Types

CS242
Lecture 5
Type Systems

• There is a split in the world of programming between
  • Typed languages
  • Untyped languages

• Leave the religious debate aside for now ...
  • We will come back to why there is a debate at all

• Today the focus is on the basics of type systems
What is a Type?

• Consensus
  • A set of values

• Examples
  • Int is the set of all integers
  • Float is the set of all floats
  • Bool is the set \{true, false\}
More Examples

- **List(Int)** is the set of all lists of integers
  - **List** is a *type constructor*
  - A function from types to types

- **Foo**, in Java, is the set of all objects of class **Foo**

- **Int → Int** is the set of functions mapping an integer to an integer
  - E.g., increment, decrement, and many others
What is a Type?

• Consensus
  • A set of values

• In typed languages
  • Every concrete value is an element of some type or types
  • Every legal program has a type

• Type systems have a well-developed notation
  • Useful for more than just type systems ...
Rules of Inference

• Inference rules have the form

   *If Hypothesis is true, then Conclusion is true*

• Type checking computes via reasoning

   *If $E_1$ and $E_2$ have certain types, then $E_3$ has a certain type*

• Rules of inference are a compact notation for “If-Then” statements
From English to an Inference Rule

• Start with a simplified system and gradually add features

• Building blocks
  • Symbol $\land$ is “and”
  • Symbol $\Rightarrow$ is “if-then”
  • $x:T$ is “$x$ has type $T$”
From English to an Inference Rule (2)

If $e_1$ has type $\text{Int}$ and $e_2$ has type $\text{Int}$, then $e_1 + e_2$ has type $\text{Int}$

$$(e_1 \text{ has type } \text{Int} \land e_2 \text{ has type } \text{Int}) \implies e_1 + e_2 \text{ has type } \text{Int}$$

$$(e_1 : \text{Int} \land e_2 : \text{Int}) \implies e_1 + e_2 : \text{Int}$$
From English to an Inference Rule (3)

The statement

\[(e_1: \text{Int} \land e_2: \text{Int}) \Rightarrow e_1 + e_2: \text{Int}\]

is a special case of

\[\text{Hypothesis}_1 \land \ldots \land \text{Hypothesis}_n \Rightarrow \text{Conclusion}\]

This is an inference rule.
Notation for Inference Rules

• By tradition inference rules are written

\[ \vdash \text{Hypothesis}_1 \ldots \vdash \text{Hypothesis}_n \]

\[ \vdash \text{Conclusion} \]

• Type rules have hypotheses and conclusions

\[ \vdash e : T \]

• \( \vdash \) means “it is provable that . . .”
Two Rules

\[
\frac{i \text{ is an integer}}{\vdash i : \text{Int}} \quad \text{[Int]}
\]

\[
\frac{\vdash e_1 : \text{Int} \quad \vdash e_2 : \text{Int}}{\vdash e_1 + e_2 : \text{Int}} \quad \text{[Add]}
\]
Two Rules (Cont.)

• These rules give templates describing how to type integers and + expressions

• By filling in the templates, we can produce complete typings for expressions

• Note that
  • Hypotheses prove facts about subexpressions
  • Conclusions prove facts about the entire expression
Example: $1 + 2$

\[
\begin{align*}
1 & \text{ is an integer} \\
\vdash & 1: \text{Int} \\
\hline
2 & \text{ is an integer} \\
\vdash & 2: \text{Int} \\
\hline
1+2 & \vdash 1+2: \text{Int}
\end{align*}
\]
A Problem

• What is the type of a variable reference?

\[ \text{x is a variable} \]
\[ \vdash x : ? \]

• The rule does not carry enough information to give \( x \) a type.
A Solution

• Put more information in the rules!

• An *environment* gives types for free variables
  • An environment is a function from variables to types
  • Recall that a variable is free in an expression if it is not defined within the expression
Type Environments

Let $A$ be a function from Variables to Types

The sentence $A \vdash e : T$ is read:

Under the assumption that variables have the types given by $A$, it is provable that the expression $e$ has the type $T$
Modified Rules

The type environment is added to all rules:

\[
\begin{align*}
A \leftarrow e_1 : \text{Int} \\
A \leftarrow e_2 : \text{Int} & \quad \text{[Add]} \\
A \leftarrow e_1 + e_2 : \text{Int}
\end{align*}
\]
New Rules

And we can write new rules:

\[ A(x) = T \]

\[
\frac{A(x) = T}{A \vdash x : T} \quad \text{[Var]}
\]
Summary

Describe type systems using logics of the form:

\[ A' \vdash e' : T' \]
\[ A'' \vdash e'' : T'' \]

Assumptions about the free variables of \( e \).

The type for \( e \).

Type of an expression is recursively defined using types of subexpressions.

