Type Inference Algorithms: A Survey

Sunil Kothari
Department of Computer Science
University of Wyoming
Abstract

In this paper we discuss various type inference algorithms. We first introduce the notations and concepts to be used, followed by a formal definition of type inference. Then we discuss Algorithm \( W \), Algorithm \( J \), and Algorithm \( M \) all of which are characterized by intermittent constraint generation and constraint solving. After that we describe Wand’s type inference algorithm, which is a constraint-based algorithm and is characterized by a distinct separation between constraint generation and constraint solving. We also describe an implementation of the mentioned algorithms in OCaml.

Type inference algorithms form an important component of many functional language compilers. Traditional type inference algorithms, for example, Alg. \( W \), Alg. \( M \) etc., are characterized by intermittent constraint generation and constraint solving. Over the years, however, focus has shifted to algorithms which have a clear separation between constraint generation and constraint solving phases.

This paper is organized as follows: Section 1 introduces the ideas of monomorphic and polymorphic types, the type inference problem, and the two categories of type inference algorithms; Section 2 introduces the terms, types and constraints together with other concepts; Section 3 introduces substitution and related concepts, including the first-order unification algorithm; Section 4 introduces type system; Section 5 introduces a formal definition of type inference; Section 6 describes the various type inference algorithms.

1 An Overview of Type Inference

Types are a ubiquitous feature of most modern programming languages. Milner’s slogan: “Well-typed programs can’t go wrong.” [Mil78] characterizes the advantage of strong typing. Well-typed programs do not have operator operand mismatches (like trying to add a string to a number). There are many methods of checking whether a program is well-typed. Type inference, i.e. automatically assigning a type to a program (if it has one) is a prominent feature of modern functional programming languages like Haskell [Jon] and the ML family of languages [MTHM97, Ler97, LDF+, See10].

Programming languages where a program can have one and only one type are called monomorphic languages. For example, Pascal and Java are monomorphic languages. In contrast, polymorphic languages, like ML and Haskell, programs can have many types.
To present examples, we will use an ML-like notation augmented with a bit of mathematical typesetting to make things easier to read. e.g. we will write $\lambda x.e$ for the ML form $(\text{fun } x \rightarrow e)$. The list constructors in ML are Nil (denoted []) and Cons (denoted using the infix operator ::). We write $e : T$ to indicate that the expression $e$ has type $T$. For expressions $e_1$ and $e_2$ we will write $e_1 \rightsquigarrow e_2$ to indicate that $e_1$ evaluates to $e_2$ in ML.

Since functional languages are expression based, programs are often called *terms* and we will use *term*, *program* and *expression* interchangeably.

As an example of a polymorphic function, consider the $\text{map}$ function over lists. Map takes function and a list as arguments and applies the function to all the elements of the list. The $\text{map}$ function is defined as follows:

$$
\text{map } f \ [\ ] = \ [\ ]
\text{map } f \ (h::t) = (f\ h) :: (\text{map } f \ t)
$$

We elaborate on the $\text{map}$ function by looking at several examples.

**Example 1.**
Consider the following evaluation:

$$
\text{map } (\lambda x.x+1) \ [1;2;3] \rightsquigarrow [2;3;4]
$$

The argument $(\lambda x.x+1)$ is an anonymous function that adds one to its argument $x$ and has the type $(\text{int} \rightarrow \text{int})$. The second argument $[1;2;3]$ is a list of integers, as is the return value. They have type $(\text{int list})$.

Thus, in this instance:

$$
\text{map} : (\text{int} \rightarrow \text{int}) \rightarrow (\text{int list}) \rightarrow (\text{int list})
$$

**Example 2.**
The expression

$$
\text{map } (\lambda x.x>1) \ [1;-2;3] \rightsquigarrow \text{[false;false;true]}
$$

Thus, in this instance:

$$
\text{map} : (\text{int} \rightarrow \text{bool}) \rightarrow (\text{int list}) \rightarrow (\text{bool list})
$$

**Example 3.**
Finally, we give the following example:

$$
\text{map } (\lambda x.\neg x) \ [\text{true;false;true}] \rightsquigarrow \text{[false;true;false]}
$$
The term \((\lambda x. \neg x)\) is a function which negates its input and so has type \((\text{bool} \rightarrow \text{bool})\). The second argument is necessarily a list of Booleans.

\[
\text{map} : (\text{bool} \rightarrow \text{bool}) \rightarrow (\text{bool list}) \rightarrow (\text{bool list})
\]

The examples show that the \textit{map} has at least three different types. By now, the reader realizes that \textit{map} function can be applied to functions and lists of different types. Each of the types is essentially an instance of a more general type known as the \textit{principal type}. The constraints imposed by the definition of \textit{map} on the arguments are that: (i.) the first argument be a function of any function type, say \(\alpha \rightarrow \beta\) and, (ii.) that the second argument be a list whose elements range over the domain \(\alpha\) of the function. Thus, the principal type for \textit{map} is:

\[
\forall \alpha \forall \beta. (\alpha \rightarrow \beta) \rightarrow (\alpha \text{ list} \rightarrow \beta \text{ list})
\]

Following mathematical practice, the \textit{let} construct allows for the introduction of local definitions. If \(x\) is a variable and \(M\) and \(N\) are expressions, let constructs are of the following form:

\[
\text{let } x = M \text{ in } N
\]

The variable \(x\) is called the \textit{let-bound variable}, \(M\) is the \textit{let-binding} and \(N\) is the \textit{body of the let}. The \textit{scope} of \(x\) is the term \(N\). Computationally, \((\text{let } x = M \text{ in } N)\) is just syntactic sugar for the function application \((\lambda x. N) M\). However, in the typed system, it is possible for the let term to be well-typed while its expansion to a lambda-term is not. Example 4 shows a let-bound variable \(f\) and its two polymorphic occurrence.

**Example 4.** Consider the following term involving let-bound and lambda-bound variables.

\[
\text{let } f = \lambda x. x \text{ in } \\
\quad \text{if } (f \text{ true}) \text{ then } (f \text{ 8}) \text{ else } 0
\]

Here, in the body of the let, the function \(f\) is used polymorphically. It has type \((\text{bool} \rightarrow \text{bool})\) when its applied to the Boolean constant \textit{true} but has type \((\text{int} \rightarrow \text{int})\) when its applied to the integer constant \textit{8}. The principal type of the expression \(f\) is \(\forall \alpha. \alpha \rightarrow \alpha\). On the other hand, in the standard typing systems, the following term is not well-typed.

\[
(\lambda f. \text{ if } (f \text{ true}) \text{ then } (f \text{ 8}) \text{ else } 0) (\lambda x. x)
\]
Once the type of \( f \) has been instantiated from its principal type \( \forall \alpha. \alpha \to \alpha \) to the more specific \((\mathsf{bool} \to \mathsf{bool})\), the standard typing rules do not allow it to be reinstantiated to have type \((\mathsf{int} \to \mathsf{int})\).

Since the let construct allows let-bound variables to be used polymorphically, it is clear that there is no intrinsic reason that lambda bound variables must be restricted to have monomorphic usages, they simply are.

Functional programs often come with terms annotated with types. When a program is compiled or interpreted, these annotations are type checked to make sure they are consistent with the meaning of the program. To type check a polymorphic let, the principal type of the let-binding needs to be known before the let-body is type checked.

A natural question one may ask is why can’t we assign types manually. Many imperative languages work this way, however aside from the freedom of not having to make the annotations, manual assigning of types to each and every expression in a functional program, in general, may not feasible due to some (or all) of the following reasons:

1. The type assigned to the term is too big.
2. An erroneous type can be assigned to a term.
3. Multiple occurrences of the same variable may or may not have the same type.

If the language is sufficiently constrained, then principal types can be inferred automatically from the program syntax. The process itself is called type inference. Various type inference algorithms have been proposed to infer the type of an expression automatically. For example, Algorithm W [Mil78, DM82], Algorithm \( J \) [Mil78], Algorithm \( M \) [LY98], Wand’s algorithm [Wan87]. An overview of type inference for a ML-like language can be found in [PR05].

All of the algorithms just mentioned return the principal type if the program has a type. Most type inference algorithms can be broadly classified into two distinct categories: substitution-based and constraint-based. Throughout this paper we use the term substitution-based algorithms to refer to those algorithms which intermix constraint generation and constraint solving, whereas algorithms with a clear separation between the two tasks are termed constraint-based algorithms.

Substitution-based algorithms have been widely used in implementations of ML and Haskell, but in recent years, constraint-based algorithms have
become more important. The constraint-based algorithms are characterized by a separation of constraint generation and constraint solving phases. In [Hee05, PR05] the authors argue that the separation leads to the possibility of better error messages when the constraint set is unsatisfiable (since a larger set of constraints is available to reason about the error).

Wand’s algorithm does not handle the polymorphic let construct. So the comparison here is perhaps somewhat uneven but the point is to highlight the basic difference between the substitution-based algorithms and the constraint-based algorithms.

