CSCI 740 - Programming Language Theory

Lecture 12

Simple Types: Progress & Preservation

Instructor: Hossein Hojjat

September 27, 2017

Recap

- A λ-calculus expression is defined as

$$\begin{array}{rcll} e & ::= & x & \text{variable} \\ & | & \lambda x.e & \text{abstraction} \\ & | & e\ e & \text{application} \end{array}$$

α-conversion

$$\lambda x.e \ \rightarrow_\alpha \ \lambda y.e[x \mapsto y] \qquad \text{if } y \notin \mathsf{FV}(e)$$

β-reduction

$$(\lambda x.e)\ e' \ \rightarrow_\beta \ e[x \mapsto e']$$

η-conversion

$$\lambda x.(e\ x) \ \rightarrow_\eta \ e \qquad \text{if } \ x \notin \mathsf{FV}(e)$$

Recap: Formalizing a Type System

Typing Rules

- Typing rules tell us how to derive typing judgments
- Very similar to derivation rules in Big Step Operational Semantics

$$\frac{\text{Premises}}{\text{Judgment}}$$

- Example: Language of Expressions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

Example: Language of Expressions

$$\frac{x : T \in \Gamma}{\Gamma \vdash x : T} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}}$$

- Show that the following Judgment is valid

$$x : \mathtt{Int}, y : \mathtt{Int} \ \vdash \ x + (y + 5) : \mathtt{Int}$$

$$\dfrac{\dfrac{x : \mathtt{Int} \in \{x : \mathtt{Int}, y : \mathtt{Int}\}}{x : \mathtt{Int}, y : \mathtt{Int} \ \vdash \ x : \mathtt{Int}} \qquad \dfrac{\dfrac{y : \mathtt{Int} \in \{x : \mathtt{Int}, y : \mathtt{Int}\}}{x : \mathtt{Int}, y : \mathtt{Int} \vdash y : \mathtt{Int}} \qquad \dfrac{}{x : \mathtt{Int}, y : \mathtt{Int} \vdash 5 : \mathtt{Int}}}{x : \mathtt{Int}, y : \mathtt{Int} \vdash (x + 5) \ : \ \mathtt{Int}}}{x : \mathtt{Int}, y : \mathtt{Int} \ \vdash \ x + (y + 5) : \mathtt{Int}}$$

Static Semantics of Simply Typed λ-calculus (F_1)

- Basic Typing Rules

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1 \quad e_2 : \tau}$$

- Extensions

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 \ = \ e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$

Exercise

- Is this a valid typing judgment?

$$\left(\lambda x : \mathsf{Bool}.\lambda y : \mathsf{Int}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y + 1\right)\ :\ \mathsf{Bool} \to \mathsf{Int} \to \mathsf{Int}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ \ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$

Exercise

- Is this a valid typing judgment?

$$\left(\lambda x : \mathsf{Bool}.\lambda y : \mathsf{Int}.\ \mathsf{if}\ x\ \mathsf{then}\ y\ \mathsf{else}\ y+1\right)\ :\ \mathsf{Bool} \to \mathsf{Int} \to \mathsf{Int}$$

- How about this one?

$$\left(\lambda x : \mathsf{Int}.\lambda y : \mathsf{Bool}.\ x+y\right)\ :\ \mathsf{Int} \to \mathsf{Bool} \to \mathsf{Int}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2} \qquad \frac{\Gamma \vdash e_1 : \tau' \to \tau \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ \ e_2 : \tau}$$

$$\frac{}{\Gamma \vdash n : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 + e_2 : \mathsf{Int}} \qquad \frac{\Gamma \vdash e_1 : \mathsf{Int} \qquad \Gamma \vdash e_2 : \mathsf{Int}}{\Gamma \vdash e_1 = e_2 : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$

Simply Typed λ-calculus (F_1)

- We have defined a really strong type system on λ-calculus
- It is so strong:

  it won't even let us write non-terminating computation
- We can actually prove this!

Progress and Preservation

Type System Correctness

- What makes a type system “correct”?

Type System Correctness

- What makes a type system "correct"?

