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
- First approximate:
  $$\{ \} \vdash (\text{fun } x \to \text{fun } f \to f(f x)) : \alpha$$
- Second approximate: use fun rule
  $$\{ x : \beta \} \vdash (\text{fun } f \to f(f x)) : \gamma$$
  $$\{ \} \vdash (\text{fun } x \to \text{fun } f \to f(f x)) : \alpha$$
- Remember constraint $\alpha = (\beta \to \gamma)$

Type Inference - Example
- Third approximate: use fun rule
  $$\{ f : \delta ; x : \beta \} \vdash f(f x) : \epsilon$$
  $$\{ x : \beta \} \vdash (\text{fun } f \to f(f x)) : \gamma$$
  $$\{ \} \vdash (\text{fun } x \to \text{fun } f \to f(f x)) : \alpha$$
- $\alpha = (\beta \to \gamma)$; $\gamma = (\delta \to \epsilon)$
Type Inference - Example

Fourth approximate: use app rule
\[\{f: \delta; x: \beta\} |- f : \phi \rightarrow \varepsilon\]
\[\{f : \delta ; x : \beta\} |- (f (f x)) : \varepsilon\]
\[\{x : \beta\} |- (\text{fun } f -> f (f x)) : \gamma\]
\[\{\} |- (\text{fun } x -> \text{fun } f -> f (f x)) : \alpha\]
\[\alpha = (\beta \rightarrow \gamma); \gamma = (\delta \rightarrow \varepsilon)\]

Type Inference - Example

Fifth approximate: use var rule, get constraint \(\delta \equiv \phi \rightarrow \varepsilon\), Solve with same

Apply to next sub-proof
\[\{f: \delta; x: \beta\} |- f : \phi \rightarrow \varepsilon\]
\[\{f : \delta ; x : \beta\} |- (f (f x)) : \varepsilon\]
\[\{x : \beta\} |- (\text{fun } f -> f (f x)) : \gamma\]
\[\{\} |- (\text{fun } x -> \text{fun } f -> f (f x)) : \alpha\]
\[\alpha = (\beta \rightarrow \gamma); \gamma = (\delta \rightarrow \varepsilon)\]
Type Inference - Example

- Current subst: \(\{\varepsilon = \text{\varphi}, \zeta = \varepsilon, \delta = \varepsilon \rightarrow \varepsilon\}\)
- Apply to next sub-proof
  
  \[
  \begin{align*}
  \ldots & \quad \{f : \varepsilon \rightarrow \varepsilon; x : \beta\} \vdash x : \varepsilon \\
  \ldots & \quad \{f : \varphi \rightarrow \varepsilon; x : \beta\} \vdash f : \varphi \\
  \{f : \delta; x : \beta\} & \vdash (f \,(f \,x)) : \varepsilon \\
  \{x : \beta\} & \vdash (\text{fun} \ f \rightarrow f \,(f \,x)) : \gamma \\
  \{\} & \vdash (\text{fun} \ x \rightarrow \text{fun} \ f \rightarrow f \,(f \,x)) : \alpha
  \end{align*}
\]

- \(\alpha = (\delta \rightarrow \varepsilon); \gamma = (\delta \rightarrow \varepsilon)\)

Type Inference - Example

- Current subst: \(\{\varepsilon = \beta, \zeta = \beta, \varphi = \beta, \delta = \beta \rightarrow \beta\}\)
- Solves subproof; return one layer
  
  \[
  \begin{align*}
  \ldots & \quad \{f : \varepsilon \rightarrow \varepsilon; x : \beta\} \vdash x : \varepsilon \\
  \ldots & \quad \{f : \varphi \rightarrow \varepsilon; x : \beta\} \vdash f : \varphi \\
  \{f : \delta; x : \beta\} & \vdash (f \,(f \,x)) : \varepsilon \\
  \{x : \beta\} & \vdash (\text{fun} \ f \rightarrow f \,(f \,x)) : \gamma \\
  \{\} & \vdash (\text{fun} \ x \rightarrow \text{fun} \ f \rightarrow f \,(f \,x)) : \alpha
  \end{align*}
\]

- \(\alpha = (\beta \rightarrow \gamma); \gamma = (\delta \rightarrow \varepsilon)\)

Type Inference - Example

- Current subst: \(\{\varepsilon = \beta, \zeta = \beta, \varphi = \beta, \delta = \beta \rightarrow \beta\}\)
- Need to satisfy constraint \(\gamma = (\delta \rightarrow \varepsilon)\),
  
given subst, becomes: \(\gamma = ((\beta \rightarrow \beta) \rightarrow \beta)\)

  \[
  \begin{align*}
  \ldots & \quad \{f : \delta; x : \beta\} \vdash (f \,(f \,x)) : \varepsilon \\
  \{x : \beta\} & \vdash (\text{fun} \ f \rightarrow f \,(f \,x)) : \gamma \\
  \{\} & \vdash (\text{fun} \ x \rightarrow \text{fun} \ f \rightarrow f \,(f \,x)) : \alpha
  \end{align*}
\]

