




Hindley Milner, Conclusion



1




Announcements

- Paper list is up
 1. Pick available paper and slot
 2. Send me an email
 3. If still available, I'll pencil you in; otherwise, goto 1

Programming in Haskell, A Milanova

2

2




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)

Programming in Haskell, A Milanova 3

3



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

Programming in Haskell, A Milanova 4

4




Outline

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

Programming in Haskell, A Milanova 5

5



Expression Syntax (to study Hindley Milner)

Expressions:

$$E ::= c \mid x \mid \lambda x \rightarrow E_1 \mid E_1 E_2 \mid \text{let } x = E_1 \text{ in } E_2$$

only place to introduce a polyomorphic type
 $x : \sigma$
 σ is instantiated 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](#)

Programming in Haskell, A Milanova (from MIT's 2015 Program Analysis OCW) 6

6

Type Syntax (to study Hindley Milner)

Types (aka monotypes):

$\tau ::= \mathbf{b} \mid \tau_1 \rightarrow \tau_2 \mid \mathbf{t}$ ← \mathbf{t} is a type variable

E.g., \mathbf{Int} , \mathbf{Bool} , $\mathbf{Int} \rightarrow \mathbf{Bool}$, $\mathbf{t}_1 \rightarrow \mathbf{Int}$, $\mathbf{t}_1 \rightarrow \mathbf{t}_1$, etc.

Type schemes (aka polymorphic types):

$\sigma ::= \tau \mid \forall \mathbf{t}. \sigma$ ← \mathbf{t}_3 is a "free" type variable as it isn't bound under \forall
 $\forall \mathbf{t}_1. \forall \mathbf{t}_2. \forall \mathbf{t}_3. \tau$

E.g., $\forall \mathbf{t}_1. \forall \mathbf{t}_2. (\mathbf{Int} \rightarrow \mathbf{t}_1) \rightarrow \mathbf{t}_2 \rightarrow \mathbf{t}_3$

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

Type environment now

$\Gamma ::= \text{Identifiers} \rightarrow \text{Type schemes}$

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

7

7

Instantiations $\sigma \rightarrow \tau$

Type scheme $\sigma = \forall \mathbf{t}_1 \dots \mathbf{t}_n. \tau$ can be instantiated into a type τ' by substituting types for the bound variables (BV) under the universal quantifier \forall

$\tau' = \mathbf{S} \tau$ \mathbf{S} is a substitution s.t. $\text{Domain}(\mathbf{S}) \supseteq \mathbf{BV}(\sigma)$


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

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

Programming in Haskell, A Milanova (modified from MIT's 2015 Program Analysis OCW)

8

8



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

$(\text{Int} \rightarrow u_1) \xrightarrow{\tau} u_2 \rightarrow t_3$

$(\text{Int} \rightarrow u_4) \rightarrow u_5 \rightarrow t_3$

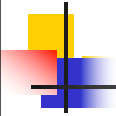
E.g., $\sigma = \forall t_1. t_1 \rightarrow t_1$

$u_1 \rightarrow u_1$

$u_2 \rightarrow u_2$

Programming in Haskell, A Milanova (modified from MIT's 2015 Program Analysis OCW) 9

9



Generalization (aka Closing) $\tau \rightarrow \sigma$

We can generalize a type τ as follows


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

where $\{t_1, \dots, t_n\} = \text{FV}(\tau) - \text{FV}(\Gamma)$

Generalization introduces polymorphism

Programming in Haskell, A Milanova (from MIT's 2015 Program Analysis OCW) 10

10



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

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


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

instantiates into

$u_1 \rightarrow t_2$ $u_2 \rightarrow t_2$

Programming in Haskell, A Milanova (from MIT's 2015 Program Analysis OCW) 11

11




ϵ_1 ϵ_2

let f = $\lambda x \rightarrow x$ in if (f True) then (f 1) else 1

1. Infer type for $\lambda x \rightarrow x : t_x \rightarrow t_x$ (a monotype)
2. Generalize type using $\text{Gen}([], t_x \rightarrow t_x)$: $\forall t_x. t_x \rightarrow t_x$ (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**
 - $(t_x \rightarrow t_x) [u_1/t_x]$ where u_1 is fresh tyvar at (f True)
 - $(t_x \rightarrow t_x) [u_2/t_x]$ where u_2 is fresh tyvar at (f 1)

Programming in Haskell, A Milanova 12

12



When can we generalize? $(\underline{t}_x \rightarrow \underline{t}_3) \rightarrow \underline{t}_x \rightarrow \underline{t}_3$

