Lecture 15 – Continuations (2)
COSE212: Programming Languages

Jihyeok Park

⚠ PLRG

2023 Fall

PLRG

- 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

- 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.

Contents

Contents

Recall: Interpreter of FAE in CPS

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 = ...
```

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?

$$
\begin{array}{llll}
\text{Values} & \mathbb{V} \ni v ::= n & (\texttt{NumV}) \\
& \quad\quad\ | \ \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) \\
\text{Environments} & \sigma \ \in \ \mathbb{X} \xrightarrow{\text{fin}} \mathbb{V} & (\texttt{Env}) \\
\text{Continuations} & = \ \text{???}
\end{array}
$$

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

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

$$\text{Add } \cfrac{\text{Add } \cfrac{\text{Num } \cfrac{}{\varnothing \vdash 1 \Rightarrow 1} \quad \text{Num } \cfrac{}{\varnothing \vdash 2 \Rightarrow 2}}{\varnothing \vdash 1 + 2 \Rightarrow 3} \quad \text{Num } \cfrac{}{\varnothing \vdash 4 \Rightarrow 4}}{\varnothing \vdash (1+2) + 4 \Rightarrow 7}$$

$$\text{Num} \; \frac{}{\sigma \vdash n \Rightarrow n} \qquad\qquad \text{Add} \; \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad\qquad \cdots$$

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

$$\text{Add} \; \frac{\text{Add} \; \dfrac{\text{Num} \; \dfrac{}{\varnothing \vdash 1 \Rightarrow 1} \quad \text{Num} \; \dfrac{}{\varnothing \vdash 2 \Rightarrow 2}}{\varnothing \vdash 1 + 2 \Rightarrow 3} \quad \text{Num} \; \dfrac{}{\varnothing \vdash 4 \Rightarrow 4}}{\varnothing \vdash (1 + 2) + 4 \Rightarrow 7}$$

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

$$\text{Num} \; \frac{}{\sigma \vdash n \Rightarrow n} \qquad \text{Add} \; \frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \qquad \cdots$$

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

$$\text{Add} \; \frac{\text{Add} \; \frac{\text{Num} \; \frac{}{\varnothing \vdash 1 \Rightarrow 1} \quad \text{Num} \; \frac{}{\varnothing \vdash 2 \Rightarrow 2}}{\varnothing \vdash 1 + 2 \Rightarrow 3} \quad \text{Num} \; \frac{}{\varnothing \vdash 4 \Rightarrow 4}}{\varnothing \vdash (1 + 2) + 4 \Rightarrow 7}$$

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**.

Contents

OPLRG

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

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

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

$$\langle \kappa \mid\mid s \rangle \rightarrow \langle \kappa \mid\mid s \rangle$$

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

$$
\begin{aligned}
\text{Continuations} \quad \mathbb{K} \ni \kappa ::= \;& \square \\
& \mid (\sigma \vdash e) :: \kappa \\
& \mid (+) :: \kappa \\
& \mid (\times) :: \kappa \\
& \mid (@) :: \kappa
\end{aligned}
$$

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

Numbers

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

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

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

```
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))
      })
    })
```

$$\langle \kappa \mid\mid s \rangle \rightarrow \langle \kappa \mid\mid s \rangle$$

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

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

Multiplication

```
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))
      })
    })
```

$$\langle \kappa \mid\mid s \rangle \rightarrow \langle \kappa \mid\mid s \rangle$$

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

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

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

$$\boxed{\langle \kappa \mid\mid s \rangle \to \langle \kappa \mid\mid s \rangle}$$

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

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

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

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

