Languages with Higher Types

Hongfei Fu

John Hopcroft Center for Computer Science
Shanghai Jiao Tong University

May 7th

Today's Topic

Typed Languages

- a functional programming language
- eager operational semantics
- lazy operational semantics

A Basic Functional Language

Textbook, Page 183 – 186

A Basic Functional Language

Types

The types τ are generated from the grammar:

$$\tau ::= \textbf{int} \mid \tau_1 * \tau_2 \mid \tau_1 \to \tau_2$$

- **int**: the basic type for integers
- $\tau_1 * \tau_2$: product type for ordered pairs
  (e.g., type **int** $*$ **int** for integer pairs $(0, 1), (-1, 2), \ldots$)
- $\tau_1 \to \tau_2$: function type from τ_1 to τ_2
  (e.g., type **int** $\to$ **int** for functions from integers to integers)

A Basic Functional Language

Variables

- **Var** $= \{x, y, \dots\}$: a set of variables
- **type**(x): the uniquely-fixed type for the variable x

We write $x : \tau$ to stress that **type**$(x) = \tau$.

A Basic Functional Language

Terms

The terms t are generated from the grammar:

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$t \quad ::= \quad x \text{ (variables)}$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$\begin{array}{rcl} t & ::= & x \text{ (variables)} \\ & | & n \text{ (integer constants)} \end{array}$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)}
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)}
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)}
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)} \\
& | & \textbf{snd}(t) \text{ (second entry of ordered pairs)}
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)} \\
& | & \textbf{snd}(t) \text{ (second entry of ordered pairs)} \\
& | & \lambda x.t \ (\lambda\text{-abstraction})
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)} \\
& | & \textbf{snd}(t) \text{ (second entry of ordered pairs)} \\
& | & \lambda x.t \ (\lambda\text{-abstraction}) \\
& | & (t_1 \ t_2) \text{ (function application } t_1(t_2)\text{)}
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)} \\
& | & \textbf{snd}(t) \text{ (second entry of ordered pairs)} \\
& | & \lambda x.t \ (\lambda\text{-abstraction}) \\
& | & (t_1 \ t_2) \text{ (function application } t_1(t_2)) \\
& | & \textbf{let } x \Leftarrow t_1 \textbf{ in } t_2 \text{ (let-notation } t_2\,[t_1/x]))
\end{array}
$$

A Basic Functional Language

Terms

The terms t are generated from the grammar:

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{+, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)} \\
& | & \textbf{snd}(t) \text{ (second entry of ordered pairs)} \\
& | & \lambda x.t \ (\lambda\text{-abstraction}) \\
& | & (t_1 \ t_2) \text{ (function application } t_1(t_2)) \\
& | & \textbf{let } x \Leftarrow t_1 \textbf{ in } t_2 \text{ (let-notation } t_2\,[t_1/x]) \\
& | & \textbf{rec} y.(\lambda x.t) \text{ (recursion)}
\end{array}
$$

A Basic Functional Language

Example

- x : **int**
- y : **int** $\rightarrow$ **int**

A Basic Functional Language

Example

- x : **int**
- y : **int** $\to$ **int**

"Legal" Terms:

- $\lambda x.(x + 1)$
- $(\lambda x.(x + 1), 2)$
- $(\lambda x.(x + 1)\ 2)$
- **rec** $y.(\lambda x.$**if** x **then** 1 **else** $x \times (y\ (x - 1)))$

A Basic Functional Language

Example

- x : **int**

A Basic Functional Language

Example

- x : **int**

"Illegal" Terms

- $(\lambda x.x) + 1$
- $(\lambda x.x) + (\lambda x.(x + 1))$

Typing Rules

Variables

$$\frac{}{x : \tau} \quad (\mathbf{type}(x) = \tau)$$

Typing Rules

Arithmetic Operations

$$\frac{}{n : \textbf{int}}\ (n \in \mathbb{Z}) \qquad \frac{t_1 : \textbf{int},\ t_2 : \textbf{int}}{t_1\ \mathrm{op}\ t_2 : \textbf{int}}\ (\mathrm{op} \in \{+, -, \times\})$$

Typing Rules

Conditional Branch

$$\frac{t_0 : \textbf{int},\ t_1 : \tau,\ t_2 : \tau}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 : \tau}$$

Typing Rules

Products

$$\frac{t_1 : \tau_1,\ t_2 : \tau_2}{(t_1, t_2) : \tau_1 * \tau_2} \qquad \frac{t : \tau_1 * \tau_2}{\mathbf{fst}(t) : \tau_1} \qquad \frac{t : \tau_1 * \tau_2}{\mathbf{snd}(t) : \tau_2}$$

