Lecture 9: The Simply Typed λ-Calculus

Polyvios Pratikakis

Computer Science Department, University of Crete

Type Systems and Programming Languages

Last time

- A type system: a way to recognize only well-behaved programs
  - Statically, without running the program
  - Conservative: might reject programs that run OK
- Defined inductively, using inference rules
  - Here called *type rules*
  - Used to define a typing relation between terms and types
  - Only terms that have a type are accepted
  - All bad programs are not accepted
- Can be proved
  - Progress: a well typed program is not stuck
  - Preservation: a well typed program is still well-typed after a step

Function types

- Going to the λ-calculus
  - What happens with functions?
- Let's add a type for functions: $\rightarrow$
  - $\lambda x.e :\rightarrow$
  - Too simple: $\lambda x.0$ and $\lambda x.\lambda y.\text{true}$ have the same type $\rightarrow$
  - What happens when we call both?
- Solution: function type needs to say more about the function
  - What is the function expecting: argument type
  - What does the function return: result type
  - These can recursively be anything

Function types (cont'd)

- Extend *type language*

$$T ::= \ldots \mid T \rightarrow T$$

  - E.g. $Bool \rightarrow Bool$: a function that takes a boolean and returns a boolean
  - $(Bool \rightarrow Bool) \rightarrow Bool$ a function that takes another function on booleans, and returns a boolean
- Now $\rightarrow$ is a *type constructor*:
  - A function in the type grammar
  - Takes two other types and constructs a new type
- $\rightarrow$ is right-associative, for readability
  - $Bool \rightarrow Bool \rightarrow Bool$ means $Bool \rightarrow (Bool \rightarrow Bool)$

Typing relation

- To assign a type to a term $\lambda x.e$ we need to know what x will be when it is applied
- Two ways to find the type of the argument
  - Require a user annotation $\lambda x : T.e$
  - Analyze the whole program, find where $\lambda x.e$ is applied and find the type of the actual argument passed to x
  - We will see the first
- To compute the result type, compute the type of the body e, assuming x has type T:

$$\frac{x : T \vdash e : T'}{\vdash (\lambda x : T.e) : (T \to T')}$$

Typing relation (cont'd)

- We change the typing relation from $e : T$ to $\Gamma \vdash e : T$
  - Also called a typing *judgement*
  - Γ is a set of assumptions, $x : T, y : T', \ldots$ assigning types to variables
  - Also called a *typing context* or *type environment*
  - In $\vdash e : T$, e has type T under the empty set of assumptions
- Generalized type rule:

$$\frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x : T_1 . e_2 : T_1 \to T_2}$$

- Ensure all variables in Γ are distinct
  - Might need α-renaming of bound variables
  - But always possible

Typing relation (cont'd)

- The rule for typing a variable x follows
- A variable has whatever type it has in the assumptions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T}$$

- If it is not in the assumptions the program is not well-typed
- Follows: open terms are not well typed in an empty environment

Typing relation (cont'd)

- Last syntactic case: function application
- To have $e_1\ e_2$ have a type
  - e_1 must have a function type $T \rightarrow T'$
  - e_2 must have the same type as the function argument T
  - The whole term will have the same type as the result of the function T'
- The type rule

$$\frac{\Gamma \vdash e_1 : T \rightarrow T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'}$$

All together

$$
\begin{array}{lrcl}
\text{Term language} & e & ::= & e\ e \mid \lambda x.e \mid x \\
\text{Type language} & T & ::= & T \rightarrow T
\end{array}
$$

$$
[\text{T-Abs}]\frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x : T_1.e_2 : T_1 \rightarrow T_2}
\qquad
[\text{T-Var}]\frac{x : T \in \Gamma}{\Gamma \vdash x : T}
$$

$$
[\text{T-App}]\frac{\Gamma \vdash e_1 : T \rightarrow T' \qquad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1\ e_2 : T'}
$$

- Not enough!
- Type language is empty: only has inductive case
- We need a *base type*
- Use $Bool$ from last time

Fixed: Add booleans

$$
\begin{array}{llll}
\text{Term language} & e & ::= & e\ e \mid \lambda x.e \mid x \\
 & & \mid & \mathsf{true} \mid \mathsf{false} \mid \mathsf{if}\ e\ \mathsf{then}\ e\ \mathsf{else}\ e \\
\text{Values} & v & ::= & \lambda x.e \mid \mathsf{true} \mid \mathsf{false} \\
\text{Type language} & T & ::= & T \to T \mid Bool
\end{array}
$$

$$
[\text{T-ABS}]\ \frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x : T_1.e_2 : T_1 \to T_2}
\qquad
[\text{T-VAR}]\ \frac{x : T \in \Gamma}{\Gamma \vdash x : T}
$$

$$
[\text{T-APP}]\ \frac{\begin{array}{c}\Gamma \vdash e_1 : T \to T' \\ \Gamma \vdash e_2 : T\end{array}}{\Gamma \vdash e_1\ e_2 : T'}
\qquad
[\text{T-IF}]\ \frac{\begin{array}{c}\Gamma \vdash e_1 : Bool \\ \Gamma \vdash e_2 : T \quad \Gamma \vdash e_3 : T\end{array}}{\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T}
$$

$$
[\text{T-TRUE}]\ \frac{}{\Gamma \vdash \mathsf{true} : Bool}
\qquad
[\text{T-FALSE}]\ \frac{}{\Gamma \vdash \mathsf{false} : Bool}
$$

