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))
Refresh your skills!

-> (val second (lambda (xs) (car (cdr xs))))
second : ...

-> (val two (lambda (f) (lambda (x) (f (f x))))))
two : ...
Skills refreshed

-> (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 polymorphic
• *forall* appears only outermost in types
We infer stratified “Hindley-Milner” types

Two layers: Monomorphic types $\tau$

Polymorphic type schemes $\sigma$

$$\begin{align*}
\tau & ::= \alpha \quad \text{type variables} \\
& \quad | \mu \quad \text{type constructors: int, list} \\
& \quad | (\tau_1, \ldots \tau_n) \tau \quad \text{constructor application} \\
\sigma & ::= \forall \alpha_1, \ldots \alpha_n . \tau \quad \text{type scheme}
\end{align*}$$

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

Each variable in $\Gamma$ introduced via LAMBDA has a degenerate type scheme $\forall . \tau$—a type, wrapped
Representing Hindley-Milner types

datatype ty
    = TYCON of name
    | CONAPP of ty * ty list
    | TYVAR of name

datatype type_scheme
    = FORALL of name list * ty

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

Type environment $\Gamma$ binds var to type scheme $\sigma$

- $\text{app2} : \forall \alpha, \beta. (\alpha \rightarrow \beta) \times \alpha \times \alpha \rightarrow \beta$
- $\text{cc} : \forall \alpha. \alpha \text{ list list} \rightarrow \alpha$
- $\text{car} : \forall \alpha. \alpha \text{ list} \rightarrow \alpha$
- $n : \forall. \text{int}$ (note empty $\forall$)

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

(Transitions happen automatically!)
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: $\sigma$, expressions: $\tau$
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 $\Gamma$ and $e$, compute $C$ and $\tau$ such that

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

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

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

becomes (note equality constraints with $\sim$)

$$
\frac{
C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3
}{
C \land \tau_1 \sim \text{bool} \land \tau_2 \sim \tau_3, \Gamma \vdash \text{IF}(e_1, e_2, e_3) : \tau_3}
$$
Apply rule

\[
\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n
\]

\[
\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau
\]

(Appl)

becomes

\[
C, \Gamma \vdash e, e_1, \ldots, e_n : \tau_f, \tau_1, \ldots, \tau_n \quad \alpha \text{ is fresh}
\]

\[
C \land \tau_f \sim \tau_1 \times \cdots \times \tau_n \rightarrow \alpha, \Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \alpha
\]

(Appl)
Your turn: Begin Rule

\[
\frac{\Gamma \vdash e_i : \tau_i \quad 1 \leq i \leq n}{\Gamma \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n}
\]  

(\text{BEGIN})

\[
\frac{C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n}{C, \Gamma \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n}
\]  

(\text{BEGIN})
Type inference, operationally

Like type checking:
  • Top-down, bottom up pass over abstract syntax
  • Use $\Gamma$ 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 $\Gamma$, returns $\tau$ and $C$

(Except for `let` forms.)

Next up: solving constraints!