Slides ready to start. . .

Type Inference

Infer types of expressions and declarations.

Infer where to introduce polymorphic types.
(Programmers don't write typed lambdas).

Infer how to instantiate polymorphic types.
(Programmers don't write type applications).

Key Idea

Ideas:

- **Fresh type variables represent unknown types.**
- **Constraints record requirements on those types.**

Judgement forms

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

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

Judgement forms

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

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

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

Formalizing Type Inference

Sad news: Full type inference for polymorphism is not decidable.

Solution: Parameters have monomorphic types.

τ	$::=$	α	type variables
	$\mid$	μ	type constructors: `int`, `list`
	$\mid$	$(\tau_1, \ldots \tau_n)\tau$	constructor application
σ	$::=$	$\forall\alpha_1, \ldots \alpha_n.\tau$	type schema

Variables in Γ introduced via LET, LETREC, VAL, and VAL-REC have type schema.

Variables in Γ introduced via LAMBDA have types.

Moving between type schema and types

$$\sigma \Leftrightarrow \tau$$

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

From Type Schema to Types

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

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

From Types to Type Schema

Goal:

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

From Types to Type Schema

Goal:

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

First derive:

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

From Types to Type Schema

Goal:

```
> (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 free type vars and add to environment:

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

Generalize Function

Useful tool for calculating quantification set:

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

where

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

Example:

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

First Candidate VAL rule

$$\frac{T, \emptyset \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \emptyset)}{\langle \texttt{VAL (x e)}, \emptyset \rangle \rightarrow \{x \mapsto \sigma\}} \qquad \textbf{(VAL 1)}$$

First Candidate VAL rule

$$\frac{T, \emptyset \vdash e : \tau \qquad \sigma = \texttt{generalize}(\tau, \emptyset)}{\langle \texttt{VAL (x e)}, \emptyset \rangle \rightarrow \{x \mapsto \sigma\}} \qquad \textbf{(VAL 1)}$$

But we need a rule that can handle non-trivial constraint sets.

Example

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

During inference, we derive the judgment:

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

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

Example

```
(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

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

which yields the type:

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

Second Candidate VAL rule

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

Second Candidate VAL rule

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

But we need a rule that can handle non-empty contexts.

VAL rule

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

Let Examples

```
(lambda (ys)
   (let ((s (lambda (x) (cons x '()))))
      (pair (s 1) (s #t))))

(lambda (ys)
   (let ((extend (lambda (x) (cons x ys))))
      (pair (extend 1) (extend #t))))

(lambda (ys)
    (let ((extend (lambda (x) (cons x ys))))
       (extend 1)))
```

Let

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

Let

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

- 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

$$\frac{C, \Gamma\{x \mapsto \alpha\} \vdash e : \tau \qquad \alpha \text{ is fresh} \qquad \theta(C \wedge \alpha \sim \tau) \text{ is satisfied} \qquad \theta\Gamma = \Gamma \qquad \sigma = \texttt{generalize}(\theta\alpha, \mathit{ftv}(\Gamma))}{\langle \texttt{VAL-REC(x, e)}, \Gamma \rangle \to \Gamma\{x \mapsto \sigma\}} \quad \textbf{(VALREC)}$$

LetRec

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

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

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

(LETREC)