

CS421 Topic 10: Type Inference¹

Sameer Sundresh
sundresh@uiuc.edu

University of Illinois at Urbana-Champaign

June 12, 2007

¹Based on slides by Mattox Beckman, as updated by Vikram Adve, Gul Agha, Elsa Gunter, and Mark Hills

Objectives

- ▶ Unification recap
- ▶ Nondeterminism
- ▶ Type inference

Unification Problem

Unification: Given a set of **syntactic equations** E between **terms**, find a **substitution** θ which is a **most general unifier** for the equations.

- ▶ $E \subseteq \text{Equation}^*$
- ▶ $\text{Equation} ::= (\text{Term} = \text{Term})$
- ▶ $\text{Term} ::= \text{Constant}(\text{TermList}) \mid \text{Variable}$

Though we will be using it for type inference, unification is a general algorithm on terms, not specific to types.

Unifiers

A **substitution** is a mapping from *variables* to *terms*,
 $\theta : \text{Variable} \rightarrow \text{Term}$

Example: $\theta = [x \leftarrow f(g(), y), z \leftarrow h()]$

A **unifier** for a set of equations E is a substitution which, when applied to the terms in E , transforms all equations to **identities**:

$$E = \{s_1 = t_1, \dots, s_n = t_n\}$$

$$E\theta = \{u_1 = u_1, \dots, u_n = u_n\}$$

where $u_i = s_i\theta = t_i\theta$

Most General Unifiers

A **most general unifier** for a set of equations E is a unifier θ s.t. for any other unifier θ' , there exists a substitution S s.t.

$$E\theta' = E\theta S$$

Most general unifiers are unique *modulo variable renaming*.

Example: $\theta = [x \leftarrow y]$ and $[y \leftarrow x]$ are equivalent most general unifiers for $\{f(x) = f(y)\}$.

Non-example: $\theta' = [x \leftarrow h(), y \leftarrow h()]$ is a *unifier*, but not a *most general unifier*.

Unification Algorithm

Given $E \subseteq Term \times Term$,
compute a most general unifier $\theta \subseteq Var \rightarrow Term$.

Equation from E

1. $f(s_1, \dots, s_n) = f(t_1, \dots, t_n)$
2. $f(s_1, \dots, s_n) = g(t_1, \dots, t_m)$
3. $t = t$
4. $X = t$ or $t = X$, $X \notin t$
5. $X = t$ or $t = X$, $X \in t$, $X \neq t$

Action

- replace by the equations
 $s_1 = t_1, \dots, s_n = t_n$
- return *not unifiable*
- delete equation
- apply $[X \leftarrow t]$ to E and θ ;
add $X \leftarrow t$ to θ
- return *not unifiable*

Unification Algorithm is All-Paths Nondeterministic

The unification algorithm is not **deterministic**: there are many different orders in which you could follow the rules.

However, **all** evaluations lead to **equivalent** results (e.g. $[x \leftarrow y]$ vs. $[y \leftarrow x]$, but not $[x \leftarrow h(), y \leftarrow k()]$).

- ▶ *Intuitively, this is because we only have a choice amongst **independent** operations: reducing different equations.*

Nondeterminism

There are two ways we can specify algorithms:

- ▶ **Deterministic algorithm:** specifies exactly what to do at each step.
- ▶ **Nondeterministic algorithm:** some choices are left unspecified.
 - ▶ **Some path nondeterminism:** there exists *some* set of decisions which allow the algorithm to terminate; but other choices may not return a result.
 - ▶ The N in NP-complete refers to *some path nondeterminism*
 - ▶ **All paths nondeterminism:** no matter which choices you make, the algorithm will terminate with an equivalent result (*but performance may vary*).
 - ▶ *Multi-threaded programs* should exhibit *all paths nondeterminism* with respect to scheduling.

Type Inference is Some-Path Nondeterministic

Type inference is actually a some-path nondeterministic problem. We have to somehow guess the correct type τ_1 for function argument x ; if we guess wrong, the function body e won't type check.

$$\frac{\Gamma \cup [x : \tau_1] \vdash e : \tau_2}{\Gamma \vdash \text{fun } x \rightarrow e : \tau_1 \rightarrow \tau_2}$$

Fortunately, with the restrictions we've placed on our type system, we can refine it to an all-paths nondeterministic algorithm based on unification.

Approach

- ▶ When we don't know what type an expression should have, we create a new type variable for it.
- ▶ Build up equations relating type expressions.
- ▶ Use unification to solve for a **principle typing**.
- ▶ Let-polymorphism requires us to *incrementally* perform unification at each `let rec` statement, not just at the end.