Typing Rules

Functions

$$\frac{x : \tau',\ t : \tau}{\lambda x.t : \tau' \to \tau} \qquad \frac{t_1 : \tau' \to \tau,\ t_2 : \tau'}{(t_1\ t_2) : \tau}$$

Typing Rules

"Let" Notation

$$\frac{x : \tau_1,\ t_1 : \tau_1,\ t_2 : \tau_2}{\textbf{let}\ x \Leftarrow t_1\ \textbf{in}\ t_2 : \tau_2}$$

Typing Rules

Recursion

$$\frac{y : \tau,\ \lambda x.t : \tau}{\mathbf{rec}y.(\lambda x.t) : \tau}$$

Typing Rules

Typable Terms

- A term t is typable if $t : \tau$ for some type τ.

Typing Rules

Typable Terms

- A term t is typable if $t : \tau$ for some type τ.
- A term t is uniquely typable if $t : \tau$ for some unique type τ. (i.e., $t : \tau_1$ and $t : \tau_2$ implies $\tau_1 = \tau_2$)

Typing Rules

Typable Terms

- A term t is typable if $t : \tau$ for some type τ.
- A term t is uniquely typable if $t : \tau$ for some unique type τ. (i.e., $t : \tau_1$ and $t : \tau_2$ implies $\tau_1 = \tau_2$)

Exercise

Every typable term is uniquely typable.

Free Variables $FV(-)$

Definition through Well-Founded Recursion

Free Variables $FV(-)$

Definition through Well-Founded Recursion

- $FV(n) := \emptyset$;
- $FV(x) := \{x\}$;
- $FV(t_1 \ \mathrm{op} \ t_2) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2) := FV(t_0) \cup FV(t_1) \cup FV(t_2)$;
- $FV((t_1,\ t_2)) = FV((t_1\ t_2)) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{fst}(t)) = FV(\textbf{snd}(t)) := FV(t)$;

Free Variables $FV(-)$

Definition through Well-Founded Recursion

- $FV(n) := \emptyset$;
- $FV(x) := \{x\}$;
- $FV(t_1 \ \mathrm{op} \ t_2) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2) := FV(t_0) \cup FV(t_1) \cup FV(t_2)$;
- $FV((t_1,\ t_2)) = FV((t_1\ t_2)) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{fst}(t)) = FV(\textbf{snd}(t)) := FV(t)$;
- $FV(\lambda x.t) := FV(t) \setminus \{x\}$;

Free Variables $FV(-)$

Definition through Well-Founded Recursion

- $FV(n) := \emptyset$;
- $FV(x) := \{x\}$;
- $FV(t_1 \ \mathrm{op} \ t_2) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2) := FV(t_0) \cup FV(t_1) \cup FV(t_2)$;
- $FV((t_1, \ t_2)) = FV((t_1 \ t_2)) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{fst}(t)) = FV(\textbf{snd}(t)) := FV(t)$;
- $FV(\lambda x.t) := FV(t) \setminus \{x\}$;
- $FV(\textbf{let } x \Leftarrow t_1 \textbf{ in } t_2) := FV(t_1) \cup (FV(t_2) \setminus \{x\})$;

Free Variables $FV(-)$

Definition through Well-Founded Recursion

- $FV(n) := \emptyset$;
- $FV(x) := \{x\}$;
- $FV(t_1 \ \mathrm{op} \ t_2) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2) := FV(t_0) \cup FV(t_1) \cup FV(t_2)$;
- $FV((t_1,\ t_2)) = FV((t_1\ t_2)) := FV(t_1) \cup FV(t_2)$;
- $FV(\textbf{fst}(t)) = FV(\textbf{snd}(t)) := FV(t)$;
- $FV(\lambda x.t) := FV(t) \setminus \{x\}$;
- $FV(\textbf{let } x \Leftarrow t_1 \textbf{ in } t_2) := FV(t_1) \cup (FV(t_2) \setminus \{x\})$;
- $FV(\textbf{rec}y.(\lambda x.t)) := FV(\lambda x.t) \setminus \{y\}$.

Free Variables $FV(-)$

Closed Terms

A term t is closed if $FV(t) = \emptyset$.

