Two Problems

Type checking
- Question: Does exp. $e$ have type $\tau$ in env $\Gamma$?
- Answer: Yes / No
- Method: Type derivation

Typability
- Question Does exp. $e$ have some type in env. $\Gamma$?
  - If so, what is it?
- Answer: Type $\tau$ / error
- Method: Type inference

Type Inference - Outline
- Begin by assigning a type variable as the type of the whole expression
- Decompose the expression into component expressions
- Use typing rules to generate constraints on components and whole
- Recursively find substitution that solves typing judgment of first subcomponent
- Apply substitution to next subcomponent and find substitution solving it; compose with first, etc.
- Apply comp of all substitution to orig. type var. to get answer

Type Inference - Example
- What type can we give to $(\text{fun } x -> \text{fun } f -> f (f x))$?
- Start with a type variable and then look at the way the term is constructed

Type Inference - Example
- First approximate:
  - $\{ \} |- (\text{fun } x -> \text{fun } f -> f (f x)) : \alpha$
  - Second approximate: use fun rule
    - $\{ x : \beta \} |- (\text{fun } f -> f (f x)) : \gamma$
    - $\{ \} |- (\text{fun } x -> \text{fun } f -> f (f x)) : \alpha$
  - Remember constraint $\alpha = (\beta \rightarrow \gamma)$

Type Inference - Example
- Third approximate: use fun rule
  - $\{ f : \delta ; x : \beta \} |- f (f x) : \varepsilon$
  - $\{ x : \beta \} |- (\text{fun } f -> f (f x)) : \gamma$
  - $\{ \} |- (\text{fun } x -> \text{fun } f -> f (f x)) : \alpha$
  - $\alpha = (\beta \rightarrow \gamma); \gamma = (\delta \rightarrow \varepsilon)$
Type Inference - Example

Fourth approximate: use app rule
\[ \{f: \delta; x: \beta\} \vdash f : \phi \rightarrow \epsilon \]
\[ \{f: \delta; x: \beta\} \vdash f \ x : \phi \]
\[ \{x: \beta\} \vdash (\text{fun } f \rightarrow f \ (f \ x)) : \gamma \]
\[ \{\} \vdash (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ (f \ x)) : \alpha \]
\[ \alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \epsilon) \]

Current subst: \( \{\delta \equiv \phi \rightarrow \epsilon\} \)

Var rule: Solve \( \zeta \rightarrow \phi \equiv \phi \rightarrow \epsilon \) Unification
\[ \{f: \phi \rightarrow \epsilon; x: \beta\} \vdash f: \zeta \rightarrow \phi \quad \{f: \phi \rightarrow \epsilon; x: \beta\} \vdash x: \zeta \]
\[ \quad \{f: \phi \rightarrow \epsilon; x: \beta\} \vdash f \ x : \phi \]
\[ \quad \{f: \delta; x: \beta\} \vdash f \ x : \phi \]
\[ \{x: \beta\} \vdash (\text{fun } f \rightarrow f \ (f \ x)) : \gamma \]
\[ \{\} \vdash (\text{fun } x \rightarrow \text{fun } f \rightarrow f \ (f \ x)) : \alpha \]
\[ \alpha \equiv (\beta \rightarrow \gamma); \gamma \equiv (\delta \rightarrow \epsilon) \]
**Type Inference - Example**

- Current subst: \{ζ≡ε, ϕ≡ε, δ≡ε→ε\}
- Apply to next sub-proof

  ... \{f:e→e; x:β\} |- x:e

  ... \{f:φ→ε; x:β\} |- f x : φ

  \{f : δ ; x : β\} |- (f (f x)) : ε

  \{x : β\} |- (fun f -> f (f x)) : γ

  \{ \} |- (fun x -> fun f -> f (f x)) : α

  \[ α ≡ (β → γ); γ ≡ (δ → ε) \]

10/14/14 13

---

**Type Inference - Example**

- Current subst: \{ε≡β, ζ≡β, ϕ≡β, δ≡β→β\}
- Need to satisfy constraint γ ≡ ...

