Last time

\[\rightarrow (\text{val cc (lambda (nss) (car (car nss))))}\]
\[\text{cc : (forall ('a) ((list (list 'a)) -> 'a))}\]
Refresh your skills!

\[
\begin{align*}
\text{->} \quad & \text{(val second (lambda (xs) (car (cdr xs)))))} \\
& \text{second : ...} \\
\text{->} \quad & \text{(val two (lambda (f) (lambda (x) (f (f x))))))} \\
& \text{two : ...}
\end{align*}
\]
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 $\sigma$

\[
\begin{align*}
\tau & ::= \alpha & \text{type variables} \\
     & | \mu & \text{type constructors: int, list} \\
     & | (\tau_1, \ldots, \tau_n) \tau & \text{constructor application}
\end{align*}
\]

\[
\sigma ::= \forall \alpha_1, \ldots, \alpha_n . \tau \quad \text{type scheme}
\]

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
Key ideas

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

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

 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 `let`
Type-inference algorithm

Given $\Gamma$ and $e$, compute $C$ and $\tau$ such that

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

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

$$\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau$$

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

becomes (note equality constraints with $\sim$)

$$C, \Gamma \vdash e_1, e_2, e_3 : \tau_1, \tau_2, \tau_3$$

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

\[
\begin{align*}
\Gamma & \vdash e : \tau_1 \times \cdots \times \tau_n \to \tau \\
\Gamma & \vdash e_1 : \tau_1 \\
\vdots & \\
\Gamma & \vdash e_n : \tau_n \\
\hline
\Gamma & \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau
\end{align*}
\]

(APPLY)

becomes

\[
\begin{align*}
C, \Gamma & \vdash e, e_1, \ldots, e_n : \tau_f, \tau_1, \ldots, \tau_n \\
\alpha & \text{ is fresh} \\
C \land \tau_f & \sim \tau_1 \times \cdots \times \tau_n \to \alpha, \Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \alpha
\end{align*}
\]

(APPLY)
Your turn: Begin Rule

\[
\Gamma \vdash e_i : \tau_i \quad 1 \leq i \leq n
\]

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

\[
C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n
\]

\[
C, \Gamma \vdash \text{BEGIN}(e_1, \ldots, e_n) : \tau_n
\]
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!
Representing Constraints

datatype con = ~ of ty * ty
            | /
            | of con * con
            | TRIVIAL

infix 4 ~
infix 3 /
\
Solving Constraints

We *solve* a constraint $C$ by finding a substitution $\theta$ such that the constraint $\theta C$ is satisfied.

Substitutions distribute over constraints:

\[
\begin{align*}
\theta(\tau_1 \sim \tau_2) &= \theta\tau_1 \sim \theta\tau_2 \\
\theta(C_1 \land C_2) &= \theta C_1 \land \theta C_2 \\
\theta T &= T
\end{align*}
\]
What is a substitution?

Formally, $\theta$ is a function:
- Replaces a \textit{finite} set of type variables with types
- Apply to type, constraint, type environment, \ldots

In code, a data structure:
- “Applied” with `tysubst`, `consubst`
- Made with `idsubst`, $a \mid\rightarrow\tau$`
- Find domain with `dom`
When is a constraint satisfied?

\[
\frac{\tau_1 = \tau_2}{\tau_1 \sim \tau_2 \text{ is satisfied}} \quad \text{(EQ)}
\]

\[
\frac{C_1 \text{ is satisfied} \quad C_2 \text{ is satisfied}}{C_1 \land C_2 \text{ is satisfied}} \quad \text{(AND)}
\]

\[
\frac{}{T \text{ is satisfied}} \quad \text{(TRIVIAL)}
\]
Examples

Which have solutions?

1. int ~ bool
2. (list int) ~ (list bool)
3. 'a ~ int
4. 'a ~ (list int)
5. 'a ~ ((args int) -> int)
6. 'a ~ 'a
7. (args 'a int) ~ (args bool 'b)
8. (args 'a int) ~ ((args bool) -> 'b)
9. 'a ~ (pair 'a int)
10. 'a ~ tau  // arbitrary tau