Type-Based Analysis

Lecture 3

Outline

- A language
  - Lambda calculus
- Types
  - Type checking
  - Type inference
- Applications to program analysis
  - Representation analysis
  - Tagging optimization
  - Ailal analysis

The Typed Lambda Calculus

- Lambda calculus
  - But types are assigned to bound variables.
  - Pascal, or C
- Add integers, addition, if-then-else
- Note: Not every expression generated by this grammar is a properly typed term.

\[ e = x | \lambda x : \tau . e | e | i | e + e | if e e e \]

Types

- Function types
- Integers
- Type variables
  - Stand for definite, but unknown, types

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

Function Types

- Intuitively, a type \( \tau \rightarrow \tau \) stands for the set of functions that map arguments of type \( \tau \) to results of type \( \tau \).
- Placeholder for any other structured datatype
  - Lists
  - Trees
  - Arrays

Comments on Abstract Interpretation

- Why is abstract interpretation either forwards or backwards?
- Answer 1
  - Polynomial to compute in one direction
  - Exponential to compute in the other direction
- Answer 2
  - Abstract functions often implemented as functions
  - Impossible to invert---they're code!
Types are Trees

- Types are terms
- Any term can be represented by a tree
  - The parse tree of the term
  - Tree representation is important in algorithms
    \[(\alpha \rightarrow \text{int}) \rightarrow \alpha \rightarrow \text{int}\]

Examples

- We write \( e : \tau \) for the statement "\( e \) has type \( \tau \)."

\[
\lambda x:\alpha . x : \alpha \\
\lambda x:\alpha . y : \beta . x : \beta \rightarrow \alpha \\
\emptyset \vdash \lambda x:\alpha . \lambda y : \beta . x : \alpha \rightarrow \beta \rightarrow \alpha
\]

Untypable Terms

- Some terms have no valid typing.
  - \( \lambda x.x \)
  - \( \lambda x.\lambda y.(x\cdot y)\cdot x \)

- Focus on first example
  - Types are finite
  - Becomes typable if we allow recursive types
    - Recursive types are possibly infinite, regular trees

Type Environments

- To determine whether the types in an expression are correct we perform type checking.
- But we need types for free variables, too
- A type environment is a function from variables to types. The syntax of environments is:

\[
A = \emptyset | A, x : \tau
\]

- The meaning is:

\[
(A, x : \tau)(y) = \tau \text{ if } x = y \\
A(y) \text{ if } x \neq y
\]

Type Checking Rules

- Type checking is done by structural induction.
  - One inference rule for each form
  - Assumptions contain types of free variables
  - A term is well-typed if \( \emptyset \vdash e : \tau \)

Example

\[
x : \alpha, y : \beta \vdash x : \alpha \\
x : \alpha + \lambda y : \beta . x : \beta \rightarrow \alpha \\
\emptyset \vdash \lambda x : \alpha . \lambda y : \beta . x : \alpha \rightarrow \beta \rightarrow \alpha
\]
Type Checking Algorithm

- There is a simple algorithm for type checking.
- Observe that there is only one possible "shape" of the type derivation: only one inference rule applies to each form.

\[
\begin{align*}
? :: X & \Rightarrow ? :: X \\
? :: \lambda y : \beta x : ? & \Rightarrow ? :: \lambda x : \alpha, y : \beta x : ? \\
\varnothing & \Rightarrow \lambda x : \alpha, y : \beta x : a \rightarrow \beta \rightarrow a
\end{align*}
\]

Algorithm (Cont.)

- Walk the proof tree from the root to the leaves, generating the correct environments.
- Assumptions are simply gathered from lambda abstractions.

\[
\begin{align*}
x :: \alpha, y :: \beta & \Rightarrow x :: \alpha, y :: \beta \\
x :: \alpha, y :: \beta, x :: ? & \Rightarrow x :: \alpha, y :: \beta, x :: ? \\
\varnothing & \Rightarrow \lambda x :: \alpha, y :: \beta x :: a \rightarrow \beta \rightarrow a
\end{align*}
\]

A Bigger Example

\[
\begin{align*}
x :: \alpha & \Rightarrow x :: \alpha \\
x :: \alpha & \Rightarrow x :: \alpha \\
\varnothing & \Rightarrow \lambda x :: \alpha, y :: \beta x :: a \rightarrow \beta \rightarrow a \\
\varnothing & \Rightarrow \lambda x :: \alpha, y :: \beta x :: a \rightarrow \beta \rightarrow a
\end{align*}
\]

What Do Types Mean?