  \( f x \) : ε

  \{x : β\} |- (fun f -> f (f x)) : γ

  \{ } |- (fun x -> fun f -> f (f x)) : α

  \[ α ≡ (β → γ); γ ≡ (δ → ε) \]

10/14/14 17

---

**Type Inference - Example**

- Current subst: \{ε≡β\}o\{ζ≡ε, ϕ≡ε, δ≡ε→ε\}
- Solves subproof; return one layer

  ... \{f:e→e; x:β\} |- x:e

  ... \{f:φ→ε; x:β\} |- f x : φ

  \{f : δ ; x : β\} |- (f (f x)) : ε

  \{x : β\} |- (fun f -> f (f x)) : γ

  \{ \} |- (fun x -> fun f -> f (f x)) : α

  \[ α ≡ (β → γ); γ ≡ (δ → ε) \]

10/14/14 15

---

**Type Inference - Example**

- Current subst: \{ε≡β, ζ≡β, ϕ≡β, δ≡β→β\}
- Solves this subproof; return one layer

  ... \{f:φ→ε; x:β\} |- f x : φ

  \{f : δ ; x : β\} |- (f (f x)) : ε

  \{x : β\} |- (fun f -> f (f x)) : γ

  \{ \} |- (fun x -> fun f -> f (f x)) : α

  \[ α ≡ (β → γ); γ ≡ (δ → ε) \]

10/14/14 16

---

**Type Inference - Example**

- Current subst: \{γ≡((β→β)→β),ε≡β, ζ≡β, ϕ≡β, δ≡β→β\}
- Solves subproof; return one layer

  ... \{f:δ; x:β\} |- (f (f x)) : ε

  \{x : β\} |- (fun f -> f (f x)) : γ

  \{ \} |- (fun x -> fun f -> f (f x)) : α

  \[ α ≡ (β → γ); γ ≡ (δ → ε) \]

10/14/14 18

---

**Type Inference - Example**

- Current subst:

  \[ γ = ((β→β) → β),ε≡β, ζ≡β, ϕ≡β, δ≡β→β \]

- Solves subproof; return one layer

  ... \{f:δ; x:β\} |- (f (f x)) : ε

  \{x : β\} |- (fun f -> f (f x)) : γ

  \{ \} |- (fun x -> fun f -> f (f x)) : α

  \[ α ≡ (β → γ); γ ≡ (δ → ε) \]

10/14/14 14
Type Inference - Example

Current subst:
{α ≡ (β → ((β→β) → β)), γ ≡ ((β→β) → β), ε≡β, ζ≡β, ϕ≡β, δ≡β→β}

Need to satisfy constraint α = (β → γ)
given subst: α = (β → ((β→β) → β))

{x : β} |- (fun f -> f (f x)) : γ

{x : β} |- (fun x -> fun f -> f (f x)) : α

Done: α ≡ (β → ((β→β) → β))

Type Inference Algorithm

Let infer (Γ, e, τ) = α
- Γ is a typing environment (giving polymorphic types to expression variables)
- e is an expression
- τ is a type (with type variables)
- α is a substitution of types for type variables
- Idea: α is the constraints on type variables necessary for Γ |- e : τ
- Should have α(Γ) |- e : α(τ)

Type Inference - Example

Current subst:
{α ≡ (β → ((β→β) → β))}, γ ≡ ((β→β) → β), ε≡β, ζ≡β, ϕ≡β, δ≡β→β

Solves subproof; return on layer

{x : β} |- (fun f -> f (f x)) : γ

{x : β} |- (fun x -> fun f -> f (f x)) : α

Type Inference Algorithm (cont)

Case exp of
- App (e₁ e₂) -->
  - Let α be a fresh variable
  - Let α₁ = infer(Γ, e₁, α → τ)
  - Let α₂ = infer(α(Γ), e₂, α(α))
  - Return α₂ o α₁

