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
  - 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

Type Inference

The task of type inference is to

E.g. E => 1 + True × NO

Reject bad programs with a decent error message

Elaborate good programs

Type Inference

Every well-formed expression in Haskell has a type

In general, we don't need to write type signatures, Haskell can figure out (many) signatures. E.g.:

```
> fun = \x -> \y -> x
> :t fun
```

$t_1 \to t_2 \to t_1$

```
> twice = \f -> \x -> f (f x)
> :t twice
```

$(t \to t) \to t \to t$

> twice (+1) 0 → 2

> twice not True → True

> twice twice (+1) 0 →

Foundation is an algorithm known as Hindley Milner type inference (also, Milner Damas)

Haskell builds on Hindley Milner to account for, most notably, user-defined types, ADTs and pattern matching, type classes and type class constraints

Classical Hindley Milner solves constraints on-the-fly

Haskell first generates constraints, then solves them "offline"

Simple Type Inference

Inference of the so-called simple types

Formally known as System F1 or Simply Typed Lambda Calculus

length1 :: [Bool] -> Int
length2 :: [Int] -> Int

NO polymorphism, i.e., functions work on a single type

Hindley Milner extends simple type inference with so-called let-polymorphism. Functions work on many different types

Important concepts: expressions and types, type environment, equality constraints, substitution, unification

Expressions

A minimal language, very close to Lambda calculus:

E ::= c | x | \x -> E_1 | E_1 E_2 |
let x = E_1 in E_2 |
E_1 + E_2 |
if E_1 then E_2 else E_3

There are no types in syntax

The type of each subexpression is derived by simple type inference

Types

Types (as known as simple types or monotypes):

τ ::= b | $\tau_1 \rightarrow \tau_2$ | t

t is a type variable (tyvar)

b is a base type
Assume **Int** and **Bool**

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

Int → Bool , Int , $t_1 \rightarrow t_2 \rightarrow t_1$

Type Environment

Type environment Gamma maps identifiers (variables) to types:

Gamma ::= Identifiers -> Types

For example, we can only type subexpression

`(f x)` [f::t→t, x::t] ⊢ (f x) ::t

in a type environment that binds identifies `f` and x to types. E.g., in `Gamma = [f :: t -> t, x :: t]`

Goal and Intuition

Given `\x -> \y -> x`

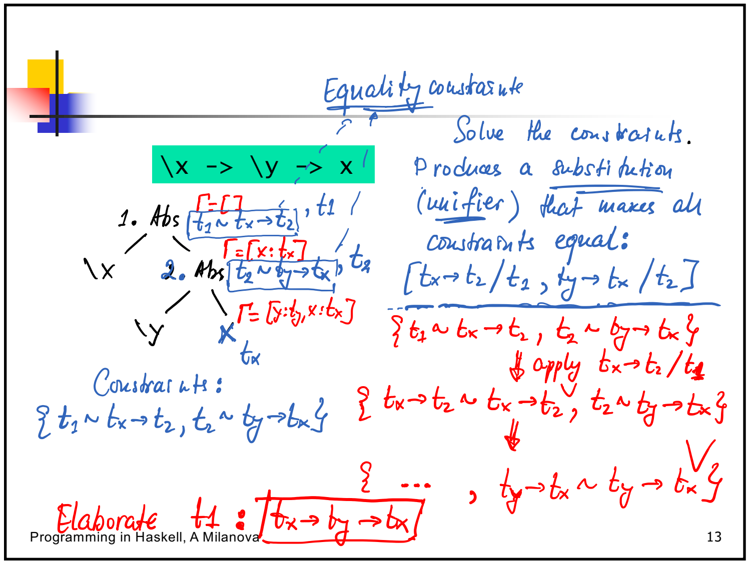
Deduce `\x -> \y -> x :: t1 -> t2 -> t1`