Format of a Type Inference Judgment

An *type inference judgment* has the following form:

$$\Gamma \vdash e : \tau \mid C$$

where

- ▶ Γ, e, τ as before.
- ▶ C is a set of **constraints**: equations on type expressions involving type variables.
- ▶ like before, we can compute $\tau \mid C$ from Γ and e .

Axioms – Constants

$$\frac{}{\vdash n : \tau \mid \{\tau = \text{int}\}} \text{(assuming } n \text{ is an int)}$$

$$\frac{}{\vdash \text{true} : \tau \mid \{\tau = \text{bool}\}}$$

$$\frac{}{\vdash \text{false} : \tau \mid \{\tau = \text{bool}\}}$$

- ▶ These are rules that are true no matter what the typing context
- ▶ n is a *meta-variable* representing any int

Axioms – Variables

We'll get back to this later, after we've discussed polymorphic functions and type schemas.

Simple Rules

- ▶ Rules for arithmetic operators (+,*,etc):

$$\frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 \oplus e_2 : \tau | \{ \tau = \text{int} , \tau_1 = \text{int} , \tau_2 = \text{int} \} \cup C_1 \cup C_2}$$

- ▶ Rules for relational operators (=,<,etc) are similar:

$$\frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 \sim e_2 : \tau | \{ \tau = \text{bool} , \tau_1 = \text{int} , \tau_2 = \text{int} \} \cup C_1 \cup C_2}$$

Simple Rules

Booleans look similar, except for not, which has only one operand:

$$\frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 \ \&\& \ e_2 : \tau | \{ \tau = \text{bool} , \tau_1 = \text{bool} , \tau_2 = \text{bool} \} \cup C_1 \cup C_2}$$

$$\frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 \ || \ e_2 : \tau | \{ \tau = \text{bool} , \tau_1 = \text{bool} , \tau_2 = \text{bool} \} \cup C_1 \cup C_2}$$

$$\frac{\Gamma \vdash e_1 : \tau | C_1}{\Gamma \vdash ! \ e_1 : \tau | \{ \tau = \text{bool} , \tau_1 = \text{bool} \} \cup C_1}$$

Function Application

$$\frac{\Gamma \vdash e_1 : \tau_1 | C_1 \quad \Gamma \vdash e_2 : \tau_2 | C_2}{\Gamma \vdash e_1 e_2 : \tau | \{\tau_1 = \tau_2 \rightarrow \tau\} \cup C_1 \cup C_2}$$

Type Schemas

As before, we use **type schemas** to represent the types of *polymorphic* functions.

$$\forall\{\alpha, \dots\}. \tau \rightarrow \tau'$$

And plain types are just null type schemas:

$$\tau = \forall\{\}. \tau$$

Polymorphic Functions

$$\frac{\Gamma \cup [x : \tau_1] \vdash e : \tau_2 \mid C}{\Gamma \vdash \mathbf{fun} \ x \rightarrow e : \tau \mid \{\tau = \forall \{\tau_1\}. \tau_1 \rightarrow \tau_2\} \cup C}$$

Let and Letrec

$$\frac{\Gamma \cup [x : \tau] \vdash e_1 : \tau_1 \mid C_1 \quad \Gamma \cup [x : \tau'] \vdash e_2 : \tau_2 \mid C_2}{\Gamma \vdash \text{let rec } x = e_1 \text{ in } e_2 : \tau_2 \mid C_1 \cup C_2}$$

Where $\tau' = \tau\theta$ for most general unifier θ of $C_1 \cup \{\tau = \tau_1\}$.
 We also need to iteratively simplify $\forall A.\tau_1 \rightarrow \forall B.\tau_2$ to
 $\forall A \cup B.\tau_1 \rightarrow \tau_2$.

Variables, Type Schema Instantiation

Finally, since our type environments contain *type schemas* rather than plain *types*, we need a way to **instantiate** a type expression containing **fresh type variables** from a type schema.

$$\frac{}{\Gamma \vdash x : \tau_1 | \{\tau_1 = \text{inst}(\tau)\}} \text{ if } (x : \tau) \in \Gamma$$

Where $\text{inst}(\tau)$ creates a copy of τ , replacing all quantified type variables by **unquantified** fresh type variables.

Example: $\text{inst}(\forall\{\alpha\}.\alpha \rightarrow \alpha) = \tau_3 \rightarrow \tau_3$