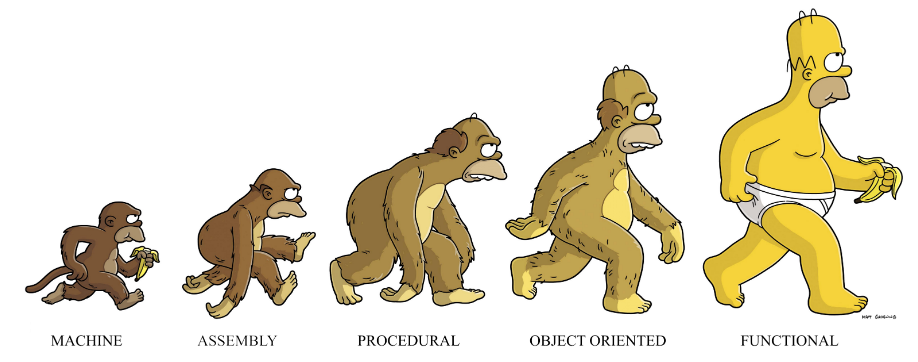
COMP302: Programming Languages and Paradigms

Week 12: Polymorphic Type Inference

Prof. Brigitte Pientka <bpientka@cs.mcgill.ca>

School of Computer Science, McGill University



Recap: typing with contexts

$$
\begin{array}{lcl}
\text{Operations op} & ::= & + \mid - \mid * \mid < \mid = \\
\text{Expressions } e & ::= & n \mid e_1 \text{ op } e_2 \mid \texttt{true} \mid \texttt{false} \mid \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \\
 & & \mid x \mid \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} \\
\text{Types } T & ::= & \texttt{int} \mid \texttt{bool} \\
\text{Context } \Gamma & ::= & \cdot \mid \Gamma, x : T
\end{array}
$$

$\boxed{\Gamma \vdash e : T}$ Expression e has type T given the typing context Γ.

$$\frac{}{\Gamma \vdash n : \texttt{int}}\ \text{T-NUM} \qquad \frac{\Gamma \vdash e_1 : \texttt{int} \quad \Gamma \vdash e_2 : \texttt{int}}{\Gamma \vdash e_1 + e_2 : \texttt{int}}\ \text{T-PLUS}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash e_1 : T \quad \Gamma \vdash e_2 : T}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 : T}\ \text{T-IF} \qquad \frac{\Gamma(x) = T}{\Gamma \vdash x : T}\ \text{T-VAR}$$

$$\frac{\Gamma \vdash e_1 : T_1 \quad \Gamma, x{:}T_1 \vdash e_2 : T}{\Gamma \vdash \texttt{let } x = e_1 \texttt{ in } e_2 \texttt{ end} : T}\ \text{T-LET} \quad x \text{ must be new}$$

Generalizing to functions and function application

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \texttt{fn } x{:}T_1 \texttt{ => } e : T_1 \to T_2}\ \textsc{T-Fn} \qquad \frac{\Gamma \vdash e_1 : T_2 \to T \quad \Gamma \vdash e_2 : T_2}{\Gamma \vdash e_1\ e_2 : T}\ \textsc{T-App}$$

Read T-FN rule as :

> Expression `fn` x `=>` e has type $T_1 \to T_2$ in a typing context Γ, if expression e has type T_2 in the extended context $\Gamma, x{:}T_1$

Issue: The rule T-FN cannot be used for type inference...
Where is the type of $x{:}T_1$ coming from?

Simple Answer: Provide type annotation and write `fn` $x{:}T_1$ `=>` e!

A better anwer: Hindley-Milner Polymorphic Type Inference

Main Idea

$$\frac{\Gamma, x{:}T_1 \vdash e : T_2}{\Gamma \vdash \texttt{fn}\ x \Rightarrow e : T_1 \rightarrow T_2}\ \text{T-FN}$$

Type Inference:

Given assumptions in Γ and $\texttt{fn}\ x$ => e, we want to infer a type

- Make a recursive call and infer in the extended context $\Gamma, x{:}T_1$ the type of $\texttt{e}$
- We don't have T_1! – We can't make that recursive call!
- Introduce a place holder (type variable) α for T_1 and let's figure out later, when analyzing e what type x must have.