Semantics (eager, small-step)

$$[\textsc{E-App}]\ \frac{}{(\lambda x : T.e)\ v \to e[v/x]}$$

$$[\textsc{E-App1}]\ \frac{e_1 \to e_1'}{e_1\ e_2 \to e_1'\ e_2} \qquad [\textsc{E-App2}]\ \frac{e_2 \to e_2'}{v\ e_2 \to v\ e_2'}$$

$$[\textsc{E-If}]\ \frac{e_1 \to e_1'}{\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \to \text{if } e_1' \text{ then } e_2 \text{ else } e_3}$$

$$[\textsc{E-If-True}]\ \frac{}{\text{if true then } e_2 \text{ else } e_3 \to e_2}$$

$$[\textsc{E-If-False}]\ \frac{}{\text{if false then } e_2 \text{ else } e_3 \to e_3}$$

Examples

$$\dfrac{\dfrac{\dfrac{x : Bool \in x : Bool}{x : Bool \vdash x : Bool}}{\vdash (\lambda x : Bool.x) : Bool \to Bool} \qquad \dfrac{}{\vdash \text{true} : Bool}}{\vdash (\lambda x : Bool.x)\ \text{true} : Bool}$$

$$\dfrac{\dfrac{}{\vdash \text{true} : Bool} \qquad \dfrac{\dfrac{}{x : Bool \vdash \text{true} : Bool}}{\vdash (\lambda x : Bool.\text{true}) : Bool \to Bool} \qquad D_3 : \vdash (\lambda x : Bool.\text{false}) : Bool \to Bool}{\vdash \text{if true then } (\lambda x : Bool.\text{true}) \text{ else } (\lambda x : Bool.\text{false}) : Bool \to Bool}$$

$$D_3 : \dfrac{\dfrac{}{x : Bool \vdash \text{false} : Bool}}{\vdash (\lambda x : Bool.\text{false}) : Bool \to Bool}$$

Inversion lemma

- Inversion of the typing relation
  - If $\Gamma \vdash x : T$ then $x : T \in \Gamma$
  - If $\Gamma \vdash (\lambda x : T_1.e) : T$ then there is a T_2 such that $T = T_1 \to T_2$ and $\Gamma, x : T_1 \vdash e : T_2$
  - If $\Gamma \vdash e_1\ e_2 : T$ then there is a T' such that $\Gamma \vdash e_1 : T' \to T$ and $\Gamma \vdash e_2 : T'$
  - If $\Gamma \vdash \mathsf{true} : T$ then $T = Bool$
  - If $\Gamma \vdash \mathsf{false} : T$ then $T = Bool$
  - If $\Gamma \vdash \mathsf{if}\ e_1\ \mathsf{then}\ e_2\ \mathsf{else}\ e_3 : T$ then $\Gamma \vdash e_1 : Bool$ and $\Gamma \vdash e_2 : T$ and $\Gamma \vdash e_3 : T$
- Proof follows from the definition of typing

Canonical forms

- We can reason about values based on their type
  - If v has type $Bool$ then it is either true or false
  - If v has type $T \to T'$ then $v = (\lambda x : T.e)$
- Proof by case analysis on the syntax of v

Progress theorem

- If $\vdash e : T$ then either e is a value, or it can take a step $e \rightarrow e'$ to some e'
- Proof like last time
- Differences:
  - Variable case can never happen ($\vdash x : T$ is impossible)
  - Lambda case is a value
  - Application case: apply lemma recursively to e_1, e_2
    - If e_1 is not a value, then apply [E-App1]
    - If e_1 is a value and e_2 isn't, apply [E-App2]
    - If they are both values, apply inversion lemma and canonical form to e_1, and then [E-App]

Permutation lemma

- If $\Gamma \vdash e : T$ and Γ' is a permutation of Γ, then $\Gamma' \vdash e : T$
- Proof is straightforward by induction on $\Gamma \vdash e : T$
- Case analysis:
    - For each typing rule
    - Apply inductively on premises (if any)
    - Reapply typing rule to construct judgement with Γ'
    - Remember all variables in Γ are different (ensured by α-renaming terms when necessary)

Weakening lemma

- If $\Gamma \vdash e : T$ and $x \notin dom(\Gamma)$, then $\Gamma, x : T' \vdash e : T$
- Proof by induction on $\Gamma \vdash e : T$ (as above)
- Intuitively: we can add irrelevant declarations around a term without affecting its type

Substitution lemma

- If $\Gamma, x : T' \vdash e : T$ and $\Gamma \vdash e' : T'$, then $\Gamma \vdash e[e'/x] : T$
- Proof by induction on $\Gamma, x : T' \vdash e : T$
  - Case analysis on typing relation (for each type rule)
  - For most cases, simply apply inductively on premises and then reapply the same type rule to reconstruct the wanted conclusion
  - Except two cases: Variable and Lambda

Substitution lemma (cont'd)

- In case the term is a variable
  - If the variable is the one replaced, the wanted conclusion is given in the assumption
  - If not, construct the wanted conclusion using [T-Var]
- In case the term is a Lambda
  - We cannot apply the lemma inductively on the premises, they have different environments
  - We must bring the two environments to the same form first
  - Use permutation on premise
  - Use weakening on second assumption
  - We can now apply the lemma inductively and reconstruct the conclusion using [T-Abs]

Preservation theorem

- *If* $\Gamma \vdash e : T$ *and* $e \to e'$ *then* $\Gamma \vdash e' : T$
- Proof by induction on $e \to e'$ (each semantic rule)
- Uses the substitution lemma for the β-reduction in $[\textsc{E-App}]$
  - Intuitively, β-reduction replaces all occurrences of a variable x in e with e'
  - Similarly, substitution lemma replaces all typings of x (using $[\textsc{T-Var}]$) in the typing of e, with the typing of e'
  - Might have to adjust the environments using weakening

Next time

- Implementing the type-system in OCaml