Lecture 8 – Lambda Calculus

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

- FVAE – VAE with First-Class Functions
  - First-Class Functions
  - Concrete and Abstract Syntax
  - Interpreter and Natural Semantics with Closures
  - Static and Dynamic Scoping
- In this lecture, we will learn **syntactic sugar** and **lambda calculus**

Contents

Contents

No More `val`

```
/* FVAE */
val x = 1; x + 2
```

It assigns a **value** 1 to the **variable** `x`, and then evaluates the **body expression** `x` + 2 with the environment $[\mathrm{x} \mapsto 1]$.

It is same as:

```
/* FVAE */
(x => x + 2)(1)
```

It assigns a **value** (argument) 1 to the **parameter** `x`, and then evaluates the **body expression** `x` + 2 with the environment $[\mathrm{x} \mapsto 1]$.

No More `val`

In general, the following two expressions are equivalent:

$$\texttt{val}\ x\ \texttt{=}\ e_1;\ e_2 \qquad \text{is equivalent to} \qquad (\lambda x.e_2)(e_1)$$

Why?

The following inference rule for the semantics of $\texttt{val}\ x\ \texttt{=}\ e_1;\ e_2$:

$$\textsc{Val}\ \frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash \texttt{val}\ x = e_1;\ e_2 \Rightarrow v_2}$$

is equivalent to the following inference rule for the semantics of the combination $(\lambda x.e_2)(e_1)$ of a anonymous function and an application:

$$\texttt{App}\ \frac{\texttt{Fun}\ \dfrac{}{\sigma \vdash \lambda x.e_2 \Rightarrow \langle \lambda x.e_2, \sigma \rangle} \qquad \sigma \vdash e_1 \Rightarrow v_1 \qquad \sigma[x \mapsto v_1] \vdash e_2 \Rightarrow v_2}{\sigma \vdash (\lambda x.e_2)(e_1) \Rightarrow v_2}$$

FAE – Removing `val` from FVAE

Then, we can define a smaller language FAE

$$
\begin{array}{llll}
\text{Expressions} & e ::= & n & (\texttt{Num}) \\
& \mid & e \;\texttt{+}\; e & (\texttt{Add}) \\
& \mid & e \;\texttt{*}\; e & (\texttt{Mul}) \\
& \mid & x & (\texttt{Id}) \\
& \mid & \lambda x.e & (\texttt{Fun}) \\
& \mid & e(e) & (\texttt{App})
\end{array}
$$

by removing `val` from FVAE using the following equivalence:

$$\texttt{val}\; x \;\texttt{=}\; e_1\texttt{;}\; e_2 \qquad \text{is equivalent to} \qquad (\lambda x.e_2)(e_1)$$

Syntactic Sugar and Desugaring

Definition (Syntactic Sugar)

Syntactic elements that can be expressed in terms of other syntactic elements are called **syntactic sugar**.

Definition (Desugaring)

Desugaring is a translation for removing syntactic sugar.

$$\mathcal{D}[\![-]\!] : \mathbb{E} \to \mathbb{E}$$

For example, we can define the desugaring of `val` as follows:

$$\mathcal{D}[\![\texttt{val}\ x\ \texttt{=}\ e_1;\ e_2]\!] = (\lambda x.\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e]\!])$$

Is it enough? **No!** Why?

Syntactic Sugar and Desugaring

$$\mathcal{D}[\![\texttt{val}\ x\ \texttt{=}\ e_1;\ e_2]\!] = (\lambda x.\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e]\!])$$

We need to **recursively desugar** sub-expressions of the given expression even if they are not syntactic sugars.

$$\begin{array}{lcllcl}
\mathcal{D}[\![n]\!] & = & n & \mathcal{D}[\![x]\!] & = & x \\
\mathcal{D}[\![e_1\ \texttt{+}\ e_2]\!] & = & \mathcal{D}[\![e_1]\!]\ \texttt{+}\ \mathcal{D}[\![e_2]\!] & \mathcal{D}[\![\lambda x.e]\!] & = & \lambda x.\mathcal{D}[\![e]\!] \\
\mathcal{D}[\![e_1\ \texttt{*}\ e_2]\!] & = & \mathcal{D}[\![e_1]\!]\ \texttt{*}\ \mathcal{D}[\![e_2]\!] & \mathcal{D}[\![e_1(e_2)]\!] & = & \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])
\end{array}$$

For example,

$$\mathcal{D}[\![\texttt{val x = }1;\ 2\ \texttt{+}\ (\texttt{val y = }3;\ \texttt{x * y})]\!] = \lambda\texttt{x}.2\ \texttt{+}\ (\lambda\texttt{y}.\texttt{x * y})(3)(1)$$

Without desugaring rule for addition, the expression $(\texttt{val y = }3;\ \texttt{x * y})$ in the right-hand side of the addition cannot be desugared.