Free Variables $FV(-)$

Substitution

- t: a term
- s: a closed term
- x: a free variable in t

Free Variables $FV(-)$

Substitution

- t: a term
- s: a closed term
- x: a free variable in t

Then we have

- $t\,[s/x]$: the term obtained from substituting all free occurrences of x by s in t
- $t[s_1/x_1, \ldots, s_k/x_k]$: the term obtained from substituting all free occurrences of x_i by closed terms s_i $(1 \leq i \leq k)$ in t

Free Variables $FV(-)$

Example

- $x : \mathbf{int}$
- $t = \mathbf{let}\ x \Leftarrow x\ \mathbf{in}\ (x + 1)$
- $t\,[4/x] = \mathbf{let}\ x \Leftarrow 4\ \mathbf{in}\ (x + 1)$

Eager Operational Semantics

Textbook, Page 186 – 188

Eager Operational Semantics

Ordered Pairs

- $t = (3 + 1, (\lambda x.(x + 1)\ 4))$

Eager Operational Semantics

Ordered Pairs

- $t = (3 + 1, (\lambda x.(x + 1)\ 4))$

How can we evaluate **fst**(t) eagerly:

Eager Operational Semantics

Ordered Pairs

- $t = (3 + 1, (\lambda x.(x + 1)\ 4))$

How can we evaluate **fst**(t) eagerly:

- first we evaluate both $3 + 1$ and $(\lambda x.(x + 1)\ 4)$;

Eager Operational Semantics

Ordered Pairs

- $t = (3 + 1, (\lambda x.(x + 1)\ 4))$

How can we evaluate **fst**(t) eagerly:

- first we evaluate both $3 + 1$ and $(\lambda x.(x + 1)\ 4)$;
- then the final result is the evaluation from $3 + 1$.

Eager Operational Semantics

Function Application

- $t_1 = \lambda x.1$;
- $t_2 = (\mathbf{rec} y.(\lambda x.(y\ x))\ 4)$;
- $t = (t_1\ t_2)$

Eager Operational Semantics

Function Application

- $t_1 = \lambda x.1$;
- $t_2 = (\mathbf{rec} y.(\lambda x.(y\ x))\ 4)$;
- $t = (t_1\ t_2)$

How can we evaluate t eagerly:

Eager Operational Semantics

Function Application

- $t_1 = \lambda x.1$;
- $t_2 = (\mathbf{rec} y.(\lambda x.(y\ x))\ 4)$;
- $t = (t_1\ t_2)$

How can we evaluate t eagerly:

- first we evaluate both t_1 and t_2;

Eager Operational Semantics

Function Application

- $t_1 = \lambda x.1$;
- $t_2 = (\mathbf{rec} y.(\lambda x.(y\ x))\ 4)$;
- $t = (t_1\ t_2)$

How can we evaluate t eagerly:

- first we evaluate both t_1 and t_2;
- then the final result is the function application.

Eager Operational Semantics

Canonical Forms (Values)

- τ: a type

The set $C^{\mathfrak{e}}_{\tau}$ of canonical forms of type τ is a subset of terms recursively defined as follows:

- $C^{\mathfrak{e}}_{\mathbf{int}} := \mathbb{Z}$;
- $C^{\mathfrak{e}}_{\tau_1 * \tau_2} := C^{\mathfrak{e}}_{\tau_1} \times C^{\mathfrak{e}}_{\tau_2}$;
- $C^{\mathfrak{e}}_{\tau_1 \to \tau_2} := \{\lambda x.t \mid \lambda x.t : \tau_1 \to \tau_2 \text{ is closed.}\}$

Eager Operational Semantics

Canonical Forms (Values)

- τ: a type

The set $C^{\mathfrak{e}}_{\tau}$ of canonical forms of type τ is a subset of terms recursively defined as follows:

- $C^{\mathfrak{e}}_{\mathbf{int}} := \mathbb{Z}$;
- $C^{\mathfrak{e}}_{\tau_1 * \tau_2} := C^{\mathfrak{e}}_{\tau_1} \times C^{\mathfrak{e}}_{\tau_2}$;
- $C^{\mathfrak{e}}_{\tau_1 \to \tau_2} := \{\lambda x.t \mid \lambda x.t : \tau_1 \to \tau_2 \text{ is closed.}\}$

Exercise

Prove that for any type τ and term $t \in C^{\mathfrak{e}}_{\tau}$, t is closed.