- Thm. If \( A :: e :: \tau \) and \( e \rightarrow^* d \), then \( A :: d :: \tau \). Evaluation preserves types.
- This is the basis of a claim that there can be no runtime type errors.
  - Functions applied to data of the wrong type:
    - Adding to a function
    - Using an integer as a function

Type Inference

- The type erasure of \( e \) is \( e \) with all type information removed (i.e., the untyped term).
- Is an untyped term the erasure of some simply typed term? And what are the types?
- This is a type inference problem. We must infer, rather than check, the types.
Outline

- We will develop the inference algorithm in the following steps:
  - recast the type rules in an equivalent form
  - show typing in the new rules reduces to a constraint satisfaction problem
  - show the constraint problem is solvable via term unification.

- We will use this outline again.

The Problems

- There are three problems in developing an algorithm
  - How do we construct the right type assumptions?
  - How do we ensure types match in application?
  - How do we ensure types match in if-then-else?

New Rules

- Sidestep the problems by introducing explicit unknowns and constraints

- Type assumption for variable x is a fresh variable αx

New Rules

- Equality conditions represented as side constraints

- Hypotheses are all arbitrary
  - Can always complete a derivation, pending constraint resolution
Notes

- The introduction of unknowns and constraints works only because the shape of the proof is already known.
  - This tells us where to put the constraints and unknowns.
- The revised rules are trivial to implement, except for handling the constraints.

Solutions of Constraints

- The new rules generate a system of type equations.
- Intuitively, a solution of these equations gives a derivation.
- A solution is a substitution Vars → Types such that the equations are satisfied.

Example

\[ \alpha = \beta \rightarrow \gamma \]
\[ \alpha = \gamma \rightarrow \beta \]
\[ \beta = \text{int} \]

A solution is
\[ \alpha = \text{int} \rightarrow \text{int}, \beta = \text{int}, \gamma = \text{int} \]

Solving Type Equations

- Term equations are a unification problem.
  - Solvable in near-linear time using a union-find based algorithm.
- No solutions \( \alpha = T[\alpha] \) are permitted
  - The occurs check
  - The check is omitted if we allow infinite types.

Unification

- Close constraints under four rules.
- If no inconsistency or occurs check violation found, system has a solution.
  - \( \text{int} \times \rightarrow \gamma \)
\[
\begin{align*}
S \cup \{ \alpha = \alpha \} & \Rightarrow S \\
S \cup \{ \alpha = \tau \} & \Rightarrow S[\tau / \alpha] \cup \{ \alpha \equiv \tau \} \\
S \cup \{ \tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4 \} & \Rightarrow S \cup \{ \tau_1 = \tau_2, \tau_3 = \tau_4 \} \\
S \cup \{ \text{int} = \text{int} \} & \Rightarrow S
\end{align*}
\]

Syntax

- We distinguish solved equations \( \alpha \equiv \tau \)
- Each rule manipulates only unsolved equations.
\[
\begin{align*}
S \cup \{ \alpha = \alpha \} & \Rightarrow S \\
S \cup \{ \alpha = \tau \} & \Rightarrow S[\tau / \alpha] \cup \{ \alpha \equiv \tau \} \\
S \cup \{ \tau_1 \rightarrow \tau_2 = \tau_3 \rightarrow \tau_4 \} & \Rightarrow S \cup \{ \tau_1 = \tau_2, \tau_3 = \tau_4 \} \\
S \cup \{ \text{int} = \text{int} \} & \Rightarrow S
\end{align*}
\]
### Rules 1 and 4
- Rules 1 and 4 eliminate trivial constraints.
- Rule 1 is applied in preference to rule 2 when the only such possible conflict is:
  
  \[
  S \cup \{a = a\} \Rightarrow S
  \]
  
  \[
  S \cup \{a = r\} \Rightarrow S \cup \tau / a \cup \{a \equiv r\}
  \]
  
  \[
  S \cup \{a \rightarrow b \rightarrow a \rightarrow a\} \Rightarrow S \cup \{a = b, b = a, a = a\}
  \]
  
  \[
  S \cup \{\text{int} = \text{int}\} \Rightarrow S
  \]