The program (or program fragment) to be analyzed.
Simply Typed Lambda Calculus
A Language of Typed Functions

Untyped lambda calculus:

\[
\begin{align*}
e & \rightarrow x \mid \lambda x.e \mid e
e & \rightarrow x \mid \lambda x.e \mid e
e & \rightarrow x \mid \lambda x.e \mid e
\end{align*}
\]

Simply typed lambda calculus:

\[
\begin{align*}
e & \rightarrow x \mid \lambda x: t.e \mid e \mid e \mid i
t & \rightarrow \alpha \mid t \rightarrow t \mid \text{int}
\end{align*}
\]
Type Rules

[Var]  
$$A, x : t \vdash x : t$$

[Int]  
$$A \vdash i : \text{int}$$

[Abs]  
$$A, x : t \vdash e : t'$$

$$A \vdash \lambda x : t. e : t'$$

[App]  
$$A \vdash e_1 : t \rightarrow t'$$

$$A \vdash e_2 : t$$

$$A \vdash e_1 e_2 : t'$$
Examples

\[ x: \alpha \vdash x: \alpha \]

\[ \vdash \lambda x: \alpha. \ x : \alpha \to \alpha \]

\[ z : \alpha \to \beta \to \alpha \vdash z : \alpha \to \beta \to \alpha \]

\[ \vdash \lambda z : \alpha \to \beta \to \alpha. \ z : (\alpha \to \beta \to \alpha) \to (\alpha \to \beta \to \alpha) \]

\[ \vdash (\lambda z : \alpha \to \beta \to \alpha. \ z) : (\alpha \to \beta \to \alpha) : \alpha \to \beta \to \alpha \]
Examples

\[
\begin{align*}
  &x: ? \vdash 1 : \text{int} \quad x: ? \vdash x: ? \\
  &\quad \quad \quad \vdash \lambda x: ?. 1 \ x : ? \\
  &x: ? \vdash x: ? \\
  &\quad \quad \quad \vdash (\lambda x: ?. \ x \ x) (\lambda y: ?. y) : ? \\
  &\vdash (\lambda x: \alpha \rightarrow \alpha. \ x): (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha) \\
  &\quad \quad \quad \vdash \lambda y: \alpha.y: \alpha \rightarrow \alpha \\
  &\quad \quad \quad \vdash (\lambda x: (\alpha \rightarrow \alpha) \rightarrow (\alpha \rightarrow \alpha). \ x) (\lambda y: \alpha.y): \alpha \rightarrow \alpha
\end{align*}
\]
Discussion

These examples illustrate two common issues with type systems:

• Code duplication may be required to type programs
  • Each version of the identity used at a different type requires a separate function definition
  • Historical example: Pascal

• The programmer may be required to write in lots of types
  • At least variable declarations are required for type checking
Type Inference
Idea

• Instead of the programmer writing in the types, have an algorithm infer the needed types automatically.

• Obviously results in less typing and less cluttered code

• Less obviously, makes it easier to reuse code

• Type inference is becoming more common
Type Rules

[Var]

\[ A, x: \alpha_x \vdash x: \alpha_x \]

[Abs]

\[ A, x: \alpha_x \vdash e: t \]
\[ A \vdash \lambda x: \alpha_x \cdot e: \alpha_x \rightarrow t \]

[App]

\[ t = t' \rightarrow \beta \]
\[ A \vdash e_1 : t \]
\[ A \vdash e_2 : t' \]
\[ A \vdash e_1 e_2 : \beta \]
Discussion

• Every place a type is required, a fresh type variable is used
  • Stands for some definite, but unknown type

• At function applications, an equation captures what must be true of the types for the program to type check
  • The expression in function position must have a function type
  • The function domain and the function argument must have the same type

• Two steps to constructing a valid typing (or showing none exists)
  • Solve the equations
  • Substitute the solution back into the type derivation to obtain a valid proof
Solving the Constraints

Apply the following rewrite rules until no new constraints can be added

\[ S, t = \alpha \Rightarrow S, t = \alpha, \alpha = t \]  \[ \text{[Reflexivity]} \]

\[ S, \alpha = t_1, \alpha = t_2 \Rightarrow S, \alpha = t_1, \alpha = t_2, t_1 = t_2 \]  \[ \text{[Transitivity]} \]

\[ S, t_1 \rightarrow t_2 = t_3 \rightarrow t_4 \Rightarrow S, t_1 \rightarrow t_2 = t_3 \rightarrow t_4, t_1 = t_3, t_2 = t_4 \]  \[ \text{[Structure]} \]
Solutions

When no constraints can be added, the constraints are *saturated*

- If \( x \rightarrow y = \text{int} \) is in the saturated constraints, then there is no solution, and the program has a type error.

