Lecture 8 – Lambda Calculus

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 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

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

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

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

Contents

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

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

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 $[x \mapsto 1]$.

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 $[x \mapsto 1]$.

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 $[x \mapsto 1]$.

No More `val`

In general, the following two expressions are equivalent:

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

Why?

No More `val`

In general, the following two expressions are equivalent:

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

Why?

The following inference rule for the semantics of `val` x=e; e':

$$\textsc{Val}\ \frac{\sigma \vdash e \Rightarrow v \qquad \sigma[x \mapsto v] \vdash e' \Rightarrow v'}{\sigma \vdash \texttt{val}\ x = e;\ e' \Rightarrow v'}$$

is equivalent to the following inference rule for the semantics of $(\lambda x.e')(e)$:

$$\texttt{App}\ \frac{\texttt{Fun}\ \dfrac{}{\sigma \vdash \lambda x.e' \Rightarrow \langle \lambda x.e', \sigma\rangle} \qquad \sigma \vdash e \Rightarrow v \qquad \sigma[x \mapsto v] \vdash e' \Rightarrow v'}{\sigma \vdash (\lambda x.e')(e) \Rightarrow v'}$$

FAE – Removing `val` from FVAE

Then, we can define a smaller language FAE

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

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

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

Syntactic Sugar and Desugaring

PLRG

Definition (Syntactic Sugar)

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

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.

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

Syntactic Sugar and Desugaring

PLRG

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.

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

$$\begin{array}{lcl}
\mathcal{D}[\![n]\!] & = & n \\
\mathcal{D}[\![e + e']\!] & = & \mathcal{D}[\![e]\!] + \mathcal{D}[\![e']\!] \\
\mathcal{D}[\![e \times e']\!] & = & \mathcal{D}[\![e]\!] \times \mathcal{D}[\![e']\!]
\end{array}
\qquad
\begin{array}{lcl}
\mathcal{D}[\![\texttt{val}\ x\texttt{=}e\texttt{;}\ e']\!] & = & (\lambda x.\mathcal{D}[\![e']\!])(\mathcal{D}[\![e]\!]) \\
\mathcal{D}[\![x]\!] & = & x \\
\mathcal{D}[\![\lambda x.e]\!] & = & \lambda x.\mathcal{D}[\![e]\!] \\
\mathcal{D}[\![e(e')]\!] & = & \mathcal{D}[\![e]\!](\mathcal{D}[\![e']\!])
\end{array}$$

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.

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

$$\begin{array}{rcl}
\mathcal{D}[\![n]\!] & = & n \\
\mathcal{D}[\![e + e']\!] & = & \mathcal{D}[\![e]\!] + \mathcal{D}[\![e']\!] \\
\mathcal{D}[\![e \times e']\!] & = & \mathcal{D}[\![e]\!] \times \mathcal{D}[\![e']\!]
\end{array}
\qquad
\begin{array}{rcl}
\mathcal{D}[\![\texttt{val}\ x\texttt{=}e;\ e']\!] & = & (\lambda x.\mathcal{D}[\![e']\!])(\mathcal{D}[\![e]\!]) \\
\mathcal{D}[\![x]\!] & = & x \\
\mathcal{D}[\![\lambda x.e]\!] & = & \lambda x.\mathcal{D}[\![e]\!] \\
\mathcal{D}[\![e(e')]\!] & = & \mathcal{D}[\![e]\!](\mathcal{D}[\![e']\!])
\end{array}$$

For example,

$$\mathcal{D}[\![\texttt{val}\ x\texttt{=}42;\ \texttt{val}\ y\texttt{=}x + 1;\ y + 2]\!] = (\lambda x.(\lambda y.y + 2)(x + 1))(42)$$

Syntactic Sugar and Desugaring

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

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

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

Syntactic Sugar and Desugaring

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

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

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

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

```
val e1: Expr = Expr("val x = 42; val y = x + 1; y + 2")
val e2: Expr = Expr("(x => (y => y + 2)(x + 1))(42)")
desugar(e1) == e2
```

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

```
x += y; x *= y;
```
≡
```
x = x + y; x = x * y;
```

- Haskell

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

- ...

Contents

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

2. Lambda Calculus
    Definition
    Church Encodings
    Church-Turing Thesis

Lambda Calculus

What is the minimal language that can express all the syntactic elements of FVAE?

