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:

$$\theta(\tau_1 \sim \tau_2) = \theta\tau_1 \sim \theta\tau_2$$

$$\theta(C_1 \land C_2) = \theta C_1 \land \theta C_2$$

$$\theta T = T$$
What is a substitution?

Formally, $\theta$ is a function:
- Replaces a *finite* set of type variables with types
- Apply to type, constraint, type environment, …

In code, a data structure:
- “Applied” with `tysubst`, `consubst`
- Made with `idsubst`, $a \mapsto \tau$, `compose`
- 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
Substitution preserves type structure

Type structure:

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

Substitution replaces only type variables:

• Every type constructor is unchanged
• Distributes over type-constructor application

\[
\theta(\text{TYCON } \mu) = \text{TYCON } \mu
\]
\[
\theta(\text{CONAPP } (\tau, [\tau_1, \ldots, \tau_n])) = \text{CONAPP } (\theta \tau, [\theta_1 \tau_1, \ldots \theta_n \tau_n])
\]
Key: Simple type-equality constraint

Solving simple type equalities $\tau_1 \sim \tau_2$

- What are the cases?
- How will you handle them?

datatype ty
  = TYVAR of tyvar
  | TYCON of name
  | CONAPP of ty * ty list
Solving Conjunctions

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

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

Food for thought (or recitation): Find examples to illustrate that UNSOLVED CONJUNCTION is bogus.
Review: Inference for \texttt{IF}

The nano-ML rule is

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

\[
C \wedge \tau_1 \sim \texttt{bool} \wedge \tau_2 \sim \tau_3, \Gamma \vdash \texttt{IF}(e_1, e_2, e_3) : \tau_3
\]
Inference for APPLY

The Typed $\mu$Scheme rule

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

(APPLY)

becomes

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

(APPLY)
Your turn: Begin Rule

The Typed $\mu$Scheme rule

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

becomes

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

From \( \sigma \) to \( \tau \): instantiate

From \( \tau \) to \( \sigma \): generalize

\[
\tau ::= \alpha \\
| \mu \\
| (\tau_1, \ldots \tau_n) \tau \\
\sigma ::= \forall \alpha_1, \ldots \alpha_n . \tau
\]
Instantiation: From Type Scheme to Type

VAR rule instantiates type scheme with fresh and distinct type variables:

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

\[ \alpha_1', \ldots, \alpha_n' \text{ are fresh and distinct} \]

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

(No constraints necessary.)
Generalization: From Type to Type Scheme

Goal is to get \texttt{forall}:

\[ \rightarrow (\text{val fst (lambda (x y) x))} \]
\[ \text{fst} : (\text{forall ('a 'b) ('a 'b -> 'a)}) \]

First derive:

\[ T, \emptyset \vdash (\text{lambda (x y) x}) : \alpha \times \beta \rightarrow \alpha \]

Abstract over \( \alpha, \beta \) and add to environment:

\[ \text{fst} : \forall \alpha, \beta . \alpha \times \beta \rightarrow \alpha \]
Generalize Function

Useful tool for finding quantified type variables:

\[ \text{generalize}(\tau, A) = \forall \alpha_1, \ldots, \alpha_n \cdot \tau \]

where

\[ \{\alpha_1, \ldots \alpha_n\} = \text{ftv}(\tau) - A \]

Example:

\[ \text{generalize}(\alpha \times \beta \rightarrow \alpha, \emptyset) = \forall \alpha, \beta . \alpha \times \beta \rightarrow \alpha \]
First candidate VAL rule (no constraints)

\[ T, \emptyset \vdash e : \tau \]

\[
\sigma = \text{generalize}(\tau, \emptyset) \\
\langle \text{VAL}(x, e), \emptyset \rangle \rightarrow \{x \mapsto \sigma\} 
\]

(VAL WITH \(T\))

But we need to handle nontrivial constraints
Example with nontrivial constraints

(val pick (lambda (x y z) (if x y z)))

During inference, we derive the judgment:

$\alpha_x \sim \text{bool} \land \alpha_y \sim \alpha_z, \emptyset \vdash$

$(\lambda(x \ y \ z) \ (\text{if } x \ y \ z)) : \alpha_x \times \alpha_y \times \alpha_z \rightarrow \alpha_z$

Before generalization, solve the constraint:

$\theta = \{ \alpha_x \mapsto \text{bool}, \alpha_y \mapsto \alpha_z \}$

So the type we need to generalize is

$\theta(\alpha_x \times \alpha_y \times \alpha_z \rightarrow \alpha_z) = \text{bool} \times \alpha_z \times \alpha_z \rightarrow \alpha_z$

And generalize$(\text{bool} \times \alpha_z \times \alpha_z \rightarrow \alpha_z, \emptyset)$ is

$\forall \alpha_z . \text{bool} \times \alpha_z \times \alpha_z \rightarrow \alpha_z$
2nd candidate VAL rule (no context)

\[ C, \emptyset \vdash e : \tau \]
\[ \theta C \text{ is satisfied} \]
\[ \sigma = \text{generalize}(\theta \tau, \emptyset) \]
\[ \langle \text{VAL}(x, e), \emptyset \rangle \rightarrow \{ x \mapsto \sigma \} \quad (\text{VAL 2}) \]