- If \( x = \ldots \rightarrow \ldots x \ldots \) or \( x = \ldots x \ldots \rightarrow \ldots \) is implied by the saturated constraints, then there are no finite solutions
  - Treat the constraints as an undirected graph of equalities, if \( x \) occurs inside a \( \rightarrow \) reachable from \( x \)
  - Example: \( x = x \rightarrow x \)
  - An *occurs check*

- If there are finite solutions, then they can be obtained by back substitution
Example

\[
\begin{align*}
z : \alpha \to \beta \to \alpha & \vdash z : \alpha \to \beta \to \alpha \\
\vdash \lambda z : \alpha \to \beta \to \alpha . z : (\alpha \to \beta \to \alpha) \to (\alpha \to \beta \to \alpha) \\
\vdash (\lambda z : \alpha \to \beta \to \alpha . z) (\lambda x : \alpha . \lambda y : \beta . x : \alpha \to \beta \to \alpha) : \alpha \to \beta \to \alpha
\end{align*}
\]

\[
\begin{align*}
x : \alpha, y : \beta & \vdash x : \alpha \\
x : \alpha & \vdash \lambda y : \beta . x : \beta \to \alpha \\
\vdash \lambda x : \alpha . \lambda y : \beta . x : \alpha \to \beta \to \alpha
\end{align*}
\]
Example

\[ \vdash \lambda z : \alpha_z. \ z : \alpha_z \rightarrow \alpha_z \]

\[ \vdash (\lambda z : \alpha_z. \ z) \ (\lambda x : \alpha_x. \ \lambda y : \alpha_y. \ x : \alpha_x) : \beta \]
Solving ...

\[ \alpha_z \rightarrow \alpha_z = (\alpha_x \rightarrow \alpha_y \rightarrow \alpha_x) \rightarrow \beta \]

\[ \alpha_z = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \alpha_z = \beta \]

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \beta = \alpha_z \]
Convert the Solution to a Substitution

\[ \alpha_z \rightarrow \alpha_z = (\alpha_x \rightarrow \alpha_y \rightarrow \alpha_x) \rightarrow \beta \]

\[ \alpha_z = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \alpha_z = \beta \]

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \beta = \alpha_z \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Example

\[
\begin{align*}
\Gamma & : \alpha_z \vdash z : \alpha_z \\
\vdash \lambda z : \alpha_z. z : \alpha_z \\
\alpha_z \rightarrow \alpha_z &= (\alpha_x \rightarrow \alpha_y \rightarrow \alpha_x) \rightarrow \beta \\
\vdash (\lambda z : \alpha_z. z) (\lambda x : \alpha_x. \lambda y : \alpha_y. x : \alpha_x) : \beta
\end{align*}
\]
Convert the Solution to a Substitution

\[ \alpha_z \rightarrow \alpha_z = (\alpha_x \rightarrow \alpha_y \rightarrow \alpha_x) \rightarrow \beta \]
\[ \alpha_z = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \alpha_z = \beta \]
\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \beta = \alpha_z \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Convert the Solution to a Substitution

\[ \alpha_z = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \alpha_z = \beta \]
\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \beta = \alpha_z \]

Repeat in any order:
Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Convert the Solution to a Substitution

\[ \alpha_z = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \alpha_z = \beta \]
\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \beta = \alpha_z \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Convert the Solution to a Substitution

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \beta = \beta \]

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \beta = \beta \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Example

\[ x : \alpha_x, \ y : \alpha_y \vdash x : \alpha_x \]
\[ x : \alpha_x \vdash \lambda y : \alpha_y. \ x : \alpha_y \rightarrow \alpha_x \]
\[ \vdash \lambda x : \alpha_x. \ \lambda y : \alpha_y. \ x : \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

\[ \vdash (\lambda z : \alpha_z. \ z) \ (\lambda x : \alpha_x. \ \lambda y : \alpha_y. \ x : \alpha_x) : \beta \]
Convert the Solution to a Substitution

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \beta = \beta \]
\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \beta = \beta \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Convert the Solution to a Substitution

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]
\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Convert the Solution to a Substitution

\[ \beta = \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \]

Repeat in any order:

Pick any equation between variables \( \alpha = \beta \) in the solution. Replace \( \beta \) by \( \alpha \) in the solution and in the type derivation.

Drop any equations \( \alpha = \alpha \)

Drop any repeated equations

Drop any equations without a variable on the lhs
Apply the Final Substitution to the Derivation

\[
\begin{align*}
z : \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x & \vdash z : \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \\
\vdash \lambda z : \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \cdot z : (\alpha_x \rightarrow \alpha_y \rightarrow \alpha_x) \rightarrow (\alpha_x \rightarrow \alpha_y \rightarrow \alpha_x) & \vdash (\lambda z : \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x \cdot z) (\lambda x : \alpha_x \cdot \lambda y : \alpha_y \cdot x : \alpha_x) : \alpha_x \rightarrow \alpha_y \rightarrow \alpha_x
\end{align*}
\]
Next Time ...

• More on types!