Type Inference

1

Project Schedule

				PS6
Tue Oct 22 / Fri Oct 25	Parametric Polymorphism and Hindley Milner Type Inference		Quiz 3 on Fri Lecture_Week9	PS6 due Friday
Tue Oct 29 / Fri Nov 1	Type inference in Haskell			PS7 **Start work on project this week (or earlier)**
Tue Nov 5 / Fri Nov 8	Standard Monads: Maybe, List, State, IO, Continuation	Ch. 12		PS7 due Tuesday, PS8
Tue Nov 12 / Fr Nov 15	Parsing Theory; Parsing with Monads	Ch.12		PS8 due Tuesday **Checkpoint #1: attend office hours this week (or earlier)**
Tue Nov 19 / Fri Nov 22	Functors and Applicative Functors	Ch.12		**5-8 min presentation in class on Friday**
Tue Nov 26	Effectful Programming	Ch. 12		PS9
Tue Dec 3 Fri Dec 6	TBD			PS9 due Tuesday **Checkpoint #2: attend office hours this week (or earlier)**
Tue Dec 10	Project presentations			**Project due 5-8 min presentation in class**

2

Outline

- Simple type inference (last week)
  - Expressions, types and type environment
  - Goal and intuition
  - Equality constraints
  - Substitution
  - Robinson's unification
  - Type inference strategies
    - Algorithm V (Strategy One) and
    - Algorithm V (Strategy Two)

3

Outline

- Hindley Milner (also known as Milner Damas)
  - Monotypes (types) and polytypes (type schemes)
  - Instantiation and generalization
  - Algorithm W
  - Observations
- Type inference in Haskell

  Extends classical system
  - Type signatures
  - Class constraints
  - Implication constraints

4

Simple Type Inference

Covered last week

Moving on

Type Inference Strategies

Strategy One aka constraint-based typing (Haskell)

Traverse expression's parse tree and generate constraints. Solve constraints offline producing substitution map S. Finally, apply S on expression tyvar to infer the <u>principal type</u> of expression

Strategy Two (Classical Hindley Milner)

Generate and solve constraints on-the-fly while traversing parse tree. Build and apply substitution map incrementally

Constraint Generation Strategy One

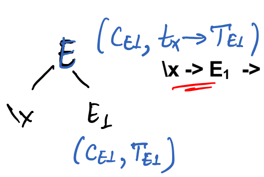
Type Env. Expression

def V(Γ, E) = case E of

Constraints Type

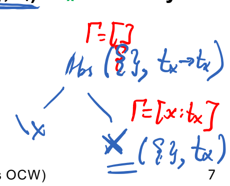
c -> ({}, TypeOf(c))

x -> if (x NOT in Dom(Γ)) then *fail*
else ({}, Γ(x))



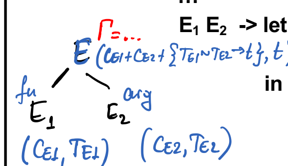
Constraints Type

\x -> E_1 -> let (C_{E1}, T_{E1}) = V(Γ+{x:t_x},E_1) – t_x is fresh tyvar
in $(C_{E1}, t_x \to T_{E1})$



def V(Γ, E) = case E of

...



E_1 E_2 -> let (C_{E1}, T_{E1}) = V(Γ,E_1)
(C_{E2}, T_{E2}) = V(Γ,E_2)
in $(C_{E1} + C_{E2} + \{T_{E1} \sim T_{E2} \to t\}, t)$ -- t is fresh tyvar

let x = E_1 in E_2 -> let (C_{E1}, T_{E1}) = V(Γ+{x:t_x},E_1)
(C_{E2}, T_{E2}) = V(Γ+{x:T_{E1}},E_2)
in $(C_{E1} + C_{E2} + \{t_x \sim T_{E1}\}, T_{E2})$

Constraints: $\{t_f \sim Int \to t_1\}$ Constraints: $\{t_x \sim Int\}$

```
(\f -> f 5) (\x -> x + 1)
```