Motivating Example

Consider the following simple language for arithmetic expressions:

$$\mathit{Expr} ::= \texttt{true} \mid \texttt{false} \mid \texttt{zero}$$

$$\mathit{Expr} ::= \texttt{if}\ \mathit{Expr}\ \texttt{then}\ \mathit{Expr}\ \texttt{else}\ \mathit{Expr}$$

$$\mathit{Expr} ::= \texttt{succ}(\mathit{Expr})$$

$$\mathit{Expr} ::= \texttt{pred}(\mathit{Expr})$$

$$\mathit{Expr} ::= \texttt{isZero}(\mathit{Expr})$$

$$\mathit{Val} ::= \texttt{true} \mid \texttt{false} \mid \mathit{NVal}$$

$$\mathit{NVal} ::= \texttt{zero} \mid \texttt{succ}\ \mathit{NVal}$$

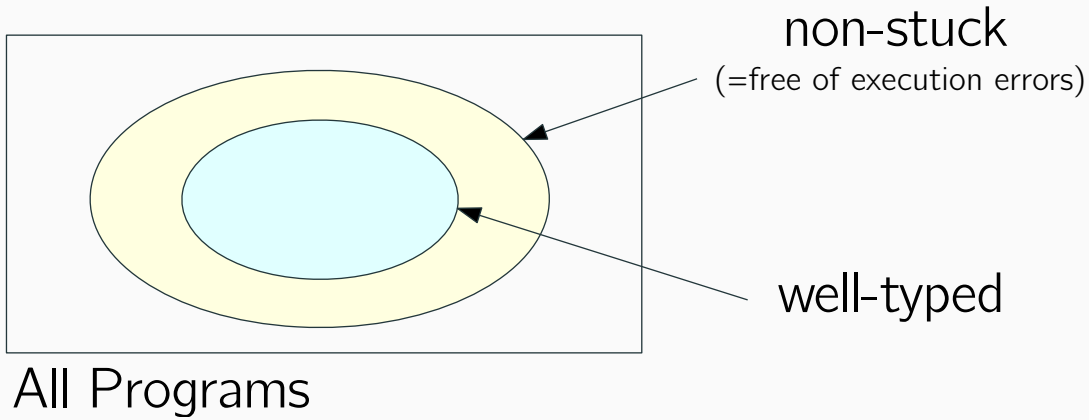
- "Stuck" terms:
  - `succ(true)`
  - `isZero(false)`
  - `if zero then true else false`
- Cannot evaluate, but are not values: no semantics = execution error

Find a Type System for Expr, so that well-typed terms do NOT get stuck

A Type System for Arithmetic Expressions

Find a Type System for Expr, so that well-typed terms do NOT get stuck

- The converse is not necessarily true:
  `if true then zero else succ(true)` is not stuck
  (evaluates to zero),
  but is not well-typed!



- Introduce two types `Bool` and `Int`,
  representing Booleans and Numbers
- Every Expr will be of type `Bool` or `Int`

A Type System for Arithmetic Expressions

Find a Type System for Expr, so that well-typed terms do NOT get stuck

- We can define the following typing rules for the language:

$$\frac{}{\Gamma \vdash \texttt{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{zero} : \mathsf{Int}}$$

$$\frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash \texttt{succ}(e) : \mathsf{Int}} \qquad \frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash \texttt{pred}(e) : \mathsf{Int}} \qquad \frac{\Gamma \vdash e : \mathsf{Int}}{\Gamma \vdash \texttt{isZero}(e) : \mathsf{Bool}}$$

$$\frac{\Gamma \vdash e : \mathsf{Bool} \quad \Gamma \vdash e_t : \tau \quad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$

- This type system does not allow expressions like `succ(true)`, `isZero(false)`,`if zero then true else false`
- How can we formally **prove** that the type system does not allow any of those expressions?

Proving Type Safety

"well-typed terms do not go wrong"

$$\text{Safety} = \text{Progress} + \text{Preservation}$$

- Progress: A well-typed term is NOT stuck
- Preservation: Evaluation preserves well-typedness

well-typed $\xrightarrow{\text{Progress}}$ NOT stuck $\rightarrow$ either value or we can evaluate $\xrightarrow{\text{Preserve}}$ result is well-typed

Preservation

- Using Big-Step semantics we can argue global preservation

$$\Gamma \vdash e_1 : \tau \ \wedge \ e_1 \Downarrow e_2 \implies \Gamma \vdash e_2 : \tau$$

- Prove by induction on the structure of derivation of $e_1 \Downarrow e_2$

Preservation

- Proof by induction on Structure of Evaluation
- Base cases: trivial

$$\frac{}{x \Downarrow x} \qquad \frac{}{\lambda x.e \Downarrow \lambda x.e}$$

- Inductive case is a little trickier

$$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_1'[x \mapsto e_2] \Downarrow e_3}{e_1\ e_2 \Downarrow e_3}$$