1. Construct parse tree for expression. Associate a fresh tyvar to each identifier and each subexpression

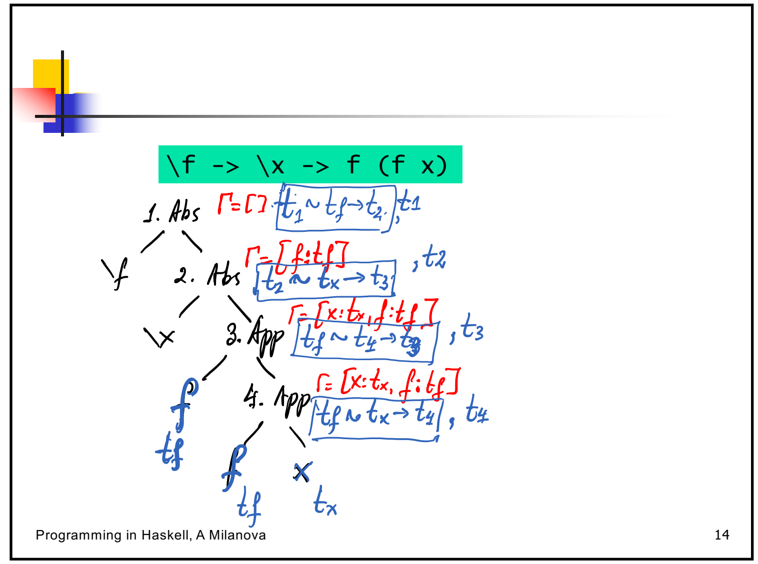
2. Generate equality constraints

3. Solve equality constraints using unification

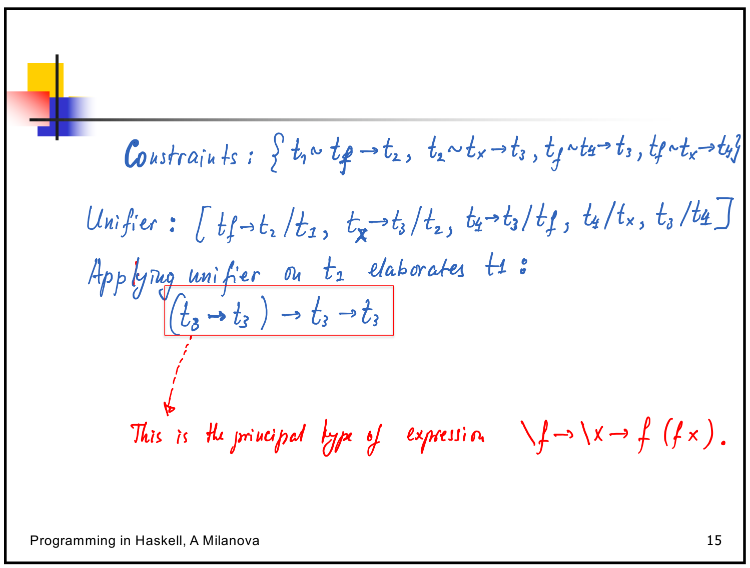
4. Deduce type for expression


Equality constraints
Solve the constraints.
Produces a substitution (unifier) that makes all constraints equal:
\x -> \y -> x
1. Abs
2. Abs
Constraints:
Elaborate t1 : tx → ty → tx
Programming in Haskell, A Milanova
13

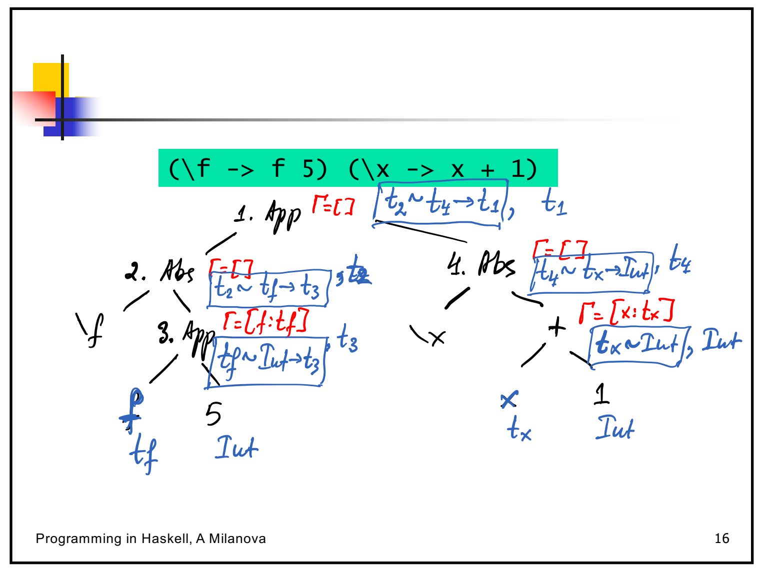
13


\f -> \x -> f (f x)
1. Abs
2. Abs
3. App
4. App
Programming in Haskell, A Milanova
14

14


Constraints :
Unifier :
Applying unifier on t1 elaborates t1 :
This is the principal type of expression \f → \x → f (f x).
Programming in Haskell, A Milanova
15

15


(\f -> f 5) (\x -> x + 1)
1. App
2. Abs
3. App
4. Abs
5
1
Programming in Haskell, A Milanova
16

16

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


Equality Constraints

Two key concepts

- <u>Equality</u>
  - What does it mean for two types to be equal?
  - Structural equality
- <u>Unification</u>
  - Can two types be made equal by choosing appropriate <u>substitutions</u> for their type variables?
  - Robinson's unification algorithm

What does it mean for two types τ_a and τ_b to be equal?

