Review of type inference ...

- Untyped terms $\lambda x . e$
- Introduce type variables
  - $x : \alpha$
- Typing rules generate constraints
  - $\alpha = \beta$, $\gamma$
  - $\alpha = \gamma$, $\beta$
  - $\beta = \text{int}$
- Solve constraints
  - $\alpha \equiv \text{int}$, $\text{int}$
  - $\beta \equiv \text{int}$
  - $\gamma \equiv \text{int}$
- Conclude
  - $\lambda x : \text{int}, \text{int}. e$
Representation Analysis
and
Polymorphic Types
Lecture 4
Representation Analysis

• Which values in a program must have the same representation?
  - Not all values of a type need be represented identically
  - Shows abstraction boundaries

• Which values must have the same representation?
  - Those that are used “together”
The Idea

• Old type language

\[ \tau = \alpha \mid \tau \rightarrow \tau \mid \text{int} \]

• New type language
  - Every type is a pair: old type \( \times \) variable

\[ \tau = [\alpha, \beta] \mid [\tau \rightarrow \tau, \beta] \mid [\text{int}, \beta] \]
Old Type Inference Rules

\[ \frac{A(x) = \alpha_x}{A \vdash x : \alpha_x} \quad \frac{A, x : \alpha_x \vdash e : \tau}{A \vdash \lambda x.e : \alpha_x \rightarrow \tau} \quad \frac{\tau_1 = \tau_2 \rightarrow \beta}{A \vdash e_1 e_2 : \beta} \]

\[ \frac{A \vdash e_1 : \tau_1 \quad A \vdash e_2 : \tau_2 \quad \tau_1 = \tau_2 = \text{int}}{A \vdash i : \text{int} \quad A \vdash e_1 + e_2 : \text{int}} \quad \frac{\tau_1 = \text{int} \quad \tau_2 = \tau_3}{A \vdash \text{if } e_1 \ e_2 \ e_3 : \tau_2} \]
New Type Inference Rules

\[
A(x) = [\alpha_x, \beta_x] \quad A,x : [\alpha_x, \beta_x] \vdash e : \tau \\
A \vdash x : [\alpha_x, \beta_x] \quad A \vdash \lambda x.e : [(\alpha_x, \beta_x) \rightarrow \tau, \gamma] \quad \tau_1 = \tau_2 \rightarrow [\alpha, \beta]
\]

\[
A \vdash e_1 : \tau_1 \\
A \vdash e_2 : \tau_2 \\
A \vdash i : [\text{int}, \beta] \\
n + n : [\text{int}, \beta] \\
A \vdash \text{if } e_1 \text{ e}_2 \text{ e}_3 : \tau_2
\]
Example

• A lambda term:
  \[ \lambda x. \lambda y. \lambda z. \lambda w. \text{if } (x + y) \ (z + 1) \ w \]

• Equivalence classes:
  \[ \lambda x. \lambda y. \lambda z. \lambda w. \text{if } (x + y) \ (z + 1) \ w \]
Lackwit

- Representation analysis for C
- Very simple, efficient, and probably useful
- Some ugly pieces
  - E.g., handling of casts
Applications

• “Re-engineering”
  - Make some values more abstract

• Find bugs
  - Every equivalence class with a malloc should have a free

• Just explain what pieces of the program interact
Polymorphism
Polymorphism

• What is type of $\lambda x. x$?
• Is it
  - int, int
  - $\alpha$, $\alpha$
  - bool, bool
  - $(\alpha, \beta), (\alpha, \beta)$
  - all of the above?
Context Sensitivity: Polymorphic Types

• Add a new class of types called type schemes:

\[ \sigma = \forall \alpha. \sigma \mid \tau \]

• Example: A polymorphic identity function

\[ \forall \alpha. \alpha \rightarrow \alpha \]

• Note: All quantifiers are at top level.
The Key Idea

\[ A \vdash e : \tau \]
\[ \alpha \text{ not free in } A \]
\[ \underline{A \vdash e : \forall \alpha . \tau} \]

- This is called *generalization*. 
Instantiation

- Polymorphic assumptions can be used as usual.
- But we still need to turn a polymorphic type into a monomorphic type for the other type rules to work.

\[ \frac{A \vdash e : \forall \alpha. \sigma}{A \vdash e : \sigma[\tau' / \alpha]} \]
Where is Type Inference Strong?

- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forwards/backwards distinction
- Type polymorphism for context sensitivity
Where is Type Inference Weak?

• No flow sensitivity
  - does not work for “null”, “only” etc (LCLint)

• “Flow-sensitive” type systems an ongoing research area