# 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



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)

# 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

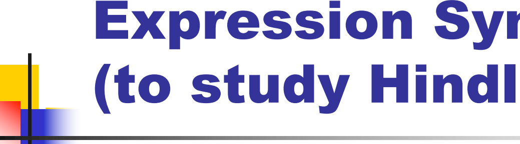
Strategy Two (Classical Hindley Milner)

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

# Outline

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





# Expression Syntax (to study Hindley Milner)

Expressions:

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

only place to introduce a polymorphic type

$x : \sigma$

$\sigma$ 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

# Type Syntax (to study Hindley Milner)

Types (aka monotypes):

$\tau$ ::= **b** | $\tau_1 \rightarrow \tau_2$ | **t** ← **t** is a type variable

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

Type schemes (aka polymorphic types):

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

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

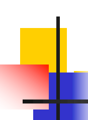
Note: all quantifiers appear in the beginning, $\tau$ cannot contain schemes

Type environment now

Gamma ::= Identifiers → Type schemes







# Instantiations

Type scheme $\sigma = \forall t_1 \ldots t_n.\tau$ can be instantiated into a type $\tau'$ 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**($\sigma$)

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

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

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

$(\text{Int} \to u_1) \to u_2 \to t_3$

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

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

$u_1 \to u_1$

$u_2 \to u_2$



## Generalization (aka Closing)

We can generalize a type $\tau$ as follows

$$\text{Gen}(\Gamma,\tau) = \forall t_1,\ldots t_n.\tau$$
where $\{t_1\ldots t_n\} = \text{FV}(\tau) - \text{FV}(\Gamma)$

Generalization introduces polymorphism

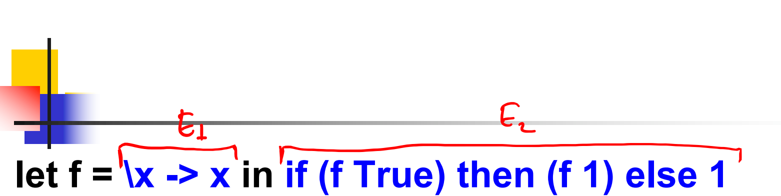
Quantify type variables that are free in $\tau$ but are not free in the type environment $\Gamma$

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

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

instantiates into $u_1 \rightarrow t_2$ and $u_2 \rightarrow t_2$





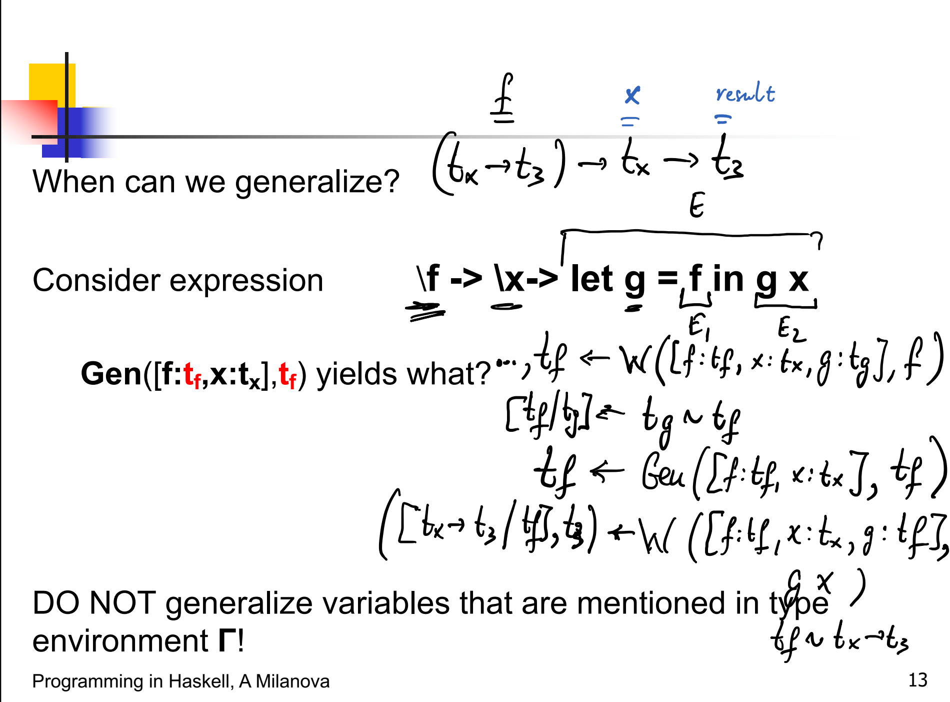
**let f = \x -> x in if (f True) then (f 1) else 1**