The introduction of a polymorphic let construct in a functional language does not change the decidability of type inference but does make it harder. The typability problem for pure lambda terms is PTIME-complete [Mit96, Tys88]. The introduction of polymorphic let in a language makes the type inference PSPACE-hard [PJ89] and DEXPTIME-complete (DTIME(2^n)) [KTU90, Mai89] in the size of the nested let statements. In practice, the worst case is rarely encountered and type inference for language with polymorphic lets work very efficiently.

```latex
let x1 = \lambda y.\lambda z.(z y) y in 
let x2 = \lambda z.x1 (x1 z) in 
let x3 = \lambda z.x2 (x2 z) in 
let x4 = \lambda z.x3 (x3 z) in 
x4 (\lambda z.z)
```

Figure 1: A program which makes type inference hard

Figure 1 shows a classic example (taken from Mairson [Mai89]), where type inference algorithms display a double exponential running time. The type of this term is four pages long. A more detailed discussion of type inference is by Tiuryn [Tiu90] and Giannini et. al. [GHR93].

\section{Terms and Types}

To make the discussion more concrete and also to formally state the type inference problem, we introduce the various terminologies and notations used throughout this paper. First, we introduce the term language and the type language. Our term language is Core-ML [MH88]. This is the pure untyped lambda calculus extended with Milner’s let. The syntax for Core-ML is
shown in Figure 2.

\[
\Lambda ::= \ x \quad \text{(Variables)} \\
| \ MN \quad \text{(Application)} \\
| \ \lambda x.M \quad \text{(Abstraction)} \\
| \ \text{let } x = M \ \text{in } N \quad \text{(Milner-Let)}
\]

where \( M, N \) are \( \Lambda \) terms and \( x \) is a variable.

Figure 2: Core-ML: Term Language syntax

We follow the usual conventions: function application associates to the left, application binds more tightly than abstraction and abstractions associate to the right. We illustrate the convention by two examples.

**Example 5.** Consider the term \( \lambda x.(\lambda y.(x y)) \). This term represents a function which takes two inputs and returns a value which results from the application of the first argument to the second. As per our convention, we can drop the parenthesis since we know application binds tighter than an abstraction and abstraction associates to the right. So the resulting term is \( \lambda x.\lambda y.x y \).

**Example 6.** Consider the term \( \lambda f.(\lambda g.(\lambda x.f (g x))) \). This term represents a function which takes three inputs and returns a value which is the result of applying first argument to the result obtained by the application of second argument to the third argument. As per the convention, we do not need parenthesis for abstraction but we do need one for the application. So the resulting term is \( \lambda f.\lambda g.\lambda x.f (g x) \).

For the purposes of this paper, it is not necessary to formalize the evaluation semantics for Core-ML programs. Evaluation in Core-ML proceeds by substitution and the reduction strategy is call-by-value. A good presentation of the details of Core-ML evaluation including an implementation in OCaml can be found in [Rém02].

Figure 3 shows the type language for Core-ML. Types belong to two syntactic categories: simple types, which include type variables and function types; and type schemes, which provide a binding operator for universally quantifying over a type.

We follow the convention that lower case Greek letters early in the alphabet denote type variables. So that we do not run out of variables we allow them to be possibly primed, hatted or subscripted (e.g. \( \alpha, \beta, \gamma, \alpha_7, \beta' \)).
\[ \tau ::= \alpha \mid \tau \to \tau' \]
\[ \theta ::= \tau \mid \forall \alpha.\theta \]

where \( \alpha \) is a type variable
\( \tau \) and \( \tau' \) are simple types
\( \theta \) is a previously constructed type scheme.

Figure 3: Syntax of Simple Types and Type Schemes

etc.). Arbitrary type expressions will be written as \( \tau \), also possibly annotated by primes, hats or subscripts. Type schemes are denoted by the (possibly annotated) Greek letter \( \theta \).

Note that by these definitions, quantifiers may only occur at the top level of a type scheme, and do not appear in a type expression at all. By convention, the function type constructor \((\to)\) associates to the right. For example, \( \alpha \to \beta \to \alpha \) should be read as \( \alpha \to (\beta \to \alpha) \). A type scheme of the form \( \forall \alpha_1, \ldots, \forall \alpha_n.\tau \), where two or more type variables are universally quantified, is often written as \( \forall \alpha_1, \ldots, \alpha_n.\tau \).

Constraints are of the form \( \tau = \tau' \), where \( \tau \) and \( \tau' \) are types. We usually denote lists of constraints by \( C \).

A type environment is a partial map of the variables to types/type schemes. We use the Greek letter \( \Gamma \) (possibly annotated) as meta-variables to denote type environments. We sometimes write \( \langle x, \tau \rangle \in \Gamma \) to mean \( x \) is mapped to \( \tau \) by \( \Gamma \) and will write \( \Gamma(x) \) to denote the type associated with the variable \( x \) in the environment \( \Gamma \). If \( x \) is not in the domain of \( \Gamma \), \( \Gamma(x) \) is undefined. We define the removal of \( x \) from \( \Gamma \) as follows:

\[ \Gamma/x \overset{\text{def}}{=} \{ \langle y, \tau \rangle \in \Gamma \mid x \neq y \} \]

This is the environment like \( \Gamma \) but that does not contain a pair having \( x \) as its domain element. The following lemma characterizes environments of the form \( \Gamma/x \).

**Lemma 1.** \( \forall \Gamma. \forall x, y : \text{Var}. \ x \neq y \Rightarrow (\Gamma/x)(y) = \Gamma(y) \) and \( (\Gamma/x)(x) \) is undefined.

The domain elements of a type environment are defined as follows:

\[ \text{dom}(\Gamma) \overset{\text{def}}{=} \{ x \mid \langle x, \tau \rangle \in \Gamma \} \]
In the next few definitions, we use the minus symbol $-$ to denote the set difference. The free type variables (FTV) of a type and a type scheme are defined as:

$$\text{FTV}(\alpha) \overset{\text{def}}{=} \{ \alpha \}$$

$$\text{FTV}(\tau \to \tau') \overset{\text{def}}{=} \text{FTV}(\tau) \cup \text{FTV}(\tau')$$

$$\text{FTV}(\forall \alpha.\tau) \overset{\text{def}}{=} \text{FTV}(\tau) - \{ \alpha \}$$

The free type variables of a type environment $\Gamma$, also denoted by FTV, is defined as:

$$\text{FTV}(\Gamma) \overset{\text{def}}{=} \bigcup_{x \in \text{dom}(\Gamma)} \text{FTV}(\Gamma(x))$$

## 3 Type Substitutions

A type substitution is a finite function from type variables to types. We will use the Greek letter $\rho$ (possibly annotated) to name type substitutions. We denote the empty substitution by $\text{Id}$ and small finite substitutions will be represented by enumerating the elements in the function. For example, a finite map that binds $\alpha$ to $\tau \to \tau'$ is denoted as $\{ \alpha \mapsto \tau \to \tau' \}$.

The domain, range and free type variables in the range of a substitution are defined as follows.

$$\text{dom}(\rho) \overset{\text{def}}{=} \{ \alpha \mid \exists \tau. \langle \alpha, \tau \rangle \in \rho \}$$

$$\text{rng}(\rho) \overset{\text{def}}{=} \{ \tau \mid \exists \alpha. \langle \alpha, \tau \rangle \in \rho \}$$

$$\text{FTV}_{\text{range}}(\rho) \overset{\text{def}}{=} \bigcup_{\tau \in \text{rng}(\rho)} \text{FTV}(\tau)$$

The application of a substitution to a type is defined as follows:

$$\text{apply}_t \rho \alpha \overset{\text{def}}{=} \left\{ \begin{array}{ll} \tau & \text{if } \langle \alpha, \tau \rangle \in \rho \\ \alpha & \text{otherwise} \end{array} \right.$$  

$$\text{apply}_t \rho (\tau_1 \to \tau_2) \overset{\text{def}}{=} (\text{apply}_t \rho \tau_1) \to (\text{apply}_t \rho \tau_2)$$

We will usually simply write $\rho(\tau)$ for $\text{apply}_t \rho \tau$. The following lemma about application of the identity substitution is proved by induction on the structure of the type.

**Lemma 2.** $\forall \tau. \text{apply}_t \text{Id} \tau = \tau$

Type schemes have a binding operator ($\forall$). For generality, we define the application of a substitution to a type scheme to be capture avoiding.
apply\_ts \ \rho \ \tau \ \overset{\text{def}}{=} \ \text{apply\_t} \ \rho \ \tau

apply\_ts \ \rho \ \forall \alpha.\theta \ \overset{\text{def}}{=} \ \text{let} \ \rho' = \ \{ \langle \beta, \tau \rangle \in \rho \mid \beta \neq \alpha \} \ \text{in}
\begin{align*}
\text{let} \ \beta \ \text{be fresh w.r.t.} \ \alpha, \ \rho \ \text{and} \ \theta \ \text{in} \\
\text{let} \ \theta' = \ \text{apply\_ts} \ \{ \alpha \mapsto \beta \} \ \theta \\
\forall \beta. \ \text{apply\_ts} \ \rho' \ \theta'
\end{align*}
\begin{align*}
\text{else} \\
\forall \alpha. \ \text{apply\_ts} \ \rho' \ \theta
\end{align*}

In practice we will typically apply substitutions to type schemes to instantiate them by eliminating a quantifier and replacing the bound variables by fresh ones. In this case, avoiding capture is unnecessary since it will never occur.