Syntactic Sugar and Desugaring

$$\mathcal{D}[\![\texttt{val}\ x\ \texttt{=}\ e_1;\ e_2]\!] = (\lambda x.\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e]\!])$$

$$\begin{array}{lcl@{\qquad}lcl}
\mathcal{D}[\![n]\!] & = & n & \mathcal{D}[\![x]\!] & = & x \\
\mathcal{D}[\![e_1\ \texttt{+}\ e_2]\!] & = & \mathcal{D}[\![e_1]\!]\ \texttt{+}\ \mathcal{D}[\![e_2]\!] & \mathcal{D}[\![\lambda x.e]\!] & = & \lambda x.\mathcal{D}[\![e]\!] \\
\mathcal{D}[\![e_1\ \texttt{*}\ e_2]\!] & = & \mathcal{D}[\![e_1]\!]\ \texttt{*}\ \mathcal{D}[\![e_2]\!] & \mathcal{D}[\![e_1(e_2)]\!] & = & \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])
\end{array}$$

We can also implement **desugaring** in Scala:

```scala
def desugar(expr: Expr): Expr = expr match
  case Val(x, i, b) => App(Fun(x, desugar(b)), desugar(i))
  case Num(n)       => Num(n)
  case Add(l, r)    => Add(desugar(l), desugar(r))
  case Mul(l, r)    => Mul(desugar(l), desugar(r))
  case Id(x)        => Id(x)
  case Fun(p, b)    => Fun(p, desugar(b))
  case App(f, e)    => App(desugar(f), desugar(e))
```

Then, we can desugar the example FVAE expression as follows:

```scala
val expr: Expr = Expr("val x = 1; 2 + (val y = 3; x * y)")
desugar(expr) == Expr("(x => 2 + (y => x * y)(3))(1)")
```

Syntactic Sugar and Desugaring

Most programming languages have **syntactic sugar**:

- Scala

```
for (x <- list) yield x * 2
```
≡
```
list.map(x => x * 2)
```

- C++

```
arr[i] + obj->field
```
≡
```
*(arr + i) + (*obj).field
```

- JavaScript[1]

```
x ||= y; x &&= y;
```
≡
```
x || (x = y); x && (x = y);
```

- Haskell

```
do x <- f; g x
```
≡
```
f >>= (\x -> g x)
```

- ...

[1] https://babeljs.io/repl

Contents

What is the minimal language that can express all the syntactic elements of FVAE? **Lambda calculus (LC)**!

The **lambda calculus (LC)** is a language only consisting of 1) **variables**, 2) **functions**, and 3) **applications**:

$$\begin{array}{lrcl} \text{Expressions} & e & ::= & x \\ & & | & \lambda x.e \\ & & | & e(e) \end{array}$$

We already showed that the **variable definition** can be desugared to a combination of a **function definition** and an **application**:

$$\mathcal{D}[\![\texttt{val}\ x\ \texttt{=}\ e_1;\ e_2]\!] = (\lambda x.\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e]\!])$$

Then, how can we desugar other syntactic elements of FVAE?

Let's learn the **Church encodings**!

Church Encodings – Church Numerals

Church encodings are ways to encode **data** and **operations** in the **lambda calculus (LC)**.

For example, **Church numerals** are a way to encode **natural numbers** in the **lambda calculus (LC)**.