Type Inference Algorithm

has_type (Γ, exp, τ) =
- Case exp of
  - Var v --> return Unify(τ = freshInstance(Γ(ν)))
  - Const c --> return Unify(τ = freshInstance ψ )
  - where Γ |- c : ψ by the constant rules
  - fun x -> e -->
    - Let α, β be fresh variables
    - Let α = infer ({x: α} + Γ, e, β)
    - Return Unify( {α(τ) = α(α → β)} ) o σ
Type Inference Algorithm (cont)

Case `exp` of
- If `e_1` then `e_2` else `e_3` -->
  - Let `\sigma_1 = \text{infer}(\Gamma, e_1, \text{bool})`
  - Let `\sigma_2 = \text{infer}(\alpha, e_2, \sigma_1(\tau))`
  - Let `\sigma_3 = \text{infer}(\sigma_2 \circ \sigma_1(\Gamma), e_2, \sigma_2 \circ \sigma_1(\tau))`
  - Return `\sigma_3 \circ \sigma_2 \circ \sigma_1`

Type Inference Algorithm (cont)

Case `exp` of
- Let `x = e_1` in `e_2` -->
  - Let `\alpha` be a fresh variable
  - Let `\sigma_1 = \text{infer}(\Gamma, e_1, \alpha)`
  - Let `\sigma_2 = \text{infer}(\{x: \text{GEN}(\sigma_1(\Gamma), \sigma_1(\alpha))\} + \sigma_1(\Gamma), e_2, \sigma_1(\tau))`
  - Return `\sigma_2 \circ \sigma_1`

Type Inference Algorithm (cont)

Case `exp` of
- Let rec `x = e_1` in `e_2` -->
  - Let `\alpha` be a fresh variable
  - Let `\sigma_1 = \text{infer}(\Gamma, e_1, \alpha)`
  - Let `\sigma_2 = \text{infer}(\{x: \text{GEN}(\sigma_1(\Gamma), \sigma_1(\alpha))\} + \sigma_1(\Gamma), e_2, \sigma_1(\tau))`
  - Return `\sigma_2 \circ \sigma_1`

To infer a type, introduce `type_of`

Let `\alpha` be a fresh variable

**type_of** `(\Gamma, e) =`
- Let `\sigma = \text{infer}(\Gamma, e, \alpha)`
- Return `\sigma(\alpha)`

Need an algorithm for Unif

Simple Implementation Background

type `term` = Variable of string | Const of (string * term list)

let rec subst var_name residue term =
  match term with
  | Variable name ->
    if var_name = name then residue else term
  | Const (c, tys) ->
    Const (c, List.map (subst var_name residue) tys);
Unification Problem

Given a set of pairs of terms ("equations")
\{ (s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n) \}
(the unification problem) does there exist a substitution \( \sigma \) (the unification solution) of terms for variables such that \( \sigma(s_i) = \sigma(t_i) \), for all \( i = 1, \ldots, n \)?

Uses for Unification

- Type Inference and type checking
- Pattern matching as in OCAML
- Can use a simplified version of algorithm
- Logic Programming - Prolog
- Simple parsing

Unification Algorithm

- Let \( S = \{ (s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n) \} \) be a unification problem.
- Case \( S = \{ \} \) : \( \text{Unif}(S) = \text{Identity function} \) (i.e., no substitution)
- Case \( S = \{ (s, t) \} \cup S' \) : Four main steps
  - Delete: if \( s = t \) (they are the same term) then \( \text{Unif}(S) = \text{Unif}(S') \)
  - Decompose: if \( s = f(q_1, \ldots, q_m) \) and \( t = f(r_1, \ldots, r_m) \) (same \( f \), same \( m! \)), then \( \text{Unif}(S) = \text{Unif}(\{ (q_i, r_i), \ldots, (q_m, r_m) \} \cup S') \)
  - Orient: if \( t = x \) is a variable, and \( s \) is not a variable, \( \text{Unif}(S) = \text{Unif}(\{ (x, s) \} \cup S') \)
  - Eliminate: if \( s = x \) is a variable, and \( x \) does not occur in \( t \) (the occurs check), then
    - Let \( \psi = x \mapsto t \)
    - Let \( \psi = \text{Unif}(\psi(S')) \)
    - \( \text{Unif}(S) = \{ x \mapsto \psi(t) \} \circ \psi \)
      - Note: \( \{ x \mapsto a \} \circ \{ y \mapsto b \} = \{ y \mapsto \{ x \mapsto a \}(b) \} \circ \{ x \mapsto a \} \) if \( y \) not in \( a \)