$:: t_2$

Type: $t_f \to t_1$ Type: $t_x \to Int$

$(\{t_f \sim Int \to t_1, t_x \sim Int, t_f \to t_1 \sim (t_x \to Int) \to t_2\}, t_2)$

$(Int \to t_1) \to t_1 \sim (t_x \to Int) \to t_2$

$\Gamma = []$

Abs $(\{t_x \sim Int\}, t_x \to Int)$ Abs $(\{t_f \sim Int \to t_1\}, t_f \to t_1)$

\x + $(\{t_x \sim Int\}, Int)$ \f $\Gamma = [f : t_f]$

x 1 App $(\{t_f \sim Int \to t_1\}, t_1)$

$(\{\}, t_x)$ $(\{\}, Int)$ f 5

$(\{\}, t_f)$ $(\{\}, Int)$

All constraints:
$\{t_f \sim Bool \to t_2, Num\ t_x, t_f \to t_2 \sim (t_x \to t_x) \to t_1\}$

Constraints: $\{t_f \sim Bool \to t_2\}$ $\{Num\ t_x\}$

```
> (\f -> f True) (\x -> x + 1)
```

Type: $t_f \to t_2$ $t_x \to t_x$

```
• No instance for (Num Bool) arising from a use of '+'
• In the expression: x + 1
  In the first argument of '\ f -> f True',
    namely '(\ x -> x + 1)'
  In the expression: (\ f -> f True) (\ x -> x + 1)
```

Solving all constraints: first subst $Bool \to t_2 / t_f$

$\{Num\ t_x, (Bool \to t_2) \to t_2 \sim (t_x \to t_x) \to t_1\}$ // Decompose

$\{Num\ t_x, Bool \to t_2 \sim t_x \to t_x, t_2 \sim t_1\}$ // Decompose

$\{Num\ t_x, Bool \sim t_x, t_2 \sim t_x, t_2 \sim t_1\}$ // Subst Bool for t_x, t_1, t_2

$\{Num\ Bool\}$ // Residual constraint. No solution, ERROR!


```
let f = \x -> x in f 1
```


On-the-fly Generation and Resolution

Strategy Two

Substitution Map Type

def V(Γ, E) = case E of

c -> ([], TypeOf(c))

x -> if (x NOT in Dom(Γ)) then *fail*

else ([], T_E)

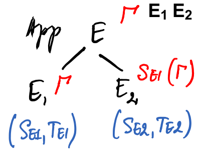
Abs E Γ... \x -> E_1 -> let (S_{E1},T_{E1}) = V(Γ+{x:t_x},E_1)

$(S_{E1}, S_{E1}(t_x) \to T_{E1})$ in (S_{E1}, S_{E1}(t_x)→T_{E1})

\x E_1 $\Gamma = [x : t_x, ..]$

(S_{E1}, T_{E1})

def V(Γ, E) = case E of

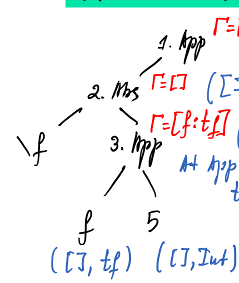


E₁ E₂ -> let $(S_{E1}, T_{E1}) = V(\Gamma, E_1)$
$(S_{E2}, T_{E2}) = V(S_{E1}(\Gamma), E_2)$
$S = \text{Unify}(S_{E2}(T_{E1}), T_{E2} \to t)$
in $(S\ S_{E2}\ S_{E1}, S(t))$ // $S\ S_{E2}\ S_{E1}$

Need to UNIFY: $S_{E2}(T_{E1}) \sim T_{E2} \to t$

let x = E_1 in E_2 -> let $(S_{E1}, T_{E1}) = V(\Gamma + \{x : t_x\}, E_1)$
$S = \text{Unify}(S_{E1}(t_x), T_{E1})$
$(S_{E2}, T_{E2}) = V(S\ S_{E1}(\Gamma) + \{x : S(T_{E1})\}, E_2)$
in $(S_{E2}\ S\ S_{E1}, T_{E2})$


```
(\f -> f 5) (\x -> x + 1)
```



Applied subst on t_f here.

