

The Hindley-Milner type system

A restricted form of polymorphism:

- \forall only at outermost level
- Type variable stands only for a monotype, never a quantified type

Advantage: **no notation—type abstractions, apps are implicit**

- Type abstraction introduced at let binding
- Type application automatically at use

Basis for ML, Haskell, Objective Caml, ...

ML examples

```
val id = fn x => x
```

Becomes

```
(val id (type-lambda ('a) (lambda (('a x)) x)))
```

and

```
3 :: []
```

becomes

```
((@ cons int) 3 (@ '() int))
```

Representing Hindley-Milner types

Quantifier at outermost level

- **Type** τ is unquantified (tycon, tyvar, conapp)
- **Type scheme** σ is quantified type

Form of σ is always $\forall \alpha_1, \dots, \alpha_n. \tau$, where **possibly** $n = 0$

$\tau \Rightarrow \alpha$	tyvar	'c
μ	tycon	int, pair, ...
$(\tau_1, \dots, \tau_n) \tau$ where $n > 0$	conapp	(int) list
$\sigma \Rightarrow \forall \alpha_1, \dots, \alpha_n. \tau$ where $n \geq 0$		

Key notation $\tau <: \sigma$: **instantiation**

- types to substitute are **unspecified**

Hindley-Milner: Key ideas elaborated

Type environment Γ binds variable to **type scheme**

- Frequently have degenerate case with **zero quantified variables** (really a monotype)

Judgment $\Gamma \vdash e : \tau$ gives expression a **type**

At use, **automatically instantiate type scheme**

At **let** binding, **automatically abstract over tyvars**

- So called “Milner’s **let**”
- Refinement: abstract only over type variables that are **not free in environment**

Key ideas formalized: Some ML type rules

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

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{IF}(e_1, e_2, e_3) : \tau} \quad \text{(IF)}$$

$$\frac{\Gamma \vdash e_i : \tau_i, 1 \leq i \leq n \quad \Gamma \vdash e : \tau_1 \times \dots \times \tau_n \rightarrow \tau}{\Gamma \vdash \mathbf{APPLY}(e, e_1, \dots, e_n) : \tau} \quad \text{(APPLY)}$$

ML type rules, continued

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \mathbf{LAMBDA}(\langle x_1, \dots, x_n \rangle, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau} \quad \text{(LAMBDA)}$$

where $\{x_i \mapsto \tau_i\}$ stands for $\{x_i \mapsto \forall. \tau_i\}$

$$\frac{\Gamma \vdash e' : \tau' \quad \{\alpha_1, \dots, \alpha_n\} = \text{fv}(\tau') - \text{fv}(\Gamma) \quad \Gamma\{x \mapsto \forall \alpha_1, \dots, \alpha_n. \tau'\} \vdash e : \tau}{\Gamma \vdash \mathbf{MLET}(x, e', e) : \tau} \quad \text{(MILNER'S LET)}$$

Things to notice

Lambda-bound variables are monomorphic

Let-bound variables are polymorphic

We have to guess the right types!

Type inference

Guess the types

- aka type reconstruction

How does it work?

- Type variable for each **unknown type**
- As information becomes available, **substitute** for type variables
- Accumulated info represented by substitutions
- Substitution driven by **unification**

Instances and substitution

Examples

```
int <: 'a
int list <: 'a
int list <: 'a list
not int <: 'a list
```

And the **instance relation** $\tau_1 <: \tau_2$:

τ_1 is an instance of τ_2 (τ_2 is more general than τ_1)
if and only if $\exists \theta : \tau_1 = \theta(\tau_2)$

Use substitutions in type inference

Instance properties

Theorem: 'a is the most general type
(every type τ is an instance of α)

Proof: let $\theta = \{\alpha \mapsto \tau\}$

Theorem: $<:$ is a partial order.

