

# Lecture 15 – Continuations (2)

## COSE212: Programming Languages

Jihyeok Park



2023 Fall

- We will learn about **continuations** with the following topics:
  - **Continuations** (Lecture 14 & 15)
  - **First-Class Continuations** (Lecture 16)
  - **Compiling with continuations** (Lecture 17)
- A **continuation** represents the **rest of the computation**.
  - Continuation Passing Style (CPS)
  - Interpreter of FAE in CPS
- In this lecture, let's define the semantics of FAE in CPS.

## 1. Recall

Recall: Interpreter of FAE in CPS

Recall: Natural Semantics of FAE

## 2. Reduction Semantics of FAE

Number

Addition

Multiplication

Identifier Lookup

Function Definition

Function Application

Semantic Equivalence

Example

## 3. First-Order Representations of Continuations

## 1. Recall

Recall: Interpreter of FAE in CPS

Recall: Natural Semantics of FAE

## 2. Reduction Semantics of FAE

Number

Addition

Multiplication

Identifier Lookup

Function Definition

Function Application

Semantic Equivalence

Example

## 3. First-Order Representations of Continuations

In the previous lecture, we represented the **continuation** of each expression in the interpreter of FAE as a **function** and implemented the interpreter in **continuation passing style** (CPS):

```
enum Value:
  case NumV(number: BigInt)
  case CloV(param: String, body: Expr, env: Env)
type Env = Map[String, Value]
type Cont = Value => Value

def interpCPS(expr: Expr, env: Env, k: Cont): Value = ...
```

Then, how can we define the **continuations** for the semantics of FAE?

Values	$\mathbb{V} \ni v ::= n$	(NumV)
	$\mid \langle \lambda x. e, \sigma \rangle$	(CloV)
Environments	$\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$	(Env)
Continuations	$= ???$	

$$\text{Num } \frac{}{\sigma \vdash n \Rightarrow n} \quad \text{Add } \frac{\sigma \vdash e_1 \Rightarrow n_1 \quad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad \dots$$

The derivation of **big-step operational (natural) semantics** of FAE is shaped as a tree:

$$\begin{array}{c} \text{Num } \frac{}{\emptyset \vdash 1 \Rightarrow 1} \quad \text{Num } \frac{}{\emptyset \vdash 2 \Rightarrow 2} \\ \text{Add } \frac{}{\emptyset \vdash 1 + 2 \Rightarrow 3} \quad \text{Num } \frac{}{\emptyset \vdash 4 \Rightarrow 4} \\ \text{Add } \frac{}{\emptyset \vdash (1 + 2) + 4 \Rightarrow 7} \end{array}$$

It is possible but non-trivial to represent **continuations** in the **big-step operational (natural) semantics**.

Let's define the **small-step operational (reduction) semantics** of FAE to represent **continuations**.

## 1. Recall

Recall: Interpreter of FAE in CPS

Recall: Natural Semantics of FAE

## 2. Reduction Semantics of FAE

Number

Addition

Multiplication

Identifier Lookup

Function Definition

Function Application

Semantic Equivalence

Example

## 3. First-Order Representations of Continuations

- **Big-step operational (natural) semantics:**

$$\sigma \vdash e \Rightarrow v$$

- **Small-step operational (reduction) semantics:**

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

where  $\rightarrow \in (\mathbb{K} \times \mathbb{S}) \times (\mathbb{K} \times \mathbb{S})$  is a **reduction relation** between **states**.

$$\begin{array}{lcl} \text{Continuations } \mathbb{K} \ni \kappa ::= & \square & \\ & | (\sigma \vdash e) :: \kappa & \\ & | (+) :: \kappa & \\ & | (\times) :: \kappa & \\ & | (@) :: \kappa & \end{array}$$

$$\text{Value Stacks } \mathbb{S} \ni s ::= \blacksquare \mid v :: s$$



```
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
  ...
  case Num(n) => k(NumV(n))
```

$$\boxed{\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle}$$

$$\text{Num} \quad \langle (\sigma \vdash n) :: \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel n :: s \rangle$$

```
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
...
case Add(l, r) =>
  interpCPS(l, env, {
    lv => interpCPS(r, env, {
      rv => k(numAdd(lv, rv))
    })
  })
})
```

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

$$\text{Add}_1 \quad \langle (\sigma \vdash e_1 + e_2) :: \kappa \parallel s \rangle \rightarrow \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (+) :: \kappa \parallel s \rangle$$

$$\text{Add}_2 \quad \langle (+) :: \kappa \parallel n_2 :: n_1 :: s \rangle \rightarrow \langle \kappa \parallel (n_1 + n_2) :: s \rangle$$

```
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
...
case Mul(l, r) =>
  interpCPS(l, env, {
    lv => interpCPS(r, env, {
      rv => k(numMul(lv, rv))
    })
  })
})
```