We usually simply write \( \rho(\theta) \) for the application \( \text{apply\_ts} \ \rho \ \theta \).

The application of a substitution to a type environment is defined pointwise; it is simply \( \rho \) applied to each type/type scheme in the environment.

\[
\rho(\Gamma) \ \overset{\text{def}}{=} \ \{ \langle x, \rho(\tau) \rangle \mid \langle x, \tau \rangle \in \Gamma \}
\]

Application of a substitution to a constraint and a constraint set are defined similarly:

\[
\rho(\tau_1 \ e = \tau_2) \ \overset{\text{def}}{=} \ \rho(\tau_1) = \rho(\tau_2)
\]
\[
\rho(C) \ \overset{\text{def}}{=} \ \{ \rho(\tau_1 \ e = \tau_2) \mid \tau_1 \ e = \tau_2 \in C \}
\]

### 3.1 Generalization and Instantiation

Following [CDKD86], we also define two operations on type schemes called \( \text{gen} \) (generalization) and \( \text{inst} \) (instantiation). Generalizing a type \( \tau \) with respect to a type environment \( \Gamma \) entails quantifying over the free variables of \( \tau \) that are not free in \( \Gamma \).

\[
\text{gen}(\Gamma, \tau) \ \overset{\text{def}}{=} \ \begin{cases} 
\tau & \text{if } \text{FTV}(\tau) - \text{FTV}(\Gamma) = \emptyset \\
\forall \alpha_1, \ldots, \alpha_n.\tau & \text{where } \alpha_i \in \text{FTV}(\tau) - \text{FTV}(\Gamma) \text{ and } 1 \leq i \leq n
\end{cases}
\]

Instantiation of a type scheme involves eliminating some or all of the quantifiers and substituting a simple type for the free occurrences of the
bound variable being eliminated. Obviously, in its general form, we may need to avoid variable capture. We define an \textit{inst} operation that takes a type scheme and list of simple types (say of length \(k\)) and, eliminates the leftmost \(k\) quantifiers and substitutes the \(i\)th type in the list for the free occurrences of the \(i\)th bound variable in the body. This operation is known as \textit{generic instantiation} and is defined as follows:

\[
\text{inst}(\theta, []) \overset{\text{def}}{=} \theta,
\text{inst}(\tau, 
\text{inst}(\forall \alpha_1, \ldots, \alpha_m. \tau, [\tau_1, \ldots, \tau_k]) \overset{\text{def}}{=}
\begin{cases}
\{\alpha_i \mapsto \tau_i \mid 1 \leq i \leq k\}(\forall \alpha_{k+1}, \ldots, \alpha_m. \tau) & \text{if } k < m \\
\{\alpha_i \mapsto \tau_i \mid 1 \leq i \leq m\}(\tau) & \text{else}
\end{cases}
\]

It turns out that in our uses of generic instantiation we eliminate all the quantifiers, replacing the free occurrences of their bound variables by fresh type variables. In this case instantiations have the form \(\text{inst}(\forall \alpha_1, \ldots, \alpha_n. \tau, [\beta_1, \ldots, \beta_n])\), where \(\beta_1, \ldots, \beta_n\) are fresh variables (at least) with respect to the type \(\tau\).

### 3.2 A Partial Ordering on Types

We follow [CDKD86, DM82] and define a partial ordering on type schemes based on instantiation instances. Specifically, a type scheme \(\theta = \forall \alpha_1, \ldots, \alpha_m. \tau\) has a \textit{generic instance} of the form \(\theta' = \forall \beta_1, \ldots, \beta_n. \tau'\) if there is a substitution \(\rho\) such that

\[
\tau' = \rho(\tau) \text{ where } \text{dom}(\rho) \subseteq \{\alpha_1, \ldots, \alpha_m\}
\]

and no variable \(\beta_i\) is free in \(\theta\), \textit{i.e.}

\[
\beta_i \notin \text{FV}(\theta) \text{ where } 1 \leq i \leq n
\]

The following are some examples of this ordering, it may or may not be immediately obvious that the statements are true.

\[
\begin{align*}
\forall \alpha. \alpha & \succeq \alpha \\
\forall \alpha. \alpha & \succeq \forall \beta. \alpha. \alpha \\
\forall \alpha. \alpha & \succeq \forall \alpha, \beta. \alpha \rightarrow \beta \\
\forall \alpha. \alpha & \succeq \forall \beta, \gamma. \beta \rightarrow \gamma \\
\forall \alpha, \beta. \alpha \rightarrow \beta & \succeq \forall \beta, \beta \rightarrow \beta
\end{align*}
\]
This relation is a partial ordering \( i.e \) it is reflexive, antisymmetric and transitive. We also have the following lemma saying that if a relation holds where the larger type in the ordering is simple, then the relation is just equality.

**Lemma 3.** \( \forall \tau, \tau'. \tau \succeq \tau' \Rightarrow \tau = \tau' \)

**Proof.** Since the domain of the substitution comes from the quantified variables in the simple type \( \tau \) which has no quantified variables, the substitution must be \( Id. \) \( Id(\tau) = \tau \) so \( \tau' = \tau. \)

Also, in [DM82] it is proved that the ordering is preserved under substitution.

**Lemma 4.** \( \forall \theta, \theta'. \forall \rho. \theta \succeq \theta' \Rightarrow \rho(\theta) \succeq \rho(\theta') \)

3.3 Composition

The composition of substitutions (say \( \rho \) and \( \rho' \)) is written \( \rho \circ \rho' \). The following three lemmas characterize the behavior\(^1\) of composition.

**Lemma 5.** \( \forall \rho, \rho'. \forall \tau. (\rho \circ \rho')(\tau) = \rho'(\rho(\tau)) \)

**Lemma 6.** \( \forall \rho, \rho'. \forall \Gamma. (\rho \circ \rho')(\Gamma) = \rho'(\rho(\Gamma)) \)

**Lemma 7.** \( \forall \rho, \rho'. \forall C. (\rho \circ \rho')(C) = \rho'(\rho(C)) \)

We can actually construct the substitution resulting from composition of the two substitutions as shown later in Appendix A. Here we show the main idea. Let \( \rho_1, \rho_2 \) be two substitutions then the composition of substitution \( \rho_1 \circ \rho_2 \) is given as the result of applying substitution \( \rho_2 \) to the range of \( \rho_1 \) and adding all pairs from \( \rho_2 \) where domain element is not in the domain of \( \rho_1 \). Example 7 shows the result of composition of two simple substitutions.

**Example 7.** If \( \rho_1 = \{ \alpha_1 \mapsto \alpha_2 \rightarrow \alpha_3 \} \) and \( \rho_2 = \{ \alpha_1 \mapsto \alpha_5, \alpha_2 \mapsto \alpha_3 \} \) then \( \rho_1 \circ \rho_2 = \{ \alpha_1 \mapsto \alpha_3 \rightarrow \alpha_3, \alpha_2 \mapsto \alpha_3 \}. \)

Composition of substitutions is associative but non-commutative. Consider substitution \( \rho_1, \rho_2, \rho_3 \). Then \( (\rho_1 \circ \rho_2) \circ \rho_3 = \rho_1 \circ (\rho_2 \circ \rho_3) \) but \( \rho_1 \circ \rho_2 \neq \rho_2 \circ \rho_1 \). A substitution is **idempotent** if \( \rho \circ \rho = \rho. \)

\(^1\)The reader should note the order of of composition. We read compositions from left to right, in the composition \( \rho \circ \rho' \), the function \( \rho \) is applied first, and then \( \rho' \). We adopted this convention early on because it was built into some of the Coq libraries we used in our formalization.
3.4 Unification

Simple types \( \tau_1 \) and \( \tau_2 \) are unifiable if there exists a substitution \( \rho \) such that \( \rho(\tau_1) = \rho(\tau_2) \). In such a case, \( \rho \) is called a unifier. We denote satisfiability of a constraint by \( \models \) (read “solves”). We write \( \rho \models (\tau_1 \equiv \tau_2) \), if \( \rho(\tau_1) = \rho(\tau_2) \).

We extend the satisfiability notion to a list of constraints and we write \( \rho \models C \) if and only if for every \( c \in C \), \( \rho \models c \). A unifier \( \rho \) is called the most general unifier if for any unifier \( \rho' \) there is a substitution \( \rho' \) such that \( \rho \circ \rho' = \rho'' \) holds.

