Programming Languages and Compilers (CS 421)

Elsa L Gunter
2112 SC, UIUC
http://courses.engr.illinois.edu/cs421

Based in part on slides by Mattox Beckman, as updated by Vikram Adve and Gul Agha

Two Problems

- Type checking
  - Question: Does exp. e have type τ in env Γ?
  - Answer: Yes / No
  - Method: Type derivation
- Typability
  - Question Does exp. e have some type in env. Γ? If so, what is it?
  - Answer: Type τ / 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

  (fun x -> 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:

  { }|- (fun x -> fun f -> f (f x)) : α
- Second approximate: use fun rule

$$\frac{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}{\{\ \} \mid\!- (\text{fun x -> fun f -> f (f x)}) : \alpha}$$

- Remember constraint $\alpha \equiv (\beta \rightarrow \gamma)$

Type Inference - Example

- Third approximate: use fun rule

$$\frac{\dfrac{\{f : \delta\ ;\ x : \beta\} \mid\!- \text{f (f x)} : \varepsilon}{\{x : \beta\} \mid\!- (\text{fun f -> f (f x)}) : \gamma}}{\{\ \} \mid\!- (\text{fun x -> fun f -> f(f x)}) : \alpha}$$

- $\alpha \equiv (\beta \rightarrow \gamma);\ \gamma \equiv (\delta \rightarrow \varepsilon)$

Type Inference - Example

- Fourth approximate: use app rule

$$\frac{\dfrac{\dfrac{\{f:\delta;\ x:\beta\} \vdash f : \varphi \to \varepsilon \quad \{f:\delta;\ x:\beta\} \vdash f\ x : \varphi}{\{f : \delta\ ;\ x : \beta\} \vdash (f\ (f\ x)) : \varepsilon}}{\{x : \beta\} \vdash (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \} \vdash (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

- Fifth approximate: use var rule, get constraint $\delta \equiv \varphi \to \varepsilon$, Solve with same
- Apply to next sub-proof

$$\frac{\dfrac{\dfrac{\{f:\delta;\ x:\beta\} \vdash f : \varphi \to \varepsilon \quad \{f:\delta;\ x:\beta\} \vdash f\ x : \varphi}{\{f : \delta\ ;\ x : \beta\} \vdash (f\ (f\ x)) : \varepsilon}}{\{x : \beta\} \vdash (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \} \vdash (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

- Current subst: $\{\delta \equiv \varphi \to \varepsilon\}$

$$\frac{\dfrac{\dfrac{\ldots \quad \{f:\varphi \to \varepsilon;\ x:\beta\} \vdash f\ x : \varphi}{\{f : \delta\ ;\ x : \beta\} \vdash (f\ (f\ x)) : \varepsilon}}{\{x : \beta\} \vdash (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \} \vdash (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

- Current subst: $\{\delta \equiv \varphi \to \varepsilon\}$ Use App Rule

$$\frac{\dfrac{\dfrac{\ldots \quad \dfrac{\{f:\varphi\to\varepsilon;\ x:\beta\} \vdash f:\zeta\to\varphi \quad \{f:\varphi\to\varepsilon;\ x:\beta\} \vdash x:\zeta}{\{f:\varphi \to \varepsilon;\ x:\beta\} \vdash f\ x : \varphi}}{\{f : \delta\ ;\ x : \beta\} \vdash (f\ (f\ x)) : \varepsilon}}{\{x : \beta\} \vdash (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \} \vdash (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

- Current subst: $\{\delta \equiv \varphi \to \varepsilon\}$
- Var rule: Solve $\zeta\to\varphi \equiv \varphi\to\varepsilon$ Unification

$$\frac{\dfrac{\dfrac{\ldots \quad \dfrac{\{f:\varphi\to\varepsilon;\ x:\beta\} \vdash f:\zeta\to\varphi \quad \{f:\varphi\to\varepsilon;\ x:\beta\} \vdash x:\zeta}{\{f:\varphi \to \varepsilon;\ x:\beta\} \vdash f\ x : \varphi}}{\{f : \delta\ ;\ x : \beta\} \vdash (f\ (f\ x)) : \varepsilon}}{\{x : \beta\} \vdash (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \} \vdash (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