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

$$\text{Mul}_1 \quad \langle (\sigma \vdash e_1 \times e_2) :: \kappa \parallel s \rangle \rightarrow \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (\times) :: \kappa \parallel s \rangle$$

$$\text{Mul}_2 \quad \langle (\times) :: \kappa \parallel n_2 :: n_1 :: s \rangle \rightarrow \langle \kappa \parallel (n_1 \times n_2) :: s \rangle$$

```
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
...
case Id(x) => k(lookupId(x, env))
```

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

$$\text{Id} \quad \langle (\sigma \vdash x) :: \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel \sigma(x) :: s \rangle$$

```
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
...
case Fun(p, b) => k(CloV(p, b, env))
```

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

$$\text{Fun} \quad \langle (\sigma \vdash \lambda x.e) :: \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel \langle \lambda x.e, \sigma \rangle :: s \rangle$$

```
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
...
case App(f, e) => interpCPS(f, env, v => v match
  case CloV(p, b, fenv) =>
    interpCPS(e, env, v => {
      interpCPS(b, fenv + (p -> v), k)
    })
  case v => error(s"not a function: ${v.str}")
)
```

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

$$\text{App}_1 \langle (\sigma \vdash e_1(e_2)) :: \kappa \parallel s \rangle \rightarrow \langle (\sigma \vdash e_2) :: (\sigma \vdash e_1) :: (@) :: \kappa \parallel s \rangle$$

$$\text{App}_2 \langle (@) :: \kappa \parallel v_2 :: \langle \lambda x. e, \sigma \rangle :: s \rangle \rightarrow \langle (\sigma[x \mapsto v_2] \vdash e) :: \kappa \parallel s \rangle$$

- The **reflexive transitive closure**  $(\rightarrow^*)$  of  $(\rightarrow)$ :

$$\langle \kappa \parallel s \rangle \rightarrow^* \langle \kappa \parallel s \rangle$$

$$\frac{\langle \kappa \parallel s \rangle \rightarrow^* \langle \kappa' \parallel s' \rangle \quad \langle \kappa' \parallel s' \rangle \rightarrow \langle \kappa'' \parallel s'' \rangle}{\langle \kappa \parallel s \rangle \rightarrow^* \langle \kappa'' \parallel s'' \rangle}$$

- The **semantic equivalence** between natural and reduction semantics:

$$\emptyset \vdash e \Rightarrow v \quad \Longleftrightarrow \quad \langle (\emptyset \vdash e) :: \square \parallel \blacksquare \rangle \rightarrow^* \langle \square \parallel v :: \blacksquare \rangle$$

More generally, the following are equivalent:

$$\sigma \vdash e \Rightarrow v \quad \Longleftrightarrow \quad \langle (\sigma \vdash e) :: \kappa \parallel s \rangle \rightarrow^* \langle \kappa \parallel v :: s \rangle$$

for all  $\sigma \in \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V}$ ,  $e \in \mathbb{E}$ ,  $v \in \mathbb{V}$ ,  $\kappa \in \mathbb{K}$ , and  $s \in \mathbb{S}$ .

Let's interpret the expression  $(\lambda x.1 + x)(2)$ :

	$\langle (\emptyset \vdash (\lambda x.1 + x)(2)) :: \square$	$\parallel \blacksquare$
(App <sub>1</sub> )	$\rightarrow \langle (\emptyset \vdash \lambda x.1 + x) :: (\emptyset \vdash 2) :: (@) :: \square$	$\parallel \blacksquare$
(Fun)	$\rightarrow \langle (\emptyset \vdash 2) :: (@) :: \square$	$\parallel \langle \lambda x.1 + x, \emptyset \rangle :: \blacksquare$
(Num)	$\rightarrow \langle (@) :: \square$	$\parallel 2 :: \langle \lambda x.1 + x, \emptyset \rangle :: \blacksquare$
(App <sub>2</sub> )	$\rightarrow \langle ([x \mapsto 2] \vdash 1 + x) :: \square$	$\parallel \blacksquare$
(Add <sub>1</sub> )	$\rightarrow \langle ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square$	$\parallel \blacksquare$
(Num)	$\rightarrow \langle ([x \mapsto 2] \vdash x) :: (+) :: \square$	$\parallel 1 :: \blacksquare$
(Num)	$\rightarrow \langle (+) :: \square$	$\parallel 2 :: 1 :: \blacksquare$
(Add <sub>2</sub> )	$\rightarrow \langle \square$	$\parallel 3 :: \blacksquare$

Thus,  $\langle (\emptyset \vdash (\lambda x.1 + x)(2)) :: \square \parallel \blacksquare \rangle \rightarrow^* \langle \square \parallel 3 :: \blacksquare \rangle$ .