Consider expression $\lambda f \rightarrow \lambda x \rightarrow \text{let } g = f \text{ in } g \ x$

$\text{Gen}([\underline{f}:t_f, x:t_x], t_f)$ yields what? $\dots, t_f \leftarrow W([\underline{f}:t_f, x:t_x, g:t_g], f)$

$[t_f/t_f] \leftarrow t_g \sim t_f$

$t_f \leftarrow \text{Gen}([\underline{f}:t_f, x:t_x], t_f)$


$([\underline{t}_x \rightarrow \underline{t}_3 / t_f], t_3) \leftarrow W([\underline{f}:t_f, x:t_x, g:t_f], \dots)$

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

Programming in Haskell, A Milanova 13

13

Hindley Milner Typing Rules



$$\frac{\Gamma, x:\tau \vdash E_1 : \tau \quad \Gamma, x:\text{Gen}(\Gamma, \tau) \vdash E_2 : \tau'}{\Gamma \vdash \text{let } x = E_1 \text{ in } E_2 : \tau'} \quad \text{(Let)}$$

- Type of x as inferred for E_1 is τ . Type of x in E_2 is the generalized type scheme $\sigma = \text{Gen}(\Gamma, \tau)$

$$\frac{x:\sigma \in \Gamma \quad \tau < \sigma}{\Gamma \vdash x : \tau} \quad \text{(Var)}$$

- x in E_2 of **let: x** is of type τ if its type σ in the environment can be instantiated to τ

Note: Rules for **c**, **App**, **Abs**, etc. are as in F_1

Program Analysis CSCI 4450/6450, A Milanova (from MIT's 2015 Program Analysis OCW) 14

14

Hindley Milner Type Inference, Rough Sketch

let $x = E_1$ in E_2

1. Run W to get **type** T_{E_1} for E_1 in $\Gamma, x:t_x$; T_{E_1} is a monotype
2. Generalize free type variables in T_{E_1} to get the **type scheme** for T_{E_1} (generalizing in Γ not in $\Gamma, x:t_x$)
3. Extend environment with $x:\text{Gen}(\Gamma, T_{E_1})$ and type E_2
4. Every time algorithm sees x in E_2 , it instantiates x 's type scheme into a fresh generic instance
E.g., id 's type scheme is $\forall t_1. t_1 \rightarrow t_1$ so id is instantiated to $u_k \rightarrow u_k$ at $(\text{id } 1) // u_k$ is fresh tyvar

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

$$\lambda x \rightarrow \overbrace{\text{let } f = \overbrace{\lambda y \rightarrow x}^{E_1} \text{ in } \overbrace{(f \text{ True}, f \text{ !})}^{E_2}}^E$$

$(\{\}, ty \rightarrow tx) \leftarrow W([\!f:ty, x:tx\!], E_1)$
 $tf \sim ty \rightarrow tx$
 $\forall ty. ty \rightarrow tx \leftarrow Gen([x:tx], ty \rightarrow tx)$
 $(C, (t_1, t_2)) \leftarrow W([x:tx, f: \forall ty. ty \rightarrow tx], E_2)$

$C = \{u_1 \rightarrow tx \sim Bool \rightarrow t_1, u_2 \rightarrow tx \sim Int \rightarrow t_2\}$
 $f \text{ True} \parallel \text{instantiate } f \text{ into } u_1 \rightarrow tx$
 $f \text{ True} :: t_1$
 $f \text{ !} \parallel \text{instantiate } f \text{ into } u_2 \rightarrow tx$
 $f \text{ !} :: t_2$

\parallel Returns a set of constraints C and (unelaborated) type (t_1, t_2)

Programming in Haskell, A Milanova 17

17

Strategy Two: Algorithm W

def W(Γ, E) = case E of

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

$c \rightarrow ([], \text{TypeOf}(c))$

$x \rightarrow$ if (x NOT in $\text{Domain}(\Gamma)$) then fail

else let $T_E = \Gamma(x)$

in case T_E of

NEW! $\forall t_1, \dots, t_n. \tau \rightarrow ([], [u_1/t_1 \dots u_n/t_n] \tau) \parallel \text{poly}$

$_ \rightarrow ([], T_E) \parallel \text{monotype case}$

$\lambda x \rightarrow E_1 \rightarrow$ let $(S_{E_1}, T_{E_1}) = W(\Gamma + \{x:t_x\}, E_1)$

in $(S_{E_1}, S_{E_1}(t_x) \rightarrow T_{E_1})$ *As in FL.*


// ...