Proof: reflexive by $\lambda\tau.\tau$ (identity substitution),
transitive by composition,
antisymmetric because only $\lambda\tau.\tau$ is its own
inverse

Instantiation intuition

Consider application $(e_1 e_2)$

- e_1 must have some arrow type, call it $\alpha \rightarrow \beta$
- e_2 must have some type, call it γ
- Instantiated γ must equal instantiated α
- Result type is instantiated β

Want **most general** instantiation

Example: `length [true,false]`

- Type of e_1 is `'a list -> int`
- Type of e_2 is `bool list`
- Instantiate with $\{\text{'a} \mapsto \text{bool}\}$; result type is `int`

More instantiation

Another example:

```
val f = fn (x, n) => n + 1
val g = fn y => f (0, y)
```

In application `f (0, y)`,

```
f : 'a * int -> int
(0, y) : int * 'b
```

Must instantiate both `f` and `y` correctly

Application is clearly OK, assuming that

$\text{'a} \mapsto \text{int}, \text{'b} \mapsto \text{int}$

Finding the right substitution is **unification**

Unification

To make the idea precise, we say that

τ_1 and τ_2 are unified by θ

if $\theta\tau_1 = \theta\tau_2$

So `'a * int` and `int * 'b` are unified by

$(\text{'a} \mapsto \text{int}) \circ (\text{'b} \mapsto \text{int})$

Unifier – substitution θ represents assumptions about type variables

Unifier satisfies an equality constraint

Most General Unifiers

Makes no unnecessary assumptions

`'a list` and `'b` are unified by

`'a \mapsto int list, 'b \mapsto int list list`

`'a list` and `'b` are unified by `'b \mapsto 'a list`

Which unifier is more general?

θ_1 is an instance of θ_2 iff $\exists \theta$ such that $\theta_1 = \theta \circ \theta_2$

A *most general unifier* of types τ_1 and τ_2 is a substitution θ such that

- τ_1 and τ_2 are unified by θ ($\theta\tau_1 = \theta\tau_2$)
- there is no more general θ that unifies τ_1 and τ_2

It's easy to implement

Implementing unification

`unify(τ_1, τ_2)` returns most general unifier

It's easy: Recall types as

$\tau \Rightarrow$ TYVAR α

| TYCON μ

| CONAPP ($\tau, [\tau_1, \dots, \tau_n]$)

TYVAR α unifies with any type τ by $\alpha \mapsto \tau$

provided α is not free in τ (or $\alpha = \tau$)

can't unify `'a` with `'a list!`

`x :: x` must not have a type!

TYCON μ unifies only with itself

CONAPP unifies with CONAPP if arguments unify

Unification more formally

Robinson's unification algorithm, disagreement sets.

Note: not the way to implement unification in uML interpreter!

Insert here

Type inference for simple-typed λ -calculus terms
(no Milner let).

From type rules to type inference

Key idea: given Γ and e , compute θ and τ such that

$$\theta\Gamma \vdash e : \tau$$

Idea #2: extend to list of e_i : $\theta\Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n$

$$\frac{\Gamma \vdash e_1 : \mathbf{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \mathbf{IF}(e_1, e_2, e_3) : \tau} \quad (\mathbf{IF})$$

becomes (note equality constraints)

$$\frac{\theta\Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3 \quad \theta'\tau_1 = \theta'\mathbf{bool} \quad \theta'\tau_2 = \theta'\tau_3}{(\theta' \circ \theta)\Gamma \vdash \mathbf{IF}(e_1, e_2, e_3) : \theta'\tau_3} \quad (\mathbf{IF})$$

W.T.F.?

IF rule from book and simplifications explained!
Emphasize (1) all substitutions involved in final types and env. (2) rules up, substitutions down.

