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 $\forall \alpha_1, \ldots, \alpha_n \,.\, \tau$, where possibly $n = 0$

$\tau \Rightarrow \alpha$		tyvar
$\mid$		tycon
$\mid\ (\tau_1, \ldots, \tau_n)\ \tau$	where $n > 0$	conapp
$\sigma \Rightarrow \forall \alpha_1, \ldots, \alpha_n . \tau$	where $n \geq 0$	

Key notation $\tau <: \sigma$ means 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 not bound in environment

Key ideas formalized: Some ML type rules

$$\frac{\Gamma(x) = \sigma \qquad \tau <: \sigma}{\Gamma \vdash x : \tau} \qquad \textsc{(Var)}$$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau} \qquad \textsc{(If)}$$

$$\frac{\Gamma \vdash e_i : \tau_i, 1 \leq i \leq n \qquad \Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \tau} \qquad \textsc{(Apply)}$$

ML type rules, continued

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \textsc{lambda}(\langle x_1, \ldots, x_n \rangle, e) : \tau_1 \times \cdots \times \tau_n \to \tau}$$

(LAMBDA)

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

$$\frac{\begin{array}{c}\Gamma \vdash e' : \tau' \\ \{\alpha_1, \ldots, \alpha_n\} = \mathrm{fv}(\tau') - \mathrm{fv}(\Gamma) \\ \Gamma\{x \mapsto \forall\alpha_1, \ldots, \alpha_n.\tau'\} \vdash e : \tau\end{array}}{\Gamma \vdash \textsc{mlet}(x, e', e) : \tau}$$

(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 $\lambda\tau.\tau$, 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** $\alpha \to \beta$
- 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** e_1 **at** `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

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

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

Substitution θ represents assumptions about type variables

Unifier satisfies the equality constraint $\tau_1 = \tau_2$

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

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: $\theta\Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n$

$$\frac{\Gamma \vdash e_1 : \texttt{bool} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \tau} \quad \textsc{(if)}$$

becomes (note equality constraints)

$$\frac{\theta\Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3 \qquad \theta'\tau_1 = \theta'\texttt{bool} \qquad \theta'\tau_2 = \theta'\tau_3}{(\theta' \circ \theta)\Gamma \vdash \textsc{if}(e_1, e_2, e_3) : \theta'\tau_3} \quad \textsc{(if)}$$

Type rules to type inference, cont'd

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \textsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e) : \tau_1 \times \cdots \times \tau_n \to \tau}$$

(LAMBDA)

becomes

$$\frac{\theta(\Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}) \vdash e : \tau \qquad \alpha_i \notin \mathrm{fv}(\Gamma)}{\theta\Gamma \vdash \textsf{LAMBDA}(\langle x_1, \ldots, x_n \rangle, e) : \theta\alpha_1 \times \cdots \times \theta\alpha_n \to \tau}$$

(LAMBDA)

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

Type rules to type inference, finish

$$\frac{\Gamma \vdash e_i : \tau_i, 1 \leq i \leq n \qquad \Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau}{\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \tau}$$

(APPLY)

becomes

$$\frac{\theta\Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \theta'(\hat{\tau}) = \theta'(\tau_1 \times \cdots \times \tau_n \to \alpha), \text{ where } \alpha \text{ is fresh}}{(\theta' \circ \theta)\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \theta'\alpha}$$

(APPLY)

More explicit substitutions

$$\frac{\Gamma \vdash e' : \tau' \qquad \{\alpha_1, \ldots, \alpha_n\} = \mathrm{fv}(\tau') - \mathrm{fv}(\Gamma) \qquad \Gamma\{x \mapsto \forall \alpha_1, \ldots, \alpha_n.\tau'\} \vdash e : \tau}{\Gamma \vdash \textsf{MLET}(x, e', e) : \tau}$$

(MILNER'S LET)

becomes

$$\frac{\theta'\Gamma \vdash e' : \tau' \qquad \{\alpha_1, \ldots, \alpha_n\} = \mathrm{fv}(\tau') - \mathrm{fv}(\theta'\Gamma) \qquad \theta((\theta'\Gamma)\{x \mapsto \forall \alpha_1, \ldots, \alpha_n.\tau'\}) \vdash e : \tau}{(\theta \circ \theta')\Gamma \vdash \textsf{MLET}(x, e', e) : \tau}$$

(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:

- **Use equality constraint to 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` (any unknown monotype)

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

Look up `+` in Γ, find `+` : `int` $\times$ `int` $\rightarrow$ `int`

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

most general unifier is $\theta = $ `'a` $\mapsto$ `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 `(tau, theta)` such that `theta on Gamma` $\vdash$ `e` : `tau`

$$\frac{\theta\Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \theta'(\hat{\tau}) = \theta'(\tau_1 \times \cdots \times \tau_n \to \alpha), \text{ where } \alpha \text{ is fresh}}{(\theta' \circ \theta)\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \theta'\alpha} \quad (\textsc{Apply})$$

Write `funty` $= \hat{\tau}$, `actualtypes` $= \tau_1, \ldots, \tau_n$, `rettype` $= \alpha$:

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

Type-inference example

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**