- Current subst: $\{\zeta\equiv\varepsilon,\ \varphi\equiv\varepsilon\} \circ \{\delta \equiv \varphi \to \varepsilon\}$
- Var rule: Solve $\zeta\to\varphi \equiv \varphi\to\varepsilon$ Unification

$$\frac{\dfrac{\dfrac{\ldots \quad \dfrac{\{f:\varphi\to\varepsilon;\ x:\beta\} \vdash f:\zeta\to\varphi \quad \{f:\varphi\to\varepsilon;\ x:\beta\} \vdash x:\zeta}{\{f:\varphi \to \varepsilon;\ x:\beta\} \vdash f\ x : \varphi}}{\{f : \delta\ ;\ x : \beta\} \vdash (f\ (f\ x)) : \varepsilon}}{\{x : \beta\} \vdash (\text{fun } f \text{ -> } f\ (f\ x)) : \gamma}}{\{\ \} \vdash (\text{fun } x \text{ -> fun } f \text{ -> } f\ (f\ x)) : \alpha}$$

- $\alpha \equiv (\beta \to \gamma);\ \gamma \equiv (\delta \to \varepsilon)$

Type Inference - Example

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

… {f:ε→ε; x:β}|- x:ε

… {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)

Type Inference - Example

- Current subst: {ζ≡ε, φ≡ε, δ≡ε→ε}
- Var rule: ε≡β

… {f:ε→ε; x:β}|- x:ε

… {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)

Type Inference - Example

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

… {f:ε→ε; x:β}|- x:ε

… {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)

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)

Type Inference - Example

- Current subst: {ε≡β, ζ≡β, φ≡β, δ≡β→β}
- Need to satisfy constraint γ ≡ (δ → ε), given subst, becomes: γ ≡ ((β→β) → β)

…

{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)

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)

Type Inference - Example

- Current subst:

$\{\gamma \equiv ((\beta \to \beta) \to \beta), \varepsilon \equiv \beta, \zeta \equiv \beta, \varphi \equiv \beta, \delta \equiv \beta \to \beta\}$

- Need to satisfy constraint $\alpha \equiv (\beta \to \gamma)$ given subst: $\alpha \equiv (\beta \to ((\beta \to \beta) \to \beta))$

...

{x : β} |- (fun f -> f (f x)) : γ

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

- $\alpha \equiv (\beta \to \gamma)$;

Type Inference - Example

- Current subst:

$\{\alpha \equiv (\beta \to ((\beta \to \beta) \to \beta)),$
$\gamma \equiv ((\beta \to \beta) \to \beta), \varepsilon \equiv \beta, \zeta \equiv \beta, \varphi \equiv \beta, \delta \equiv \beta \to \beta\}$

- Solves subproof; return on layer

{x : β} |- (fun f -> f (f x)) : γ

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

Type Inference - Example

- Current subst:

$\{\alpha \equiv (\beta \to ((\beta \to \beta) \to \beta)),$
$\gamma \equiv ((\beta \to \beta) \to \beta), \varepsilon \equiv \beta, \zeta \equiv \beta, \varphi \equiv \beta, \delta \equiv \beta \to \beta\}$

- Done: $\alpha \equiv (\beta \to ((\beta \to \beta) \to \beta))$

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

Type Inference Algorithm

Let 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 $\sigma(\Gamma)$ |- *e* : $\sigma(\tau)$ valid

