Lecture 15 – Continuations (2)

COSE212: Programming Languages

Jihyeok Park

PLRG

2023 Fall

Recall

- 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

Recall

- 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

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

Contents

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

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 = ...
```

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 = ...
```

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

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

Recall: Natural Semantics of FAE

PLRG

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

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

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

Recall: Natural Semantics of FAE

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

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

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

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

Recall: Natural Semantics of FAE

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

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

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

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

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

Contents

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

Reduction Semantics of FAE

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

$$\boxed{\sigma \vdash e \Rightarrow v}$$

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

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

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

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

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

Numbers

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

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

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

Addition

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

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

$$\begin{array}{llll} \texttt{Add}_1 & \langle (\sigma \vdash e_1 + e_2) :: \kappa \mid\mid s \rangle & \rightarrow & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (+) :: \kappa \mid\mid s \rangle \\ \texttt{Add}_2 & \langle (+) :: \kappa \mid\mid n_2 :: n_1 :: s \rangle & \rightarrow & \langle \kappa \mid\mid (n_1 + n_2) :: s \rangle \end{array}$$

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))
      })
    })
```

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

$$\begin{array}{llcl}
\texttt{Mul}_1 & \langle (\sigma \vdash e_1 \times e_2) :: \kappa \mid\mid s \rangle & \rightarrow & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (\times) :: \kappa \mid\mid s \rangle \\
\texttt{Mul}_2 & \langle (\times) :: \kappa \mid\mid n_2 :: n_1 :: s \rangle & \rightarrow & \langle \kappa \mid\mid (n_1 \times n_2) :: s \rangle
\end{array}$$

Identifier Lookup

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

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

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

Function Definition

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

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

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

Function Application

```
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}")
  )
```

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

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

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

Semantic Equivalence

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

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

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

Semantic Equivalence

PLRG

- 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 $s \in \mathbb{S}$.

Example

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

Example

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

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

Example

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

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

Example

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

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

Example

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

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

Example

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

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

Example

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

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

Example

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

$$
\begin{array}{rll}
 & \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\mathtt{App}_1)} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\mathtt{Fun})} & \langle\ (\varnothing \vdash 2) :: (@) :: \square & \mid\mid \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\mathtt{Num})} & \langle\ (@) :: \square & \mid\mid 2 :: \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\mathtt{App}_2)} & \langle\ ([x \mapsto 2] \vdash 1 + x) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\mathtt{Add}_1)} & \langle\ ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\mathtt{Num})} & \langle\ ([x \mapsto 2] \vdash x) :: (+) :: \square & \mid\mid 1 :: \blacksquare \ \rangle
\end{array}
$$

Example

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

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

Example

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

$$
\begin{array}{rllll}
 & \langle & (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & \mid\mid \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle & (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (@) :: \square & \mid\mid \blacksquare & \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle & (\varnothing \vdash 2) :: (@) :: \square & \mid\mid \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle & (@) :: \square & \mid\mid 2 :: \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_2)} & \langle & ([x \mapsto 2] \vdash 1 + x) :: \square & \mid\mid \blacksquare & \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle & ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square & \mid\mid \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle & ([x \mapsto 2] \vdash x) :: (+) :: \square & \mid\mid 1 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Id})} & \langle & (+) :: \square & \mid\mid 2 :: 1 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Add}_2)} & \langle & \square & \mid\mid 3 :: \blacksquare & \rangle
\end{array}
$$

Example

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

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash (\lambda x.1 + x)(2)) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash \lambda x.1 + x) :: (\varnothing \vdash 2) :: (\texttt{@}) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 2) :: (\texttt{@}) :: \square & \mid\mid \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\texttt{@}) :: \square & \mid\mid 2 :: \langle \lambda x.1 + x, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_2)} & \langle\ ([x \mapsto 2] \vdash 1 + x) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ ([x \mapsto 2] \vdash 1) :: ([x \mapsto 2] \vdash x) :: (+) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ ([x \mapsto 2] \vdash x) :: (+) :: \square & \mid\mid 1 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Id})} & \langle\ (+) :: \square & \mid\mid 2 :: 1 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Add}_2)} & \langle\ \square & \mid\mid 3 :: \blacksquare \ \rangle
\end{array}
$$

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

Contents

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

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 = ...
```

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)?

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**).

First-Order Representations of Continuations

PLRG

```
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]
```

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

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

First-Order Representations of Continuations

PLRG

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) = ???
```

First-Order Representations of Continuations

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) = ???
```

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
```

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

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)
  ...
```

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

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

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

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

Exercise #8

- 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.

Next Lecture

- First-Class Continuations

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