Unification plays a key role in all the type inference algorithms; it is the mechanism for solving constraints. We focus on first-order unification in this paper. For an excellent overview of unification, interested readers should consult [Kni89, BS01]. Note that the first-order unification always returns an idempotent substitution.

Our presentation of first-order unification below is based on Robinson’s algorithm [Rob65]:

\[
\begin{align*}
\text{unify}([]) & \equiv Id \\
\text{unify}(\alpha \equiv \beta :: C) & \equiv \begin{cases} \\
\text{if } \alpha = \beta & \text{then } \text{unify}(C) \\
\text{else } \{\alpha \mapsto \beta\} \circ \text{unify}(\{\alpha \mapsto \beta\}(C)) & \\
\end{cases} \\
\text{unify}(\alpha \equiv \tau :: C) & \equiv \begin{cases} \\
\text{if } \alpha \in \text{FTV}(\tau) & \text{then Fail} \\
\text{else } \{\alpha \mapsto \tau\} \circ \text{unify}(\{\alpha \mapsto \tau\}(C)) & \\
\end{cases} \\
\text{unify}(\tau \equiv \alpha :: C) & \equiv \text{unify}(\{\alpha \equiv \tau\} :: C) \\
\text{unify}(\tau_1 \rightarrow \tau_2 \equiv \tau_3 \rightarrow \tau_4 :: C) & \equiv \text{unify}(\{\tau_1 \equiv \tau_3\} :: (\tau_2 \equiv \tau_4) :: C)
\end{align*}
\]

A note on the time complexity of first-order unification; the unification algorithm runs in linear time [PW76, BS01] in the size of the input if the inputs and outputs are coded as directed acyclic graphs (DAGs). We have not done that here. The time complexity of the algorithm is exponential if the inputs and outputs are coded as strings (or trees as they are here) since a DAG of size \( n \) can encode a string of \( 2^{O(n)} \).

4 Type Systems

Terms and types are related by sequents of the form \( \Gamma \vdash M : \tau \), where \( \Gamma \) is an environment, \( M \) is a term and \( \tau \) is a type. The sequent can be read as “Under the assumptions in \( \Gamma \), \( M \) has type \( \tau \)”. A type rule is a relation between sequents. Type rules are specified schematically and are:
1-ary relations specifying axioms; 2-ary relations specifying rules with one hypothesis; or 3-ary relations specifying rules with two hypothesis. A type system is a collection of type rules.

\[
\begin{align*}
(VAR) & \quad \Gamma \vdash x : \theta & \quad \langle x : \theta \rangle \in \Gamma \\
(INST) & \quad \frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \theta'} & \quad \theta \succeq \theta' \\
(GEN) & \quad \frac{\Gamma \vdash M : \theta}{\Gamma \vdash M : \forall \alpha.\theta} & \quad \alpha \notin \text{FTV}(\Gamma) \\
(APP) & \quad \frac{\Gamma \vdash M : \tau' \rightarrow \tau \quad \Gamma \vdash N : \tau'}{\Gamma \vdash MN : \tau} \\
(ABS) & \quad \frac{\Gamma \vdash \lambda x. M : \tau \rightarrow \tau'}{\Gamma \vdash \lambda x. M : \theta} \\
(LET) & \quad \frac{\Gamma \vdash M : \theta \quad \Gamma / x \cup \{x : \theta\} \vdash N : \tau}{\Gamma \vdash \text{let } x = M \text{ in } N : \tau}
\end{align*}
\]

Figure 4: Non-Deterministic Damas-Hindley-Milner Type System

A derivation of a sequent \( \Gamma \vdash M : \tau \) is an inverted tree of sequents concluding with \( \Gamma \vdash M : \tau \), and every node in the tree is an instance of some rule from the underlying type system. If a sequent \( \Gamma \vdash M : \tau \) is derivable, we write \( \vdash \Gamma \vdash M : \tau \). When needed, we also affix the name of the type system to the derivation symbol to denote the derivation in a particular type system. For example,

\[
\vdash_{\text{DHM}} \emptyset \vdash \lambda f.\lambda g.\lambda x.g(f x) : (\alpha \rightarrow \beta) \rightarrow (\beta \rightarrow \gamma) \rightarrow \alpha \rightarrow \gamma
\]

indicates that the sequent is derivable in the DHM type system. We will use the Greek letter Δ (possibly annotated with primes, or subscripts) as meta-variables for derivations.

The Damas-Hindley-Milner type system (DMH) was first presented in [DM82]. Fig. 4 shows the DHM type system for Core-ML. Note that the meta-variables \( \tau \) and \( \tau' \) range over simple types and \( \theta \) and \( \theta' \) range over type schemes. This system is non-deterministic i.e. more than rule might apply at any given stage in building the derivation. To make the type system deterministic and syntax-directed, we use the modified DMH type system
presented in [CDKD86], the deterministic system is shown in Fig. 5. Notice that the APP and ABS rules remain the same but the VAR and LET rules have been modified. The non-determinism arises in the original system from the INST and GEN rules. In the modified system, the GEN and INST rules are eliminated and their effects have been built into the LET and VAR rules using the operators gen and inst. This eliminates the non-determinism. We will refer to the modified type system in Fig. 5 as the Damas-Hindley-Milner type system and denote it by DHM throughout this paper.

**Example 8.** Derivation showing that the compose function

\[ \lambda f.\lambda g.\lambda x.g(f\ x) \] 

has type \((\alpha \to \beta) \to (\beta \to \gamma) \to \alpha \to \gamma\) in the DHM type system.

We also consider Wand’s constraint-based type system in this paper. In Wand’s system, sequents are 4-place relations of the following form:

\[ \Gamma, C \triangleright M : \tau \]
As before, $\Gamma$ is an environment, $M$ is a term, $\tau$ is a type and the new component $(C)$ is a set of equational constraints. The meaning of a sequent in Wand’s system is more complex since derivations in the system simply generate a set of constraints $C$ that, if solvable by unification, will yield a substitution that applied to $\tau$ gives the type of $M$. In Wand’s system, the typing rules specify how to compute the set $C$. The constraint generation phase is separated from the generation of the solution.

We extend the earlier notion of free type variables of a type to a set of free type variables for a constraint set to be defined pointwise:

$$FTV(C) \overset{\text{def}}{=} \{ \alpha \in FTV(\tau \rightarrow \tau') \mid \tau \overset{\alpha}{=} \tau' \in C\}$$

If we restrict ourselves to a language without polymorphic let, we can define a simple constraint-based type system. At this time, we do not have any specific algorithm in mind, but a little later we describe Wand’s algorithm, which serves as an implementation of the type system given in Fig. 6.

Figure 6: Wand Type System

To illustrate this type system, we reconsider the type derivation of the compose function from Example 8 in Wand type system. To represent the derivation compactly, we display the constraint generated (if any) on the top of sequent at that node in the derivation. At each node in the derivation, the constraint set consists of the constraint labeling the assertion at that node together with all the constraints generated higher in the derivation tree.

**Example 9.** A derivation in Wand’s system for the term $\lambda f.\lambda g.\lambda x.g(f\ x)$
starting with the type variable $\alpha_0$ generates the following constraint set:

\[
\begin{align*}
\{ & \alpha_3 \overset{e}{=} \alpha_6 \rightarrow \alpha_7, \\
 & \alpha_1 \overset{e}{=} \alpha_8 \rightarrow \alpha_7, \\
 & \alpha_5 \overset{e}{=} \alpha_8, \\
 & \alpha_4 \overset{e}{=} \alpha_5 \rightarrow \alpha_6, \\
 & \alpha_2 \overset{e}{=} \alpha_3 \rightarrow \alpha_4, \\
 & \alpha_0 \overset{e}{=} \alpha_1 \rightarrow \alpha_2 \}\end{align*}
\]

The derivation generating this set is shown here.

\[
\begin{align*}
\Delta_1 : & \quad \text{(W-Var)} \quad \frac{\text{ } \frac{\Delta_1}{f : \alpha_1} \triangleright f : \alpha_8 \rightarrow \alpha_7 \quad \frac{\Delta_1}{f : \alpha_1, g : \alpha_3, x : \alpha_5} \triangleright x : \alpha_8 \quad f \quad \alpha_7}{f : \alpha_1, g : \alpha_3, x : \alpha_5 \triangleright f \ x : \alpha_7} \\
& \quad \text{(W-Var)} \quad \frac{g : \alpha_3 \triangleright g : \alpha_6 \rightarrow \alpha_7 \quad f : \alpha_1, g : \alpha_3, x : \alpha_5 \triangleright f \ x : \alpha_7}{f : \alpha_1, g : \alpha_3, x : \alpha_5 \triangleright g(f \ x) : \alpha_6} \quad \text{(W-App)} \\
& \quad \frac{f : \alpha_1, g : \alpha_3, x : \alpha_5 \triangleright g(f \ x) : \alpha_6 \quad f : \alpha_1, g : \alpha_3 \rightarrow \alpha_4 \quad \frac{\Delta_1}{f \ x : \alpha_7 \triangleright \lambda x. g(f \ x) : \alpha_2} \quad \frac{f : \alpha_1, \lambda x. g(f \ x) : \alpha_2}{g : \alpha_6 \triangleright \lambda f. \lambda g. \lambda x. g(f \ x) : \alpha_0} \quad \text{(W-Abs)} \\
& \quad \frac{f : \alpha_1, \lambda x. g(f \ x) : \alpha_2 \triangleright \lambda f. \lambda g. \lambda x. g(f \ x) : \alpha_0}{\lambda f. \lambda g. \lambda x. g(f \ x) : \alpha_0}
\end{align*}
\]

Derivations in Wand’s system build constraint sets. The constraints so generated may or may not be satisfiable. In this case they are. Unification is the constraint solving mechanism needed to derive the substitution which, applied to $\alpha_0$ yields the actual type of the input expression.