Type Inference Algorithm

infer (Γ, *exp*, τ) =

- Case *exp* of
  - Var *v* --> return Unify{$\tau \equiv$ freshInstance(Γ(*v*))}
    - Replace all quantified type vars by fresh ones
  - Const *c* --> return Unify{$\tau \equiv$ freshInstance φ } where Γ |- *c* : φ by the constant rules
  - fun *x* -> *e* -->
    - Let α, β be fresh variables
    - Let σ = infer ({x: α} + Γ, *e*, β)
    - Return Unify({$\sigma(\tau) \equiv \sigma(\alpha \to \beta)$}) $\circ\ \sigma$

Type Inference Algorithm (cont)

- Case *exp* of
  - App (e_1 e_2) -->
    - Let α be a fresh variable
    - Let σ_1 = infer(Γ, e_1, $\alpha \to \tau$)
    - Let σ_2 = infer($\sigma(\Gamma)$, e_2, $\sigma(\alpha)$)
    - Return $\sigma_2 \circ \sigma_1$

Type Inference Algorithm (cont)

- Case *exp* of
  - If e_1 then e_2 else e_3 -->
    - Let σ_1 = infer(Γ, e_1, bool)
    - Let σ_2 = infer($\sigma\Gamma$, e_2, $\sigma_1(\tau)$)
    - Let σ_3 = infer($\sigma_2 \circ \sigma_1(\Gamma)$,$e_2$,$\sigma_2 \circ \sigma(\tau)$)
    - Return $\sigma_3 \circ \sigma_2 \circ \sigma_1$

Type Inference Algorithm (cont)

- Case *exp* of
  - let x = e_1 in e_2 -->
    - Let α be a fresh variable
    - Let σ_1 = infer(Γ, e_1, α)
    - Let σ_2 = infer({x: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 α be a fresh variable
    - Let σ_1 = infer({x: α} + Γ, e_1, α)
    - Let σ_2 = infer({x: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)

- To infer a type, introduce type_of
- Let α be a fresh variable
- type_of (Γ, e) =
  - Let σ = infer (Γ, e, α)
  - Return σ (α)
- Need an algorithm for 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")

$\{(s_1, t_1), (s_2, t_2), \ldots, (s_n, t_n)\}$

(the *unification problem*) does there exist a substitution σ (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 $S = \{\ \}$: $Unif(S)$ = Identity function (i.e., no substitution)
- Case $S = \{(s, t)\} \cup S'$: Four main steps

Unification Algorithm

- Delete: if $s = t$ (they are the same term) then $Unif(S) = Unif(S')$
- Decompose: if $s = f(q_1, \ldots, q_m)$ and $t = f(r_1, \ldots, r_m)$ (same f, same m!), then $Unif(S) = Unif(\{(q_1, r_1), \ldots, (q_m, r_m)\} \cup S')$
- Orient: if $t = x$ is a variable, and s is not a variable, $Unif(S) = Unif(\{(x = s)\} \cup S')$

Unification Algorithm

- Eliminate: if $s = x$ is a variable, and x does not occur in t (the occurs check), then
  - Let $\varphi = \{x \to t\}$
  - Let $\psi = Unif(\varphi(S'))$
  - $Unif(S) = \{x \to \psi(t)\} \circ \psi$
    - Note: $\{x \to a\} \circ \{y \to b\} = \{y \to (\{x \to a\}(b))\} \circ \{x \to a\}$ 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
- 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)
- 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))), (g(y,y) = x)} = 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
- {(f(x) = f(g(f(z),y))), (x = g(y,y))} is non-empty
- 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: (x = g(y,y))
- Eliminate x with substitution {x→ g(y,y)}
  - Check: x not in 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: (x = g(y,y))
- Eliminate x with substitution {x→ g(y,y)}
- Unify {(f(x) = f(g(f(z),y))), (x = g(y,y))} =
  Unify {(f(g(y,y)) = f(g(f(z),y)))}
  o {x→ g(y,y)}

Example

- x,y,z variables, f,g constructors
- Unify {(f(g(y,y)) = f(g(f(z),y)))}
  o {x→ g(y,y)} = ?