- \(\alpha = (\beta \rightarrow \gamma); \gamma = (\delta \rightarrow \varepsilon)\)

Type Inference - Example

- Current subst: \(\{\gamma = ((\beta \rightarrow \beta) \rightarrow \beta), \varepsilon = \beta, \zeta = \beta, \varphi = \beta, \delta = \beta \rightarrow \beta\}\)
- Solves subproof; return one layer
  
  \[
  \begin{align*}
  \ldots & \quad \{f : \delta; x : \beta\} \vdash (f \,(f \,x)) : \varepsilon \\
  \{x : \beta\} & \vdash (\text{fun} \ f \rightarrow f \,(f \,x)) : \gamma \\
  \{\} & \vdash (\text{fun} \ x \rightarrow \text{fun} \ f \rightarrow f \,(f \,x)) : \alpha
  \end{align*}
\]

- \(\alpha = (\beta \rightarrow \gamma); \gamma = (\delta \rightarrow \varepsilon)\)
Type Inference - Example

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

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

\[ \{x : β\} |- (\text{fun } f -> f (f x)) : γ \]
\[ \{\} |- (\text{fun } x -> \text{fun } f -> f (f x)) : α \]

Type Inference Algorithm

Let \(\text{infer } (Γ, e, τ) = σ\)

- \(Γ\) is a typing environment (giving polymorphic types to expression variables)
- \(e\) is an expression
- \(τ\) is a type (with type variables),
- \(σ\) is a substitution of types for type variables

Idea: \(σ\) is the constraints on type variables necessary for \(Γ |- e : τ\)

Should have \(σ(Γ) |- e : σ(τ)\) valid

Type Inference Algorithm (cont)

- Case \(\text{exp}\) of
  - \(\text{App } (e_1 e_2) \rightarrow\)
    - Let \(α\) be a fresh variable
    - Let \(σ_1 = \text{infer}(Γ, e_1, α \to τ)\)
    - Let \(σ_2 = \text{infer}(σ(Γ), e_2, α(α))\)
    - Return \(σ_2 \circ σ_1\)
Type Inference Algorithm (cont)

Case $exp$ of
  
  If $e_1$ then $e_2$ else $e_3$ -->
  
  Let $\alpha_1 = \text{infer}(\Gamma, e_1, \text{bool})$
  
  Let $\alpha_2 = \text{infer}(\alpha_1 \Gamma, e_2, \alpha_1(\tau))$
  
  Let $\alpha_3 = \text{infer}(\alpha_2 \circ \alpha_1(\Gamma), e_2, \alpha_2 \circ \alpha_1(\tau))$
  
  Return $\alpha_3 \circ \alpha_2 \circ \alpha_1$

Type Inference Algorithm (cont)

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

Type Inference Algorithm (cont)

To infer a type, introduce $\text{type_of}$
  
  Let $\alpha$ be a fresh variable
  
  $\text{type_of}(\Gamma, e) =$
  
  Let $\sigma = \text{infer}(\Gamma, e, \alpha)$
  
  Return $\sigma(\alpha)$
  
  Need an algorithm for $\text{Unif}$

Background for Unification

Terms made from constructors and variables (for the simple first order case)
  
  Constructors may be applied to arguments (other terms) to make new terms
  
  Variables and constructors with no arguments are base cases
  
  Constructors applied to different number of arguments (arity) considered different
  
  Substitution of terms for variables

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")
\[ \{(s_1, t_1), (s_2, t_2), ..., (s_n, t_n)\} \]
(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 \( i = 1, ..., n \)?

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 \( S = \{(s_1 = t_1), (s_2 = t_2), ..., (s_n = t_n)\} \) 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

- Eliminate: if \( s = x \) is a variable, and
\( x \) does not occur in \( t \) (the occurs check), then
  - Let \( \psi = \{x \rightarrow t\} \)
  - Let \( \psi = \text{Unif}(\psi(S')) \)
  - \( \text{Unif}(S) = \{x \rightarrow \psi(t)\} \circ \psi \)
  - Note: \( \{x \rightarrow a\} \circ \{y \rightarrow b\} = \{y \rightarrow \{(x \rightarrow a)(b)\}\} \circ \{x \rightarrow a\} \) if \( y \) not in \( a \)

- Delete: if \( s = t \) (they are the same term) then \( \text{Unif}(S) = \text{Unif}(S') \)

- Decompose: if \( s = f(q_1, ..., q_m) \) and
\( t = f(r_1, ..., r_m) \) (same \( f \), same \( m \)!), then
\( \text{Unif}(S) = \text{Unif}((\{(q_1, r_1), ..., (q_m, r_m)\} \cup S') \)

- Orient: if \( t = x \) is a variable, and \( s \) is not a
variable, \( \text{Unif}(S) = \text{Unif}((\{x = s\} \cup S') \)

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

- Unify \{ (f(x) = f(g(f(z),y))), (g(y,y) = x) \} = ?

Example

- x, y, z variables, f, g constructors
- \( S = \{(f(x) = f(g(f(z),y))), (g(y,y) = x)\} \) is nonempty

- Unify \{ (f(x) = f(g(f(z),y))), (g(y,y) = x) \} = ?