$([Int \to t_3 / t_f], (Int \to t_3) \to t_3)$

$([Int \to t_3 / t_f], t_3)$

At App we need to unify:
$t_f \sim Int \to t_3$ leading to $[Int \to t_3 / t_f]$

Outline

- Hindley Milner (also known as Milner Damas)
  - Monotypes (types) and polytypes (type schemes)
  - Instantiation and generalization
  - Algorithm W
  - Observations
- Back to Haskell
  - Type signatures
  - Class constraints
  - Implication constraints

Motivating Example

A sound type system rejects some good programs

Canonical example

let f = \x -> x
in
if (f True) then (f 1) else 1

This is a good program, it does not “get stuck“
Term is NOT typable in Simple types
It is typable in Hindley Milner!

Towards Hindley Milner

let f = \x -> x

in

if (f True) then (f 1) else 1

Constraints

$t_f \sim t_1 \to t_1$

$t_f \sim bool \to t_2$ // at call **(f True)**

$t_f \sim int \to t_3$ // at call **(f 1)**

Does not unify!

Towards Hindley Milner

Solution:

Generalize the type variable in type of **f**

$t_f : t_1 \to t_1$ becomes $t_f : \forall t_1 . t_1 \to t_1$

Different uses of generalized type variables are instantiated differently

(f True) instantiates t_f into $u_1 \to u_1$ (u_1 is fresh)

$u_1 \to u_1$ unifies with $Bool \to t_2$, no problem

E.g., **(f 1)** instantiates t_f into $u_2 \to u_2$ (u_2 is fresh)

When can we generalize?

Expression Syntax (to study Hindley Milner)

Expressions:

Only place we can generalize

E ::= c | x | \x -> E_1 | E_1 E_2 | let x = E_1 in E_2

Only place we instantiate

There are no types in the syntax

fun1 = expr1
fun2 = expr2 equiv
> expr3

let
fun1 = expr1
fun2 = expr2
in
expr3

The type of each sub-expression is derived by the Hindley Milner type inference algorithm

Type Syntax (to study Hindley Milner)

Types (aka monotypes):

$\tau ::= b \mid \tau_1 \to \tau_2 \mid t$ — **t** is a type variable

E.g., **Int, Bool, Int→Bool, $t_1 \to Int$, $t_1 \to t_1$, etc.**

Type schemes (aka polymorphic types):

$\sigma ::= \tau \mid \forall t.\sigma$ $\forall t_1 . \forall t_2 . \forall t_3 . t_1 \to t_2 \to t_3$

E.g., $\forall t_1 . \forall t_2 . (Int \to t_1) \to t_2 \to t_3$ — t_3 is a "free" type variable as it isn't bound under ∀

Note: all quantifiers appear in the beginning, τ cannot contain schemes

Type environment now

Gamma ::= Identifiers → Type schemes

Instantiations

Type scheme $\sigma = \forall t_1 \ldots t_n . \tau$ can be instantiated into a type τ' by substituting types for the bound variables (**BV**) under the universal quantifier $\forall$

$\tau' = S\,\tau$ **S** is a substitution s.t. Domain(**S**) $\supseteq$ **BV**(σ)

τ' is said to be an instance of σ ($\sigma > \tau'$)

τ' is said to be a generic instance when **S** maps type variables to new (i.e., fresh) type variables

E.g., $\sigma = \forall t_1 t_2 . (\text{Int} \to t_1) \to t_2 \to t_3$

$(\text{Int} \to u_1) \to u_2 \to t_3$ // can't touch t_3

$(\text{Int} \to u_3) \to u_4 \to t_3$

E.g., $\sigma = \forall t_1 . t_1 \to t_1$

$b_1 \to b_1$

$u_1 \to u_1$

Generalization (aka Closing)