  Make a recursive call in the he extended context $\Gamma, x{:}\alpha$ – **We succeed if there exists an instantiation for α s.t. $\texttt{fn}\ x$ => e has type $\alpha \rightarrow T_2$**

$$\frac{\Gamma, x{:}\alpha \vdash e : T_2}{\Gamma \vdash \texttt{fn}\ x \Rightarrow e : \alpha \rightarrow T_2}\ \text{T-FN}$$

Type Variables – Two Different Views

$$\text{Types } T \ ::= \ \texttt{int} \mid \texttt{bool} \mid T_1 \to T_2 \mid \alpha$$

$\Gamma \vdash e : T$ “Expression e has type T in the context Γ”
where T and Γ may contain type variables

View A. Are *all* substitution instances of e well-typed? That is for every type substitution σ, we have $[\sigma]\Gamma \vdash e : [\sigma]T$. Type checking

Examples for A. T/α

$$\vdash \texttt{fn } x \texttt{ => } x \quad \text{has type} \quad \alpha \to \alpha$$

$$\vdash \texttt{fn } \overset{\alpha}{f} \texttt{ => fn } \overset{\alpha}{x} \texttt{ => } f(f(x)) \quad \text{has type} \quad (\underbrace{\alpha \to \alpha}_{f}) \to \underset{x}{\alpha} \to \alpha$$

$$x : \alpha \vdash \texttt{fn } f \texttt{ => } f\ x \quad \text{has type} \quad (\underbrace{\alpha \to \beta}_{f}) \to \underset{\text{result}}{\beta}$$

Type Variables – Two Different Views

$$\text{Types } T \; ::= \; \texttt{int} \mid \texttt{bool} \mid T_1 \to T_2 \mid \alpha$$

$\Gamma \vdash e : T$ "Expression e has type T in the context Γ"
where T and Γ may contain type variables

View B. Is *some* substitution instance of e well-typed? That is we can find a type substitution σ, such that $[\sigma]\Gamma \vdash e : [\sigma]T$. Type inference

Examples for B.

$\vdash \texttt{fn } x \texttt{ => } x + 1$	has type	$\alpha \to \alpha$	choosing $\texttt{int}$ for α (i.e. $\texttt{int}/\alpha$)
$\vdash \texttt{fn } x \texttt{ => } x + 1$	has type	$\alpha \to \beta$	choosing $\texttt{int}$ for α
			choosing $\texttt{int}$ for β
			(i.e. $\texttt{int}/\alpha$, $\texttt{int}/\beta$)
$x : \alpha \vdash \texttt{fn } f \texttt{ => } f\ x$	has type	$\beta \to \gamma$	choosing $(\alpha \to \gamma)$ for β

Which substitution to pick, under the inference view?

$x : \alpha \vdash$ `fn f => f x` has type $\beta \to \gamma$ choosing $(\alpha \to \gamma)$ for β
(i.e. $(\alpha \to \gamma)/\beta$)

What about choosing $\mathtt{int}/\alpha$, $(\mathtt{int} \to \gamma)/\beta$?

This gives us that

$\boxed{\texttt{fn f => f x}}$ has type $\boxed{(\mathtt{int} \to \gamma) \to \gamma}$ under the assumption $x : \mathtt{int}$

which is **a** solution.

But it's *not the most general* solution!

Damas-Hindley-Milner Style Type Inference - Recipe

$\Gamma \vdash e \Rightarrow T$ Given a typing context Γ and an expression e, infer a type T (and some constraints)

The type T is a skeleton that may contain type variables.

- Analyze e as before following the given typing rules
- When we analyze e recursively and we miss type information, introduce a new type variable α and possibly generate constraints.

  For example:

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1 \quad \Gamma \vdash e_2 \Rightarrow T_2}{\Gamma \vdash e_1\ e_2 \Rightarrow \alpha}\ \text{T-APP} \quad \text{where } \alpha \text{ is new and } T_1 = (T_2 \rightarrow \alpha)$$

Damas-Hindley-Milner Style Type Inference - Recipe

$\Gamma \vdash e \Rightarrow T$ Given a typing context Γ and an expression e, infer a type T (and some constraints)

The type T is a skeleton that may contain type variables.

- Analyze e as before following the given typing rules
- When we analyze e recursively and we miss type information, introduce a new type variable α and possibly generate constraints.

  For example:

$$\frac{\Gamma \vdash e \Rightarrow T \qquad \Gamma \vdash e_1 \Rightarrow T_1 \quad \Gamma \vdash e_2 \Rightarrow T_2}{\Gamma \vdash \texttt{if } e \texttt{ then } e_1 \texttt{ else } e_2 \Rightarrow T_1} \textsc{T-If} \text{ where } T = \texttt{bool} \text{ and } T_1 = T_2$$

Damas-Hindley-Milner Style Type Inference - Recipe

$\Gamma \vdash e \Rightarrow T$ Given a typing context Γ and an expression e, infer a type T (and some constraints)

The type T is a skeleton that may contain type variables.

- Analyze e as before following the given typing rules
- When we analyze e recursively and we miss type information, introduce a new type variable α and possibly generate constraints.