($t_1$ = \x -> x, $t_2$ = if (f True) then (f 1) else 1)

1. Infer type for **\x -> x** : $t_x \rightarrow t_x$ (a monotype)
2. Generalize type using **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)**




13

# Hindley Milner Typing Rules

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

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

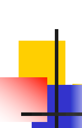
$$\frac{x{:}\sigma \in \Gamma \quad \tau < \sigma}{\Gamma \vdash x : \tau} \quad \text{(Var)}$$

- **x** in $\mathbf{E_2}$ of **let: x** is of type $\tau$ if its type $\sigma$ in the environment can be instantiated to $\tau$

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



14



## Hindley Milner Type Inference, Rough Sketch

**let x = $E_1$ in $E_2$**

1. Run W to get 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 $T_{E1}$ (generalizing in $\Gamma$ not in $\Gamma$,x:$t_x$)
3. Extend environment with **x:Gen($\Gamma$,$T_{E1}$)** 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 $\forall t_1.t_1 \to t_1$ so **id** is instantiated to $u_k \to u_k$ at **(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

$E$ ; $E_1$ ; $E_2$

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

$(\{\}, t_y \to t_x) \leftarrow W([f{:}t_f, x{:}t_x], E_1)$

$t_f \sim t_y \to t_x$

$\forall t_y .\, t_y \to t_x \leftarrow Gen([x{:}t_x], t_y \to t_x)$

$(C, (t_1,t_2)) \leftarrow W([x{:}t_x, f{:}\forall t_y . t_y \to t_x], E_2)$

$C = \{u_1 \to t_x \sim Bool \to t_1,\ u_2 \to t_x \sim Int \to t_2\}$

// Returns a set of constraints C and (unelaborated) type $(t_1,t_2)$

f True // instantiate f into $u_1 \to t_x$

f True :: $t_x$

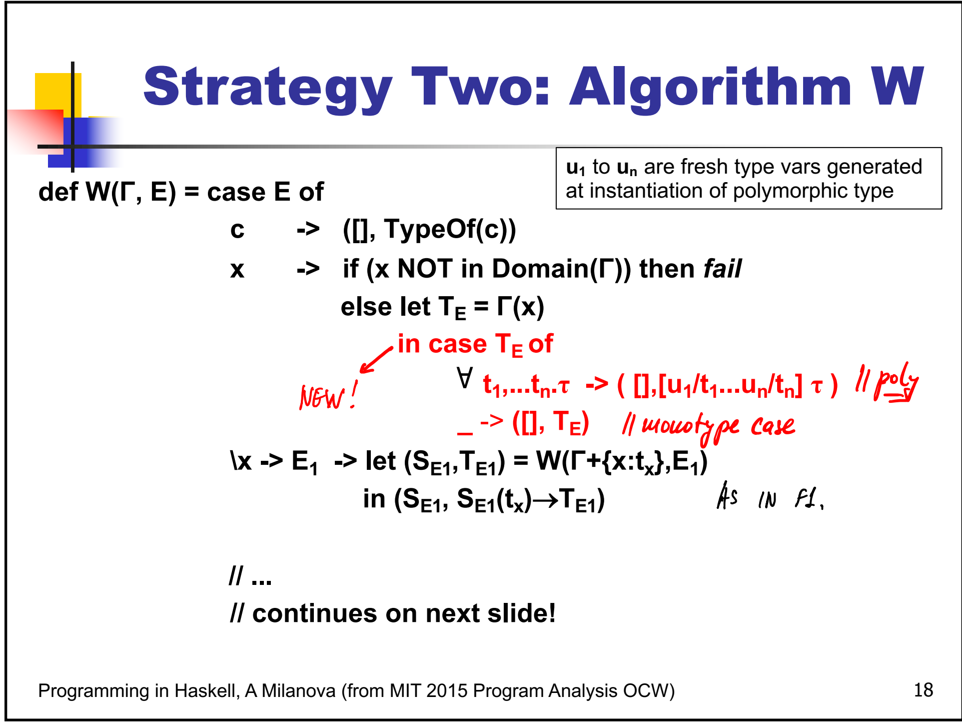
f 1 // instantiate f into $u_2 \to t_x$

f 1 :: $t_x$

$u_1 \to t_x$ / Bool $\to t_1$ ; $u_2 \to t_x$ / Int $\to t_2$



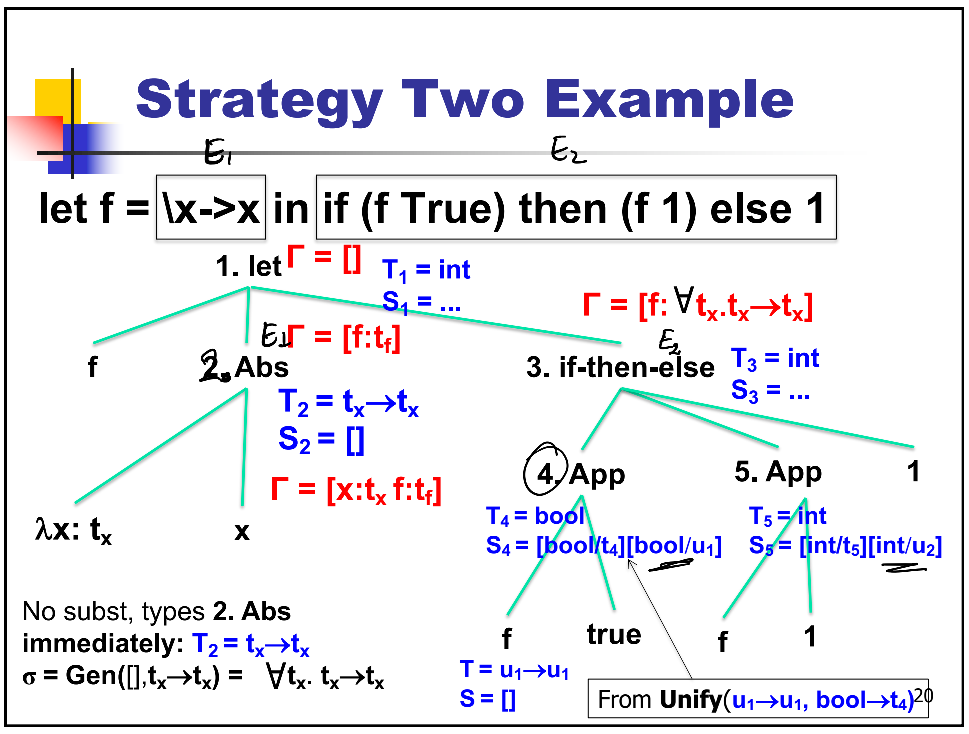




```
