Lecture 9: The Simply Typed $\lambda$-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 \( \lambda \)-calculus
  - What happens with functions?
- Let's add a type for functions: \( \to \)
  - \( \lambda x.e : \to \)
  - Too simple: \( \lambda x.0 \) and \( \lambda x.\lambda y.\text{true} \) have the same type \( \to \)
  - 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*

\[ T ::= \ldots | T \rightarrow T \]

- E.g. \( \text{Bool} \rightarrow \text{Bool} \): a function that takes a boolean and returns a boolean
- \( (\text{Bool} \rightarrow \text{Bool}) \rightarrow \text{Bool} \) 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
  - \( \text{Bool} \rightarrow \text{Bool} \rightarrow \text{Bool} \) means \( \text{Bool} \rightarrow (\text{Bool} \rightarrow \text{Bool}) \)
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 $T$:

$$
\frac{x : T \vdash e : T'}{\vdash (\lambda x : T. e) : (T \rightarrow T')}
$$
Typing relation (cont’d)

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

- Generalized type rule:
  $$
  \Gamma, x : T_1 \vdash e_2 : T_2 \quad \frac{\frac{\Gamma \vdash \lambda x : T_1.e_2 : T_1 \rightarrow T_2}{\Gamma, x : T_1 \vdash e_2 : T_2}}
  $$

- Ensure all variables in $\Gamma$ are distinct
  - Might need $\alpha$-renaming of bound variables
  - But always possible
The rule for typing a variable $x$ follows:

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

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

\[
\Gamma \vdash e_1 : T \rightarrow T' \quad \Gamma \vdash e_2 : T \\
\hline
\Gamma \vdash e_1 \ e_2 : T'
\]
All together

Term language \[ e ::= e \mathbin{e} | \lambda x. e | x \]

Type language \[ T ::= T \to T \]

\[
\frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x : T_1.e_2 : T_1 \to T_2} \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T}
\]

\[
\frac{\Gamma \vdash e_1 : T \to T' \quad \Gamma \vdash e_2 : T}{\Gamma \vdash e_1 \, e_2 : T'}
\]

- Not enough!
- Type language is empty: only has inductive case
- We need a base type
- Use \textit{Bool} from last time
Fixed: Add booleans

Term language \( e ::= e \ e \mid \lambda x. e \mid x \)
\[ \mid \text{true} \mid \text{false} \mid \text{if } e \text{ then } e \text{ else } e \]

Values \( v ::= \lambda x. e \mid \text{true} \mid \text{false} \)

Type language \( T ::= T \to T \mid \text{Bool} \)

\( [\text{T-Abs}] \quad \frac{\Gamma, x : T_1 \vdash e_2 : T_2}{\Gamma \vdash \lambda x : T_1.e_2 : T_1 \to T_2} \)
\( [\text{T-VAR}] \quad \frac{x : T \in \Gamma}{\Gamma \vdash x : T} \)

\( [\text{T-APP}] \quad \frac{\Gamma \vdash e_1 : T \to T'}{\Gamma \vdash e_1 \ e_2 : T'} \)
\( [\text{T-IF}] \quad \frac{\Gamma \vdash e_1 : \text{Bool} \quad \Gamma \vdash e_3 : T}{\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T} \)

\( [\text{T-TRUE}] \quad \frac{\Gamma \vdash \text{true} : \text{Bool}}{} \)
\( [\text{T-FALSE}] \quad \frac{\Gamma \vdash \text{false} : \text{Bool}}{} \)
Semantics (eager, small-step)

\[
\begin{align*}
\text{[E-APP] } & : (\lambda x : T.e) \nu \rightarrow e[\nu/x] \\
\text{[E-APP1] } & : e_1 \rightarrow e'_1 \\
\text{[E-APP2] } & : e_2 \rightarrow e'_2 \\
\text{[E-IF] } & : e_1 \rightarrow e'_1 \\
\text{if } e_1 \text{ then } e_2 \text{ else } e_3 \rightarrow & \text{ if } e'_1 \text{ then } e_2 \text{ else } e_3 \\
\text{[E-IF-TRUE] } & : \text{if true then } e_2 \text{ else } e_3 \rightarrow e_2 \\
\text{[E-IF-FALSE] } & : \text{if false then } e_2 \text{ else } e_3 \rightarrow e_3
\end{align*}
\]
Examples

\[ x : \text{Bool} \vdash x : \text{Bool} \]

\[ \vdash (\lambda x : \text{Bool}. x) : \text{Bool} \rightarrow \text{Bool} \]

\[ \vdash \text{true} : \text{Bool} \]

\[ \vdash (\lambda x : \text{Bool}. x) \text{ true} : \text{Bool} \]

\[ \vdash \text{true} : \text{Bool} \]

\[ \vdash (\lambda x : \text{Bool}. \text{true}) : \text{Bool} \rightarrow \text{Bool} \]

\[ D_3 : \vdash (\lambda x : \text{Bool}. \text{false}) : \text{Bool} \rightarrow \text{Bool} \]

\[ \vdash \text{if true then} (\lambda x : \text{Bool}. \text{true}) \text{ else} (\lambda x : \text{Bool}. \text{false}) : \text{Bool} \rightarrow \text{Bool} \]

\[ D_3 : \]

\[ x : \text{Bool} \vdash \text{false} : \text{Bool} \]

\[ \vdash (\lambda x : \text{Bool}. \text{false}) : \text{Bool} \rightarrow \text{Bool} \]
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 $T = T_1 \rightarrow T_2$ and $\Gamma, x : T_1 \vdash e : T_2$
  - If $\Gamma \vdash e_1 e_2 : T$ then there is a $T'$ such that $\Gamma \vdash e_1 : T' \rightarrow T$ and $\Gamma \vdash e_2 : T'$
  - If $\Gamma \vdash \text{true} : T$ then $T = \text{Bool}$
  - If $\Gamma \vdash \text{false} : T$ then $T = \text{Bool}$
  - If $\Gamma \vdash \text{if } e_1 \text{ then } e_2 \text{ else } e_3 : T$ then $\Gamma \vdash e_1 : \text{Bool}$ 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 $\text{Bool}$ then it is either true or false
  - If $v$ has type $T \rightarrow T'$ 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 $\Gamma'$ is a permutation of $\Gamma$, 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 $\Gamma'$
  - Remember all variables in $\Gamma$ are different (ensured by $\alpha$-renaming terms when necessary)
Weakening lemma

- If $\Gamma \vdash e : T$ and $x \notin \text{dom}(\Gamma)$, 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 $e \rightarrow e'$ then $\Gamma \vdash e' : T$

Proof by induction on $e \rightarrow e'$ (each semantic rule)

- Uses the substitution lemma for the $\beta$-reduction in $[E\text{-}APP]$
  
  - Intuitively, $\beta$-reduction replaces all occurrences of a variable $x$ in $e$ with $e'$
  
  - Similarly, substitution lemma replaces all typings of $x$ (using $[T\text{-}VAR]$) 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