  For example:

$$\frac{\Gamma \vdash e_1 \Rightarrow T_1 \quad \Gamma \vdash e_2 \Rightarrow T_2}{\Gamma \vdash e_1 + e_2 \Rightarrow \texttt{int}} \textsc{T-Plus} \text{ where } T_1 = \texttt{int} \text{ and } T_2 = \texttt{int}$$

- To determine whether e is well-typed, solve the constraints! – If the constraints can be solved, then there exists a substitution instance for the type variables s.t. e is well-typed.

Inferring Types and Constraints by Example

How to infer the type of `fn` x `=>` `fn` y `=>` `if` f x `then` y `else` $2 + x$?

$\alpha = \beta \to \beta'$

$$\dfrac{\dfrac{\dfrac{}{f:\alpha,\ x:\beta,\ y:\gamma \vdash f \Rightarrow \alpha}\ \text{TVAR} \qquad \dfrac{}{f:\alpha,\ x:\beta,\ y:\gamma \vdash x \Rightarrow \beta}\ \text{TVAR}}{f:\alpha,\ x:\beta,\ y:\gamma \vdash f\ x \Rightarrow \beta'}\ \text{TAPP} \qquad \dfrac{}{\cdots \vdash y \Rightarrow \gamma}\ \text{TVAR} \qquad \dfrac{\dfrac{}{\cdots \vdash x \Rightarrow \beta}\ \text{TVAR} \quad \cdots \vdash 2 \Rightarrow int}{\cdots \vdash 2 + x \Rightarrow int}}{f:\alpha,\ x:\beta,\ y:\gamma \vdash if\ f\ x\ then\ y\ else\ 2+x \Rightarrow \gamma}$$

$\beta = int$

3× T-FN

$\vdash$ `fn` f `=>` `fn` x `=>` fn y => `if` f x `then` y `else` $2 + x \Rightarrow$ $\alpha \to \beta \to \gamma \to \gamma$ skeleton

$\beta' = bool$ $\quad \gamma = int$

$[int \to bool/\alpha,\ int/\beta,\ int/\gamma]$

solution.

How to solve constraints?

Examples ...Can we solve the following constraints?

- $\{\alpha = \texttt{int},\ \alpha \to \beta = \texttt{int} \to \texttt{bool}\}$ int/α, bool/β
- $\{\alpha_1 \to \alpha_2 = \texttt{int} \to \beta,\ \beta = \texttt{bool}\}$ bool/β, bool/α₂, int/α₁
- $\{\alpha_1 \to \alpha_2 = \texttt{int} \to \beta,\ \beta = \alpha_2 \to \alpha_2\}$ int/α₁

$\alpha_2 = \beta \quad \beta = \alpha_2 \to \alpha_2$

no instantiation.

Constraint Solving via Unification

> Two types T_1 and T_2 are *unifiable*
> if there exists an instantiation σ for the type variables in T_1 and T_2
> s.t. $[\sigma]T_1 = [\sigma]T_2$, i.e $[\sigma]T_1$ is syntactically equal to $[\sigma]T_2$.

Unification via Rewriting Constraints

Given a set of constraints $\mathcal{C}$ try to simplify the set until we derive the empty set.

We write C for $C_1, \ldots, C_n$ and we assume constraints can be reordered.

$$
\begin{array}{lll}
\{C,\ \texttt{int} = \texttt{int}\} & \Longrightarrow \{C\} & \\
\{C,\ \texttt{bool} = \texttt{bool}\} & \Longrightarrow \{C\} & \\
\{C,\ \alpha = \alpha\} & \Longrightarrow \{C\} & \\
\{C,\ (T_1 \to T_2) = (S_1 \to S_2)\} & \Longrightarrow \{C,\ T_1 = S_1,\ T_2 = S_2\} & \\
\{C,\ \alpha = T\} & \Longrightarrow \{[T/\alpha]C\} & \text{provided that } \alpha \notin FV(T) \\
\{C,\ T = \alpha\} & \Longrightarrow \{[T/\alpha]C\} & \text{provided that } \alpha \notin FV(T)
\end{array}
$$

occurs check

Example : $\alpha = \beta \to \gamma,\ \beta = \alpha \quad \Longrightarrow \quad ?$ Fail

To summarize ...

Unification is a fundamental algorithm to determine whether two objects can be made syntactically equal.

Take-Away

Two Uses and Views of Type variables :

- Polymorphism : For **all** instantiations of a type variable, the expression is well-typed.
- Polymorphic Type Inference: There **exists** an instantiation for the type variables s.t. the expression is well-typed

Unification :

- Find an instantiations for (type) variables s.t. all equations (constraints) are true

Polymorphic Type Inference:

- Follows typing rules, introduces type variables for unknown types, and generates constraints.
- We succeed, if the constraints are unifiable (i.e. can be solved).