def W(Γ, E) = case E of
            // continues from previous slide
            // ...
            E1 E2  -> let (S_E1,T_E1) = W(Γ,E1)
                          (S_E2,T_E2) = W(S_E1(Γ),E2)          As in F1.
                          S = Unify(S_E2(T_E1),T_E2→t)
                       in (S S_E2 S_E1, S(t))
            let x = E1 in E2 -> let (S_E1,T_E1) = W(Γ+{x:t_x},E1)
                          S = Unify( S_E1(t_x),T_E1 )
                    NEW ← σ = Gen( S S_E1(Γ), S(T_E1) )
                          (S_E2,T_E2) = W(S S_E1(Γ)+{x:σ},E2)
                       in (S_E2 S S_E1, T_E2)
```



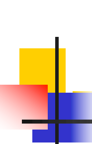




## Example

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





## Hindley Milner Observations

Notes

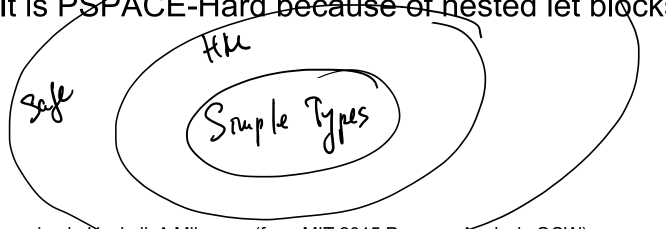
- Do not generalize over type variables mentioned in type environment (they are used elsewhere)
- let is the only way of defining polymorphic constructs
- Generalize the types of let-bound identifiers **only after** processing their definitions

let x = E1 in E2 || (λx. E2) E1

x: ∀t1∀t2.τ (polytype) || x:τ (monotype)

# 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

quad :: $(t_x \to t_x) \to t_x \to t_x$

twice :: $(t_x \to t_x) \to t_x \to t_x$ Simple types: NO Hindley Milner: YES

**let twice f x = f (f x)**

**in twice twice succ 4** // let-bound polymorphism

$(u_1 \to u_1) \to u_1 \to u_1$ $(u_2 \to u_2) \to u_2 \to u_2$ // $u_1 \to u_1$ $(u_2 \to u_2) \to u_2 \to u_2$ succeeds

**let twice f x = f (f x)**

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

foo = \g. g g succ 4

**in foo twice**

$t_x \sim$ Int (fail) $\rightarrow$ ?
$t_x \sim (t_1 \rightarrow t_1) \rightarrow$ ?

Simple types : Type-incorrect
HM : Type-n correct

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

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