Type inference: Review of the basics

1. For each unknown type, a fresh type variable
2. Instantiate every variable automatically
3. Every typing rule adds equality constraints
4. Solve constraints to get substitution
5. Apply substitution to constraints and types
6. Introduce polymorphism at let/val bindings

Review: Using polymorphic names

```
-> (val cc (lambda (nss) (car (car nss))))
```

Using polymorphic names

```
-> (val cc (lambda (nss) (car (car nss))))

cc : (forall ('a) ((list (list 'a)) -> 'a))
```

Your turn!

Given

```
empty : (forall ['a] (list 'a))
cons  : (forall ['a] ('a (list 'a) -> (list 'a)))
```

For

```
(cons empty empty)
```

You fill in:

1. Fresh instances
2. Constraints
3. Final type

Bonus example

```
-> (val second (lambda (xs) (car (cdr xs))))
second : ...
-> (val two    (lambda (f) (lambda (x) (f (f x)))))
two : ...
```

Bonus example solved

```
-> (val second (lambda (xs) (car (cdr xs))))
second : (forall ('a) ((list 'a) -> 'a))
-> (val two    (lambda (f) (lambda (x) (f (f x)))))
two :    (forall ('a) (('a -> 'a) -> ('a -> 'a)))
```

Making Type Inference Precise

Sad news:

- **Type inference for polymorphism is undecidable**

Solution:

- **Each formal parameter has a monomorphic type**

Consequences:

- **The *argument* to a higher-order function *cannot* be *mandated* to be polymorphic**
- `forall` **appears only outermost in types**

We infer stratified "Hindley-Milner" types

Two layers: Monomorphic types τ
Polymorphic type schemes σ

τ	$::=$	α	type variables
	$\vert$	μ	type constructors: `int`, `list`
	$\vert$	$(\tau_1, \ldots \tau_n)\ \tau$	constructor application
σ	$::=$	$\forall \alpha_1, \ldots \alpha_n \,.\, \tau$	type scheme

Each variable in Γ introduced via LET, LETREC, VAL, and VAL-REC has a type scheme σ with $\forall$

Each variable in Γ introduced via LAMBDA has a degenerate type scheme $\forall \,.\, \tau$—a type, wrapped

Representing Hindley-Milner types

```
type tyvar = name
datatype ty
  = TYVAR  of tyvar
  | TYCON  of name
  | CONAPP of ty * ty list

datatype type_scheme
  = FORALL of tyvar list * ty

fun funtype (args, result) =
  CONAPP (TYCON "function",
          [CONAPP (TYCON "arguments", args),
           result])
```

Key ideas

Type environment Γ binds var to type scheme σ

- `singleton` : $\forall\alpha . \alpha \rightarrow \alpha$ `list`
- `cc` : $\forall\alpha . \alpha$ `list list` $\rightarrow \alpha$
- `car` : $\forall\alpha . \alpha$ `list` $\rightarrow \alpha$
- `n` : $\forall . $ `int` **(note empty $\forall$)**

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

(Transitions inserted by algorithm!)

Key ideas

Definitions are polymorphic with type schemes

Each use is monomorphic with a (mono-) type

Transitions:

- At use, type scheme instantiated automatically
- At definition, automatically abstract over tyvars

All the pieces

1. Hindley-Milner types
2. Bound names : σ, expressions : τ
3. Type inference yields type-equality constraint
4. Constraint solving produces substitution
5. Substitution refines types
6. Call solver, introduce polytypes at `val`
7. Call solver, introduce polytypes at all `let` forms

Type-inference algorithm

Given Γ and e, compute C and τ such that

$$C, \Gamma \vdash e : \tau$$

Idea #2: Extend to list of e_i: $C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n$

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

becomes (note equality constraints with $\sim$)

$$\frac{C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3}{C \wedge \tau_1 \sim \texttt{bool} \wedge \tau_2 \sim \tau_3, \Gamma \vdash \textsf{IF}(e_1, e_2, e_3) : \tau_3} \quad (\textsf{IF})$$

Apply rule

$$\frac{\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau \qquad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n}{\Gamma \vdash \textsf{APPLY}(e, e_1, \ldots, e_n) : \tau} \quad (\textsc{Apply})$$

becomes

$$\frac{C, \Gamma \vdash e, e_1, \ldots, e_n : \tau_f, \tau_1, \ldots, \tau_n \qquad \alpha \text{ is fresh}}{C \wedge \tau_f \sim \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \textsf{APPLY}(e, e_1, \ldots, e_n) : \alpha} \quad (\textsc{Apply})$$

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:

- **Create fresh type variables when needed**
- **Accumulate equality constraints**

Your skills so far

You can complete typeof

- Takes e and Γ, returns τ and C

(Except for let forms.)

Next up: solving constraints