## 1. Recall

Recall: Interpreter of FAE in CPS

Recall: Natural Semantics of FAE

## 2. Reduction Semantics of FAE

Number

Addition

Multiplication

Identifier Lookup

Function Definition

Function Application

Semantic Equivalence

Example

## 3. First-Order Representations of Continuations

In our new implementation for FAE using CPS, we define the type of continuations using the **closures (first-class functions)** in Scala.

```
enum Value:
  case NumV(number: BigInt)
  case CloV(param: String, body: Expr, env: Env)
type Env = Map[String, Value]
type Cont = Value => Value

def interpCPS(expr: Expr, env: Env, k: Cont): Value = ...
```

How can we define continuations if we want to implement the interpreter for FAE in CPS using a **non-functional** language (e.g., C)?

Let's define the continuations as **data structures** (e.g., algebraic data types) in such languages (**first-order representations**).

```
enum Cont:
  case EmptyK
  case EvalK(env: Env, expr: Expr, k: Cont)
  case AddK(k: Cont)
  case MulK(k: Cont)
  case AppK(k: Cont)

type Stack = List[Value]
```

Continuations  $\mathbb{K} \ni \kappa ::= \square \quad (\text{EmptyK})$   
 $\quad \quad \quad | (\sigma \vdash e) :: \kappa \quad (\text{EvalK})$   
 $\quad \quad \quad | (+) :: \kappa \quad (\text{AddK})$   
 $\quad \quad \quad | (\times) :: \kappa \quad (\text{MulK})$   
 $\quad \quad \quad | (@) :: \kappa \quad (\text{AppK})$

Value Stacks  $\mathbb{S} \ni s ::= \blacksquare \mid v :: s \quad (\text{List[Value]})$

We define a **reduce** function that takes a state  $\langle \kappa \parallel s \rangle$  and **reduces** it to another state  $\langle \kappa' \parallel s' \rangle$  using the reduction relation  $\rightarrow$  we defined before:

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa' \parallel s' \rangle$$

```
def reduce(k: Cont, s: Stack): (Cont, Stack) = ???
```

And the **evalK** function **iteratively reduces** the state until it reaches the empty continuation  $\square$  and returns the single value in the value stack:

```
def evalK(str: String): String =  
  import Cont.*  
  def aux(k: Cont, s: Stack): Value = reduce(k, s) match  
    case (EmptyK, List(v)) => v  
    case (k, s) => aux(k, s)  
  aux(EvalK(Map.empty, Expr(str), EmptyK), List.empty).str
```

$$\langle (\emptyset \vdash e) :: \square \parallel \blacksquare \rangle \rightarrow^* \langle \square \parallel v :: \blacksquare \rangle$$

```
def reduce(k: Cont, s: Stack): (Cont, Stack) = (k, s) match
  case (EvalK(env, expr, k), s) => expr match
    ...
    case Add(l, r) => (EvalK(env, l, EvalK(env, r, AddK(k))), s)
  ...
  case (AddK(k), r :: l :: s) => (k, numAdd(l, r) :: s)
  ...
```

$$\langle \kappa \parallel s \rangle \rightarrow \langle \kappa \parallel s \rangle$$

$$\text{Add}_1 \quad \langle (\sigma \vdash e_1 + e_2) :: \kappa \parallel s \rangle \rightarrow \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (+) :: \kappa \parallel s \rangle$$

$$\text{Add}_2 \quad \langle (+) :: \kappa \parallel n_2 :: n_1 :: s \rangle \rightarrow \langle \kappa \parallel (n_1 + n_2) :: s \rangle$$

Similarly, we can define the reduce function for the other cases.

## 1. Recall

Recall: Interpreter of FAE in CPS

Recall: Natural Semantics of FAE

## 2. Reduction Semantics of FAE

Number

Addition

Multiplication

Identifier Lookup

Function Definition

Function Application

Semantic Equivalence

Example

## 3. First-Order Representations of Continuations

- Please see this document<sup>1</sup> on GitHub.
  - Implement `interpCPS` function.
  - Implement `reduce` function.
- It is just an exercise, and you **don't need to submit** anything.
- However, some exam questions might be related to this exercise.

---

<sup>1</sup><https://github.com/ku-plrg-classroom/docs/tree/main/cose212/fae-cps>.

- First-Class Continuations

Jihyeok Park

`jihyeok_park@korea.ac.kr`

`https://plrg.korea.ac.kr`