Example

- x, y, z variables, f, g constructors
- Pick a pair: \((g(y,y) = x)\)

- Orient: \((x = g(y,y))\)

- Unify \{ (f(x) = f(g(f(z),y))), (x = g(y,y)) \} = Unify \{ (f(x) = f(g(f(z),y))), (x = g(y,y)) \} by Orient

Example

- x, y, z variables, f, g constructors

- Unify \{ (f(x) = f(g(f(z),y))), (x = g(y,y)) \} = ?
Example
- $x, y, z$ variables, $f, g$ constructors
- Pick a pair: $(x = g(y, y))$
  - Unify $\{(f(x) = f(g(f(z), y))), (x = g(y, y))\} = ?$
Example

- x, y, z variables, f, g constructors
- Pick a pair: \((f(g(y,y)) = f(g(f(z),y)))\)
- Decompose: \((f(g(y,y)) = f(g(f(z),y)))\) becomes \((g(y,y) = g(f(z),y)))\)
  - Unify \((f(g(y,y)) = f(g(f(z),y)))\)
    - \(o \{x \rightarrow g(y,y)\} = \)
  - Unify \((g(y,y) = g(f(z),y)))\) \(o \{x \rightarrow g(y,y)\}\)

Example

- x, y, z variables, f, g constructors
- Decompose: \((g(y,y) = g(f(z),y)))\) becomes \((y = f(z)); (y = y))\)
  - Unify \((g(y,y) = g(f(z),y)))\)
    - \(o \{x \rightarrow g(y,y)\} = ?\)

Example

- x, y, z variables, f, g constructors
  - Pick a pair: \((g(y,y) = g(f(z),y)))\)
  - Unify \((g(y,y) = g(f(z),y)))\)
    - \(o \{x \rightarrow g(y,y)\} = ?\)

Example

- x, y, z variables, f, g constructors
- Decompose: \((g(y,y) = g(f(z),y)))\) becomes \((y = f(z)); (y = y))\)
  - Unify \((g(y,y) = g(f(z),y)))\) \(o \{x \rightarrow g(y,y)\} = \)
  - Unify \((y = f(z)); (y = y))\) \(o \{x \rightarrow g(y,y)\}\)

Example

- x, y, z variables, f, g constructors
  - Unify \((y = f(z)); (y = y))\) \(o \{x \rightarrow g(y,y)\} = ?\)
Example

- \( x, y, z \) variables, \( f, g \) constructors
- Pick a pair: \((y = f(z))\)

Unify \{\(y = f(z)\); (y = y)\} \circ \{x \mapsto g(y, y)\} = ?

Example

- \( x, y, z \) variables, \( f, g \) constructors
- Pick a pair: \((y = f(z))\)
- Eliminate \( y \) with \{\(y \mapsto f(z)\)\}

Unify \{\(y = f(z)\); (y = y)\} \circ \{x \mapsto g(y, y)\} = \{\(f(z) = f(z)\)\} 
  \circ \{y \mapsto f(z)\} \circ \{x \mapsto g(y, y)\} = 
  \{\(f(z) = f(z)\)\} 
  \circ \{x \mapsto g(f(z), f(z))\} = ?

Example

- \( x, y, z \) variables, \( f, g \) constructors

Unify \{\(f(z) = f(z)\)\} 
  \circ \{y \mapsto f(z); x \mapsto g(f(z), f(z))\} = ?

Example

- \( x, y, z \) variables, \( f, g \) constructors
- Pick a pair: \((f(z) = f(z))\)

Unify \{\(f(z) = f(z)\)\} 
  \circ \{x \mapsto g(f(z), f(z))\} = ?

Example

- \( x, y, z \) variables, \( f, g \) constructors
- Pick a pair: \((f(z) = f(z))\)
- Delete

Unify \{\(f(z) = f(z)\)\} 
  \circ \{x \mapsto g(f(z), f(z))\} = 
  \{\} \circ \{y \mapsto f(z); x \mapsto g(f(z), f(z))\} =
Example

- $x,y,z$ variables, $f,g$ constructors

- Unify $\{\} \circ \{y \rightarrow f(z); x \rightarrow g(f(z), f(z))\} = \?$

Example of Failure: Decompose

- Unify $\{(f(x,g(y)) = f(h(y), x))\}$
- Decompose: $\{(f(x,g(y)) = f(h(y), x))\} \Rightarrow \{x = h(y)\}, \{g(y) = x\}$
- Orient: $\{g(y) = x\}$
- Unify $\{x = h(y)\}, \{x = g(y)\}$
- Eliminate: $\{x = h(y)\}$
- Unify $\{h(y) = g(y)\} \circ \{x \rightarrow h(y)\}$
- No rule to apply! Decompose fails!

Example of Failure: Occurs Check

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

Major Phases of a Compiler

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

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>yawned</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 balled</td>
<td></td>
</tr>
<tr>
<td>The professor</td>
<td>wants to retire</td>
<td></td>
</tr>
<tr>
<td>The jury</td>
<td>found the defendant guilty</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)
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