We can generalize a type τ as follows

$\textbf{Gen}(\Gamma,\tau) = \forall t_1,\ldots t_n . \tau$

where $\{t_1 \ldots t_n\} = \textbf{FV}(\tau) - \textbf{FV}(\Gamma)$

Generalization introduces polymorphism

Quantify type variables that are free in τ but are not free in the type environment Γ

E.g., $\textbf{Gen}([], t_1 \to t_2)$ yields $\forall t_1 t_2 . t_1 \to t_2$

E.g., $\textbf{Gen}([x{:}t_2], t_1 \to t_2)$ yields $\forall t_1 . t_1 \to t_2$

instantiates into $u_1 \to t_2$

instantiates into $u_2 \to t_2$

let f = $\overbrace{\backslash x \to x}^{E_1}$ in $\overbrace{\text{if (f True) then (f 1) else 1}}^{E_2}$

1. Infer type for **\x -> x** : $t_x \to t_x$ (a monotype)
2. Generalize type using **Gen**([],$t_x \to t_x$): $\forall t_x . t_x \to t_x$ (a type scheme)

$\Gamma = [f : \forall t_x . t_x \to t_x]$

3. Pass type scheme to **if (f True) then (f 1) else 1**
4. Instantiate for each **f** in **if (f True) then (f 1) else 1**

   [u_1/t_x] ($t_x \to t_x$) where u_1 is fresh tyvar at **(f True)**

   $u_1 \to u_1 \sim Bool \to t_k$

   [u_2/t_x] ($t_x \to t_x$) where u_2 is fresh tyvar at **(f 1)**

   $u_2 \to u_2 \sim Int \to t_{k+1}$

When can we generalize?

Correct type: $\overbrace{(t_1 \to t_2)}^{f} \to \overbrace{t_1}^{x} \to \overbrace{t_2}^{result}$

Consider expression **\f -> \x-> let g = f in g x**

Gen([f:t_f,x:t_x],t_f) yields what? $\cancel{\forall tf. tf}$ or just tf ?

If we generalize f: ∀tf.tf it will lead to type tf → tx → t2 for the term, which is incorrect. It loses the connection between f and x and allow typing of (\f → \x → let g = f in g x) (+1) True. UNSOUND.

DO NOT generalize variables that are mentioned in type environment **Γ**!

Hindley Milner Type Inference, Rough Sketch

let x = E_1 in E_2

1. Calculate type T_{E1} for E_1 in **Γ;x:t_x** ; T_{E1} is a monotype
2. Generalize free type variables in T_{E1} to get the type scheme for T_{E1} (be mindful of caveat!)
3. Extend environment with **x:Gen(Γ,T_{E1})** and start typing E_2
4. Every time algorithm sees **x** in E_2, it instantiates x's type scheme using fresh type variables

   E.g., **id**'s type scheme is $\forall t_1 . t_1 \to t_1$ so **id** is instantiated to $u_k \to u_k$ at **(id 1)**

Hindley Milner Type Inference

Just like with Simple types, there are two strategies

Strategy One

Simple types extended with generalization and instantiation

Generate all constraints, then solve

Strategy Two

Again, simple types with generalization and instantiation

Generate and solve constraints on-the-fly

This is classical Algorithm W

Example

$\backslash x \text{ -> let } f = \overbrace{\backslash y \text{ -> } x}^{E_1} \text{ in } \overbrace{(f \text{ True}, f\ 1)}^{E_2} :: t_x \to (t_x, t_x)$

1. Type $\backslash y \to x$ Yields $t_y \to t_x$
2. Generalize $t_y \to t_x$ $Gen([x{:}t_x], t_y \to t_x)$ yields $\forall t_y . t_y \to t_x$
3. Type E_2 in $\Gamma = [f : \forall t_y . t_y \to t_x,\ x{:}t_x]$
4. $(f \text{ True})$ instantiates $\forall t_y . t_y \to t_x$ into $u_1 \to t_x \sim Bool \to V_1$

$(f\ 1)$ instantiates $\forall t_y . t_y \to t_x$ into $u_2 \to t_x \sim Int \to V_2$