Lambda Calculus

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**:

$$
\begin{array}{rrcl}
\text{Expressions} & \mathbb{E} \ni e & ::= & x \\
& & \mid & \lambda x.e \\
& & \mid & e(e)
\end{array}
$$

Lambda Calculus

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**:

$$\begin{array}{llcl} \text{Expressions} & \mathbb{E} \ni e & ::= & x \\ & & \mid & \lambda x.e \\ & & \mid & e(e) \end{array}$$

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

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

Lambda Calculus

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**:

$$\begin{array}{lrcl} \text{Expressions} & \mathbb{E} \ni e & ::= & x \\ & & \mid & \lambda x.e \\ & & \mid & e(e) \end{array}$$

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

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

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

Lambda Calculus

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**:

$$\begin{array}{llll} \text{Expressions} & \mathbb{E} \ni e & ::= & x \\ & & \mid & \lambda x.e \\ & & \mid & e(e) \end{array}$$

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

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

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

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

Church Encodings – Church Numerals

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

Church Encodings – Church Numerals

Church encodings are a way 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)**.

Church Encodings – Church Numerals

Church encodings are a way 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:

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

$$
\begin{array}{rcl}
\mathcal{D}[\![e_0 + e_1]\!] & = & \lambda f.\lambda x.\mathcal{D}[\![e_0]\!](f)(\mathcal{D}[\![e_1]\!](f)(x)) \\
\mathcal{D}[\![e_0 \times e_1]\!] & = & \lambda f.\lambda x.\mathcal{D}[\![e_0]\!](\mathcal{D}[\![e_1]\!](f))(x)
\end{array}
$$

Church Encodings – Church Numerals

For example,

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

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

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.[1]

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

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.[1]

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

[1] 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`:

$$\mathcal{D}[\![\texttt{true}]\!] = \lambda t.\lambda f.t \qquad \mathcal{D}[\![\texttt{false}]\!] = \lambda t.\lambda f.f$$

$$\begin{aligned}
\mathcal{D}[\![\texttt{if}(e_1)\ e_2\ \texttt{else}\ e_3]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_3]\!]) \\
\mathcal{D}[\![e_1\ \texttt{\&\&}\ e_2]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_1]\!]) \\
\mathcal{D}[\![e_1\ \texttt{||}\ e_2]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_1]\!])(\mathcal{D}[\![e_2]\!]) \\
\mathcal{D}[\![\texttt{!}\ e_0]\!] &= \lambda t.\lambda f.\mathcal{D}[\![e_0]\!](f)(t)
\end{aligned}$$

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`:

$$\begin{array}{rl} \mathcal{D}[\![\texttt{true}]\!] &= \lambda t.\lambda f.t \\ \mathcal{D}[\![\texttt{false}]\!] &= \lambda t.\lambda f.f \end{array} \qquad \begin{array}{ll} \mathcal{D}[\![\texttt{if}(e_1)\ e_2\ \texttt{else}\ e_3]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_3]\!]) \\ \mathcal{D}[\![e_1\ \texttt{\&\&}\ e_2]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_2]\!])(\mathcal{D}[\![e_1]\!]) \\ \mathcal{D}[\![e_1\ \texttt{||}\ e_2]\!] &= \mathcal{D}[\![e_1]\!](\mathcal{D}[\![e_1]\!])(\mathcal{D}[\![e_2]\!]) \\ \mathcal{D}[\![!\ e_0]\!] &= \lambda t.\lambda f.\mathcal{D}[\![e_0]\!](f)(t) \end{array}$$

For example,

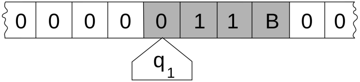
$$\begin{array}{rcl} \mathcal{D}[\![\texttt{true \&\& false}]\!] &=& \mathcal{D}[\![\texttt{true}]\!](\mathcal{D}[\![\texttt{false}]\!])(\mathcal{D}[\![\texttt{true}]\!]) \\ &=& (\lambda t.\lambda f.t)(\mathcal{D}[\![\texttt{false}]\!])(\mathcal{D}[\![\texttt{true}]\!]) \\ &=& \mathcal{D}[\![\texttt{false}]\!]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

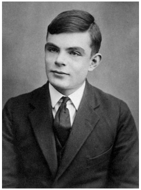
Church-Turing Thesis

Alonzo Church invented **lambda calculus (LC)** 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: **LC** 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**.

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