Lecture 14 – Continuations (1)

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 Fall

Recall

- Lazy Evaluation
  - Call-by-Name (CBN)
  - Call-by-Need (CBN')

- LFAE – FAE with Lazy Evaluation

Recall

- Lazy Evaluation
    - Call-by-Name (CBN)
    - Call-by-Need (CBN')

- LFAE – FAE with Lazy Evaluation

- We will learn about **continuations** with the following topics:
    - **Continuations** (Lecture 14 & 15)
    - **First-Class Continuations** (Lecture 16)
    - **Compiling with continuations** (Lecture 17)

Recall

- Lazy Evaluation
  - Call-by-Name (CBN)
  - Call-by-Need (CBN')

- LFAE – FAE with Lazy Evaluation

- We will learn about **continuations** with the following topics:
  - **Continuations** (Lecture 14 & 15)
  - **First-Class Continuations** (Lecture 16)
  - **Compiling with continuations** (Lecture 17)

- In this lecture, we will focus on the meaning of **continuations**.

Contents

Contents

Continuations

Many real-world programming languages support **control statements** to change the **control-flow** of a program.

Continuations

Many real-world programming languages support **control statements** to change the **control-flow** of a program.

For example, C++ supports `break`, `continue`, and `return` statements:

```
int sumEvenUntilZero(int xs[], int len) {
  if (len <= 0) return 0;           // directly return 0 if len <= 0
  int sum = 0;
  for (int i = 0; i < len; i++) {
    if (xs[i] == 0) break;          // stop the loop if xs[i] == 0
    if (xs[i] % 2 == 1) continue;   // skip the rest if xs[i] is odd
    sum += xs[i];
  }
  return sum;                       // finally return the sum
}
int xs[] = {4, 1, 3, 2, 0, 6, 5, 8};
sumEvenUntilZero(xs, 8);            // 4 + 2 = 6
```

How can we represent them in functional languages?

Continuations

Many real-world programming languages support **control statements** to change the **control-flow** of a program.

For example, C++ supports `break`, `continue`, and `return` statements:

```
int sumEvenUntilZero(int xs[], int len) {
  if (len <= 0) return 0;         // directly return 0 if len <= 0
  int sum = 0;
  for (int i = 0; i < len; i++) {
    if (xs[i] == 0) break;        // stop the loop if xs[i] == 0
    if (xs[i] % 2 == 1) continue; // skip the rest if xs[i] is odd
    sum += xs[i];
  }
  return sum;                     // finally return the sum
}
int xs[] = {4, 1, 3, 2, 0, 6, 5, 8};
sumEvenUntilZero(xs, 8);          // 4 + 2 = 6
```

How can we represent them in functional languages? **Continuations**!

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate `1`. (Result: `1`)

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate 1. (Result: 1)
2. Evaluate 3. (Result: 3)

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate `1`. (Result: `1`)
2. Evaluate `3`. (Result: `3`)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate `1`. (Result: `1`)
2. Evaluate `3`. (Result: `3`)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)
4. Evaluate `5`. (Result: `5`)

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate `1`. (Result: `1`)
2. Evaluate `3`. (Result: `3`)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)
4. Evaluate `5`. (Result: `5`)
5. Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

① Evaluate 1. (Result: 1)
② Evaluate 3. (Result: 3)
③ Add the results of step ① and ②. (Result: `1 + 3 = 4`)
④ Evaluate 5. (Result: 5)
⑤ Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

The **continuation** of k-th step is the steps from $(k+1)$-th to the last one.

Continuations

Intuitively, a **continuation** represents the **rest of the computation**.

For example, consider the following FAE expression:

```
/* FAE */
(1 + 3) * 5
```

It **implicitly** represents the following computation:

1. Evaluate 1. (Result: 1)
2. Evaluate 3. (Result: 3)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

The **continuation** of k-th step is the steps from $(k+1)$-th to the last one.

For instance, the **continuation** of the 3rd step is the 4th and 5th steps.