But we need to handle nonempty contexts
VAL rule — the full version

\[ C, \Gamma \vdash e : \tau \]

\[ \theta C \text{ is satisfied} \quad \theta \Gamma = \Gamma \]

\[ \sigma = \text{generalize}(\theta \tau, \text{ftv}(\Gamma)) \]

\[ \langle \text{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma \{ x \mapsto \sigma \} \]

(VAL)
Example of Val rule with non-empty \( \Gamma \)

\[
(val \ pick\text{-}t \ (\lambda (y \ z) \ (\text{pick} \ #t \ y \ z)))
\]

\[
\Gamma = \{\text{pick} \mapsto \forall \alpha. \text{bool} \times \alpha \times \alpha \to \alpha\}
\]

Instantiate \( \text{pick} : \text{bool} \times \alpha_p \times \alpha_p \to \alpha_p \)

Derive the judgment:

\[
\alpha_y \sim \alpha_p \land \alpha_z \sim \alpha_p, \Gamma \vdash \\
(\lambda (y \ z) \ (\text{pick} \ #t \ y \ z)) : \alpha_y \times \alpha_z \to \alpha_p
\]

Before generalization, solve the constraint: \( \theta = \{\alpha_y \mapsto \alpha_p, \alpha_z \mapsto \alpha_p\} \)

Note that \( \theta \Gamma = \Gamma \) and \( \text{ftv}(\Gamma) = \emptyset \).

The type to generalize is \( \theta(\alpha_y \times \alpha_z \to \alpha_p) = \alpha_p \times \alpha_p \to \alpha_p \)

which yields the type: \( \forall \alpha_p . \alpha_p \times \alpha_p \to \alpha_p \)

which is the same as \( \forall \alpha . \alpha \times \alpha \to \alpha \)
Let Examples

(lambda (ys) ; OK
    (let ([s (lambda (x) (cons x '('))])
      (pair (s 1) (s #t))))

(lambda (ys) ; Oops!
    (let ([extend (lambda (x) (cons x ys))])
      (pair (extend 1) (extend #t))))

(lambda (ys) ; OK
    (let ([extend (lambda (x) (cons x ys))])
      (extend 1)))
Let

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

\( \theta C \) is satisfied \hspace{1cm} \theta \) is idempotent

\[ C' = \wedge \{ \alpha \sim \theta \alpha \mid \alpha \in (\text{dom} \theta \cap \text{ftv}(\Gamma)) \} \]

\( \sigma_i = \text{generalize}(\theta \tau_i, \text{ftv}(\Gamma) \cup \text{ftv}(C')), \quad 1 \leq i \leq n \)

\[ C_b, \Gamma \{ x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n \} \vdash e : \tau \]

\[ C' \land C_b, \Gamma \vdash \text{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau \quad \text{(LET)} \]

• If it’s not mentioned in the context, it can be anything: independent

• If it is mentioned in the context, don’t mess with it: dependent
Idempotence

$$\theta \circ \theta = \theta$$

Implies: Applying once is good enough.

<table>
<thead>
<tr>
<th>Good</th>
<th>Bad</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\alpha \mapsto \text{int}$</td>
<td>$\alpha \mapsto \alpha \text{ list}$</td>
</tr>
<tr>
<td>$\alpha \mapsto \beta$</td>
<td>$\alpha \mapsto \beta, \beta \mapsto \gamma$</td>
</tr>
<tr>
<td>$\alpha_1 \mapsto \beta_1, \alpha_2 \mapsto \beta_2$</td>
<td></td>
</tr>
</tbody>
</table>

Implies: If $\alpha \mapsto \tau \in \theta$, then $\theta \alpha = \theta \tau$. 
VAL-REC rule

\[
\begin{align*}
C, \Gamma \{x \mapsto \alpha\} &\vdash e : \tau \quad \alpha \text{ is fresh} \\
\theta(C \land \alpha \sim \tau) &\text{ is satisfied} \quad \theta \Gamma = \Gamma \\
\sigma &= \text{generalize}(\theta \alpha, \text{ftv}(\Gamma)) \\
\langle \text{VAL-REC}(x, e), \Gamma \rangle &\rightarrow \Gamma \{x \mapsto \sigma\}
\end{align*}
\]
\[
\Gamma_e = \Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}, \quad \alpha_i \text{ distinct and fresh}
\]

\[
C_e, \Gamma_e \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n
\]

\[
C = C_e \land \tau_1 \sim \alpha_1 \land \ldots \land \tau_n \sim \alpha_n
\]

\[
\theta C \text{ is satisfied} \quad \theta \text{ is idempotent}
\]

\[
C' = \land \{\alpha \sim \theta \alpha \mid \alpha \in \text{dom} \theta \cap \text{ftv} (\Gamma)\}
\]

\[
\sigma_i = \text{generalize}(\theta \tau_i, \text{ftv}(\Gamma) \cup \text{ftv}(C')), \quad 1 \leq i \leq n
\]

\[
C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau
\]

\[
C' \land C_b, \Gamma \vdash \text{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau
\] (LETREC)