### Rule 2
- Rule 2 eliminates a variable from all equations but one (which is marked as solved).
  - Note the variable is eliminated from all unsolved as well as solved equations.
  
  \[
  S \cup \{a = a\} \Rightarrow S
  \]
  
  \[
  S \cup \{a = r\} \Rightarrow S \cup \tau / a \cup \{a \equiv r\}
  \]
  
  \[
  S \cup \{a \rightarrow b \rightarrow a \rightarrow a\} \Rightarrow S \cup \{a = b, b = a, a = a\}
  \]
  
  \[
  S \cup \{\text{int} = \text{int}\} \Rightarrow S
  \]

### Rule 3
- Rule 3 applies structural equality to non-trivial terms.
- Note rule 4 is a degenerate case of rule 3 for a type constructor of unit type.
  
  \[
  S \cup \{a = a\} \Rightarrow S
  \]
  
  \[
  S \cup \{a = r\} \Rightarrow S \cup \tau / a \cup \{a \equiv r\}
  \]
  
  \[
  S \cup \{a \rightarrow b \rightarrow a \rightarrow a\} \Rightarrow S \cup \{a = b, b = a, a = a\}
  \]
  
  \[
  S \cup \{\text{int} = \text{int}\} \Rightarrow S
  \]

### Correctness
- Each rule preserves the set of solutions.
  - Rules 1 and 4 eliminate trivial constraints.
  - Rule 2 substitutes equal equals for equals.
  - Rule 3 is the definition of equality on function types.
  
  \[
  S \cup \{a = a\} \Rightarrow S
  \]
  
  \[
  S \cup \{a = r\} \Rightarrow S \cup \tau / a \cup \{a \equiv r\}
  \]
  
  \[
  S \cup \{a \rightarrow b \rightarrow a \rightarrow a\} \Rightarrow S \cup \{a = b, b = a, a = a\}
  \]
  
  \[
  S \cup \{\text{int} = \text{int}\} \Rightarrow S
  \]

### Termination
- Rules 1 and 4 reduce the number of equations.
- Rule 2 reduces the number of variables in unsolved equations.
- Rule 3 decreases the height of terms.
  
  \[
  S \cup \{a = a\} \Rightarrow S
  \]
  
  \[
  S \cup \{a = r\} \Rightarrow S \cup \tau / a \cup \{a \equiv r\}
  \]
  
  \[
  S \cup \{a \rightarrow b \rightarrow a \rightarrow a\} \Rightarrow S \cup \{a = b, b = a, a = a\}
  \]
  
  \[
  S \cup \{\text{int} = \text{int}\} \Rightarrow S
  \]

### Termination (Cont.)
- Rules 1, 3, and 4 always terminate because terms must eventually be reduced to height 0.
  - Eventually rule 2 is applied, reducing the number of variables.
  
  \[
  S \cup \{a = a\} \Rightarrow S
  \]
  
  \[
  S \cup \{a = r\} \Rightarrow S \cup \tau / a \cup \{a \equiv r\}
  \]
  
  \[
  S \cup \{a \rightarrow b \rightarrow a \rightarrow a\} \Rightarrow S \cup \{a = b, b = a, a = a\}
  \]
  
  \[
  S \cup \{\text{int} = \text{int}\} \Rightarrow S
  \]
A Nitpick

- We really need one more operation.
- \( \tau \rightarrow \alpha \) should be flipped to \( \alpha \rightarrow \tau \) if \( \tau \) is not a variable.
- Needed to ensure rule 2 applies whenever possible.
- We just assume equations are maintained in this "normal form".

Solutions

- The final system is a solution.
  - There is one equation \( \alpha = \tau \) for each variable.
  - This is a substitution with all the solutions of the original system
- Must also perform occurs check to guarantee there are no recursive constraints.

