Languages with Higher Types

Hongfei Fu

John Hopcroft Center for Computer Science
Department of Computer Science
Shanghai Jiao Tong University

Today's Topic

Typed Languages

- a functional programming language
- (eager) operational semantics
- (eager) denotational semantics

A Basic Functional Language

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** $\rightarrow$ **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

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

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 $FV(-)$

Well-Founded Recursion

Free Variables $FV(-)$

Well-Founded Recursion

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

Free Variables $FV(-)$

Well-Founded Recursion

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

Free Variables $FV(-)$

Well-Founded Recursion

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

Free Variables $FV(-)$

Well-Founded Recursion

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

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

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):

Eager Operational Semantics

Ordered Pairs

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

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

- 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):

- 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:

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:

- 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:

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

Eager Operational Semantics

Canonical Terms (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 Terms (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 $C^{\mathfrak{e}}_{\tau}$

Then

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

Evaluation Rules

Canonical Forms

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

Evaluation Rules

Arithmetic Operations

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

Evaluation Rules

Conditional Branch

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

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

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

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

Evaluation Rules

Recursion

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

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

Exercise

- $\mathit{fact} := \mathbf{rec} y.(\lambda x.(\mathbf{if}\ x\ \mathbf{then}\ 1\ \mathbf{else}\ x \times (y\ (x - 1))))$
- How can one evaluate $(\mathit{fact}\ 2)$?

Eager Denotational Semantics

Eager Denotational Semantics

An Overview

- terms as functions from environments to values
- cpo's and continuous functions as mathematical backbone

Eager Denotational Semantics

Values

- τ: a type

The cpo V^{e}_{τ} of values associated with the type τ is recursively defined as follows:

- $V^{e}_{\textbf{int}} := \mathbb{Z}$ (discrete cpo);
- $V^{e}_{\tau_1 * \tau_2} := V^{e}_{\tau_1} \times V^{e}_{\tau_2}$ (product cpo);
- $V^{e}_{\tau_1 \to \tau_2} := [V^{e}_{\tau_1} \to (V^{e}_{\tau_2})_{\perp}]$ (function space);

Eager Denotational Semantics

Values

- τ: a type

The cpo V^e_τ of values associated with the type τ is recursively defined as follows:

- $V^e_{\mathbf{int}} := \mathbb{Z}$ (discrete cpo);
- $V^e_{\tau_1 * \tau_2} := V^e_{\tau_1} \times V^e_{\tau_2}$ (product cpo);
- $V^e_{\tau_1 \to \tau_2} := [V^e_{\tau_1} \to (V^e_{\tau_2})_\perp]$ (function space);

Question

Why do we incorporate $\perp$ in the last definition?

Eager Denotational Semantics

Environments

- **Var**: the set of variables

An *environment* ρ is a function

$$\rho : \mathbf{Var} \to \bigcup \{ V_\tau^{\mathfrak{e}} \mid \tau \text{ a type} \}$$

such that

$$\forall x \in \mathbf{Var}.(x : \tau \Rightarrow \rho(x) \in V_\tau^{\mathfrak{e}}) \ .$$

Eager Denotational Semantics

Environments

- **Var**: the set of variables

An *environment* ρ is a function

$$\rho : \mathbf{Var} \to \bigcup \{ V_\tau^{\mathfrak{e}} \mid \tau \text{ a type} \}$$

such that

$$\forall x \in \mathbf{Var}.(x : \tau \Rightarrow \rho(x) \in V_\tau^{\mathfrak{e}}) \ .$$

We denote by $\mathbf{Env}^{\mathfrak{e}}$ the set of environments under eager semantics.