Eager Operational Semantics

The Evaluation Relation

- t: a typable closed term with type τ
- c: a canonical term in $C_\tau^{\mathfrak{e}}$

Then

- $t \to^{\mathfrak{e}} c$: t evaluates to c in eager operational semantics

Evaluation Rules

Canonical Forms

$$\frac{}{c \to^{e} c} \quad (c \in C_{\tau}^{e})$$

Evaluation Rules

Arithmetic Operations

$$\frac{t_1 \to^{\mathfrak{e}} n_1,\ t_2 \to^{\mathfrak{e}} n_2}{(t_1\ \mathrm{op}\ t_2) \to^{\mathfrak{e}} n_1\ \mathrm{op}\ n_2}\quad \mathrm{op} \in \{+, -, \times\}$$

Evaluation Rules

Conditional Branch

$$\frac{t_0 \to^{\mathfrak{e}} 0,\ t_1 \to^{\mathfrak{e}} c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to^{\mathfrak{e}} c_1}$$

$$\frac{t_0 \to^{\mathfrak{e}} n,\ t_2 \to^{\mathfrak{e}} c_2}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to^{\mathfrak{e}} c_2}\ (n \neq 0)$$

Evaluation Rules

Product

$$\frac{t_1 \to^{e} c_1,\ \ t_2 \to^{e} c_2}{(t_1, t_2) \to^{e} (c_1, c_2)}$$

$$\frac{t \to^{e} (c_1, c_2)}{\mathbf{fst}(t) \to^{e} c_1} \qquad \frac{t \to^{e} (c_1, c_2)}{\mathbf{snd}(t) \to^{e} c_2}$$

Evaluation Rules

Function Application

$$\frac{t_1 \to^{\mathfrak{e}} \lambda x.t_1',\ t_2 \to^{\mathfrak{e}} c_2,\ t_1'\,[c_2/x] \to^{\mathfrak{e}} c}{(t_1\ t_2) \to^{\mathfrak{e}} c}$$

Evaluation Rules

"Let" Expression

$$\frac{t_1 \to^{\mathfrak{e}} c_1,\ t_2\left[c_1/x\right] \to^{\mathfrak{e}} c_2}{\textbf{let}\ x \Leftarrow t_1\ \textbf{in}\ t_2 \to^{\mathfrak{e}} c_2}$$

Evaluation Rules

Recursion

$$\frac{}{\mathbf{rec}y.(\lambda x.t) \to^{\mathfrak{e}} \lambda x.(t\,[\mathbf{rec}y.(\lambda x.t)/y])}$$

Eager Operational Semantics

Proposition

- t: a closed term with type τ
- c, c': canonical forms

Then we have that:

- if $t \to^{\mathfrak{e}} c$ and $t \to^{\mathfrak{e}} c'$ then $c = c'$;
- if $t \to^{\mathfrak{e}} c$ then $c : \tau$.

Eager Operational Semantics

Proposition

- t: a closed term with type τ
- c, c': canonical forms

Then we have that:

- if $t \to^{\mathfrak{e}} c$ and $t \to^{\mathfrak{e}} c'$ then $c = c'$;
- if $t \to^{\mathfrak{e}} c$ then $c : \tau$.

Proof

By a simple rule induction.

Eager Operational Semantics

Homework

- x : **int**
- y : **int** $\rightarrow$ **int**
- $fact := \textbf{rec}y.(\lambda x.(\textbf{if } x \textbf{ then } 1 \textbf{ else } x \times (y\ (x-1))))$
- Find the type of *fact* through the typing rules.
- Evaluate $(fact\ 2)$ under the eager operational semantics.

Lazy Operational Semantics

Textbook, Page 200 – 202

Lazy Operational Semantics

- eager semantics: evaluate every sub-term
- lazy semantics: evaluate only necessary sub-terms

Lazy Operational Semantics

Terms

$$
\begin{array}{rcl}
t & ::= & x \text{ (variables)} \\
& | & n \text{ (integer constants)} \\
& | & t_1 \bowtie t_2 \ (\bowtie \in \{=, -, \times\}) \text{ (arithmetic operations)} \\
& | & \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \\
& | & (t_1, t_2) \text{ (ordered pairs)} \\
& | & \textbf{fst}(t) \text{ (first entry of ordered pairs)} \\
& | & \textbf{snd}(t) \text{ (second entry of ordered pairs)} \\
& | & \lambda x.t \ (\lambda\text{-abstraction}) \\
& | & (t_1 \ t_2) \text{ (function application } t_1(t_2)) \\
& | & \textbf{let } x \Leftarrow t_1 \textbf{ in } t_2 \text{ (let-notation } t_2\,[t_1/x]) \\
& | & \textbf{rec}y.t \text{ (recursion)}
\end{array}
$$

Typing Rules

Recursion

$$\frac{y : \tau,\ t : \tau}{\mathbf{rec} y.t : \tau}$$

Typing Rules

Typable Terms

- A term t is *typable* if $t : \tau$ for some type τ.
- A term t is *uniquely typable* if $t : \tau$ for some unique type τ. (i.e., $t : \tau_1$ and $t : \tau_2$ implies $\tau_1 = \tau_2$)

Typing Rules

Typable Terms

- A term t is *typable* if $t : \tau$ for some type τ.
- A term t is *uniquely typable* if $t : \tau$ for some unique type τ. (i.e., $t : \tau_1$ and $t : \tau_2$ implies $\tau_1 = \tau_2$)

A Simple Exercise

Every typable term is uniquely typable.

