Formalizing Type Inference

Sad news:
• Type inference for polymorphism is undecidable.

Solution:
• Each formal parameter has a monomorphic type.

Consequences:
• Polymorphic functions are not first class.
• 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 $\sigma$

$\tau ::= \alpha$ \hspace{1cm} type variables

$\mid \mu$ \hspace{1cm} type constructors: \text{int, list}

$\mid (\tau_1, \ldots, \tau_n) \tau$ \hspace{1cm} constructor application

$\sigma ::= \forall \alpha_1, \ldots, \alpha_n . \tau$ \hspace{1cm} type scheme

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

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

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

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
Type inference Algorithm

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

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

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

\[
\frac{
\Gamma \vdash e : \tau_1 \times \cdots \times \tau_n \rightarrow \tau \quad \Gamma \vdash e_1 : \tau_1 \quad \ldots \quad \Gamma \vdash e_n : \tau_n
}{
\Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \tau}
\]

(APPLY)

becomes

\[
\frac{
C, \Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \quad \alpha \text{ is fresh}
}{
C \land \hat{\tau} \leadsto \tau_1 \times \cdots \times \tau_n \rightarrow \alpha, \Gamma \vdash \text{APPLY}(e, e_1, \ldots, e_n) : \alpha}
\]

(APPLY)
Exercise: Begin Rule

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

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

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

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

A substitution $\theta$ that makes all constraints in $C$ true is a solution for $C$. 
Examples

Which have solutions?

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

Which have solutions?

1. int  ~  bool  No
2. int list  ~  bool list  No
3. 'a  ~  int  'a |-> int
4. 'a  ~  int list  'a |-> int list
5. 'a  ~  int -> int  'a |-> int -> int
6. 'a  ~  'a  'a |-> 'a
7. 'a * int  ~  bool * 'b  'a |-> bool and 'b
8. 'a * int  ~  bool -> 'b  No
9. 'a  ~  ('a, int)  No
10. 'a  ~  tau  depends if 'a in free-vars(tau)
Substitutions over constraints

Substitutions distribute over constraints:

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

\[ \tau_1 = \tau_2 \]

\[ \tau_1 \sim \tau_2 \text{ is satisfied} \]  \hspace{1cm} \text{(EQ)}

\[ C_1 \text{ is satisfied} \quad C_2 \text{ is satisfied} \]

\[ C_1 \land C_2 \text{ is satisfied} \]  \hspace{1cm} \text{(AND)}

\[ T \text{ is satisfied} \]  \hspace{1cm} \text{(TRIVIAL)}
Solving Constraint Conjunctions

Useless rule:

\[
\begin{align*}
\theta_1 C_1 \text{ is satisfied} & \quad \tilde{\theta}_2 C_2 \text{ is satisfied} \\
(\tilde{\theta}_2 \circ \theta_1) C_1 \land C_2 \text{ is or is not satisfied} \\
\end{align*}
\]

(UNSOLVEDCONJUNCTION)

Useful rule:

\[
\begin{align*}
\theta_1 C_1 \text{ is satisfied} & \quad \theta_2(\theta_1 C_2) \text{ is satisfied} \\
(\theta_2 \circ \theta_1) C_1 \land C_2 \text{ is satisfied} \\
\end{align*}
\]

(SOLVEDCONJUNCTION)

Food for thought (or recitation): Find examples to illustrate that UNSOLVEDCONJUNCTION is bogus.
What you can do now

After this lecture, you can write `solve`, a function which, given a constraint \( C \), has one of three outcomes:

- Returns the identity substitution in the case where \( C \) is trivially satisfied
- Returns a non-trivial substitution if \( C \) is satisfiable otherwise.
- Calls `unsatisfiableEquality` in when \( C \) cannot be satisfied

You can also write a type inferencer `ty` for everything except `let` forms. (Coming Monday)
Moving between type scheme and type

From $\sigma$ to $\tau$: instantiate

From $\tau$ to $\sigma$: generalize

$$
\tau ::= \alpha \\
| \mu \\
| (\tau_1, \ldots \tau_n) \tau
$$

$$
\sigma ::= \forall \alpha_1, \ldots \alpha_n. \tau
$$
From Type Scheme to Type

**VAR** rule instantiates type schema with fresh and distinct type variables:

\[ \Gamma(x) = \forall \alpha_1, \ldots, \alpha_n. \tau \]

\[ \alpha'_1, \ldots, \alpha'_n \text{ are fresh and distinct} \]

\[ T, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \ldots \circ (\alpha_n \mapsto \alpha'_n)) \tau \]