Strategy Two: Algorithm W

u_1 to u_n are fresh type vars generated at instantiation of polymorphic type

def W(Γ, E) = case E of
- c -> ([], TypeOf(c))
- x -> if (x NOT in Domain(Γ)) then *fail*
  - else let $T_E = \Gamma(x)$
  - in case T_E of
    - $\forall t_1,\dots t_n.\tau$ -> ([],[$u_1/t_1 \dots u_n/t_n$] τ) // polytype, instantiate
    - _ -> ([], T_E) // monotype, return
- \x -> E_1 -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x{:}t_x\},E_1)$
  - in $(S_{E1}, S_{E1}(t_x) \to T_{E1})$

// ...

// continues on next slide!

def W(Γ, E) = case E of

// continues from previous slide

// ...

$E_1\ E_2$ -> let $(S_{E1},T_{E1}) = W(\Gamma,E_1)$

$(S_{E2},T_{E2}) = W(S_{E1}(\Gamma),E_2)$

$S = Unify(S_{E2}(T_{E1}),T_{E2} \to t)$

in $(S\ S_{E2}\ S_{E1}, S(t))$

let x = E_1 in E_2 -> let $(S_{E1},T_{E1}) = W(\Gamma+\{x{:}t_x\},E_1)$

$S = Unify(S_{E1}(t_x),T_{E1})$

$\sigma = Gen(S\ S_{E1}(\Gamma), S(T_{E1}))$

$(S_{E2},T_{E2}) = W(S\ S_{E1}(\Gamma)+\{x{:}\sigma\},E_2)$

in $(S_{E2}\ S\ S_{E1}, T_{E2})$

Strategy Two Example

let f = $\boxed{\backslash x\text{->}x}$ in $\boxed{\text{if (f True) then (f 1) else 1}}$

1. let Γ = [] T_1 = int, S_1 = ...
- f
- 2. Abs: Γ = [f:t_f], $T_2 = t_x \to t_x$, S_2 = []
  - λx: t_x
  - x: Γ = [x:t_x f:t_f]
- 3. if-then-else: Γ = [f: $\forall t_x . t_x \to t_x$], T_3 = int, S_3 = ...
  - 4. App: T_4 = bool, S_4 = [bool/t_4][bool/u_1]
    - f: T = $u_1 \to u_1$, S = []
    - true
  - 5. App: T_5 = int, S_5 = [int/t_5][int/u_2]
    - f
    - 1
  - 1

No constraint, types **2. Abs** immediately: $T_2 = t_x \to t_x$: $[t_x \to t_x/t_2]$

$\sigma = Gen([],t_x \to t_x) = \forall t_x .\ t_x \to t_x$

From **Unify**($u_1 \to u_1$, bool $\to t_4$)

Example

\x -> let f = \y -> x in (f True, f 1)

Hindley Milner Observations

Notes

- Do not generalize over type variables mentioned in type environment (they are used elsewhere)
- let is the only way of defining polymorphic constructs
- Generalize the types of let-bound identifiers **only after** processing their definitions

only here one can generalize τ into a σ

$\text{let } \underbrace{X = E_1}\text{ in } \underbrace{E_2}$

here one instantiates σ.

Hindley Milner Observations

- Generates the most general type (principal type) for each term/subterm
- Type system is sound
- Complexity of Algorithm W

  It is PSPACE-Hard because of nested let blocks

Hindley Milner Limitations

- Only let-bound constructs can be polymorphic and instantiated differently

let twice f x = f (f x)

in twice twice succ 4 // let-bound polymorphism

$(u_1 \to u_1) \to u_1 \to u_1 \quad (u_2 \to u_2) \to u_2 \to u_2$

let twice f x = f (f x)

foo g = g g succ 4 // lambda-bound

in foo twice

```
(\x -> x (\y -> y) (x 1)) (\z -> z)
```

```
let x = (\z -> z)
in
    x (\y -> y) (x 1)
```