Terms

Free Variables

- $FV(\mathbf{rec}y.t) := FV(t) \setminus \{y\}$.
- A term t is *closed* if $FV(t) = \emptyset$.

Lazy Operational Semantics

Canonical Forms

- τ: a type

The set $C_\tau^{\mathfrak{l}}$ is recursively defined as follows:

- $C_{\mathbf{int}}^{\mathfrak{l}} := \mathbb{Z}$;
- $C_{\tau_1 * \tau_2}^{\mathfrak{l}} := \{(t_1, t_2) \mid t_1 : \tau_1, t_2 : \tau_2 \text{ and } t_1, t_2 \text{ are closed.}\}$;
- $C_{\tau_1 \to \tau_2}^{\mathfrak{l}} := \{\lambda x.t \mid \lambda x.t : \tau_1 \to \tau_2 \text{ is closed.}\}$

Lazy Operational Semantics

The Evaluation Relation

- t: a closed term with type τ
- c: a canonical term in $C_\tau^{\mathfrak{C}}$

Then

- $t \to^{\mathfrak{l}} c$: t evaluates to c in lazy operational semantics

Lazy Operational Semantics

Evaluation Rules

- canonical terms:

$$\frac{}{c \to^{!} c} \quad (c \in C_{\tau}^{!})$$

Lazy Operational Semantics

Evaluation Rules

- arithmetic operations:

$$\frac{t_1 \to^{\mathfrak{l}} n_1, \;\; t_2 \to^{\mathfrak{l}} n_2}{t_1 \;\mathrm{op}\; t_2 \;\to^{\mathfrak{l}}\; n_1 \;\mathrm{op}\; n_2} \quad \mathrm{op} \in \{+, -, \times\}$$

Lazy Operational Semantics

Evaluation Rules

- conditional branch:

$$\frac{t_0 \to^! 0 \ , t_1 \to^! c_1}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to^! c_1}$$

$$\frac{t_0 \to^! n \ , t_2 \to^! c_2}{\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \to^! c_2} \ (n \neq 0)$$

Lazy Operational Semantics

Evaluation Rules

- Product:

$$\frac{t \to^! (t_1, t_2),\ \ t_1 \to^! c_1}{\mathbf{fst}(t) \to^! c_1} \qquad \frac{t \to^! (t_1, t_2),\ \ t_2 \to^! c_2}{\mathbf{snd}(t) \to^! c_2}$$

Lazy Operational Semantics

Evaluation Rules

- Function Application:

$$\frac{t_1 \ \to^{\iota} \ \lambda x.t_1' \ , t_1'\,[t_2/x] \to^{\iota} c}{(t_1 \ t_2) \to^{\iota} c}$$

Lazy Operational Semantics

Evaluation Rules

- "Let" Notation:

$$\frac{t_2\,[t_1/x] \to^{!} c}{\textbf{let}\ x \Leftarrow t_1\ \textbf{in}\ t_2 \to^{!} c}$$

Lazy Operational Semantics

Evaluation Rules

- Recursion:

$$\frac{t\,[\mathbf{rec}y.t/y] \to^{l} c}{\mathbf{rec}y.t \to^{l} c}$$

Lazy Operational Semantics

Proposition

- t: a closed term with type τ
- c, c': canonical terms

Then we have that:

- if $t \to^! c$ and $t \to^! c'$ then $c = c'$;
- if $t \to^! c$ then $c : \tau$.

Lazy Operational Semantics

Proposition

- t: a closed term with type τ
- c, c': canonical terms

Then we have that:

- if $t \to^! c$ and $t \to^! c'$ then $c = c'$;
- if $t \to^! c$ then $c : \tau$.

Proof.

By a simple rule induction. $\square$

Summary

Functional Programming Languages

- types and terms
- eager operational semantics
- lazy operational semantics

Homework

Problem

- x : **int**
- y : **int** $\to$ **int**
- $fact := \textbf{rec}y.(\lambda x.(\textbf{if } x \textbf{ then } 1 \textbf{ else } x \times (y\ (x-1))))$
- Find the type of *fact* through the typing rules.
- Evaluate ($fact\ 2$) under the eager operational semantics.