// continues on next slide!

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

18

18



```


def W( $\Gamma$ , E) = case E of
  // continues from previous slide
  // ...
  E1 E2 -> let (SE1, TE1) = W( $\Gamma$ , E1)
                  (SE2, TE2) = W(SE1( $\Gamma$ ), E2)
                  S = Unify(SE2(TE1), TE2 -> t)
                  in (S SE2 SE1, S(t))
  let x = E1 in E2 -> let (SE1, TE1) = W( $\Gamma$  + {x:tx}, E1)
                          S = Unify(SE1(tx), TE1)
                          in (SE2, TE2) = W(S SE1( $\Gamma$ ) + {x: $\sigma$ }, E2)
                          in (SE2 S SE1, TE2)
  
```

As in Fl.

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

Programming in Haskell, A Milanova (from MIT 2015 Program Analysis OCW) 19

19



Strategy Two Example

E_1
 E_2

let f = $\lambda x \rightarrow x$ in if (f True) then (f 1) else 1

1. let $\Gamma = []$ $T_1 = \text{int}$
 $S_1 = \dots$

2. Abs
 $T_2 = t_x \rightarrow t_x$
 $S_2 = []$
 $\Gamma = [x:t_x f:t_f]$

$\lambda x: t_x$ x

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

3. if-then-else E_2 $T_3 = \text{int}$
 $S_3 = \dots$

4. App $T_4 = \text{bool}$
 $S_4 = [\text{bool}/t_4][\text{bool}/u_1]$

5. App $T_5 = \text{int}$
 $S_5 = [\text{int}/t_5][\text{int}/u_2]$


f true f 1

$T = u_1 \rightarrow u_1$
 $S = []$

From $\text{Unify}(u_1 \rightarrow u_1, \text{bool} \rightarrow t_4)$ ²⁰

No subst, types 2. Abs immediately: $T_2 = t_x \rightarrow t_x$
 $\sigma = \text{Gen}([], t_x \rightarrow t_x) = \forall t_x. t_x \rightarrow t_x$

20




Example

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

Programming in Haskell, A Milanova 21

21



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

$$\text{let } x = E_1 \text{ in } E_2 \quad \Bigg\| \quad (\lambda x. E_2) E_1$$

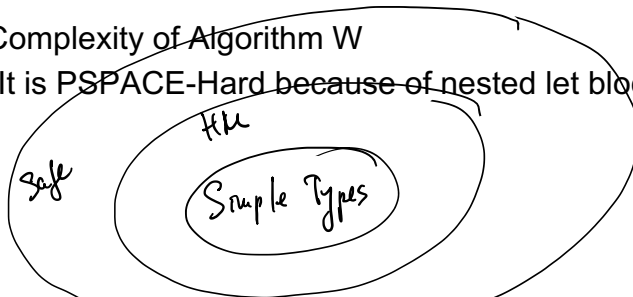
$$x : \forall t_1, t_2. \tau \text{ (polymorphic)} \quad \Bigg\| \quad \begin{matrix} x : \tau \text{ (monotype)} \\ \tau \end{matrix}$$

Programming in Haskell, A Milanova (from MIT 2015 Program Analysis OCW) 22

22

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)

23

23

Hindley Milner Limitations

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

$quad ::= (t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x$

$twice ::= (t_x \rightarrow t_x) \rightarrow t_x \rightarrow t_x$

let twice f x = f (f x)

Simple types: NO

Hindley Milner: YES

in twice twice succ 4 // let-bound polymorphism

$(u_g \rightarrow u_g) \rightarrow u_1 \rightarrow u_2$ $(u_2 \rightarrow u_2) \rightarrow u_2 \rightarrow u_2$ // $u_1 \rightarrow u_1$ $(u_2 \rightarrow u_2) \rightarrow u_2 \rightarrow u_2$ succeeds

let twice f x = f (f x)

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


$foo = \lambda g. g g \text{ succ } 4$

in foo twice

Programming in Haskell, A Milanova

24

24



$t_x \sim \text{Int}^{\text{fail}} \rightarrow ?$ Simple types: Type-incorrect
 $b_x \sim (b_1 \rightarrow b_1) \rightarrow ?$ HM: Type-incorrect

$\rightarrow (\backslash x \rightarrow x (\backslash y \rightarrow y) (x 1)) (\backslash z \rightarrow z)$

```

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

Programming in Haskell, A Milanova 25