Type rules to type inference, cont'd

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \dots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \mathbf{LAMBDA}(\langle x_1, \dots, x_n \rangle, e) : \tau_1 \times \dots \times \tau_n \rightarrow \tau} \quad (\mathbf{LAMBDA})$$

becomes

$$\frac{\theta(\Gamma\{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}) \vdash e : \tau}{\theta\Gamma \vdash \mathbf{LAMBDA}(\langle x_1, \dots, x_n \rangle, e) : \theta\alpha_1 \times \dots \times \theta\alpha_n \rightarrow \tau} \quad (\mathbf{LAMBDA})$$

N.B. $\theta(\Gamma\{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}) = (\theta\Gamma)\{x_1 \mapsto \theta\alpha_1, \dots, x_n \mapsto \theta\alpha_n\}$

Type rules to type inference, finish

$$\frac{\Gamma \vdash e_i : \tau_i, 1 \leq i \leq n \quad \Gamma \vdash e : \tau_1 \times \dots \times \tau_n \rightarrow \tau}{\Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \tau} \quad \text{(APPLY)}$$

becomes

$$\frac{\theta \Gamma \vdash e, e_1, \dots, e_n : \hat{\tau}, \tau_1, \dots, \tau_n \quad \theta'(\hat{\tau}) = \theta'(\tau_1 \times \dots \times \tau_n \rightarrow \alpha), \text{ where } \alpha \text{ is fresh}}{(\theta' \circ \theta) \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \theta' \alpha} \quad \text{(APPLY)}$$

More explicit substitutions

$$\frac{\Gamma \vdash e' : \tau' \quad \{\alpha_1, \dots, \alpha_n\} = \text{fv}(\tau') - \text{fv}(\Gamma) \quad \Gamma \{x \mapsto \forall \alpha_1, \dots, \alpha_n. \tau'\} \vdash e : \tau}{\Gamma \vdash \text{MLET}(x, e', e) : \tau} \quad \text{(MILNER'S LET)}$$

becomes

$$\frac{\theta' \Gamma \vdash e' : \tau' \quad \{\alpha_1, \dots, \alpha_n\} = \text{fv}(\tau') - \text{fv}(\theta' \Gamma) \quad \theta((\theta' \Gamma) \{x \mapsto \forall \alpha_1, \dots, \alpha_n. \tau'\}) \vdash e : \tau}{(\theta \circ \theta') \Gamma \vdash \text{MLET}(x, e', e) : \tau} \quad \text{(MILNER'S LET)}$$

Type inference, operationally

Like type checking:

- top-down, bottom up pass over abstract syntax
- use Γ to look up types of variables

Different from type checking:

- Get substitution θ
- Use θ to modify Γ

Γ represents assumptions

Operational example

Start with `val f = fn x => ...`

Add binding `x : 'a` to Γ

no assumptions about `x` (it's any unknown type)

continue with body: `val f = fn x => x + 1`

Look up `+` in Γ , find `+: int × int → int`

Unify `(x, 1) : 'a * int` with `int * int`

most general unifier is $\theta = 'a \mapsto \text{int}$

We modify the environment: “ $\theta\Gamma$ ” (theta on Gamma)

(In new environment, `x : int`)

We'll give you a suitable abstract data type...

Implementing type inference

Calling `typeof(e, Gamma)` returns pair (τ, θ) such that $\theta \vdash e : \tau$

$$\frac{\theta \Gamma \vdash e, e_1, \dots, e_n : \hat{\tau}, \tau_1, \dots, \tau_n}{\theta'(\hat{\tau}) = \theta'(\tau_1 \times \dots \times \tau_n \rightarrow \alpha), \text{ where } \alpha \text{ is fresh}} \quad (\text{APPLY})$$

$$(\theta' \circ \theta) \Gamma \vdash \text{APPLY}(e, e_1, \dots, e_n) : \theta' \alpha$$

Write `funty = $\hat{\tau}$, actualtypes = τ_1, \dots, τ_n , rettype = α :`

```
fun typeof(APPLY (f, actuals)) =
  let val (funty :: actualtypes, theta) = (*cheat*)
        typeof (f :: actuals, Gamma)
      val rettype = freshtyvar()
      val theta' =
          unify(funty, funtype(actualtypes, rettype))
    in (theta' rettype, theta' o theta)
    end
```

