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 θ such that the *constraint θC is satisfied*.

Substitutions distribute over constraints:

$$\begin{aligned} \theta(\tau_1 \sim \tau_2) &= \theta\tau_1 \sim \theta\tau_2 \\ \theta(C_1 \wedge C_2) &= \theta C_1 \wedge \theta C_2 \\ \theta T &= T \end{aligned}$$

What is a substitution?

Formally, θ 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 |--> tau`, `compose`**
- **Find domain with `dom`**

When is a constraint satisfied?

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

$$\frac{C_1 \text{ is satisfied} \qquad C_2 \text{ is satisfied}}{C_1 \wedge C_2 \text{ is satisfied}} \quad \text{(AND)}$$

$$\frac{}{T \text{ is satisfied}} \quad \text{(TRIVIAL)}$$

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

$$\theta(\texttt{TYCON}\ \mu) = \texttt{TYCON}\ \mu$$

$$\theta(\texttt{CONAPP}\ (\tau, [\tau_1, \ldots, \tau_n])) = \texttt{CONAPP}\ (\theta\tau, [\theta_1\tau_1, \ldots \theta_n\tau_n]))$$

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:

$$\frac{\theta_1 C_1 \text{ is satisfied} \quad \tilde{\theta}_2 C_2 \text{ is satisfied}}{(\tilde{\theta}_2 \circ \theta_1)(C_1 \wedge C_2) \text{ is or is not satisfied}}$$

(UNSOLVEDCONJUNCTION)

Useful rule:

$$\frac{\theta_1 C_1 \text{ is satisfied} \quad \theta_2(\theta_1 C_2) \text{ is satisfied}}{(\theta_2 \circ \theta_1)(C_1 \wedge C_2) \text{ is satisfied}}$$

(SOLVEDCONJUNCTION)

Food for thought (or recitation): Find examples to illustrate that UNSOLVEDCONJUNCTION is bogus.

Review: Inference for IF

The nano-ML rule is

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

Moving between type scheme and type

From σ to τ: instantiate

From τ to σ: generalize

$$
\begin{array}{rcl}
\tau & ::= & \alpha \\
 & | & \mu \\
 & | & (\tau_1, \ldots \tau_n)\tau \\
\sigma & ::= & \forall \alpha_1, \ldots \alpha_n \, . \, \tau
\end{array}
$$

Instantiation: From Type Scheme to Type

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

$$\frac{\Gamma(x) = \forall \alpha_1, \ldots \alpha_n \,.\, \tau \qquad \alpha'_1, \ldots \alpha'_n \text{ are fresh and distinct}}{T, \Gamma \vdash x : ((\alpha_1 \mapsto \alpha'_1) \circ \ldots \circ (\alpha_n \mapsto \alpha'_n))\tau} \quad \text{(VAR)}$$

(No constraints necessary.)

Generalization: From Type to Type Scheme

Goal is to get `forall`:

```
-> (val fst (lambda (x y) x))
fst : (forall ('a 'b) ('a 'b -> 'a))
```

First derive:

$$T, \emptyset \vdash \texttt{(lambda (x y) x)} : \alpha \times \beta \rightarrow \alpha$$

Abstract over α, β and add to environment:

$$\texttt{fst} : \forall \alpha, \beta \,.\, \alpha \times \beta \rightarrow \alpha$$

Generalize Function

Useful tool for finding quantified type variables:

$$\texttt{generalize}(\tau, A) = \forall \alpha_1, \ldots, \alpha_n \,.\, \tau$$

where

$$\{\alpha_1, \ldots \alpha_n\} = \mathrm{ftv}(\tau) - A$$

Examples:

$$\texttt{generalize}(\alpha \times \beta \to \alpha, \emptyset) \quad = \forall \alpha, \beta \,.\, \alpha \times \beta \to \alpha$$

$$\texttt{generalize}(\alpha \times \beta \to \alpha, \{\alpha\}) = \forall \beta \,.\, \alpha \times \beta \to \alpha$$

First candidate VAL rule (no constraints)

Empty environment:

$$\frac{T, \emptyset \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \emptyset)}{\langle \textsc{val}(x, e), \emptyset \rangle \rightarrow \{x \mapsto \sigma\}} \qquad \textsc{(Val with } T\textsc{)}$$

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:

$$\alpha_x \sim \texttt{bool} \wedge \alpha_y \sim \alpha_z, \emptyset \vdash$$
$$\texttt{(lambda (x y z) (if x y z))} : \alpha_x \times \alpha_y \times \alpha_z \to \alpha_z$$

Before generalization, solve the constraint:

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

So the type we need to generalize is

$$\theta(\alpha_x \times \alpha_y \times \alpha_z \to \alpha_z) = \texttt{bool} \times \alpha_z \times \alpha_z \to \alpha_z$$

And `generalize`$(\texttt{bool} \times \alpha_z \times \alpha_z \to \alpha_z, \emptyset)$ **is**

$$\forall \alpha_z . \texttt{bool} \times \alpha_z \times \alpha_z \to \alpha_z$$

2nd candidate VAL rule (no context)

$$\frac{\begin{array}{c} C, \emptyset \vdash e : \tau \\ \theta C \text{ is satisfied} \\ \sigma = \texttt{generalize}(\theta\tau, \emptyset) \end{array}}{\langle \texttt{VAL}(x, e), \emptyset \rangle \rightarrow \{x \mapsto \sigma\}} \quad \textbf{(VAL 2)}$$

But we need to handle nonempty contexts

VAL rule — the full version

$$\frac{C, \Gamma \vdash e : \tau \qquad \theta C \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \texttt{generalize}(\theta\tau, \mathrm{ftv}(\Gamma))}{\langle \texttt{VAL}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}} \quad \textbf{(VAL)}$$

Example of Val rule with non-empty Γ

```
(val pick-t (lamba (y z) (pick #t y z)))
```

$$\Gamma = \{\texttt{pick} \mapsto \forall\alpha\,.\,\texttt{bool} \times \alpha \times \alpha \to \alpha\}$$

Instantiate `pick`: $\texttt{bool} \times \alpha_p \times \alpha_p \to \alpha_p$

Derive the judgment:

$$\alpha_y \sim \alpha_p \wedge \alpha_z \sim \alpha_p, \Gamma \vdash$$
$$\texttt{(lambda (y z) (pick \#t y z))} \;:\; \alpha_y \times \alpha_z \to \alpha_p$$

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

Note that $\theta\Gamma = \Gamma$ **and** $\mathrm{ftv}(\Gamma) = \emptyset$.

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

$$\frac{\begin{array}{c} C, \Gamma \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge \{ \alpha \sim \theta\alpha \mid \alpha \in (dom\theta \cap \mathrm{ftv}(\Gamma)) \} \\ \sigma_i = \mathtt{generalize}(\theta\tau_i, \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(C')), \quad 1 \leq i \leq n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \mathsf{LET}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau} \quad (\textsc{Let})$$

- 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.**

Good	Bad
$\alpha \mapsto \texttt{int}$	$\alpha \mapsto \alpha\ \texttt{list}$
$\alpha \mapsto \beta$	$\alpha \mapsto \beta, \beta \mapsto \gamma$
$\alpha_1 \mapsto \beta_1, \alpha_2 \mapsto \beta_2$	

Implies: **If $\alpha \mapsto \tau \in \theta$, then $\theta\alpha = \theta\tau$.**

VAL-REC rule

$$\frac{C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \qquad \alpha \text{ is fresh} \qquad \theta(C \wedge \alpha \sim \tau) \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \texttt{generalize}(\theta\alpha, \mathrm{ftv}(\Gamma))}{\langle \textsc{val-rec}(x, e), \Gamma \rangle \to \Gamma\{x \mapsto \sigma\}} \quad (\textsc{ValRec})$$

LetRec

$$\Gamma_e = \Gamma\{x_1 \mapsto \alpha_1, \ldots, x_n \mapsto \alpha_n\}, \quad \alpha_i \text{ distinct and fresh}$$

$$\frac{\begin{array}{c} C_e, \Gamma_e \vdash e_1, \ldots, e_n : \tau_1, \ldots, \tau_n \\ C = C_e \wedge \tau_1 \sim \alpha_1 \wedge \ldots \wedge \tau_n \sim \alpha_n \\ \theta C \text{ is satisfied} \qquad \theta \text{ is idempotent} \\ C' = \bigwedge\{\alpha \sim \theta\alpha \mid \alpha \in dom\theta \cap \mathrm{ftv}(\Gamma)\} \\ \sigma_i = \mathtt{generalize}(\theta\tau_i, \mathrm{ftv}(\Gamma) \cup \mathrm{ftv}(C')), \quad 1 \leq i \leq n \\ C_b, \Gamma\{x_1 \mapsto \sigma_1, \ldots, x_n \mapsto \sigma_n\} \vdash e : \tau \end{array}}{C' \wedge C_b, \Gamma \vdash \mathsf{LETREC}(\langle x_1, e_1, \ldots, x_n, e_n \rangle, e) : \tau} \text{ (LETREC)}$$