Eager Denotational Semantics

Intuition

- t: a typable term with type τ
- $[\![t]\!]^{\mathfrak{e}} : \mathbf{Env}^{\mathfrak{e}} \to (V_{\tau}^{\mathfrak{e}})_{\perp}$: the denotational semantics of t

Eager Denotational Semantics

Well-Founded Recursion

- $[\![x]\!]^{\mathfrak{e}} := \lambda\rho.\lfloor\rho(x)\rfloor$
- $[\![n]\!]^{\mathfrak{e}} := \lambda\rho.\lfloor n\rfloor$

Eager Denotational Semantics

Well-Founded Recursion

- $[\![t_1 \ \mathbf{op} \ t_2]\!]^{\mathfrak{e}} := \lambda\rho.([\![t_1]\!]^{\mathfrak{e}}(\rho) \ \mathbf{op}_{\perp} \ [\![t_2]\!]^{\mathfrak{e}}(\rho))$

Eager Denotational Semantics

Well-Founded Recursion

- $[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]^{\mathfrak{e}} := \lambda\rho.\mathit{Cond}([\![t_0]\!]^{\mathfrak{e}}(\rho), [\![t_1]\!]^{\mathfrak{e}}(\rho), [\![t_2]\!]^{\mathfrak{e}}(\rho))$
- Conditional (Chapter 9.3):

$$\mathit{Cond}(z_0, z_1, z_2) := \begin{cases} z_1 & \text{if } z_0 = \lfloor 0 \rfloor \\ z_2 & \text{if } z_0 = \lfloor n \rfloor \text{ and } n \neq 0 \\ \bot & \text{if } z_0 = \bot \end{cases}$$

Eager Denotational Semantics

Well-Founded Recursion

- $[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]^{\mathfrak{e}} := \lambda\rho.\mathit{Cond}([\![t_0]\!]^{\mathfrak{e}}(\rho), [\![t_1]\!]^{\mathfrak{e}}(\rho), [\![t_2]\!]^{\mathfrak{e}}(\rho))$
- Conditional (Chapter 9.3):

$$\mathit{Cond}(z_0, z_1, z_2) := \begin{cases} z_1 & \text{if } z_0 = \lfloor 0 \rfloor \\ z_2 & \text{if } z_0 = \lfloor n \rfloor \text{ and } n \neq 0 \\ \bot & \text{if } z_0 = \bot \end{cases}$$

-

$$[\![\textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2]\!]^{\mathfrak{e}} := \lambda\rho.(\mathit{let}\ n \Leftarrow [\![t_0]\!]^{\mathfrak{e}}(\rho).[[\![t_1]\!]^{\mathfrak{e}}(\rho), [\![t_2]\!]^{\mathfrak{e}}(\rho)](n))$$

Eager Denotational Semantics

Well-Founded Recursion

- $[\![(t_1, t_2)]\!]^{\mathfrak{e}} := \lambda\rho.\mathit{let}\ v_1 \Leftarrow [\![t_1]\!]^{\mathfrak{e}}(\rho), v_2 \Leftarrow [\![t_2]\!]^{\mathfrak{e}}(\rho).\lfloor(v_1, v_2)\rfloor$;
- $[\![\mathbf{fst}(t)]\!]^{\mathfrak{e}} := \lambda\rho.\mathit{let}\ v \Leftarrow [\![t]\!]^{\mathfrak{e}}(\rho).\lfloor\pi_1(v)\rfloor$
- $[\![\mathbf{snd}(t)]\!]^{\mathfrak{e}} := \lambda\rho.\mathit{let}\ v \Leftarrow [\![t]\!]^{\mathfrak{e}}(\rho).\lfloor\pi_2(v)\rfloor$

Eager Denotational Semantics

Function Update (Chapter 9.3)

- $in_1 : \{x\} \to \mathbf{Var}$: $x \mapsto x$
- $in_2 : \mathbf{Var} \setminus \{x\} \to \mathbf{Var}$: $y \mapsto y, y \neq x$

$$\rho\,[v/x] = \lambda y.\mathbf{case}\ y\ \mathbf{of}\ in_1(y_1).v \mid in_2(y_2).\rho(y_2)$$

