CS 4110

Programming Languages & Logics

Lecture 15
Encodings

Review: λ-calculus

Syntax

$$
\begin{array}{rcl}
e & ::= & x \mid e_1\, e_2 \mid \lambda x.\, e \\
v & ::= & \lambda x.\, e
\end{array}
$$

Semantics

$$
\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e \to e'}{v\, e \to v\, e'}
$$

$$
\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta
$$

Rewind: Currying

This is just a function that returns a function:

$$\text{ADD} \triangleq \lambda x.\, \lambda y.\, x + y$$

$$\text{ADD } 38 \rightarrow \lambda y.\, 38 + y$$

$$\text{ADD } 38\ 4 = (\text{ADD } 38)\ 4 \rightarrow 42$$

Informally, you can think of it as a *curried* function that takes two arguments, one after the other.

But that's just a way to get intuition. The λ-calculus only has one-argument functions.

Review: Call-by-Value

Here are the syntax and CBV semantics of λ-calculus:

$$
\begin{aligned}
e &::= x \mid \lambda x.\, e \mid e_1\, e_2 \\
v &::= \lambda x.\, e
\end{aligned}
$$

$$
\frac{e_1 \to e_1'}{e_1\, e_2 \to e_1'\, e_2} \qquad \frac{e \to e'}{v\, e \to v\, e'}
$$

$$
\frac{}{(\lambda x.\, e)\, v \to e\{v/x\}}\ \beta
$$

There are two kinds of rules: *congruence rules* that specify evaluation order and *computation rules* that specify the "interesting" reductions.

Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

An evaluation context E is an expression with a "hole" in it: a single occurrence of the special symbol $[\cdot]$ in place of a subexpression.

$$E ::= [\cdot] \mid E\, e \mid v\, E$$

Evaluation Contexts

Evaluation contexts let us separate out these two kinds of rules.

An evaluation context E is an expression with a "hole" in it: a single occurrence of the special symbol $[\cdot]$ in place of a subexpression.

$$E ::= [\cdot] \mid E\ e \mid v\ E$$

We write $E[e]$ to mean the evaluation context E where the hole has been replaced with the expression e.

Examples

$$E_1 = [\cdot]\,(\lambda x.\, x)$$
$$E_1[\lambda y.\, y\, y] = (\lambda y.\, y\, y)\ \lambda x.\, x$$

Examples

$$
\begin{aligned}
E_1 &= [\cdot]\,(\lambda x.\, x) \\
E_1[\lambda y.\, y\, y] &= (\lambda y.\, y\, y)\, \lambda x.\, x
\end{aligned}
$$

$$
\begin{aligned}
E_2 &= (\lambda z.\, z\, z)\,[\cdot] \\
E_2[\lambda x.\, \lambda y.\, x] &= (\lambda z.\, z\, z)\,(\lambda x.\, \lambda y.\, x)
\end{aligned}
$$

Examples

$$
\begin{aligned}
E_1 &= [\cdot]\,(\lambda x.\, x) \\
E_1[\lambda y.\, y\, y] &= (\lambda y.\, y\, y)\ \lambda x.\, x
\end{aligned}
$$

$$
\begin{aligned}
E_2 &= (\lambda z.\, z\, z)\,[\cdot] \\
E_2[\lambda x.\, \lambda y.\, x] &= (\lambda z.\, z\, z)\,(\lambda x.\, \lambda y.\, x)
\end{aligned}
$$

$$
\begin{aligned}
E_3 &= ([\cdot]\ \lambda x.\, x\, x)\,((\lambda y.\, y)\,(\lambda y.\, y)) \\
E_3[\lambda f.\, \lambda g.\, f\, g] &= ((\lambda f.\, \lambda g.\, f\, g)\ \lambda x.\, x\, x)\,((\lambda y.\, y)\,(\lambda y.\, y))
\end{aligned}
$$

CBV With Evaluation Contexts

With evaluation contexts, we can define the evaluation semantics for the CBV λ-calculus with just two rules: one for evaluation contexts, and one for β-reduction.

CBV With Evaluation Contexts

With evaluation contexts, we can define the evaluation semantics for the CBV λ-calculus with just two rules: one for evaluation contexts, and one for β-reduction.

With this syntax:

$$E ::= [\cdot] \mid E\, e \mid v\, E$$

The small-step rules are:

$$\frac{e \rightarrow e'}{E[e] \rightarrow E[e']}$$

$$\frac{}{(\lambda x.\, e)\, v \rightarrow e\{v/x\}}\ \beta$$

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

$$E ::= [\cdot] \mid E\,e$$

CBN With Evaluation Contexts

We can also define the semantics of CBN λ-calculus with evaluation contexts.