Structural equality

Suppose $\tau_a = t_1 \rightarrow t_2$

$\tau_b = t_3 \rightarrow t_4$

Structural equality entails

$\tau_a \sim \tau_b$ means $t_1 \rightarrow t_2 \sim t_3 \rightarrow t_4$ iff $t_1 \sim t_3$ and $t_2 \sim t_4$

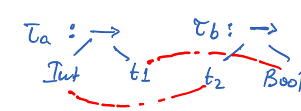
Can two types be made equal by choosing appropriate substitutions for their type variables?

Robinson's unification algorithm

Suppose $\tau_a = \textbf{Int} \rightarrow t_1$

$\tau_b = t_2 \rightarrow \textbf{Bool}$

Can we unify τ_a and τ_b?



Yes, if **Bool**/t_1 and **Int**/t_2

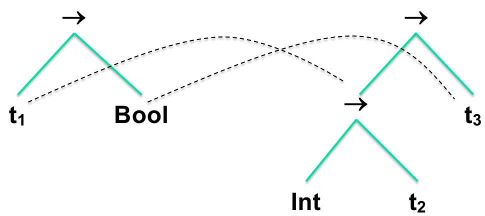
Suppose $\tau_a = \textbf{Int} \rightarrow t_1$

$\tau_b = \textbf{Bool} \rightarrow \textbf{Bool}$

Can we unify τ_a and τ_b? No.

Example

$t_1 \to \text{Bool} \sim (\text{Int} \to t_2) \to t_3$



Yes, if $\text{Int}\to t_2/t_1$ and Bool/t_3

Substitution

Language of types

τ ::= **b** // base type: **Int** and **Bool**

| **t** // type variable (tyvar)

| $\tau_1 \to \tau_2$ // function type

A substitution is a map

S : Type Variable → Type

S = [τ_1/t_1, … τ_n/t_n] // substitute type τ_i for tyvar t_i

A substitution instance $\tau' = S\,\tau$

S = [$t_0\to$Bool / t_1] $\tau = t_1\to t_1$ then

S τ = S($t_1\to t_1$) = ($t_0\to$Bool) → ($t_0\to$Bool)

Exercises

Substitutions can be composed

S_1 = [$t_0\to$Bool/t_1]

S_2 = [Int/t_0]

$\tau = t_1\to t_1$

$S_2 S_1 \tau = S_2 (S_1(t_1\to t_1))$ = ?

$S_2 S_1 = [Int/t_0] \circ [t_0\to Bool/t_1]$

same as

$S = [t_0\to Bool/t_1, Int/t_0]$

$(t_1\to t_1)[t_0\to Bool/t_1][Int/t_0] =$

$((t_0\to Bool) \to t_0\to Bool)[Int/t_0] =$

$(Int\to Bool) \to Int\to Bool$

Substitutions can be composed

S_1 = [t_x/t_1]

S_2 = [t_x/t_2]

$\tau = t_2\to t_1$

$S_2 S_1\ \tau$ = ?

$(t_2\to t_1)[t_x/t_1][t_x/t_2] =$

$t_x\to t_x$

Substitutions can be composed

$S_1 = [\, t_1/t_2 \,]$

$S_2 = [\, t_3/t_1 \,]$

$S_3 = [\, t_4 \to Int/t_3 \,]$

$\tau = t_1 \to t_2$

$S_3\, S_2\, S_1\, \tau = ?$

$(t_1 \to t_2)[t_1/t_2][t_3/t_1][t_4 \to Int/t_3] =$

$(t_1 \to t_1)[t_3/t_1][t_4 \to Int/t_3] =$

$(t_3 \to t_3)[t_4 \to Int/t_3] = (t_4 \to Int) \to t_4 \to Int$

Principal Unifier

A unifier is a substitution that unifies (i.e., makes equal) a set of constraints

