

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:

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

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

$$\theta T = T$$

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`, `cons subst`
- Made with `idsubst`, a $| \rightarrow \tau$, `compose`
- Find domain with `dom`

When is a constraint satisfied?

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

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

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

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

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

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

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:

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

Useful rule:

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

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

Review: Inference for IF

The nano-ML rule is

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

Moving between type scheme and type

From σ to τ : instantiate

From τ to σ : generalize

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

Instantiation: From Type Scheme to Type

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

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

(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:

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

Abstract over α, β and add to environment:

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

Generalize Function

Useful tool for finding quantified type variables:

$$\text{generalize}(\tau, A) = \forall \alpha_1, \dots, \alpha_n . \tau$$

where

$$\{\alpha_1, \dots, \alpha_n\} = \text{ftv}(\tau) - A$$

Examples:

$$\text{generalize}(\alpha \times \beta \rightarrow \alpha, \emptyset) = \forall \alpha, \beta . \alpha \times \beta \rightarrow \alpha$$

$$\text{generalize}(\alpha \times \beta \rightarrow \alpha, \{\alpha\}) = \forall \beta . \alpha \times \beta \rightarrow \alpha$$

First candidate VAL rule (no constraints)

Empty environment:

$$\frac{T, \emptyset \vdash e : \tau}{\sigma = \text{generalize}(\tau, \emptyset)} \quad (\text{VAL WITH } T)$$
$$\langle \text{VAL}(x, e), \emptyset \rangle \rightarrow \{x \mapsto \sigma\}$$

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:

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

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

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)

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

(VAL 2)

But we need to handle nonempty contexts

VAL rule — the full version

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

(VAL)

Example of Val rule with non-empty Γ

(val pick-t (lambda (y z) (pick #t y z)))

$$\Gamma = \{\text{pick} \mapsto \forall \alpha. \text{bool} \times \alpha \times \alpha \rightarrow \alpha\}$$

Instantiate $\text{pick}: \text{bool} \times \alpha_p \times \alpha_p \rightarrow \alpha_p$

Derive the judgment:

$$\alpha_y \sim \alpha_p \wedge \alpha_z \sim \alpha_p, \Gamma \vdash$$

$$(\lambda (y z) (\text{pick} \#t y z)) : \alpha_y \times \alpha_z \rightarrow \alpha_p$$

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 $\theta(\alpha_y \times \alpha_z \rightarrow \alpha_p) = \alpha_p \times \alpha_p \rightarrow \alpha_p$

which yields the type: $\forall \alpha_p. \alpha_p \times \alpha_p \rightarrow \alpha_p$

which is the same as $\forall \alpha. \alpha \times \alpha \rightarrow \alpha$

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

$$C, \Gamma \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n$$

θC is satisfied θ is idempotent

$$C' = \bigwedge \{\alpha \sim \theta\alpha \mid \alpha \in (\text{dom}\theta \cap \text{ftv}(\Gamma))\}$$

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

$$\frac{C_b, \Gamma\{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau}{C' \wedge C_b, \Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}$$

$$(C' \wedge C_b, \Gamma \vdash \text{LET}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau)$$

- 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

$$\alpha \mapsto \text{int}$$

$$\alpha \mapsto \beta$$

$$\alpha_1 \mapsto \beta_1, \alpha_2 \mapsto \beta_2$$

Bad

$$\alpha \mapsto \alpha \text{ list}$$

$$\alpha \mapsto \beta, \beta \mapsto \gamma$$

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

VAL-REC rule

$$\frac{\begin{array}{c} C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \quad \alpha \text{ is fresh} \\ \theta(C \wedge \alpha \sim \tau) \text{ is satisfied} \quad \theta\Gamma = \Gamma \\ \sigma = \text{generalize}(\theta\alpha, \text{ftv}(\Gamma)) \end{array}}{\langle \text{VAL-REC}(x, e), \Gamma \rangle \rightarrow \Gamma\{x \mapsto \sigma\}}$$

(VALREC)

LetRec

$\Gamma_e = \Gamma\{x_1 \mapsto \alpha_1, \dots, x_n \mapsto \alpha_n\}$, α_i distinct and fresh

$$C_e, \Gamma_e \vdash e_1, \dots, e_n : \tau_1, \dots, \tau_n$$

$$C = C_e \wedge \tau_1 \sim \alpha_1 \wedge \dots \wedge \tau_n \sim \alpha_n$$

θC is satisfied θ is idempotent

$$C' = \bigwedge \{\alpha \sim \theta\alpha \mid \alpha \in \text{dom}\theta \cap \text{ftv}(\Gamma)\}$$

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

$$C_b, \Gamma\{x_1 \mapsto \sigma_1, \dots, x_n \mapsto \sigma_n\} \vdash e : \tau$$

$$\frac{C' \wedge C_b, \Gamma \vdash \text{LETREC}(\langle x_1, e_1, \dots, x_n, e_n \rangle, e) : \tau}{(\text{LETREC})}$$