The following important lemma relates how a type derivation could be a substitution instance of another derivation.

**Lemma 8.** If $\vdash_{DHM} \Gamma \triangleright M : \tau$ then, for any $\rho$, $\vdash_{DHM} \rho(\Gamma) \triangleright M : \rho(\tau)$.

**Proof.** See [DM99] for a proof sketch. \qed

A similar but less well-known lemma holds for constraint-based derivations, but since constraints are synthesized attributes, we have to be careful about the kind of substitutions for which the lemma holds.
5 Type Inference - Formal Definition

The general type inference problem can be formulated as:

Given a well-formed term $M$ (without any type annotations),
a type environment $\Gamma$ and a type system, does there exist a type
$\tau$ such that $\Gamma \leadsto M : \tau$?

We note here the difference between type inference and type reconstruction. A type reconstruction algorithm takes an untyped term $M$ as input, and finds a type environment $\Gamma$ and a type-annotated version $M'$ of $M$, and a type $\tau$ such that $\tau$ is a type for $M'$ with respect to the type environment $\Gamma$. A far tougher problem is type inhabitation, where given a type the task is to find a term $M$ and a type environment $\Gamma$ such that $M$ has type $\tau$ in the type environment $\Gamma$ [Tiu90].

6 Type Inference Algorithms

In this section we describe several type inference algorithms.

6.1 Algorithm $W$

Milner [Mil78] implemented a type reconstruction algorithm in the context of LCF [GMM+78] metalanguage. He gave the first thorough account of not one but two type reconstruction algorithms, namely, Algorithm $W$ and Algorithm $J$. The paper also gave a proof of soundness. Later, Damas and Milner [DM82] showed that Algorithm $W$ returns the principal type/type scheme - a generalization of a previous result result by Hindley for combinatory logic [Hin69]. Later more extensions and variants of Algorithm $W$ 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 - an idea which leads to intersection types [BDCD95]. Coppo proves that type reconstruction is undecidable for a type scheme with sequence of types.

Cardelli [Car97] gives a detailed account of the algorithmic and constraint-based approach to type reconstruction and also supplies the code for type reconstruction in Modula 2 - a language not widely used now. Laufer [Lau92] and Mycroft [Myc84] have given a thorough account (including corrected typos) of Algorithm $W$. Mycroft extended Algorithm $W$ to deal with more general type schemes associated with mutual recursive definitions. Nelson
[Nel95] used a variant of Algorithm $\mathcal{W}$ to show a sound and complete type reconstruction system for first order dependent types.

$$
\begin{align*}
\text{Case Var.} & \\
\mathcal{W}(\Gamma, x) & \stackrel{\text{def}}{=} \\
\text{If } x : \theta \in \Gamma \text{ then let } \beta_1, \ldots, \beta_n \text{ be fresh variables in} \\
& \quad \text{let } \tau'' = \text{inst}(\theta, \beta_1, \ldots, \beta_n), \\
& \quad \text{in } (\text{Id}, \tau'') \\
\text{else if } x : \tau'' \in \Gamma \text{ then } (\text{Id}, \tau'') \\
\text{else Fail}
\end{align*}
$

$$
\begin{align*}
\text{Case App.} & \\
\mathcal{W}(\Gamma, M_1 M_2) & \stackrel{\text{def}}{=} \\
\text{let } (\rho_1, \tau_1) = \mathcal{W}(\Gamma, M_1) \\
\text{let } (\rho_2, \tau_2) = \mathcal{W}(\rho_1(\Gamma), M_2) \\
\text{let } \beta \text{ be a new type variable} \\
\text{let } \rho_3 = \text{unify}(\rho_2(\tau_1), \tau_2 \to \beta) \\
& \quad \text{in } ((\rho_1 \circ \rho_2) \circ \rho_3, \rho_3(\beta))
\end{align*}
$

$$
\begin{align*}
\text{Case Abs.} & \\
\mathcal{W}(\Gamma, \lambda x. M) & \stackrel{\text{def}}{=} \\
\text{let } \beta \text{ be a new type variable} \\
\text{let } (\rho_1, \tau_1) = \mathcal{W}(\Gamma/x \cup \{x : \beta\}, M) \\
& \quad \text{in } (\rho_1, \rho_1(\beta) \to \tau_1)
\end{align*}
$

$$
\begin{align*}
\text{Case Let.} & \\
\mathcal{W}(\Gamma, \text{let } x = M \text{ in } N) & \stackrel{\text{def}}{=} \\
\text{let } (\rho_1, \tau_1) = \mathcal{W}(\Gamma, M) \\
\text{let } (\rho_2, \tau_2) = \mathcal{W}(\Gamma/x \cup \{x : \text{gen}(\rho_1(\Gamma), \tau_1)\}, N) \\
& \quad \text{in } (\rho_1 \circ \rho_2, \tau_2)
\end{align*}
$

Figure 7: Algorithm $\mathcal{W}$

Given an initial environment $\Gamma$ and a term $M$, if Algorithm $\mathcal{W}$ succeeds it finds a substitution $\rho$ and a type $\tau$ such that $\rho(\Gamma) \triangleright M : \tau$. Formally, Algorithm $\mathcal{W}$ can be seen as a mapping:

Type Environment $\times$ Expression $\rightarrow$ Substitution $\times$ Type

Fig. 7 shows a full description of the algorithm. An implementation of the algorithm in OCaml is described in Appendix A.1.
6.2 Algorithm \( J \)

Algorithm \( J \) was first mentioned by [Mil78] and is a simpler version of Algorithm \( W \). Surprisingly, even though both Algorithm \( W \) and Algorithm \( J \) are mentioned in the same paper, the former is the more popular one. Although, according to Pottier [Pot05], Jones used Algorithm \( J \) in his work [Jon99]. To know the type of an expression, a global substitution is maintained and is updated during the inference process. Since the algorithms presented here are in a functional style, the notion of a global object is simulated by passing the object as an input argument and returning the updated object as a part of the output. Thus, Algorithm \( J \) is defined by recursion on the structure of the term \( M \) such that \( J(\Gamma, M, Id) = (\rho_1, \tau) \), where \( \rho_1 \) is the resulting substitution and \( \tau \) is the type assigned to \( M \) in the type environment \( \Gamma \). Fig. 8 shows a complete description of Algorithm \( J \). An implementation of the algorithm in OCaml is described in Appendix A.2.

7 Algorithm \( M \)

Algorithm \( M \) was first implemented in CamlLight [Ler93] (a lighter version of Caml). A formal description of the algorithm and proof of its soundness and completeness was given by Lee and Yi [LY98]. The algorithm is top-down and stops earlier than Algorithm \( W \) if the program is ill-typed. Given an initial type environment \( \Gamma \), a term \( M \) and a type \( \tau \), which is initially assumed to be a type variable, Algorithm \( M \) can be seen as a mapping:

\[
\text{Type Environment} \times \text{Expression} \times \text{Type} \rightarrow \text{Substitution}
\]

and is defined by induction on the structure of \( M \) such that \( M(\Gamma, M, \tau) = \rho \), where \( \rho \) is a substitution generated by the Algorithm \( M \). The algorithm is fully described in Fig. 9. An implementation of the algorithm in OCaml is described in Appendix A.3.

We end this section with a note on the how the algorithms behave differently in the application case. The App case is representative of the different approaches used in the three algorithms towards type inference. Consider the term \( MN \). Algorithm \( W \) uses an eager application strategy in the sense that substitution generated from the left branch (for computing the type of term \( M \)) is applied to the environment before computing the type for the term \( N \). In Algorithm \( J \), the substitutions resulting from computing the type of \( M \) and the type of \( N \) are composed in the end to generate the resulting substitution. However, both Algorithm \( W \) and \( J \) make no assumptions about the type of \( M \), i.e. it is an arrow type. On the other hand, Algorithm
$M$ makes the assumption that the type of $M$ is a function type. Thus, the extra step of invoking first-order unification algorithm is avoided in Algorithm $M$, although it does apply the substitution resulting from computing the type of $M$ to compute the type of $N$.