Example

rewrites

\[
\begin{align*}
\alpha &= \beta \rightarrow \gamma, \: \alpha &= \gamma \rightarrow \beta, \: \beta = \text{int} \\
\alpha &= \text{int} \rightarrow \gamma, \: \alpha &= \gamma \rightarrow \text{int}, \: \beta = \text{int} \\
\gamma &= \text{int} \rightarrow \gamma, \: \alpha &= \gamma \rightarrow \text{int}, \: \beta = \text{int} \\
\gamma &= \text{int}, \: \text{int} = \gamma, \: \alpha &= \gamma \rightarrow \text{int}, \: \beta = \text{int} \\
\text{int} &= \text{int}, \: \gamma = \text{int}, \: \alpha &= \text{int} \rightarrow \text{int}, \: \beta = \text{int} \\
\gamma &= \text{int}, \: \alpha &= \text{int} \rightarrow \text{int}, \: \beta = \text{int}
\end{align*}
\]

An Example of Failure

\[
\begin{align*}
\alpha &= \beta \rightarrow \gamma, \: \alpha &= \gamma \rightarrow (\beta \rightarrow \beta), \: \beta = \text{int} \\
\alpha &= \text{int} \rightarrow \gamma, \: \alpha &= \gamma \rightarrow (\text{int} \rightarrow \text{int}), \: \beta = \text{int} \\
\gamma &= \text{int} \rightarrow \gamma, \: \alpha &= \gamma \rightarrow \text{int}, \: \beta = \text{int} \\
\gamma &= \text{int}, \: \text{int} = \gamma, \: \alpha &= \gamma \rightarrow \text{int}, \: \beta = \text{int} \\
\text{int} &= \text{int}, \: \gamma = \text{int} \rightarrow \text{int}, \: \alpha &= \text{int} \rightarrow \text{int}, \: \beta = \text{int}
\end{align*}
\]

Notes

- The algorithm produces the most general unifier of the equations.
  - All solutions are preserved.
- Less general solutions are all substitution instances of the most general solution.

An Efficient Algorithm

- The algorithm we have sketched is polynomial, but not very efficient.
  - The repeated substitutions on types is slow.
- Idea: Maintain equivalence classes of types directly.
Union/Find

- Consider sets in which one element is the designated representative.
  - If \( \alpha \) or \( \beta \) is in a set, then it is the representative.
  - Otherwise, the representative is arbitrary.
- Two operations
  - \( \text{Union}(s,t) \) union two sets together
  - \( \text{Find}(s) \) return the representative of set \( s \)
- Equal types will be put in the same set.

Algorithm

\[
\text{fun unify}(m,n) = \begin{cases}
\text{let } s = \text{find}(m), t = \text{find}(n) \text{ in} \\
\text{if } s = t \text{ then true} \\
\text{else if } s = s_1 \rightarrow s_2 \text{ and } t = t_1 \rightarrow t_2 \text{ then} \\
\quad \text{union}(s,t); \text{unify}(s_1,t_1) \& \& \text{unify}(s_2,t_2)) \\
\text{else if } s \text{ or } t \text{ is a variable then} \\
\quad \text{union}(s,t); \text{true}) \\
\text{else false -- one is } \rightarrow \text{ and one is int.}
\end{cases}
\]

Example

\[
\alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int}
\]

Example

\[
\alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int}
\]

Example

\[
\alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int}
\]

Example

\[
\alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int}
\]

Example

\[
\alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int}
\]

Example

\[
\alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int}
\]
Example

\[ \alpha = \beta \rightarrow \gamma \quad \alpha = \gamma \rightarrow \beta \quad \beta = \text{int} \]

Notes

- Any sequence of union and find operations can be made to run in nearly linear time (amortized).
- The constants are very small, giving excellent performance in practice.

