not. If U_w is empty, then $\sigma_w = \Omega$, and the constraint is satisfied. If U_w is nonempty, then we have $\sigma_w = \sigma_{w_1} \rightarrow \sigma_{w_2}$. We claim that $L_{w_1} \subseteq L_{v_1}$, $U_{v_1} \subseteq U_{w_1}$, $L_{v_2} \subseteq L_{w_2}$, and $U_{w_2} \subseteq U_{v_2}$.

Let us do the lower-bound cases of the claim. To show $L_{w_1} \subseteq L_{v_1}$, we recall that $L_{w_1} = \bigcup\{ L_a \mid (a \rightarrow b) \in U_w \}$. So choose $(a \rightarrow b) \in U_w$. We have $(v_1 \rightarrow v_2) \in L_w$ and $(a \rightarrow b) \in U_w$, so by closure condition 4 in Step 2, $L_a \subseteq L_{v_1}$. So L_{v_1} is an upper bound for all the L_a. L_{w_1} is their least upper bound, so $L_{w_1} \subseteq L_{v_1}$. To show $L_{v_2} \subseteq L_{w_2}$, we observe that $v_1 \rightarrow v_2 \in L_w$, and $L_{w_2} = \bigcup\{ L_b \mid (a \rightarrow b) \in L_w \}$. The upper-bound cases follow symmetrically.

We can now apply lemma 3 to deduce that $\sigma_{w_1} \leq \sigma_{v_1}$, $\sigma_{v_2} \leq \sigma_{w_2}$, and therefore the constraint is satisfied.

If the constraint is of the form $w \leq v_1 \rightarrow v_2$, then U_w is nonempty, so we have $\sigma_w = \sigma_{w_1} \rightarrow \sigma_{w_2}$. The result then follows by an argument like the preceding one. □

The algorithm runs in exponential space, since it has a number of states bounded by the number of (L, U) pairs, which is $2^{|A|+1}$. If one builds a cache of these states, then the algorithm requires only exponential time.

6 Related Work

In addition to the applications cited in the introduction, there are a number of related topics. Gomard [5] used a variant of this system to do binding-type analysis, in which untyped represented an untyped run-time value. Henglein [6] gives an almost-linear-time algorithm for doing this analysis. We do not currently understand the gap between Henglein's algorithm and our exponential algorithm, but there appear to be some essential differences [7].

This work is somewhat related to the work on dynamics in ML [10]. Dynamic types are oriented towards persistent objects, which have some type that is part of its representation; the hard part about such a system is reading the object back into the system within something close to the ML type system. This is rather different from our approach.

Amadio and Cardelli have solved a related but different problem. They consider a type system with both a bottom and top element, so that $\bot \leq t \leq \top$ for all types, and with recursive types. In [1] they solve the problem of *validity* (truth under all substitutions) for recursive type expressions in this system. It would be interesting to see if this result could be extended to satisfiability, as this is the natural problem arising from type inference.

7 Conclusions and Future Work

We have shown the decidability of the satisfiability problem, and therefore the type inference problem, for Thatte's system of partial types. We have formulated the problem in the simplest way possible in order to facilitate theoretical study; the next step is to extend the system with enough features to make it practical. It should be easy to extend it to allow a bottom element, as in [1], or additional base types under an arbitrary partial order, as in [16]. Product types should also pose no difficulties. Polymorphic let can be included by translating let $x = M$ in N as $N[M/x]$, but a more efficient method would be preferable. We conjecture that this system can also be used to solve the system of flags in [14]. A more difficult problem is to extend it to handle recursive types, as in [1]. Also it would be desirable to have some lower bounds on the complexity of these problems.

Acknowledgements

Thanks to Jonathan Young for numerous conversations on types and coercions. Thanks also to Fritz Henglein who enthusiastically predicted the outcome.

References

[1] Amadio, R.M., and Cardelli, L. "Subtyping Recursive Types," *Conf. Rec. 1991 ACM Symp. on Principles of Programming Languages*, 104–118.

[2] Clément, D., Despeyroux, J., Despeyroux, T., and Kahn, G. "A Simple Applicative Language: Mini-ML" *Proc. 1986 ACM Symp. on Lisp and Functional Programming,* 13–27.

[3] Cardelli, L. "Basic Polymorphic Typechecking," *Polymorphism Newsletter 2*,1 (Jan, 1985). Also appeared as Computing Science Tech. Rep. 119, AT&T Bell Laboratories, Murray Hill, NJ.

[4] Fuh, Y.-C., and Mishra, P. "Type Inference with Subtypes," *Proc. European Symposium on Programming* (1988), 94–114.

[5] Gomard, C.K. "Partial Type Inference for Untyped Functional Programs," *Proc. 1990 ACM Conf. on Lisp and Functional Programming*, 282–287.

[6] Henglein, F. "Efficient Type Inference for Higher-Order Binding-Time Analysis," *Functional Programming Languages and Computer Architecture, 5th ACM Conference* (J. Hughes, ed.), Springer Lecture Notes in Computer Science, Vol. 523, 1991, pp. 448–472.

[7] Henglein, F. personal communication, 1991.

[8] Hindley, R. "The Principal Type-Scheme of an Object in Combinatory Logic," *Trans. Am. Math. Soc. 146* (1969) 29–60.

[9] Hindley, R. "The Completeness Theorem for Typing λ-Terms" *Theoret. Comp. Sci. 22* (1983) 1–17. See also Hindley, R. "Curry's Type-rules are Complete with Respect to the F-Semantics Too" *Theoret. Comp. Sci. 22* (1983) 127–133.

[10] LeRoy X., and Mauny, M. "Dynamics in ML," *Functional Programming Languages and Computer Architecture, 5th ACM Conference* (J. Hughes, ed.), Springer Lecture Notes in Computer Science, Vol. 523, 1991, pp. 406–426.

[11] Milner, R. "A Theory of Type Polymorphism in Programming," *J. Comp. & Sys. Sci. 17* (1978), 348–375.

[12] Mitchell, J.C. "Coercion and Type Inference (summary)," *Conf. Rec. 11th Ann. ACM Symp. on Principles of Programming Languages* (1984), 175–185.

[13] Mitchell, J.C. "Type Inference with Simple Subtypes," *J. of Functional Programming 1* (1991), 245–285.

[14] Rémy, D. "Typechecking records and variants in a natural extension of ML," *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages* (1989), 77–88.

[15] Thatte, S. "Type Inference with Partial Types," *Proc. ICALP '88* (1988), 615–629.

[16] Wand, M., and O'Keefe, P. "On the Complexity of Type Inference with Coercion," *Conf. on Functional Programming Languages and Computer Architecture* (London, September, 1989).

[17] Wand, M. "A Semantic Prototyping System," *Proc. ACM SIGPLAN '84 Compiler Construction Conference* (1984) 213–221.

[18] Wand, M. "A Simple Algorithm and Proof for Type Inference" *Fundamenta Informaticae 10* (1987), 115–122.