### 7.1 Wand’s Algorithm

In this section, we introduce the constraint-based algorithms and frameworks for type inference. The constraint-based algorithms as well as frameworks need a constraint solving mechanism. We mention a few in this section.

The first account of constraint-based type inference that we know of was given by Wand [Wan87]. In particular, Wand gave an algorithm for generating constraints out of terms and then solving those constraints to get the principal type of a term. Sulzmann et. al. [SOW97] introduced HM(X) framework parametrized by the constraint domain $X$. For the HM type systems, the parameter $X$ is instantiated to a standard Herbrand constraint system. Type schemes are treated as constrained types of the form $\forall \alpha.C \Rightarrow \sigma$, where $C$ is a constraint in $X$ that restricts the types that can be substituted for $\alpha$. Monomorphic types (say $\tau$) are treated as $\forall \alpha.true \Rightarrow \tau$ where $\alpha \notin FTV(\tau)$. The typing judgments are of the form $C, \Gamma \vdash M : \theta$, where $M$ is a term, $C$ is a satisfiable constraint set in $X$, $\Gamma$ is a type environment and $\theta$ is a type scheme. Additionally, a normalization step is performed to convert constraints in term constraint system to get a constraint $C$ in cylindrical constraint system $X$ [HMT71]. Réméy and Pottier [PR05] describe a less abstract approach to type inference but their approach is again based on constrained types much like HM(X). Qualified types [Jon95], introduced by Jones, is another constraint-based framework for type inference. Here the typing judgment is of the form $P \mid A \vdash M : \tau$, where $\tau$ ranges over the set of type expressions, and represents the assertion that term $M$ has type $\tau$ under type environment $A$ if the predicates in $P$ are satisfied.

In the next few sections, we describe Wand’s type inference algorithm for the untyped lambda calculus.

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 to look at type inference as solving constraints. However, Milner’s polymorphic let is not a part of their language syntax, therefore, it is much simpler than the algorithms mentioned so far.

The following description is from Wand’s paper [Wan87]. Let $G$ denote a
set of goals. And \( C \) a set of equations. Then the main steps of the algorithm are:

**Input.** A term \( M_0 \) of \( \Lambda \) not containing any let construct.

**Initialization.** Set \( C = \emptyset \) and \( G = \{ (\Gamma_0, M_0, \alpha_0) \} \).

**Loop Step** If \( G = \emptyset \) then return \( C \) else choose a subgoal \( (\Gamma, M, \tau) \) from \( G \) and add to \( C \) and \( G \) new verification conditions and subgoals by following the rules specified in the action table.

The action table serves as a black box effectively separating the constraint solving from constraint generation. For Core-ML without let constructs, the action table’s behavior is:

- **Case** \((\Gamma, x, \tau)\). Generate the equation \( \tau \equiv \Gamma(x) \).
- **Case** \((\Gamma, M_1M_2, \tau)\). Generate subgoals \((\Gamma, M_1, \alpha \rightarrow \tau)\) and \((\Gamma, M_2, \tau_1)\), where \( \alpha \) is a fresh type variable.
- **Case** \((\Gamma, \lambda x.M, \tau)\). Generate equation \( \tau \equiv \alpha_1 \rightarrow \alpha_2 \) and subgoal \((\Gamma/x \cup \{ x : \alpha_1 \}, M, \alpha_2)\), where \( \alpha_1 \) and \( \alpha_2 \) are fresh type variables.

The entire algorithm is summarized in Fig. 10. Given an initial type environment \( \Gamma \), a term \( M \) and a type \( \tau \), Wand’s algorithm can be seen as a mapping:

\[
\text{Type Environment} \times \text{Expression} \times \text{Type} \rightarrow \text{Constraint List}
\]

The algorithm itself is defined as a recursive function \( \text{Wand} \) such that \( C = \text{Wand}(\Gamma, M, \tau) \), where \( C \) is the constraint set generated by the Wand’s algorithm. An informal definition of \( \text{Wand} \) function is shown in Fig. 11.

The equations generated by the Wand’s algorithm are solved by a first-order unification algorithm as shown in Example 10.

**Example 10.** The goal store is shown on the left and the constraint generated for a particular goal is on the right. For brevity, we do not show all the constraints but only the newly generated constraint at each step. Initially, the constraint store is empty and the goal store contains the following goal:

\( (\emptyset, \lambda x.\lambda y.\lambda z.xz(yz), \alpha_0) \).

When the algorithm terminates, the constraint store contains all the constraints generated by the algorithm.
### Goal Store (G)

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>{∅, \lambda x.\lambda y.\lambda z.xz(yz), α₀}</td>
</tr>
<tr>
<td>{((x : α₁), λy.λz.xz(yz), α₂) }</td>
</tr>
<tr>
<td>{((x : α₁, y : α₃), λz.xz(yz), α₄) }</td>
</tr>
<tr>
<td>{((x : α₁, y : α₃, z : α₅), xz(yz), α₆) }</td>
</tr>
<tr>
<td>{((x : α₁, z : α₅), xz, α₇ → α₆) }</td>
</tr>
<tr>
<td>{((x : α₁), x, α₈ → α₇ → α₆) }</td>
</tr>
<tr>
<td>{((z : α₅), z, α₈ → α₇ → α₇) }</td>
</tr>
<tr>
<td>{((z : α₅, z, α₈) }</td>
</tr>
</tbody>
</table>

### Constraint Store (C)

<table>
<thead>
<tr>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>{}</td>
</tr>
<tr>
<td>{α₀ e = α₁ → α₂}</td>
</tr>
<tr>
<td>{α₂ e = α₃ → α₄}</td>
</tr>
<tr>
<td>{α₄ e = α₅ → α₆}</td>
</tr>
<tr>
<td>{α₁ e = α₈ → α₇ → α₆}</td>
</tr>
<tr>
<td>{α₈ e = α₅}</td>
</tr>
<tr>
<td>{α₉ → α₇ e = α₃}</td>
</tr>
<tr>
<td>{α₉ e = α₅}</td>
</tr>
</tbody>
</table>

The constraint store has the following constraints when the algorithm terminates:

\{α₀ = α₁ → α₂, α₂ = α₃ → α₄, α₄ = α₅ → α₉, α₁ = α₈ → α₇ → α₉, α₈ = α₅, α₉ → α₇ = α₃, α₉ = α₅\}

After unifying the constraints, a substitution ρ is obtained. ρ(α₀) = (α₅ → α₇ → α₉) → (α₅ → α₇) → (α₅ → α₆)

This gives the type of the original term.

An implementation of the Wand’s algorithm is described in Appendix A.4.
Case Var.
\[ J(\Gamma, M_1, \rho_0) \overset{\text{def}}{=} \]
If \( x : \theta \in \Gamma \)
then let \( \beta_1, \ldots, \beta_n \) be fresh variables in
let \( \tau'' = \text{inst}(\theta, \beta_1, \ldots, \beta_n) \),
in \( (\text{Id}, \tau'') \)
else if \( x : \tau'' \in \Gamma \) then \( (\text{Id}, \tau'') \)
else Fail

Case App.
\[ J(\Gamma, M_1, M_2, \rho_0) \overset{\text{def}}{=} \]
let \( (\rho_1, \tau_1) = J(\Gamma, M_1, \rho_0) \)
let \( (\rho_2, \tau_2) = J(\Gamma, M_2, \rho_1) \)
let \( \beta \) be a new type variable
let \( \rho_3 = \rho_2 \circ \text{unify}(\rho_2(\tau_1), \rho_2(\tau_2 \rightarrow \beta)) \)
in \( (\rho_3, \rho_3(\beta)) \)

Case Abs.
\[ J(\Gamma, \lambda x. M_1, \rho_0) \overset{\text{def}}{=} \]
let \( \beta \) be a new type variable
let \( (\rho_1, \tau_1) = J(\Gamma/x \cup \{ x : \beta \}, M_1, \rho_0) \)
in \( (\rho_1, \rho_1(\beta) \rightarrow \tau_1) \)

Case Let.
\[ J(\Gamma, \text{let } x = M_1 \text{ in } M_2, \rho_0) \overset{\text{def}}{=} \]
let \( (\rho_1, \tau_1) = J(\Gamma, M_1, \rho_0) \)
let \( \theta = \text{gen}(\rho_1(\Gamma), \rho_1(\tau_1)) \)
in \( J(\Gamma/x \cup \{ x : \theta \}, M_2, \rho_1) \)

Figure 8: Algorithm \( J \)
Case Var.
\[ \mathcal{M}(\Gamma, x, \tau) \overset{\text{def}}{=} \]
If \( x : \theta \in \Gamma \)
then let \( \beta_1, \ldots, \beta_n \) be fresh variables in
let \( \tau'' = \text{inst}(\theta, \beta_1, \ldots, \beta_n) \)
in \text{unify}(\tau, \tau'')
else if \( x : \tau'' \in \Gamma \) then \text{unify}(\tau, \tau'')
else Fail

Case App.
\[ \mathcal{M}(\Gamma, M_1 M_2, \tau) \overset{\text{def}}{=} \]
let \( \beta \) be a fresh type variable
let \( \rho_1 = \mathcal{M}(\Gamma, M_1, \beta \to \tau) \)
let \( \rho_2 = \mathcal{M}(\rho_1(\Gamma), M_2, \rho_1(\beta)) \)
in \( \rho_1 \circ \rho_2 \).