Type inference example

```
; from (define exists? ...) [syntax-
tic sugar]
```

```
(val-rec exists? (lambda (p? lis)
  (if (null? lis) #f
      (if (p? (car lis)) #t
          (exists? p? (cdr lis))))))

Type environment  $\Gamma$  for body of lambda:
exists?: 'd
p?: 'e
lis: 'f
null?: 'a, 'a list -> bool
car: 'a, 'a list -> 'a
cdr: 'a, 'a list -> 'a list
```

```
Inference:
Term      : Type      Constraints
null?     : 'g list -> bool None
(null? lis) : bool      'f = 'g list
car       : 'h list -> 'h None
(car lis) : 'g        'h = 'g
(p? (car lis)) : 'l   'e = 'g -> 'l
if (p? (car lis)) ...
:: 'l = bool
#t        : bool      None
(exists? p?) : 'm     'd = 'e -> 'm
cdr       : 'n list -> 'n list None
(cdr lis) : 'n list   'f = 'n list
(exists? p? (cdr lis))
:: 'p
(if ... #t (exists? p? (cdr lis)))
:: bool
:: ...
```

Example

Stu's notes p143: integrate order of unification w Ramsey's order of building substitutions.

Real ML type inference

μ ML has no assignment.

```
let r = ref (fn x => x) in
r := (fn x => x + 1); !r (true); end
```

would calculate `(true + 1)`. Must not be allowed to type check!

`r` has type `('a -> 'a) ref`

Problem: typing rules use substitution of any term, evaluation only substitutes values.

Fix: The value restriction the right-hand side of a let declaration is polymorphic only if it is a value.

Real ML: Type ascriptions

ML allows the user to state a type for an expression.

The type given must be an instance of the type inferred for the expression. The ascribed type will then be used as the type of the expression:

```

val ei : int list = []

val memofun : (int -> int) ref =
  fn _ => raise Fail "Not initialized"
fun f x = ... memofun (x - 1) ...
val _ = memofun := memoize f

```

What is not Hindley-Milner typeable?

Consider Church numerals, type

$\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)$

suc takes a Church numeral and returns a Church numeral.

$(\forall\alpha.(\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow (\forall\beta.(\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta))$

$= \forall\beta\exists\alpha.((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta))$

not $\forall\beta, \alpha.((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\beta \rightarrow \beta) \rightarrow (\beta \rightarrow \beta))$

not $\forall\alpha.((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$

Try add, mult, exp, ack at type

$\forall\alpha.((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow ((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha)) \rightarrow$

$((\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha))$

add, mult work, exp, ack don't.

What is not Hindley-Milner typeable?

```

- val expt = fn m => fn n => fn f => fn x => ((n m) f) x;
val expt = fn : 'a -> ('a -> 'b -> 'c -> 'd) -> 'b -> 'c -> 'd
- val expt1 = expt : (('a -> 'a) -> 'a -> 'a) -> (('a -> 'a) -> 'a -> 'a) ->
  (('a -> 'a) -> 'a -> 'a);
stdIn:64.1-64.100 Error: expression doesn't match constraint [UBOUND match]
exprn: (('a -> 'a) -> 'a -> 'a)
-> (((('a -> 'a) -> 'a -> 'a) -> 'Z -> 'Y -> 'X) -> 'Z -> 'Y -> 'X)
constr: (('a -> 'a) -> 'a -> 'a)
-> (('a -> 'a) -> 'a -> 'a) -> ('a -> 'a) -> 'a -> 'a
in expression:
  expt:
    (('a -> 'a) -> 'a -> 'a)
    -> (('a -> 'a) -> 'a -> 'a) -> ('a -> 'a) -> 'a -> 'a
- val c2 = fn f => fn x => (f (f x));
val c2 = fn : ('a -> 'a) -> 'a -> 'a
- expt c2 c2;
  val it = fn : (?..X1 -> ?..X1) -> ?..X1 -> ?..X1
-

```

Type systems: Things to remember

Compile-time checking

Type soundness

Type erasure

Examples

- Simple monomorphic: like C
- General polymorphic: super-powerful, but too much notation
- Hindley-Milner polymorphic: powerful, no notation

Some runnable terms not typeable