The key idea is to encode a **natural number** n as a **function** that takes another function f and an argument x and applies f to x n times:

$$
\begin{array}{rcl}
\mathcal{D}[\![0]\!] & = & \lambda f.\lambda x.x \\
\mathcal{D}[\![1]\!] & = & \lambda f.\lambda x.f(x) \\
\mathcal{D}[\![2]\!] & = & \lambda f.\lambda x.f(f(x)) \\
\mathcal{D}[\![3]\!] & = & \lambda f.\lambda x.f(f(f(x))) \\
& \vdots &
\end{array}
$$

$$
\begin{array}{rcl}
\mathcal{D}[\![e_1 \texttt{+} e_2]\!] & = & \lambda f.\lambda x.\mathcal{D}[\![e_1]\!](f)(\mathcal{D}[\![e_2]\!](f)(x)) \\
\mathcal{D}[\![e_1 \texttt{*} e_2]\!] & = & \lambda f.\lambda x.\mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!](f))(x)
\end{array}
$$

Church Encodings – Church Numerals

For example,

$$
\begin{array}{rcl}
\mathcal{D}[\![1 + 1]\!] & = & \lambda f.\lambda x.\mathcal{D}[\![1]\!](f)(\mathcal{D}[\![1]\!](f)(x)) \\
& = & \lambda f.\lambda x.(\lambda f.\lambda x.f(x))(f)((\lambda f.\lambda x.f(x))(f)(x)) \\
& = & \lambda f.\lambda x.f((\lambda f.\lambda x.f(x))(f)(x)) \\
& = & \lambda f.\lambda x.f(f(x)) \\
& = & \mathcal{D}[\![2]\!]
\end{array}
$$

We can represent other data or operations in the **LC** using **Church encodings**, such as **integers**, **booleans**, **pairs**, **lists**, and so on.[2]

Let's see one more example of **Church encoding** for **booleans** and **logical operations** (i.e., **Church booleans**).

[2] https://en.wikipedia.org/wiki/Church_encoding

Church Encodings – Church Booleans

The key idea is to encode a **boolean** b as a **function** that takes two arguments t and f and applies t if b is `true` or f if b is `false`:

$$
\begin{aligned}
\mathcal{D}[\![\texttt{true}]\!] &= \lambda t.\lambda f.t \\
\mathcal{D}[\![\texttt{false}]\!] &= \lambda t.\lambda f.f
\end{aligned}
\qquad
\begin{aligned}
\mathcal{D}[\![\texttt{if}(e_1)\ e_2\ \texttt{else}\ e_3]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_3]\!]) \\
\mathcal{D}[\![e_1\ \texttt{\&\&}\ e_2]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_1]\!]) \\
\mathcal{D}[\![e_1\ \texttt{||}\ e_2]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_1]\!])(\mathcal{D}[\![e_2]\!]) \\
\mathcal{D}[\![!\ e]\!] &= \lambda t.\lambda f.\mathcal{D}[\![e]\!](f)(t)
\end{aligned}
$$

For example,

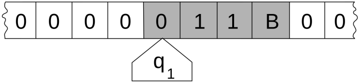
$$
\begin{aligned}
\mathcal{D}[\![\texttt{true \&\& false}]\!] &= \mathcal{D}[\![\texttt{true}]\!](\mathcal{D}[\![\texttt{false}]\!])(\mathcal{D}[\![\texttt{true}]\!]) \\
&= (\lambda t.\lambda f.t)(\mathcal{D}[\![\texttt{false}]\!])(\mathcal{D}[\![\texttt{true}]\!]) \\
&= \mathcal{D}[\![\texttt{false}]\!]
\end{aligned}
$$

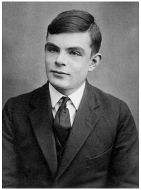
Church-Turing Thesis

Alonzo Church invented **lambda calculus** in 1930s, and it became the foundation of **programming languages**:

$$e ::= e \mid \lambda x.e \mid e(e)$$

Alan Turing invented **Turing machines (TM)** in 1936, and it became the foundation of **computers**:





Church-Turing Thesis: **Lambda Calculus** is **Turing complete**.

> *Any* **real-world computation** *can be translated into an equivalent computation involving a* **Turing machine** *or can be done using* **lambda calculus**.

Homework #2

https://github.com/ku-plrg-classroom/docs/tree/main/cose212/cobalt

- Please see above document on GitHub:
  1. Implement `interp` function.
  2. Implement `subExpr1` and `subExpr2` functions.

- The due date is 23:59 on Oct. 14 (Mon.).

- Please only submit **`Implementation.scala`** file to **Blackboard**.

Summary

1. Syntactic Sugar
   - No More `val`
   - FAE – Removing `val` from FVAE
   - Syntactic Sugar and Desugaring

2. Lambda Calculus
   - Definition
   - Church Encodings
   - Church-Turing Thesis

Next Lecture

- Recursive Functions

Jihyeok Park
jihyeok_park@korea.ac.kr
https://plrg.korea.ac.kr