Preservation

- Inductive case: $$\frac{e_1 \Downarrow \lambda x.e_1' \qquad e_1'[x \mapsto e_2] \Downarrow e_3}{e_1\ e_2 \Downarrow e_3}$$

- Given $\Gamma \vdash e_1\ e_2 : \tau_{e_{12}}$ we want to show that $\Gamma \vdash e_3 : \tau_{e_{12}}$
- By our typing rule, we have
$$\frac{\Gamma \vdash e_1 : \tau' \to \tau_{e_{12}} \qquad \Gamma \vdash e_2 : \tau'}{\Gamma \vdash e_1\ e_2 : \tau_{e_{12}}}$$
- And by the IH, we know that e_1 preserves type $(\lambda x.e_1') : \tau' \to \tau_{e_{12}}$
- Which again by the typing rule $\dfrac{\Gamma, x : \tau' \vdash e_1' : \tau_{e_{12}}}{\Gamma \vdash (\lambda x : \tau'.e_1') : \tau' \to \tau_{e_{12}}}$
- Now, we need to show that
$$\Gamma, x : \tau' \vdash e_1' : \tau_{e_{12}} \land \Gamma \vdash e_2 : \tau' \implies \Gamma \vdash e_1'[x \mapsto e_2] : \tau_{e_{12}}$$
- And by the IH, we know that $e_1'[x \mapsto e_2]$ preserves type
$$\Gamma \vdash e_1'[x \mapsto e_2] : \tau_{e_{12}} \implies \Gamma \vdash e_3 : \tau_{e_{12}}$$

Significance of Type Preservation

- The result of an evaluation has the same type as the initial expression
- The theorem does not say if the evaluation
  - never gets stuck
    (e.g., trying to apply a non-function, to add non-integers, ...)
  - terminates
- Both of the above facts are true for F_1
- We need a small-step semantics to prove that the execution never gets stuck

- Big-step goes directly from initial program to result
- Small-step evaluates one step at a time

Small-Step Semantics for F_1

- Define contexts with one *hole* $\circ$

$$H ::= \circ \mid H\ e \mid H + e \mid n + H \mid \text{if } H \text{ then } e_1 \text{ else } e_2 \mid H == e \mid n == H$$

- $H[e]$: Filling the hole in H with the expression e
- Local Reduction Rules
  - $n_1 + n_2 \to n$ (where $n = \text{PLUS } n_1\ n_2$))
  - $n_1 == n_2 \to b$ (where $b = (\text{EQUALS } n_1\ n_2)$
  - if true then e_1 else $e_2 \ \to\ e_1$
  - if false then e_1 else $e_2 \ \to\ e_2$
  - $(\lambda x : \tau . e_1)\ v \ \to\ e_1[x \mapsto v]$
- Global Reduction Rules
  - $H[r] \to H[e]$ iff $r \to e$

Proof Strategy

- Progress Theorem

If $\vdash e : \tau$ and e is not a value, then there is an e' such that $e \to e'$

- We can prove this through a decomposition lemma
- If $\vdash e : \tau$ and e is not a value, then there are H and r s.t. $e = H[r]$
- This guarantees one step of progress

Proving the Progress Theorem

If $\vdash e : \tau$ and e is not a value, then there is an e' such that $e \to e'$ or equivalently, $e = H[r]$

- Proved by induction on the derivation of $\vdash e : \tau$
- Base case: Irreducible values

$$\frac{}{\Gamma \vdash \texttt{true} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash \texttt{false} : \mathsf{Bool}} \qquad \frac{}{\Gamma \vdash n : \mathsf{Int}}$$

$$\frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau} \qquad \frac{\Gamma, x : \tau_1 \vdash e : \tau_2}{\Gamma \vdash (\lambda x : \tau_1.\ e) : \tau_1 \to \tau_2}$$

Proving the Progress Theorem

- Inductive case

$$\frac{\Gamma \vdash e : \mathsf{Bool} \qquad \Gamma \vdash e_t : \tau \qquad \Gamma \vdash e_f : \tau}{\Gamma \vdash \mathsf{if}\ e\ \mathsf{then}\ e_t\ \mathsf{else}\ e_f : \tau}$$

- by the IH, e can be irreducible,
  - in which case it must be true or false and the whole thing is a redex
- Or, it can be decomposed into $H[r]$
  - in which case "if $\circ$ then e_1 else e_2" is a valid context

- Similar proof for other cases such as application