Applications

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

Type Inference Rules

\[ \begin{align*}
A + e_1 : \eta_1 & \\
A + e_2 : \eta_2 & \\
\sigma : [\alpha_1, \beta_1] & \\
\tau : [\alpha_2, \beta_2] & \\
A + \tau : \sigma : [\alpha_1, \beta_1] & \\
A + \tau : \sigma \rightarrow \tau & \\
\gamma = [\alpha_3, \beta_3] & \\
\eta = [\alpha_4, \beta_4] & \\
\eta = [\alpha_4, \beta_4] & \\
\gamma = [\alpha_3, \beta_3] & \\
A + [\gamma, \eta] : [\tau \rightarrow \tau, \beta] & \\
A + \tau : \tau_1 & \\
A + \tau : \tau_2 & \\
\end{align*} \]
### Example

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

### Uses

- "Re-engineering"
  - Make some values more abstract
- Find bugs
  - Every equivalence class with a malloc should have a free
- Implemented for C in a tool Lackwit
  - O’Callahan & Jackson

### Dynamic Tag Optimization

- Untyped languages need runtime tags
  - To do runtime type checking
  - Eg., Lisp, Scheme
- Consider an untyped version of our language:
  \[
  e = x | \lambda x.e | e\ e | i | e + e | \text{if } e\ e\ e
  \]
- Every value carries a tag
  - For us, just 1 bit "function" or "integer"

### Term Completion

- View lambda terms as "incomplete"
  - Still need the tagging/tag checking operations
  - \( T \) Tags a value as having type \( T \)
    - Every operation that constructs a \( T \) must invoke \( T \)
  - \( T ? \) Checks if a value has the tag for type \( T \)
    - Every operation that expects to use a \( T \) must invoke \( T ? \)
- Example
  - \( \lambda f.\lambda x. (x + 1) \)
  - \( \text{fun} \lambda f.\text{fun} \lambda x. (\text{fun? } f) (\text{int! } (\text{int? } x) + (\text{int? } (\text{int! } 1)))) \)

### Tagging Optimization

Optimization problem: remove pairs of tag/untag operations without changing program semantics

```plaintext
fun \lambda f.\text{fun} \lambda x. (\text{fun? } f) (\text{int! } (\text{int? } x) + (\text{int? } (\text{int! } 1))))
```

### Coercions

- The tagging/untagging operations are coercions
  - Functions that change the type
  - But change it to what?
  - Introduce type dynamic \( \gamma \) to indicate tagged values
- New types:
  \[
  \tau = \alpha | \tau \rightarrow \tau | \text{int} | \tau | \mu \beta.\tau
  \]
Coercion Signatures

- With type dynamic, we can give signatures to the coercions:
  \[ \text{int: int} \rightarrow T \]
  \[ \text{int?: T} \rightarrow \text{int} \]
  \[ \text{func?: (T \rightarrow T)} \rightarrow T \]
  \[ \text{func?: T} \rightarrow (T \rightarrow T) \]
  \[ \text{noop: T} \rightarrow T \]

- Problem: Decide whether to insert proper coercions or noop.

Type Ordering and Constraints

- Types are related by tagging operations:
  \[ \text{int} \leq T \]
  \[ T \rightarrow T \leq T \]
  \[ T \leq T \]

- Now the choice of a proper coercion or noop can be captured by a constraint:
  \[ \text{int} \leq T \]
  - Says: is either \( T \) or int

Type Inference Rules

\[ \text{if } e_1 : \alpha \text{ then } e_2 : \beta \]
\[ A \rightarrow e_1 : \alpha \]
\[ A \rightarrow e_2 : \beta \]
\[ A \rightarrow \lambda x : \alpha \rightarrow \beta \]
\[ A \rightarrow \alpha : \alpha, \beta : \beta \]
\[ A \rightarrow e_1 : \alpha \]
\[ A \rightarrow e_2 : \beta \]
\[ \text{int} \leq T \]
\[ \text{int} \leq T \]
\[ \text{int} \leq T \]
\[ \text{int} \leq T, \text{int} \leq T \]
\[ A \rightarrow \text{if } e_1 : \alpha \text{ then } e_2 : \beta \]

Constraint Resolution Rules

\[ S \cup \{\alpha = \beta\} \]
\[ S \cup \{\alpha = \gamma\} \]
\[ S \cup \{\alpha = \beta\} \]
\[ S \cup \{\alpha = \beta\} \]

- Note: Arguments of \( \rightarrow \) and rhs of inequality constraints are always variables

Complexity

- Inequality constraints are generated only by inference rules
  - No new ones are ever added by resolution
  - All constraint resolution is of equality constraints
  - Runs at the speed of unification

