Hindley Milner

1

So Far

- Simple type inference
  - 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)

Algorithm W

2

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 principal type 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   Constraints   Type

def V(Γ, E) = case E of

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

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

$\Gamma = ..$ $(C_{E1}, t_x \to T_{E1})$

E

λx E_1 $\Gamma = [x:t_x]$

(C_{E1}, T_{E1})

\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→T_{E1})

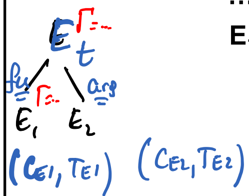
Abs $\Gamma = [\,]$ $(\{\}, t_x \to t_x)$

λx x $\Gamma = [x:t_x]$

$(\{\}, t_x)$

def V(Γ, E) = case E of

…



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

$(C_{E2},T_{E2}) = V(\Gamma,E_2)$

in $(C_{E1} + C_{E2} + \{T_{E1} \sim T_{E2} \to t\}, t)$ -- t is fresh tyvar

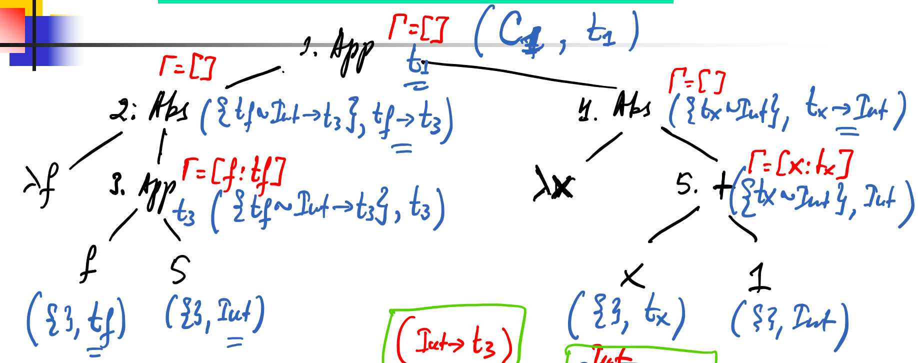
let x = E_1 in E_2 -> let $(C_{E1},T_{E1}) = V(\Gamma+\{x:t_x\},E_1)$

let is letrec!

$(C_{E2},T_{E2}) = V(\Gamma+\{x:T_{E1}\},E_2)$

in $(C_{E1} + C_{E2} + \{t_x \sim T_{E1}\}, T_{E2})$


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



$C_1 = \{ t_x \sim Int,\ tf \sim Int \to t_3,\ tf \to t_3 \sim (t_x \to Int) \to t_1 \}$

Unify Set $(C_1) = [\,Int/t_x,\ Int \to t_3/tf,\ Int/t_3,\ Int/t_1\,]$

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

$\underbrace{\texttt{\textbackslash x -> x}}_{t_x \to t_x}$ in $\overbrace{\underbrace{\texttt{f 1}}_{\Gamma = [f : t_x \to t_x]}}^{t_1}$ :: Int

$t_x \to t_x \sim Int \to t_1$

On-the-fly Generation and Resolution

Strategy Two

subst map, Type

def V(Γ, E) = case E of

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

x -> if (x NOT in Dom(Γ)) then *fail*
else ([], Γ(x))

\x -> E_1 -> let (S_{E1},T_{E1}) = V(Γ+{x:t_x},E_1)
in (S_{E1}, $S_{E1}(t_x) \to T_{E1}$)

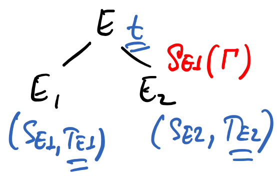
def V(Γ, E) = case E of

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

(S_{E2},T_{E2}) = V(S_{E1}(Γ),E_2)

S = Unify(S_{E2}(T_{E1}),T_{E2}→t)

in (S S_{E2} S_{E1}, S(t)) // S S_{E2} S_{E1}



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

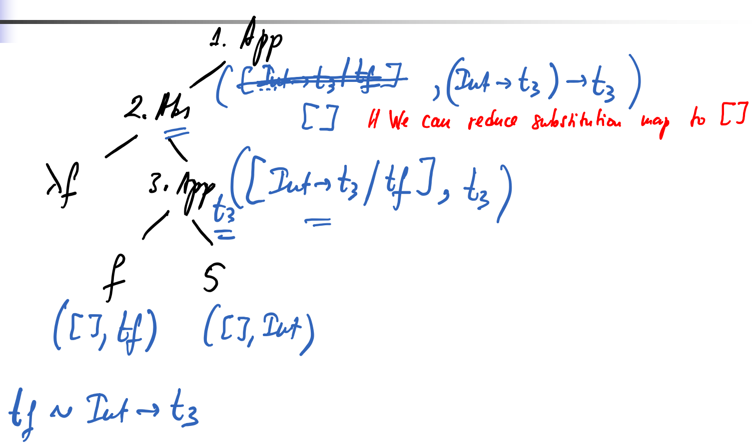
let x = E_1 in E_2 -> let (S_{E1},T_{E1}) = V(Γ+{x:t_x},E_1)

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

(S_{E2},T_{E2}) = V(S S_{E1}(Γ)+{x:S(T_{E1})},E_2)

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


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



$t_f \sim Int \to t_3$

Outline

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

Towards Hindley Milner

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!

let f = \x -> x

in

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

Constraints

$t_f \sim t_1 \rightarrow t_1$

$t_f \sim Bool \rightarrow t_2$ // at call **(f True)**

$t_f \sim Int \rightarrow t_3$ // at call **(f 1)**

Does not unify!

