CS301
Session 15

Agenda

- Interpreting the algorithmic type inference rules
- Type inference for variables
- Type inference for lambdas
- Examples

What it's all about

- The issue is how to turn *nondeterministic* rules into a *deterministic* type inference algorithm
- The algorithm is presented in terms of inference rules that "return" a substitution as well as a type!
- Unification is the way we find substitutions

Type inference judgment

- In $\theta\Gamma \vdash e : \tau$, the substitution theta and the type tau are outputs
- The type may contain type variables
- The typing context contains *type schemes*

Type inference for APPLY

- Type checking:

$$\frac{\Gamma \vdash e : \tau_1 \times \ldots \times \tau_n \to \tau \qquad \Gamma \vdash e_i : \tau_i, \quad 1 \le i \le n}{\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \tau}$$

- Inference:

$$\frac{\theta\Gamma \vdash e, e_1, \ldots, e_n : \hat{\tau}, \tau_1, \ldots, \tau_n \qquad \theta'(\hat{\tau}) = \theta'(\tau_1 \times \ldots \times \tau_n \to \alpha), \text{where } \alpha \text{ is fresh}}{(\theta' \circ \theta)\Gamma \vdash \textsc{apply}(e, e_1, \ldots, e_n) : \theta'\alpha}$$

Operational interpretation

- Infer types $\hat{\tau}, \tau_1, \ldots, \tau_n$ for $e, e_1, \ldots, e_n$, yielding substitution θ
- Pick fresh type var α and unify $\hat{\tau}$ with $\tau_1, \ldots, \tau_n \to \alpha$, yielding θ'
- Answer type is $\theta'\alpha$, answer substitution is $\theta' \circ \theta$

Soundness

- *Soundness* of the type inference rules means that if we infer a type for *e* using the type inference system, then *e* has that type according to the type checking system.
- *Soundness* can be proved by induction on the structure of a type inference.

Example

- Let's infer a type for `(car '(1))` APPLY(PRIM(car),LITERAL(PAIR(NUM(1),NIL)))
- Type scheme $\forall\alpha.\alpha$ `list` $\to \alpha$ for car is found in environment, and we take its most general instance, or α `list` $\to \alpha$; for the literal we use the rule on p. 236 to get `int list`; our substitution is still "empty", or id.
- So now we have types for the function and for its argument, and we want to match them up.

Example (cont'd)

- We pick a fresh type variable β and unify $\alpha\ \texttt{list} \rightarrow \alpha$ with $\texttt{int list} \rightarrow \beta$; the answer substitution is $\theta' = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}\}$
- So the answer type is $\theta'\beta = \texttt{int}$
- and the answer substitution is $\theta' \circ \mathrm{id} = \{\alpha \mapsto \texttt{int}, \beta \mapsto \texttt{int}\}$
- Notice how unification implicitly filled in the type application `(@ car int)`

Type inference for variables

- The typing rule for variables is nondeterministic:

$$\frac{\Gamma(x) = \sigma \qquad \tau <: \sigma}{\Gamma \vdash x : \tau}$$

- To make it algorithmic, we use the *most general instance* of the type scheme:

$$\frac{\Gamma(x) = \sigma \quad \tau = \mathsf{freshinstance}(\sigma)}{\Gamma \vdash x : \tau}$$

Most general instance

- If σ is a type scheme and τ is a most general instance of σ, what could τ be?
- Example: σ is $\forall\alpha,\beta.\alpha\times\beta\rightarrow(\alpha\times\beta)$ list

  could τ be $\beta_1\times\beta_2\rightarrow(\beta_1\times\beta_2)$ list?
  how about $\beta_1\times\beta_1\rightarrow(\beta_1\times\beta_1)$ list?
  int×bool→(int×bool) list?

Type inference for lambda

- Again the typing rule is nondeterministic:

$$\frac{\Gamma\{x_1 \mapsto \tau_1, \ldots, x_n \mapsto \tau_n\} \vdash e : \tau}{\Gamma \vdash \textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e) : \tau_1 \times \ldots \times \tau_n \rightarrow \tau}$$

- We introduce fresh type variables:

$$\frac{\begin{array}{l}\alpha_1, \ldots, \alpha_n \text{ are fresh} \\ \Gamma' = \Gamma\{x_1 \mapsto \forall.\alpha_1, \ldots, x_n \mapsto \forall.\alpha_n\} \\ \theta\Gamma' \vdash e : \tau\end{array}}{\theta\Gamma \vdash \textsc{lambda}(\langle x_1, \ldots, x_n\rangle, e) : \theta\alpha_1 \times \ldots \times \theta\alpha_n \rightarrow \tau}$$

Operational interpretation

- Pick n fresh type variables and form type schemes $\forall.\alpha_i$
- Bind the x_i to $\forall.\alpha_i$ to form the new typing environment Γ'
- Infer a type τ for e in Γ', yielding substitution θ
- The answer substitution is θ and the answer type is $\theta\alpha_1 \times \ldots \times \theta\alpha_n \rightarrow \tau$

Example

- Let's infer a type for `(lambda (x) (+ x 1))`
  LAMBDA(<x>,APPLY(PRIM(+),VAR(x),LIT(NUM(1)))
- Pick a fresh type variable α and bind x to $\forall.\alpha$
- Infer a type for the body in the new environment
  - Use the rule for APPLY

Example (cont'd)

- Environment: $\{x \rightarrow \forall.\alpha\}$
- Infer types for PRIM(+), VAR(x), and LIT(NUM(1)), getting $int \times int \rightarrow int$, α, and int; the substitution is $\theta = id$
- Pick a fresh type variable β, and unify $int \times int \rightarrow int$ with $\alpha \times int \rightarrow \beta$, yielding substitution $\theta' = \{\alpha \Rightarrow int, \beta \Rightarrow int\}$
- Answer: $\theta'\beta = int$, and $\theta' \cdot \theta = \theta'$

Example (cont'd)

- Now we have typed the body of the lambda, so the answer substitution is θ', which is $\{\alpha \Rightarrow int, \beta \Rightarrow int\}$, and the answer type is $\theta'\alpha \rightarrow int$, which is $int \rightarrow int$.
- In this example the algorithm has "filled in" the unstated type of the formal parameter x in `(lambda (x) (+ x 1))`

Let-binding

- Your homework is to work out a type inference example involving let. You need to understand free variables and the *generalize* operation.

Free variables

- The free type variables of a type scheme are those not bound by $\forall$
- For instance, in $\forall\alpha.\alpha\rightarrow\beta$, β is free (and α is bound)
- How about in $\forall.\alpha$?

Generalization

- To type let-binding, we generalize an inferred type *t* to create a type scheme, by "closing over" the variables that are free in *t*, but not over the variables free in the typing environment.
- E.g., generalize($\alpha\rightarrow\beta$,{x$\Rightarrow\forall.\alpha$}) is $\forall\beta.\alpha\rightarrow\beta$