- Solution of the constraints shows where to insert coercions

Alias Analysis

- In languages with side effects, want to know which locations may have aliases
  - More than one name
  - More than one pointer to them

- E.g.,
  \[ Y = &Z \]
  \[ X = Y \]
  \[ *X = 3 \quad /\text{ changes the value of } *Y*/ \]
The Types

- Deal just with pointers and atomic data
  \[ \tau = \alpha \mid \text{ref}(\tau) \mid \perp \]

A Type Rule

- Consider a C assignment \( x = y \)
- Intuition: \( x \) points to whatever \( y \) points to

\[
\begin{align*}
A &\vdash x: \tau_1 \\
A &\vdash y: \tau_2 \\
\beta &\leq \alpha
\end{align*}
\]

\[
A \vdash x = y : 
\]

A Problem

- \( X \) and \( Y \) are always references
  - They're variables
- But what their contents may be atomic:
  \[
  A = A \\
  X = A \\
  Y = A
  \]
- Now \( x \) and \( y \) are inferred to always "point" to the same thing:
  - But it is obvious there are no pointers here

Type Ordering

- Define an ordering on types:
  \[
  \tau_1 \leq \tau_2 \iff (\tau_1 = \perp \lor \tau_1 = \tau_2)
  \]

- Change the inference rule:
  \[
  \begin{align*}
  A &\vdash x: \tau_1 \\
  A &\vdash y: \tau_2 \\
  \beta &\leq \alpha
  \end{align*}
  \]

\[
A \vdash x = y : 
\]

Example Inference Rules

\[
\begin{align*}
A &\vdash x: \tau_1 \\
A &\vdash y: \tau_2 \\
\beta &\leq \alpha
\end{align*}
\]

\[
A \vdash x = y : 
\]

\[
\begin{align*}
A &\vdash x: \tau_1 \\
A &\vdash y: \tau_2 \\
\beta &\leq \alpha
\end{align*}
\]

\[
A \vdash x = y : 
\]

Constraint Resolution Rules

\[
\begin{align*}
S &\cup \{ \alpha = \tau \} \quad \Rightarrow \quad S \\
S &\cup \{ \alpha = \tau \} \quad \Rightarrow \quad S[\tau/\alpha] \cup \{ \alpha = \tau \} \\
S &\cup \{ \tau_1 = \tau_2 \} \quad \Rightarrow \quad S \cup \{ \tau_1 = \tau_2 \} \\
S &\cup \{ \alpha \leq \tau, \alpha = \text{ref}(\tau) \} \quad \Rightarrow \quad S \cup \{ \alpha = \tau, \alpha = \text{ref}(\tau) \}
\end{align*}
\]
Implementation

- No new inequality constraints are generated by resolution
- Keep a list of pending equality constraints for each variable \( \alpha \)
  - These constraints "fire" when \( \alpha \) is unified with a \( \text{ref} \)
  - More generally, unified with a constructor

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.

A Useful Lemma

\[ A \vdash e : \tau \Rightarrow A[\tau/\alpha] \vdash e : [\tau/\alpha] \]

- A variable in a typing proof can be instantiated to something more specific and the proof still works.
- Proof: Replace \( \alpha \) by \( \tau \) in derivation for \( e \). Show by cases the derivation is still correct.

The Key Idea

\[ \begin{align*}
A \vdash e : \tau \\
\alpha \text{ not free in } A
\end{align*} \]

\[ A \vdash e : \forall \alpha.\tau \]

- 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.

\[ \begin{align*}
A \vdash e : \forall \alpha.\sigma \\
\hline
A \vdash e : \sigma[\tau/\alpha]
\end{align*} \]

Where is Type Inference Strong?

- Handles data structures smoothly
- Works in infinite domains
  - Set of types is unlimited
- No forwards/backwards distinction
- Type polymorphism good fit for context sensitivity
  - Lexically based
  - Less sensitive to program edits than call strings
Where is Type Inference Weak?

- No flow sensitivity
  - Equality-based analysis only gets equivalence classes
  - “backflow” problem

- Context-sensitive analyses don’t always scale
  - Type polymorphism can lead to exponential blowup in constraints