A principal unifier is a most general unifier of a set of constraints

$\{ t_1 \to t_1 \sim t_2 \to t_3 \}$ $[t_1/t_2, t_1/t_3]$

$\{ (t_1 \to t_1) \to t_1 \to t_1 \sim t_2 \to t_3 \}$

$[t_1 \to t_1/t_3,\ t_1 \to t_1/t_2]$ MOST GENERAL

$[Int \to Int/t_3,\ Int \to Int/t_2,\ Int/t_1]$ MORE SPECIFIC

Exercise

A principal unifier is the most general unifier of a set of constraints

Find principal unifiers (when they exist) for

$\{ Int \to Int \sim t_1 \to t_2 \}$ $[Int/t_1, Int/t_2]$

$\{ Int \sim Int \to t_2 \}$ DNE

$\{ t_1 \sim Int \to t_2 \}$ $[Int \to t_2/t_1]$

$\{ t_1 \sim Int,\ t_2 \sim t_1 \to t_1 \}$ $[Int/t_1, Int \to Int/t_2]$

$\{ t_1 \to t_2 \sim t_2 \to t_3,\ t_3 \sim t_4 \to t_5 \}$ $[t_1/t_2, t_1/t_3, t_4 \to t_5/t_3]$

Unification

- **Unify**: tries to unify τ_1 and τ_2 and returns a **principal unifier for** $\tau_1 \sim \tau_2$ if unification is successful

def **Unify**(τ_1,τ_2) =

This is the occurs check!

case (τ_1,τ_2)

- $(\tau_1,t_2) = [\tau_1/t_2]$ provided t_2 does not occur in τ_1
- $(t_1,\tau_2) = [\tau_2/t_1]$ provided t_1 does not occur in τ_2

(b_1,b_2) = if (eq? b_1 b_2) then [ ] else ***fail***

$(\tau_{11} \to \tau_{12},\ \tau_{21} \to \tau_{22})$ = let S_1 = **Unify**(τ_{11},τ_{21})

S_2 = **Unify**$(S_1\,\tau_{12},\ S_1\,\tau_{22})$

in $S_2\, S_1$ // compose substitutions

otherwise = ***fail***

Exercise

Unify (**Int→Int**, **t_1→t_2**) yields ?

[Int/t_1, Int/t_2]

Unify (Int, t_1) → [Int/t_1]
Unify (Int, t_2) → [Int/t_2]

Unify (**Int**, **Int→t_2**) yields ?

Does Not Unify

Unify (**t_1**, **Int→t_2**) yields ?

[Int→t_2 / t_1]

Unify Set of Constraints C

Robinson's algorithm unifies (i.e., solves) a single constraint $\tau_1 \sim \tau_2$.

What if we have a set of constraints?

Intuition:

1. Pick a constraint $\tau_1 \sim \tau_2$ from the set
2. Solve $\tau_1 \sim \tau_2$ either failing or succeeding getting subst **S**
   - If fail, then done, constraints cannot be unified
   - If success, then first apply **S** on remaining constraints as S carries structure that must be taken into account , goto 1

Unify Set of Constraints C

UnifySet: tries to unify **C** and returns a **principal unifier for C** if unification is successful

def **UnifySet** (**C**) =
  if **C** is Empty Set then **[]** // Empty subsitution
  else let
    C = { $\tau_1 \sim \tau_2$ } U **C'**
    S = **Unify** (τ_1,τ_2) // **Unify** returns a substitution **S**
  in
    UnifySet (**S**(**C'**)) **S**
    // Compose the substitutions

Exercise

UnifySet { **t_1 ~ Int**, **t_2 ~ t_1→t_1** } yields ?

[Int/t_1, Int→Int/t_2]

UnifySet { **t_1→t_2 ~ t_2→t_3**, **t_3 ~ t_4→t_5** } yields ?

[t_1/t_2, t_1/t_3, t_4→t_5/t_1]

UnifySet { **t_f ~ t_2→t_1**, **t_f ~ t_x→t_2** } yields ?

[t_2→t_1/t_f, t_2/t_x, t_1/t_2]

UnifySet { **t_2 ~ t_4→t_1**, **t_2 ~ t_f→t_3**, **t_4 ~ t_x→Int**, **t_f ~ Int→t_3**, **t_x ~ Int** } yields ?

Haskell's Way

Haskell does a sequence of successive rewrites:

{ $t_2 \sim t_4 \rightarrow t_1$, $t_2 \sim t_f \rightarrow t_3$, $t_4 \sim t_x \rightarrow Int$, $t_f \sim Int \rightarrow t_3$, $t_x \sim Int$ }

$\Downarrow [Int/t_x]$

$\{ t_2 \sim t_4 \rightarrow t_1,\ t_2 \sim t_f \rightarrow t_3,\ t_4 \sim Int \rightarrow Int,\ t_f \sim Int \rightarrow t_3 \}$

$\Downarrow [Int \rightarrow Int / t_4]$

$\{ t_2 \sim (Int \rightarrow Int) \rightarrow t_1,\ t_2 \sim t_f \rightarrow t_3,\ t_f \sim Int \rightarrow t_3 \}$

$\Downarrow$

$\{ t_f \rightarrow t_3 \sim (Int \rightarrow Int) \rightarrow t_1,\ t_f \sim Int \rightarrow t_3 \}$

And so on… $\{ t_f \sim Int \rightarrow Int,\ t_3 \sim t_1,\ t_f \sim Int \rightarrow t_3 \}$ … etc.