Case Abs.
\[ \mathcal{M}(\Gamma, \lambda x. M', \tau) \overset{\text{def}}{=} \]
let \( \beta_1, \beta_2 \) be two fresh type variables
let \( \rho_1 = \text{unify}(\tau, \beta_1 \to \beta_2) \) and
let \( \rho_2 = \mathcal{M}(\rho_1(\Gamma/x) \cup \{x : \rho_1(\beta_1)\}, M', \rho_1(\beta_2)) \)
in \( \rho_1 \circ \rho_2 \).

Case Let.
\[ \mathcal{M}(\Gamma, \text{let } x = M_1 \text{ in } M_2, \tau) \overset{\text{def}}{=} \]
let \( \beta \) be a fresh type variable
let \( \rho_1 = \mathcal{M}(\Gamma, M_1, \beta) \)
let \( \rho_2 = \mathcal{M}(\Gamma/x \cup \{x : \text{gen}(\rho_1(\Gamma), \rho_1(\beta))\}, M_2, \rho_1(\tau)) \)
in \( \rho_1 \circ \rho_2 \).

Figure 9: Algorithm \( \mathcal{M} \)
Figure 10: **Wand’s Algorithm - An Overview**

<table>
<thead>
<tr>
<th>Case Var.</th>
</tr>
</thead>
</table>
| $\text{Wand}(\Gamma, x, \tau) \overset{\text{def}}{=} \begin{cases} 
\text{If } x : \tau' \text{ in } \Gamma \\
\text{then } \tau' \equiv \tau \\
\text{else } \text{Fail}.
\end{cases}$ |

<table>
<thead>
<tr>
<th>Case App.</th>
</tr>
</thead>
</table>
| $\text{Wand}(\Gamma, M_1 M_2, \tau) \overset{\text{def}}{=} \begin{cases} 
\text{let } \alpha \text{ be a fresh type variable} \\
\text{let } C_1 = \text{Wand}(\Gamma, M, \alpha \rightarrow \tau) \\
\text{let } C_2 = \text{Wand}(\Gamma, N, \alpha) \\
\text{in } C_1 \cup C_2.
\end{cases}$ |

<table>
<thead>
<tr>
<th>Case Abs.</th>
</tr>
</thead>
</table>
| $\text{Wand}(\Gamma, \lambda x. M_1, \tau) \overset{\text{def}}{=} \begin{cases} 
\text{let } \alpha, \beta \text{ be two fresh type variables} \\
\text{let } C = \text{Wand}(\Gamma/x \cup \{x : \alpha\}, M, \beta) \\
\text{in } \{\alpha \rightarrow \beta \equiv \tau\} \cup C
\end{cases}$ |

Figure 11: **Wand’s Algorithm**
A Implementation of Type Inference Algorithms in OCaml

The general layout of the implementations is as follows. First, we describe several OCaml list library used in the implementation. Next, we describe the various concepts needed to implement the type inference algorithms. The data types and functions span over the various type inference algorithms. Finally, we describe Algorithm $W$, Algorithm $J$ and Algorithm $M$. Throughout this discussion, we have used mathematical symbols to make concise and readable definitions.

We have made extensive use of the OCaml List library [LDF+] and we reproduced below the used library functions for completeness of the discussion:

- **append**: `'a list → 'a list → 'a list`
  Concatenates two lists. Same function as the infix operator `@`.

- **assoc**: `'a → ('a # 'b) list → 'b`
  `assoc a l` returns the value associated with the leftmost binding of `a` in the list of pairs `l`. Raises exception `NotFound` if there is no value associated with `a` in the list `l`.

- **concat**: `'a list list → 'a list`
  Concatenates a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result.

- **fold_right**: `(′a → ′b → ′b) → ′a list → ′b → ′b`
  `fold_right f [a1; ...; an] b` is a short form for `f a1 (f a2 (... (f an b) ...))`.

- **filter**: `(′a → bool) → ′a list → ′a list`
  `filter p l` returns all the elements of the list `l` that satisfy the predicate `p`. The order of elements in the input list is preserved.

- **map**: `(′a → ′b) → ′a list → ′b list`
  `map f [a1; ...; an]` applies function `f` to `a1, ..., an` and builds the result `[f(a1); ...; f(an)]` with the results returned by `f`.

- **mem**: `'a → ′a list → bool`
  `mem a l` is true if and only if `a` is equal to an element of `l`.

- **partition**: `(′a → bool) → ′a list → ′a list ′a list`
  `partition p l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the
elements of $l$ that satisfy the predicate $p$ and $l_2$ is the list of all elements that do not satisfy $p$. The order of the elements in the input list is preserved.

The term language is given by the following data type:

```ocaml
type term =
  | Var of string
  | Abs of string * term
  | Ap of term * term
  | Let of string * term * term
```

The type and type schemes are given by the following OCaml data types:

```ocaml
type types =
  | TVar of string
  | Arrow of types * types

type scheme =
  | TS of types
  | All of string * scheme
```

We define a union type in OCaml to represent type environments. The union type is defined as follows:

```ocaml
type ('a,'b) union = Inl of 'a | Inr of 'b
```

The inhabitants of the union type are constructed by the constructors `Inl` and `Inr`. We represent type environments as lists of pairs with the first projection being a string and the second projection being a type or type scheme.

```ocaml
type env = Env of (string * (types,scheme) union) list
```

The type substitutions are defined as follows:

```ocaml
type subst = Subst of (string * types) list
```

The constraints are represented as a list of pairs, where each component of the pair is a type. We use the constructor `E` to construct an equational constraint as given by the following datatype:

```ocaml
type constraints =
  E of types * types
```

The list of free type variables of a type is defined as:
let rec fv_type ty =  
  match ty with  
  | TVar a -> [a]  
  | Arrow(t1,t2) -> fv_type t1 @ fv_type t2

We define a utility function remove_all to remove all occurrences of a type variable from a list of type variables as follows:

let remove_all x t = List.filter (fun h -> not (x = h)) t

The function fv_scheme returns a list of free type variables of a type scheme:

let rec fv_scheme ts =  
  match ts with  
  | TS ty -> fv_type ty  
  | All (x,ts1) -> remove_all x (fv_scheme ts1)

Another utility function is unique which removes all duplicate occurrences of variables from a list of variables, and is defined as:

let unique l = List.fold_right (fun x y -> x :: remove_all x y) l []

The free type variables of a type environment is defined as:

let rec fv_env (Env e) =  
  match e with  
  | [] -> []  
  | (x,y)::t ->  
    (match y with  
    | Inl x -> (fv_type x)@ (fv_env (Env t))  
    | Inr y -> (fv_scheme y)@(fv_env (Env t))

The free type variables in the range of a subst is computed by the fv_subst function defined as:

let fv_subst (Subst s) =  
  List.fold_right (fun (_,t) vs -> fv_type t @ vs) s []

Environments can be extended with a variable bound to a type as:

let addType x t (Env e) = Env ((x,Inl t) :: e)

Similarly, type environments can be extended with a variable bound to a type scheme as:

let addScheme x t (Env e) = Env ((x,Inr t)::e)
Application of a substitution to a type is defined as:

```ml
let rec subst type (Subst s) t =
  match t with
  | TVar a -> (try List.assoc a s with Not_found -> t)
  | Arrow(t1,t2) ->
                        Arrow (subst type (Subst s) t1, subst_type (Subst s) t2)

Before we present the substitution application to a type scheme we need to talk about the fresh function which generates a fresh type variable. In here, we have represented type variables as strings. Every time a fresh variable is needed, the counter is incremented and the updated value of counter appended to the input string is assumed to be a fresh variable. This supposedly fresh variable is compared in the context (type, type environment) to ensure freshness. If the check is successful then the string with the variable name and the updated counter is sent as a result, otherwise the steps are repeated until a fresh variable is generated. This process terminates because the context is finite. The freshness function fresh is implemented as:

```ml
let fresh x xs =
  let count = ref (0) in
  let rec getit y =
    if not(List.mem y xs) then
      y
    else
      (count := !count + 1;
       getit (x ^ (string_of_int !count)))
  in
  getit x
```

The function string_of_int is used to convert an integer into a string, and is defined in the OCaml library.

Application of a substitution to a type scheme is defined as:

```ml
let rec subst_scheme (Subst s) ts =
  match ts with
  | TS ty -> TS(subst type (Subst s) ty)
  | All(x,t2) ->
    let s' = List.filter (fun (y,t) -> y <> x) s in
    if List.mem x (fv_scheme t2) then
      let vars = fv_scheme t2 in
      let z = fresh "z" vars in
```

30
All (z, (subst_scheme (Subst s'))
    (subst_scheme (Subst [(x, TVar z)]) t2))

else
All (x, subst_scheme (Subst s) t2)

Application of a substitution to a type environment is defined as follows:

let subst_env s (Env e) =
let subst (x, ty) =
match ty with
| Inl ty' -> (x, Inl (subst_type s ty'))
| Inr ts -> (x, Inr (subst_scheme s ts))
in
Env (List.map subst e)

Substitution application to an equational constraint is implemented as:

let subst_constr s (E (ty,ty')) = E (subst_type s ty, subst_type s ty')

Substitution application to a list of equational constraint is implemented:

let subst_constrs s = List.map (subst_constr s)

Composition of two substitutions is implemented as:

let compose (Subst s1) (Subst s2) =
let comp1 = map (fun (x,y) -> (x,subst_type (Subst s2) y)) s1 in
let dom = map fst comp1 in
let comp3 = filter (fun (x,_) -> not (mem x dom)) s2 in
Subst(comp1 @ comp3)

Another important function is the list_minus function which takes two lists
and returns a list which contains all the elements in the first list that are
not present in the second list. It is defined as:

let list_minus l1 l2 = List.filter (fun x -> not (List.mem x l2)) l1

The function gen for generalizing a type with respect to a type environ-
ment is implemented as:

let gen ty (Env e) =
let fv_ty = fv_type ty in
let fv_e = fv_env (Env e) in
let gen_forall vars = List.fold_right (fun x y -> All (x, y)) vars

The function \texttt{inst} for instantiating a type scheme is implemented as:

\begin{verbatim}
let rec inst ts vars =
  match ts with
    | All(x,ts') ->
      let b = fresh "b" vars in
      let ts1 = subst_scheme (Subst [x,TVar b]) ts' in
      inst ts1 (b :: vars)
    | TS ty -> (ty, vars)
\end{verbatim}

The first-order unification is implemented as follows:

\begin{verbatim}
let rec unify c =
  match c with
    | [] -> Subst []
    | h::t ->
      (match h with
       | E (TVar a, TVar b) ->
         if a = b
         then unify t
         else compose (Subst [(a, TVar b)])
         (unify (subst_constrs (Subst [(a,TVar b)]) t))
       | E (TVar a, ty) ->
         if (List.mem a (fv_type ty))
         then raise (Failure ("occurs check failure"))
         else compose (Subst [(a, ty)])
         (unify (subst_constrs (Subst [(a, ty)]) t))
       | E (ty, TVar a) -> unify (E(TVar a,ty)::t)
       | E (Arrow(t1,t2),Arrow(t3,t4)) ->
         unify (E(t1, t3)::( E(t2,t4)::t))
     )
\end{verbatim}
A.1 Algorithm $W$

The algorithm is implemented as a recursive function `infer_type_W` as follows:

```ocaml
let infer_type_W t =
  let rec infer_W t (Env env) vars =
    match t with
    | Var x ->
      let ty = try (List.assoc x env) with Not_found
      -> raise (Failure ("type of \"x\" not found")) in
        (match ty with
         | Inr ty1 -> let (ty',vars') = inst ty1 vars in
           (Subst [],ty', vars')
         | Inl ty2 -> (Subst [],ty2,vars)
        )
    | Abs (x,t1) ->
      let a = fresh "a" vars in
      let env' = addType x (TVar a) (Env env) in
      let (s, ty'', vars') = infer_W t1 env' (a::vars) in
        (s, Arrow ((subst_type s (TVar a)), ty''), vars')
    | Ap (t1,t2) ->
      let (s', ty', vars') = infer_W t1 (Env env) vars in
      let env' = subst_env s' (Env env) in
      let a = fresh "a" vars' in
      let (s'',ty'', vars'') = infer_W t2 env' (a::vars') in
      let s1 = unify [E (subst_type s'' ty',
        Arrow (ty'', TVar a))] in
      let s2 = compose (compose s' s'') s1 in
      let ty1 = subst_type s1 (TVar a) in
        (s2, ty1, vars'')
    | Let (x,t1,t2) ->
      let (s',ty',vars') = infer_W t1 (Env env) vars in
      let env' = subst_env s' (Env env) in
      let ts = gen ty' env' in
      let env'' = addScheme x ts env' in
      let (s'',ty'',vars'') = infer_W t2 env'' vars' in
        (compose_subst s' s'', ty'', vars'')
in
    infer_W t (Env []) []
```

A.2 Algorithm $J$

The algorithm is implemented as a recursive function `infer_type_J` as follows:

```ocaml
let infer_type_J t =
    let rec infer_J t (Env env) (Subst s) vars =
      match t with
      | Var x ->
        let ty = try (List.assoc x env) with Not_found
          -> raise (Failure ("type of"^"x"^"not found")) in
          (match ty with
           | Inr ty1 -> let (ty', vars') = inst ty1 vars in
                         (Subst s, ty', vars')
           | Inl ty2 -> (Subst s, ty2, vars)
           )
      | Abs (x, t1) ->
        let a = fresh "a" vars in
        let env' = addType x (TVar a) (Env env) in
        let (s1, ty'', vars') = infer_J t1 env' (Subst s)
          (a::vars) in
        (s1, Arrow ((subst_type s1 (TVar a)), ty''), vars')
      | Ap (t1, t2) ->
        let a = fresh "a" vars in
        let (s', ty', vars') = infer_J t1 (Env env)
          (Subst s) (a::vars) in
        let (s'', ty'', vars'') = infer_J t2 (Env env) s' vars' in
        let s1 = unify
          [E (subst_type s'' ty',
             (subst_type s'' (Arrow (ty'', TVar a))))]
        in
        let s2 = compose s'' s1 in
        (s2, subst_type s2 (TVar a), vars'')
      | Let (x, t1, t2) ->
        let (s', ty', vars') = infer_J t1 (Env env)
          (Subst s) vars in
        let env' = subst_env s' (Env env) in
        let ty'' = subst_type s' ty' in
        let ts = gen ty'' env' in
```

34
let env'' = addScheme x ts env' in
infer_J t2 env'' s' vars'
in
infer_J t (Env []) (Subst []) []
A.3 Algorithm $M$

The algorithm $M$ is implemented as a recursive function `infer_type_M` as follows:

```ml
let infer_type_M t =
  let rec infer_M t (Env env) ty vars =
    match t with
    | Var x ->
      let ty' = try (List.assoc x env) with Not_found
        -> raise (Failure ("type of " ^ x ^ " not found")) in
        (match ty' with
         | Inr ty1 -> let (ty'',vars') = inst ty1 vars in
           ( unify [E (ty'', ty)], vars')
         | Inl ty2 -> (unify [E(ty2,ty)],vars)
        )
    | Abs (x,t1) ->
      let a = fresh "a" vars in
      let b = fresh "b" (a::vars) in
      let s = unify [E (ty, Arrow (TVar a , TVar b))] in
      let env' = addType x (subst_type s (TVar a))
        (subst_env s (Env env)) in
      let (s',vars') = infer_M t1 env' (subst_type s (TVar b))
        (b::a::vars) in
        (compose s s', vars')
    | Ap (t1,t2) ->
      let a = fresh "a" vars in
      let (s',vars') = infer_M t1 (Env env)
        (Arrow (TVar a, ty))
        (a::vars) in
      let (s'',vars'') = infer_M t2 (subst_env s' (Env env))
        (subt_type s' (TVar a)) vars' in
        (compose s' s'', vars'')
    | Let (x,t1,t2) ->
      let a = fresh "a" vars in
      let (s',vars') = infer_M t1 (Env env) (TVar a) (a::vars) in
      let env' = subst_env s' (Env env) in
      let ty' = subst_type s' (TVar a) in
      let ts = gen ty' env' in
      let (s'',vars'') = infer_M t2 (addScheme x ts env')
        (subt_type s' ty) vars' in
        (compose s' s'', vars'')
    in
    infer_M t (Env []) (TVar "a") ["a"]

```
A.4 Wand’s Algorithm

Note here the algorithm only generates a constraint list. The recursive function \texttt{wand}, which generates a constraint list, is implemented as:

\begin{verbatim}
let rec wand env t ty vars =
    match t with
    | Var x -> let ty' = try (lookup x env) with Not_found
        -> raise (Failure "wand: failure.")
        in ([E (ty,ty')],vars)
    | Abs (x,t1) ->
        let a = fresh "a" vars in
        let b = fresh "b" (a::vars) in
        let env' = addType x (TVar a) env in
        let (c,vars') = wand env' t1 (TVar b) (a::b::vars) in
        ((E(Arrow(TVar a, TVar b),ty))::c , vars')
    | Ap (t1,t2) ->
        let a = fresh "a" vars in
        let (c1,vars1) = wand env t1 (Arrow (TVar a, ty)) (a::vars) in
        let (c2,vars2) = wand env t2 (TVar a) vars1 in
        (c1 @ c2, vars2)
end
\end{verbatim}
References