Eager Denotational Semantics

Well-Founded Recursion

- $[\![\lambda x.t]\!]^{\mathfrak{e}} := \lambda\rho.\lfloor\lambda v \in V^{\mathfrak{e}}_{\tau_1}.[\![t]\!]^{\mathfrak{e}}(\rho\,[v/x])\rfloor$ for $\lambda x.t : \tau_1 \to \tau_2$
- $[\![(t_1\ t_2)]\!]^{\mathfrak{e}} := \lambda\rho.\mathit{let}\ F \Leftarrow [\![t_1]\!]^{\mathfrak{e}}(\rho), v \Leftarrow [\![t_2]\!]^{\mathfrak{e}}(\rho).F(v)$

Eager Denotational Semantics

Well-Founded Recursion

- $[\![\textbf{let } x \Leftarrow t_1 \textbf{ in } t_2]\!]^{\mathfrak{e}} := \lambda\rho.let\ v \Leftarrow [\![t_1]\!]^{\mathfrak{e}}(\rho).[\![t_2]\!]^{\mathfrak{e}}(\rho\,[v/x])$
- $[\![\textbf{rec}y.(\lambda x.t)]\!]^{\mathfrak{e}} := \lambda\rho.\lfloor\mu F.(\lambda v.[\![t]\!]^{\mathfrak{e}}(\rho[v/x, F/y]))\rfloor$

Eager Denotational Semantics

Lemma

- t: a typable term
- ρ, ρ': two environments such that for all $x \in FV(t)$, we have $\rho(x) = \rho'(x)$

Then we have $[\![t]\!]^{\mathfrak{e}}(\rho) = [\![t]\!]^{\mathfrak{e}}(\rho')$.

Eager Denotational Semantics

Lemma

- t: a typable term
- ρ, ρ': two environments such that for all $x \in FV(t)$, we have $\rho(x) = \rho'(x)$

Then we have $[\![t]\!]^{\mathfrak{e}}(\rho) = [\![t]\!]^{\mathfrak{e}}(\rho')$.

Proof

A simple structural induction on t.

Eager Denotational Semantics

Substitution Lemma

- s: a typable closed term with type τ
- x: a variable with type τ
- t: a typable term with type τ'

If $[\![s]\!]^{\mathfrak{e}}(\rho) = \lfloor v \rfloor$, then we have that:

- $t\,[s/x] : \tau'$;
- $[\![t\,[s/x]]\!]^{\mathfrak{e}}(\rho) = [\![t]\!]^{\mathfrak{e}}(\rho\,[v/x])$.

Eager Denotational Semantics

Substitution Lemma

- s: a typable closed term with type τ
- x: a variable with type τ
- t: a typable term with type τ'

If $[\![s]\!]^{\mathfrak{e}}(\rho) = \lfloor v \rfloor$, then we have that:

- $t\,[s/x] : \tau'$;
- $[\![t\,[s/x]]\!]^{\mathfrak{e}}(\rho) = [\![t]\!]^{\mathfrak{e}}(\rho\,[v/x])$.

Proof

By structural induction on t.

Eager Denotational Semantics

Lemma

- If $t : \tau$, then for all ρ we have $[\![t]\!]^{\mathfrak{e}}(\rho) \in (V_\tau^{\mathfrak{e}})_\perp$.
- If $c \in C_\tau^{\mathfrak{e}}$, then for all ρ we have $[\![c]\!]^{\mathfrak{e}}(\rho) \neq \perp$ (the bottom element of $(V_\tau^{\mathfrak{e}})_\perp$).

Eager Denotational Semantics

Lemma

- If $t : \tau$, then for all ρ we have $[\![t]\!]^{\mathfrak{e}}(\rho) \in (V_\tau^{\mathfrak{e}})_\perp$.
- If $c \in C_\tau^{\mathfrak{e}}$, then for all ρ we have $[\![c]\!]^{\mathfrak{e}}(\rho) \neq \perp$ (the bottom element of $(V_\tau^{\mathfrak{e}})_\perp$).

Proof

By structural induction.

Summary

A Functional Programming Language

- types and terms
- eager operational semantics
- eager denotational semantics

Questions?