Outline

- 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)

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

def V(Γ, E) = case E of

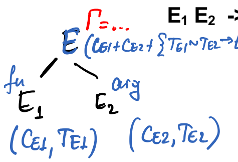
c -> ({}, TypeOf(c))

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

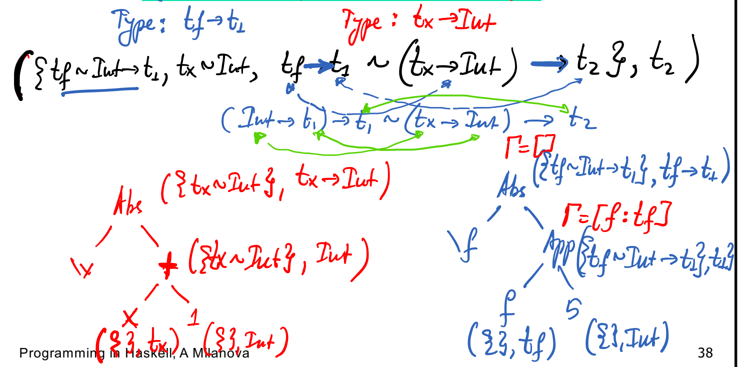
\x -> E_1 -> let (C_{E1},T_{E1}) = V(Γ+{x:t_x},E_1) – t_x is fresh tyvar in (C_{E1},$t_x \rightarrow T_{E1}$)

```
def V(Γ, E) = case E of
   …
   E1 E2 -> let (CE1,TE1) = V(Γ,E1)
                (CE2,TE2) = V(Γ,E2)
            in (CE1 + CE2 + {TE1 ~ TE2 → t}, t) -- t is fresh tyvar

   let x = E1 in E2 -> let (CE1,TE1) = V(Γ+{x:tx},E1)
                           (CE2,TE2) = V(Γ+{x:TE1},E2)
                       in (CE1 + CE2 + {tx ~ TE1}, TE2)
```




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




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

• 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)
```



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

On-the-fly Generation and Resolution

Strategy Two

def V(Γ, E) = case E of

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

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

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

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)
```


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:

$E ::= c \mid x \mid \backslash x \to E_1 \mid E_1\, E_2 \mid \text{let } x = E_1 \text{ in } E_2$

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** | $\tau_1 \rightarrow \tau_2$ | **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

E.g.,$\forall t_1.\forall t_2.(Int \rightarrow t_1) \rightarrow t_2 \rightarrow t_3$ ← 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

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., $\sigma = \forall\, t_1 t_2.(Int \rightarrow t_1) \rightarrow t_2 \rightarrow t_3$

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

Generalization (aka Closing)

We can generalize a type τ as follows

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

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

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

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

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

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)
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?

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

Gen([f:t_f,x:t_x],t_f) yields what?

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(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

\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

∀ $t_1,\ldots t_n.\tau$ -> ([],[$u_1/t_1 \ldots u_n/t_n$] 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(Γ,E_1)

(S_{E2},T_{E2}) = W(S_{E1}(Γ),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(Γ+{x:t_x},E_1)

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

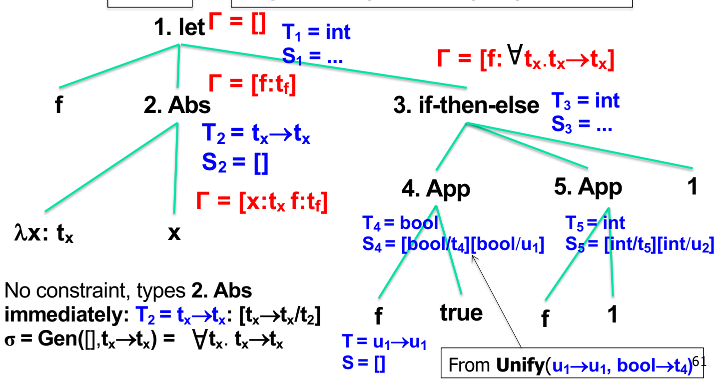
σ = Gen(S S_{E1}(Γ), S(T_{E1}))

(S_{E2},T_{E2}) = W(S S_{E1}(Γ)+{x:σ},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



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

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


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

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