```
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}")
  )
```

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

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

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

Semantic Equivalence

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

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

$$\frac{\langle \kappa \mid\mid s \rangle \rightarrow^* \langle \kappa' \mid\mid s' \rangle \qquad \langle \kappa' \mid\mid s' \rangle \rightarrow \langle \kappa'' \mid\mid s'' \rangle}{\langle \kappa \mid\mid s \rangle \rightarrow^* \langle \kappa'' \mid\mid s'' \rangle}$$

Semantic Equivalence

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

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

$$\frac{\langle \kappa \mid\mid s \rangle \rightarrow^* \langle \kappa' \mid\mid s' \rangle \qquad \langle \kappa' \mid\mid s' \rangle \rightarrow \langle \kappa'' \mid\mid s'' \rangle}{\langle \kappa \mid\mid s \rangle \rightarrow^* \langle \kappa'' \mid\mid s'' \rangle}$$

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

$$\varnothing \vdash e \Rightarrow v \qquad \Longleftrightarrow \qquad \langle (\varnothing \vdash e) :: \square \mid\mid \blacksquare \rangle \rightarrow^* \langle \square \mid\mid v :: \blacksquare \rangle$$

  More generally, the following are equivalent:

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

  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 $(\lambda x.1 + x)(2)$:

Example

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

$$\langle \; (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square \qquad\qquad || \; \blacksquare \qquad\qquad\qquad \rangle$$

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

$$
\begin{array}{ll}
\langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \Box & ||\ \blacksquare \\
\xrightarrow{\text{App}_1}\ \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (\text{@}) :: \Box & ||\ \blacksquare
\end{array}\ \rangle
$$

Example

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

$$
\begin{array}{lll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & || \ \blacksquare \\
\overset{\text{App}_1}{\rightarrow} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \square & || \ \blacksquare \\
\overset{\text{Fun}}{\rightarrow} & \langle\ (\varnothing \vdash 2) :: (@) :: \square & || \ \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \\
\end{array}
\quad
\begin{array}{l}
\rangle \\
\rangle \\
\rangle \\
\end{array}
$$

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

$$
\begin{array}{lll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & ||\ \blacksquare & \rangle \\
\overset{\text{App}_1}{\rightarrow} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (\texttt{@}) :: \square & ||\ \blacksquare & \rangle \\
\overset{\text{Fun}}{\rightarrow} & \langle\ (\varnothing \vdash 2) :: (\texttt{@}) :: \square & ||\ \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare & \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\ (\texttt{@}) :: \square & ||\ 2 :: \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare & \rangle
\end{array}
$$

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

$$
\begin{array}{lll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \Box & || \ \blacksquare \rangle \\
\overset{\text{App}_1}{\rightarrow} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (\text{@}) :: \Box & || \ \blacksquare \rangle \\
\overset{\text{Fun}}{\rightarrow} & \langle\ (\varnothing \vdash 2) :: (\text{@}) :: \Box & || \ \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\ (\text{@}) :: \Box & || \ 2 :: \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare \rangle \\
\overset{\text{App}_2}{\rightarrow} & \langle\ ([x \mapsto 2] \vdash 1 + x) :: \Box & || \ \blacksquare \rangle
\end{array}
$$

Example

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

$$
\begin{array}{ll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & || \ \blacksquare & \rangle \\
\overset{\text{App}_1}{\rightarrow} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \square & || \ \blacksquare & \rangle \\
\overset{\text{Fun}}{\rightarrow} & \langle\ (\varnothing \vdash 2) :: (@) :: \square & || \ \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare & \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\ (@) :: \square & || \ 2 :: \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare & \rangle \\
\overset{\text{App}_2}{\rightarrow} & \langle\ ([x \mapsto 2] \vdash 1 + x) :: \square & || \ \blacksquare & \rangle \\
\overset{\text{Add}_1}{\rightarrow} & \langle\ ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square & || \ \blacksquare & \rangle
\end{array}
$$

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

$$
\begin{aligned}
& \langle\; (\varnothing \vdash (\lambda x.1 + x)(2)) :: \Box && \|\; \blacksquare && \rangle \\
\overset{\text{App}_1}{\rightarrow}\; & \langle\; (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \Box && \|\; \blacksquare && \rangle \\
\overset{\text{Fun}}{\rightarrow}\; & \langle\; (\varnothing \vdash 2) :: (@) :: \Box && \|\; \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare && \rangle \\
\overset{\text{Num}}{\rightarrow}\; & \langle\; (@) :: \Box && \|\; 2 :: \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare && \rangle \\
\overset{\text{App}_2}{\rightarrow}\; & \langle\; ([x \mapsto 2] \vdash 1 + x) :: \Box && \|\; \blacksquare && \rangle \\
\overset{\text{Add}_1}{\rightarrow}\; & \langle\; ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \Box && \|\; \blacksquare && \rangle \\
\overset{\text{Num}}{\rightarrow}\; & \langle\; ([x \mapsto 2] \vdash x) :: (+) :: \Box && \|\; 1 :: \blacksquare && \rangle
\end{aligned}
$$

Example

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

$$
\begin{array}{lll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \Box & || \ \blacksquare \ \rangle \\
\overset{\text{App}_1}{\rightarrow} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \Box & || \ \blacksquare \ \rangle \\
\overset{\text{Fun}}{\rightarrow} & \langle\ (\varnothing \vdash 2) :: (@) :: \Box & || \ \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \ \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\ (@) :: \Box & || \ 2 :: \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \ \rangle \\
\overset{\text{App}_2}{\rightarrow} & \langle\ ([x \mapsto 2] \vdash 1 + x) :: \Box & || \ \blacksquare \ \rangle \\
\overset{\text{Add}_1}{\rightarrow} & \langle\ ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \Box & || \ \blacksquare \ \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\ ([x \mapsto 2] \vdash x) :: (+) :: \Box & || \ 1 :: \blacksquare \ \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\ (+) :: \Box & || \ 2 :: 1 :: \blacksquare \ \rangle
\end{array}
$$

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

$$
\begin{array}{lll}
& \langle\, (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & || \blacksquare & \rangle \\
\overset{\text{App}_1}{\rightarrow} & \langle\, (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \square & || \blacksquare & \rangle \\
\overset{\text{Fun}}{\rightarrow} & \langle\, (\varnothing \vdash 2) :: (@) :: \square & || \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare & \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\, (@) :: \square & || 2 :: \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare & \rangle \\
\overset{\text{App}_2}{\rightarrow} & \langle\, ([x \mapsto 2] \vdash 1 + x) :: \square & || \blacksquare & \rangle \\
\overset{\text{Add}_1}{\rightarrow} & \langle\, ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square & || \blacksquare & \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\, ([x \mapsto 2] \vdash x) :: (+) :: \square & || 1 :: \blacksquare & \rangle \\
\overset{\text{Num}}{\rightarrow} & \langle\, (+) :: \square & || 2 :: 1 :: \blacksquare & \rangle \\
\overset{\text{Add}_2}{\rightarrow} & \langle\, \square & || 3 :: \blacksquare & \rangle
\end{array}
$$

Example

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

$$
\begin{array}{lll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & || \ \blacksquare \quad\rangle \\
\overset{\text{App}_1}{\to} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \square & || \ \blacksquare \quad\rangle \\
\overset{\text{Fun}}{\to} & \langle\ (\varnothing \vdash 2) :: (@) :: \square & || \ \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare \quad\rangle \\
\overset{\text{Num}}{\to} & \langle\ (@) :: \square & || \ 2 :: \langle\lambda x.1 + x, \varnothing\rangle :: \blacksquare \ \rangle \\
\overset{\text{App}_2}{\to} & \langle\ ([x \mapsto 2] \vdash 1 + x) :: \square & || \ \blacksquare \quad\rangle \\
\overset{\text{Add}_1}{\to} & \langle\ ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square & || \ \blacksquare \quad\rangle \\
\overset{\text{Num}}{\to} & \langle\ ([x \mapsto 2] \vdash x) :: (+) :: \square & || \ 1 :: \blacksquare \quad\rangle \\
\overset{\text{Num}}{\to} & \langle\ (+) :: \square & || \ 2 :: 1 :: \blacksquare \quad\rangle \\
\overset{\text{Add}_2}{\to} & \langle\ \square & || \ 3 :: \blacksquare \quad\rangle
\end{array}
$$

Thus, $\langle(\varnothing \vdash (\lambda x.1 + x)(2)) :: \square \ || \ \blacksquare\rangle \to^* \langle\square \ || \ 3 :: \blacksquare\rangle$.

Contents

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