For call-by-name, the syntax for evaluation contexts is different:

$$E ::= [\cdot] \mid E\ e$$

But the small-step rules are the same:

$$\frac{e \to e'}{E[e] \to E[e']}$$

$$\frac{}{(\lambda x.\, e)\ e' \to e\{e'/x\}}\ \beta$$

Encodings

The pure λ-calculus contains only functions as values. It is not exactly easy to write large or interesting programs in the pure λ-calculus. We can however encode objects, such as booleans, and integers.

Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$
\begin{aligned}
\text{AND TRUE FALSE} &= \text{FALSE} \\
\text{NOT FALSE} &= \text{TRUE} \\
\text{IF TRUE } e_1\ e_2 &= e_1 \\
\text{IF FALSE } e_1\ e_2 &= e_2
\end{aligned}
$$

Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned} \text{AND TRUE FALSE} &= \text{FALSE} \\ \text{NOT FALSE} &= \text{TRUE} \\ \text{IF TRUE } e_1\ e_2 &= e_1 \\ \text{IF FALSE } e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\begin{aligned} \text{TRUE} &\triangleq \\ \text{FALSE} &\triangleq \end{aligned}$$

Booleans

We need to define functions TRUE, FALSE, AND, NOT, IF, and other operators that behave as follows:

$$\begin{aligned} \text{AND TRUE FALSE} &= \text{FALSE} \\ \text{NOT FALSE} &= \text{TRUE} \\ \text{IF TRUE } e_1\ e_2 &= e_1 \\ \text{IF FALSE } e_1\ e_2 &= e_2 \end{aligned}$$

Let's start by defining TRUE and FALSE:

$$\begin{aligned} \text{TRUE} &\triangleq \lambda x.\, \lambda y.\, x \\ \text{FALSE} &\triangleq \lambda x.\, \lambda y.\, y \end{aligned}$$

Booleans

We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

Booleans

We want the function IF to behave like

$$\lambda b.\ \lambda t.\ \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

Booleans

We want the function IF to behave like

$$\lambda b.\ \lambda t.\ \lambda f.\ \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\ \lambda t.\ \lambda f.\ b\ t\ f$$

We can also write the standard Boolean operators.

$$\text{NOT} \triangleq$$

$$\text{AND} \triangleq$$

$$\text{OR} \triangleq$$

Booleans

We want the function IF to behave like

$$\lambda b.\, \lambda t.\, \lambda f.\, \text{if } b \text{ is our term TRUE then } t\text{, otherwise } f$$

We can rely on the way we defined TRUE and FALSE:

$$\text{IF} \triangleq \lambda b.\, \lambda t.\, \lambda f.\, b\; t\; f$$

We can also write the standard Boolean operators.

$$\text{NOT} \triangleq \lambda b.\, b\; \text{FALSE TRUE}$$

$$\text{AND} \triangleq \lambda b_1.\, \lambda b_2.\, b_1\; b_2\; \text{FALSE}$$

$$\text{OR} \triangleq \lambda b_1.\, \lambda b_2.\, b_1\; \text{TRUE}\; b_2$$

Church Numerals

Let's encode the natural numbers!

We'll write $\overline{n}$ for the encoding of the number n. The central function we'll need is a *successor* operation:

$$\text{SUCC}\ \overline{n} = \overline{n+1}$$

Church Numerals

Church numerals encode a number n as a function that takes f and x, and applies f to x n times.

$$\begin{aligned}
\overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\
\overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\
\overline{2} &\triangleq \lambda f.\, \lambda x.\, f\, (f\, x)
\end{aligned}$$

Church Numerals

Church numerals encode a number n as a function that takes f and x, and applies f to x n times.

$$\begin{aligned} \overline{0} &\triangleq \lambda f.\, \lambda x.\, x \\ \overline{1} &\triangleq \lambda f.\, \lambda x.\, f\, x \\ \overline{2} &\triangleq \lambda f.\, \lambda x.\, f\,(f\, x) \end{aligned}$$

We can write a successor function that "inserts" another application of f:

$$\mathrm{SUCC} \triangleq \lambda n.\, \lambda f.\, \lambda x.\, f\,(n\, f\, x)$$

Addition

Given the definition of SUCC, we can define addition. Intuitively, the natural number $n_1 + n_2$ is the result of applying the successor function n_1 times to n_2.

$$\text{PLUS} \triangleq$$

Addition

Given the definition of SUCC, we can define addition. Intuitively, the natural number $n_1 + n_2$ is the result of applying the successor function n_1 times to n_2.

$$\text{PLUS} \triangleq \lambda n_1.\, \lambda n_2.\, n_1\ \text{SUCC}\ n_2$$