Tricks for Efficient Unification

- Don’t return substitution, rather do it incrementally
- Make substitution be constant time
  - Requires implementation of terms to use mutable structures (or possibly lazy structures)
  - We won’t discuss these
Example

- x, y, z variables, f, g constructors
  - \[ S = \{(f(x), f(g(y,z))), (g(y,f(y))), x\}\]
Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(f(x), f(g(y,z)))$
- Decompose: $(x, g(y,z))$
- $S \rightarrow \{(f(x), f(g(y,z))), (x, g(y,f(y)))\}$
- $\rightarrow \{(x, g(y,z)), (x, g(y,f(y)))\}$

With $\{x \mapsto g(y,f(y))\}$

Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(x, g(y,f(y)))$
- Substitute: $\{x \mapsto g(y,f(y))\}$
- $S \rightarrow \{(x, g(y,z)), (x, g(y,f(y)))\}$
- $\rightarrow \{(g(y,f(y)), g(y,z))\}$

With $\{x \mapsto g(y,f(y))\}$

Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(g(y,f(y)), g(y,z))$
- $S \rightarrow \{(g(y,f(y)), g(y,z))\}$

With $\{x \mapsto g(y,f(y))\}$

Example

- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(y, y)$
- Delete
- $S \rightarrow \{(y, y), (f(y), z)\}$

With $\{x \mapsto g(y,f(y))\}$
Example

- $x,y,z$ variables, $f,g$ constructors
- Pick a pair: $(f(y), z)$
- $S \rightarrow \{(f(y), z)\}$

With $\{x \rightarrow g(y,f(y))\}$

---

Example

- $x,y,z$ variables, $f,g$ constructors
- Pick a pair: $(f(y), z)$
- Orient: $(z, f(y))$
- $S \rightarrow \{(f(y), z)\}$
- $\rightarrow \{(z, f(y))\}$

With $\{x \rightarrow g(y,f(y))\}$

---

Example

- $x,y,z$ variables, $f,g$ constructors
- Pick a pair: $(z, f(y))$
- $S \rightarrow \{(z, f(y))\}$
- $\rightarrow \{\}$

With $\{x \rightarrow \{z \rightarrow f(y)\} \circ \{g(y,f(y))\}\}$

---

Example

- $x,y,z$ variables, $f,g$ constructors
- Pick a pair: $(z, f(y))$
- Eliminate: $\{z \rightarrow f(y)\}$
- $S \rightarrow \{(z, f(y))\}$
- $\rightarrow \{\}$

With $\{x \rightarrow \{z \rightarrow f(y)\} \circ \{g(y,f(y))\}\}$

---

Example

- $S = \{(f(x), f(g(y,z))), (g(y,f(y)), x)\}$
- Solved by $\{x \rightarrow g(y,f(y))\} \circ \{z \rightarrow f(y)\}$

\[
f(g(y,f(y))) = f(y,f(y))
\]

and

\[
g(y,f(y)) = g(y,f(y))
\]
**Example of Failure: Decompose**

- \( S = \{(f(x,g(y)), f(h(y),x))\} \)
- Decompose: \((f(x,g(y)), f(h(y),x))\)
- \( S \rightarrow \{(x,h(y)), (g(y),x)\} \)
- Orient: \((g(y),x)\)
- \( S \rightarrow \{(x,h(y)), (x,g(y))\} \)
- Eliminate: \((x,h(y))\)
- \( S \rightarrow \{(h(y), g(y))\} \) with \( \{x \mid \rightarrow h(y)\} \)
- No rule to apply! Decompose fails!