```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 = ...
```

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.

```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)?

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.

```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]
```

$$
\begin{array}{lll}
\text{Continuations} & \mathbb{K} \ni \kappa ::= \square & (\texttt{EmptyK}) \\
& \quad\mid (\sigma \vdash e) :: \kappa & (\texttt{EvalK}) \\
& \quad\mid (+) :: \kappa & (\texttt{AddK}) \\
& \quad\mid (\times) :: \kappa & (\texttt{MulK}) \\
& \quad\mid (@) :: \kappa & (\texttt{AppK})
\end{array}
$$

$$
\begin{array}{lll}
\text{Value Stacks} & \mathbb{S} \ni s ::= \blacksquare \mid v :: s & (\texttt{List[Value]})
\end{array}
$$

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

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

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

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

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

```scala
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:

```scala
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
```

$$\langle (\varnothing \vdash e) :: \square \mid\mid \blacksquare \rangle \rightarrow^* \langle \square \mid\mid v :: \blacksquare \rangle$$

First-Order Representations of Continuations

```
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)
  ...
```

$$\langle \kappa \mid\mid s \rangle \rightarrow \langle \kappa \mid\mid s \rangle$$

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

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

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

Summary

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[1] 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.

---

[1] https://github.com/ku-plrg-classroom/docs/tree/main/cose212/fae-cps.

◆ PLRG

- First-Class Continuations

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