Solution:

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

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

Different uses of generalized type variables are instantiated differently

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

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

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

When can we generalize?

Expression Syntax (to study Hindley Milner)

Expressions:

$\Gamma = [x: \forall t \ldots]$

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

Let is the only place where we introduce polymorphism

There are no types in the syntax

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

Type Syntax (to study Hindley Milner)

Types (aka monotypes):

τ ::= **b** | $\tau_1 \to \tau_2$ | **t** ← **t** is a type variable

E.g., **Int, Bool, Int→Bool, t_1→Int, t_1→t_1, etc.**

Type schemes (aka polymorphic types):

σ ::= τ | $\forall$**t.**σ $\forall t_1 . \forall t_2 . \forall t_3 . \tau$

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

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

Type environment now

Gamma ::= Identifiers → Type schemes

Instantiations

Turns a σ (polytype) into a τ (monotype)

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 ($\sigma > \tau'$)

We are interested in generic instances of type schemes.

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

$\forall t_1 . t_1 \to t_1$ $u_1 \to u_1$ $u_2 \to u_2$ $u_3 \to u_3$

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

$\tau[a/t_1, b/t_2] = ((\text{Int} \to t_1) \to t_2 \to t_3)[a/t_1, b/t_2] =$

$(\text{Int} \to a) \to b \to t_3$

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

$a \to a$

$b \to b$

Generalization (aka Closing)

Turns a τ (a monotype) into a σ (a polytype).

We can generalize a type τ as follows

Gen(Γ,τ) = $\forall t_1,\ldots t_n.\tau$

where $\{t_1 \ldots t_n\}$ = **FV**(τ) – **FV**(Γ)

Generalization introduces polymorphism

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

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

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

instantiates into

$u_1 \to t_2$ $\qquad$ $u_2 \to t_2$

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)**

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

When can we generalize? $\overbrace{(t_x \to t_3)}^{\text{type of } f} \to \overbrace{t_x}^{\text{type of } x} \to \overbrace{t_3}^{\text{result}}$ is the correct type!

Consider expression **\f -> \x-> let g = f in** $\overbrace{\textbf{g x}}^{t_3}$

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

Suppose we generalize. At g x we instantiate g : $u \sim t_x \to t_3$

Therefore, the type of the term ends up being $t_f \to t_x \to t_3$, losing the connection between t_f, t_x and t_3. It is 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 $\Gamma;x:t_x$; 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 $x:Gen(\Gamma,T_{E1})$ 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

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

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 = Γ(x)

in case T_E of

$\forall$ $t_1, \ldots t_n . \tau$ -> ([],[$u_1/t_1 \ldots u_n/t_n$] τ)

_ -> ([], T_E)

\x -> E_1 -> let (S_{E1},T_{E1}) = W(Γ+{x:t_x},E_1)

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

// ...

// continues on next slide!

def W(Γ, E) = case E of

// continues from previous slide

// ...

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

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

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

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

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

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

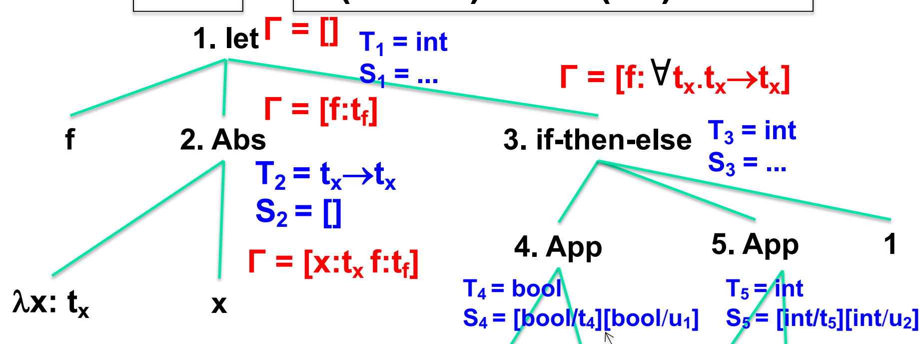
$\sigma = Gen(\ S\ S_{E1}(\Gamma),\ S(T_{E1})\ )$

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

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

Strategy Two Example

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



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

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

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

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

Programming in Haskell, A Milanova (from MIT 2015 Program Analysis OCW) 31

31

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

let twice f x = f (f x)
foo g = g g succ 4 // lambda-bound
in foo twice

Programming in Haskell, A Milanova 32

32

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

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