Example

- x,y,z variables, f,g constructors
- {(f(g(y,y)) = f(g(f(z),y)))} is non-empty
- Unify {(f(g(y,y)) = f(g(f(z),y)))}
  o {x→ g(y,y)} = ?

Example

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

Example

- x,y,z variables, f,g constructors
- {(g(y,y) = g(f(z),y))} is non-empty

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

Example

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

- Unify {(g(y,y) = g(f(z),y))} o {x→ g(y,y)} =
 Unify {(y = f(z)); (y = y)} o {x→ g(y,y)}

Example

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

- Unify {(y = f(z)); (y = y)} o {x→ g(y,y)} = ?

Example

- x,y,z variables, f,g constructors
- {(y = f(z)); (y = y)} o {x→ g(y,y) is non-empty

- Unify {(y = f(z)); (y = y)} o {x→ g(y,y)} = ?

Example

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

Example

- x,y,z variables, f,g constructors
- Pick a pair: (y = f(z))
- Eliminate y with {y → f(z)}
- Unify {(y = f(z)); (y = y)} o {x→ g(y,y)} =
  Unify {(f(z) = f(z))}
  o {y → f(z)} o {x→ g(y,y)}=
  Unify {(f(z) = f(z))}
  o {y → f(z); x→ g(f(z), f(z))}

Example

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

Example

- x,y,z variables, f,g constructors
- {(f(z) = f(z))} is non-empty
- Unify {(f(z) = f(z))}
  o {y → f(z); x→ 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))}
  o {y → f(z); x→ 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))}
  o {y → f(z); x→ g(f(z), f(z))} =
  Unify {} o {y → f(z); x→ g(f(z), f(z))}

Example

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

Example

- x,y,z variables, f,g constructors
- {} is empty
- Unify {} = identity function
- Unify {} o {y → f(z); x→ g(f(z), f(z))} =
  {y → f(z); x→ g(f(z), f(z))}

Example

- Unify {(f(x) = f(g(f(z),y))), (g(y,y) = x)} =
  {y → f(z); x→ g(f(z), f(z))}

f(x) = f(g(f(z), y))
→ f(g(f(z), f(z))) = f(g(f(z), f(z)))

g(y , y) = x
→ g(f(z),f(z)) = g(f(z), f(z))

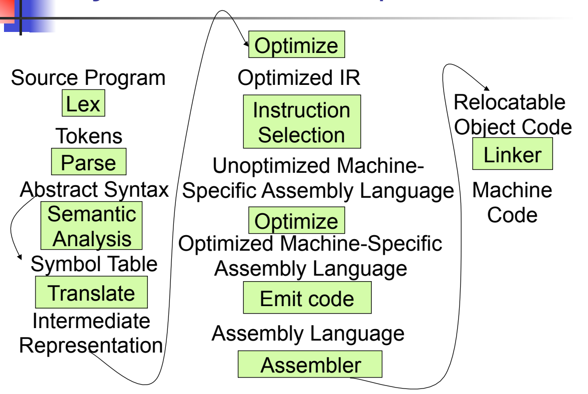
Example of Failure: Decompose

- Unify{(f(x,g(y)) = f(h(y),x))}
- Decompose: (f(x,g(y)) = f(h(y),x))
- = Unify {(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))} o {x → 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))
- = Unify {(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



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

Subject	Verb
David	*sings*
The dog	*barked*
Susan	*yawned*

- Pattern 2

Subject	Verb	Direct Object
David	*sings*	*ballads*
The professor	*wants*	*to retire*
The jury	*found*	*the defendant guilty*

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)

Elements of Syntax

- Expressions
  if ... then begin ... ; ... end else begin ... ; ... end
- Type expressions
  typexpr$_1$ -> *typexpr*$_2$
- Declarations (in functional languages)
  let *pattern*$_1$ = *expr*$_1$ in *expr*
- Statements (in imperative languages)
  a = b + c
- Subprograms
  let *pattern*$_1$ = let rec inner = … in *expr*

Elements of Syntax

- 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