**Example of Failure: Occurs Check**

- \( S = \{(f(x,g(x)), f(h(x),x))\} \)
- Decompose: \((f(x,g(x)), f(h(x),x))\)
- \( S \rightarrow \{(x,h(x)), (g(x),x)\} \)
- Orient: \((g(y),x)\)
- \( S \rightarrow \{(x,h(x)), (x,g(x))\} \)
- No rules apply.

**Major Phases of a Compiler**

- Source Program
- Lex
- Tokens
- Parse
- Abstract Syntax
- Semantic Analysis
- Symbol Table
- Translate
- Intermediate Representation
- Optimize
  - Optimized IR
  - Instruction Selection
- Optimized Machine-Specific Assembly Language
- Unoptimized Machine-Specific Assembly Language
- Emit code
- Assembly Language
- Assembler
- Relocatable Object Code
- Linker
- Machine Code

Modified from "Modern Compiler Implementation in ML", by Andrew Appel.

**Meta-discourse**

- Language Syntax and Semantics
  - Syntax
    - Regular Expressions, DFSAs and NDFSAs
    - Grammars
  - Semantics
    - Natural Semantics
    - Transition Semantics

**Language Syntax**

- Syntax is the description of which strings of symbols are meaningful expressions in a language
- It takes more than syntax to understand a language; need meaning (semantics) too
- Syntax is the entry point

**Syntax of English Language**

**Pattern 1**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
</tr>
<tr>
<td>The dog</td>
<td>barked</td>
</tr>
<tr>
<td>Susan</td>
<td>yearned</td>
</tr>
</tbody>
</table>

**Pattern 2**

<table>
<thead>
<tr>
<th>Subject</th>
<th>Verb</th>
<th>Direct Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>David</td>
<td>sings</td>
<td>ballads</td>
</tr>
<tr>
<td>The professor</td>
<td>wants</td>
<td>to retire</td>
</tr>
<tr>
<td>The jury</td>
<td>found</td>
<td>the defendant</td>
</tr>
<tr>
<td>Susan</td>
<td>yearned</td>
<td></td>
</tr>
</tbody>
</table>
Elements of Syntax

- Character set – previously always ASCII, now often 64 character sets
- Keywords – usually reserved
- Special constants – cannot be assigned to
- Identifiers – can be assigned to
- Operator symbols
- Delimiters (parenthesis, braces, brackets)
- Blanks (aka white space)

Expressions

- if ... then begin ... ; ... end else begin ... ; ... end
- Type expressions
  \[ \text{typexpr}_1 \rightarrow \text{typexpr}_2 \]
- Declarations (in functional languages)
  \[ \text{let pattern}_1 = \text{expr}_1 \text{ in expr} \]
- Statements (in imperative languages)
  \[ a = b + c \]
- Subprograms
  \[ \text{let pattern}_1 = \text{let rec inner = \ldots in expr} \]

Modules

- Interfaces
- Classes (for object-oriented languages)

Lexing and Parsing

- Converting strings to abstract syntax trees done in two phases
  - **Lexing**: Converting string (or streams of characters) into lists (or streams) of tokens (the “words” of the language)
    - Specification Technique: Regular Expressions
  - **Parsing**: Convert a list of tokens into an abstract syntax tree
    - Specification Technique: BNF Grammars

Formal Language Descriptions

- Regular expressions, regular grammars, finite state automata
- Context-free grammars, BNF grammars, syntax diagrams
- Whole family more of grammars and automata – covered in automata theory

Grammars

- Grammars are formal descriptions of which strings over a given character set are in a particular language
- Language designers write grammar
- Language implementers use grammar to know what programs to accept
- Language users use grammar to know how to write legitimate programs