Continuations

Can we **explicitly** represent the **continuations** in the expression?

Continuations

Can we **explicitly** represent the **continuations** in the expression?

Yes! Let's represent the **continuation** of the k-th step as a **function** that

- **takes** the result of the k-th step as an argument and
- **performs** the $(k+1)$-th to the last steps.

Continuations

Can we **explicitly** represent the **continuations** in the expression?

Yes! Let's represent the **continuation** of the k-th step as a **function** that

- **takes** the result of the k-th step as an argument and
- **performs** the $(k+1)$-th to the last steps.

If e' is the **current evaluation part** in the expression e:

$$e \;=\; (\ldots\; e' \;\ldots)$$

we can revise it as:

$$(\lambda x.(\ldots\; x \;\ldots))(e')$$

where $\lambda x.(\ldots\; x \;\ldots)$ is the **continuation** of e'.

Continuations

Can we **explicitly** represent the **continuations** in the expression?

Yes! Let's represent the **continuation** of the k-th step as a **function** that

- **takes** the result of the k-th step as an argument and
- **performs** the $(k+1)$-th to the last steps.

If e' is the **current evaluation part** in the expression e:

$$e \;=\; (\ldots\; e' \;\ldots)$$

we can revise it as:

$$(\lambda x.(\ldots\; x \;\ldots))(e')$$

where $\lambda x.(\ldots\; x \;\ldots)$ is the **continuation** of e'.

Let's explicitly represent the **continuations** of the previous example:

```
/* FAE */
(1 + 3) * 5
```

Continuations

1. Evaluate 1. (Result: 1)
2. Evaluate 3. (Result: 3)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

```
/* FAE */
(1 + 3) * 5
```

Continuations

1. Evaluate 1. (Result: 1)
2. Evaluate 3. (Result: 3)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

```
/* FAE */
{
  x1 => (x1 + 3) * 5      // step 2-5 (continuation of step 1)
}(1)                      // step 1
```

Continuations

① Evaluate 1. (Result: 1)

② Evaluate 3. (Result: 3)

③ Add the results of step ① and ②. (Result: `1 + 3 = 4`)

④ Evaluate 5. (Result: 5)

⑤ Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

```
/* FAE */
{
  x1 => {
    x2 => (x1 + x2) * 5   // step 3-5 (continuation of step 2)
  }(3)                    // step 2
}(1)                      // step 1
```

Continuations

1. Evaluate 1. (Result: 1)
2. Evaluate 3. (Result: 3)
3. Add the results of step ① and ②. (Result: `1 + 3 = 4`)
4. Evaluate 5. (Result: 5)
5. Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

```
/* FAE */
{
  x1 => {
    x2 => {
      x3 => x3 * 5          // step 4-5 (continuation of step 3)
    }(x1 + x2)              // step 3
  }(3)                      // step 2
}(1)                        // step 1
```

Continuations

① Evaluate 1.	(Result: `1`)
② Evaluate 3.	(Result: `3`)
③ Add the results of step ① and ②.	(Result: `1 + 3 = 4`)
④ Evaluate 5.	(Result: `5`)
⑤ Multiply the results of step ③ and ④.	(Result: `4 * 5 = 20`)

```
/* FAE */
{
  x1 => {
    x2 => {
      x3 => {
        x4 => x3 * x4     // step 5 (continuation of step 4)
      }(5)                // step 4
    }(x1 + x2)            // step 3
  }(3)                    // step 2
}(1)                      // step 1
```

Continuations

① Evaluate 1. (Result: 1)

② Evaluate 3. (Result: 3)

③ Add the results of step ① and ②. (Result: `1 + 3 = 4`)

④ Evaluate 5. (Result: 5)

⑤ Multiply the results of step ③ and ④. (Result: `4 * 5 = 20`)

```
/* FAE */
{
  x1 => {
    x2 => {
      x3 => {
        x4 => {
          x5 => x5          // no more steps (continuation of step 5)
        }(x3 * x4)          // step 5
      }(5)                  // step 4
    }(x1 + x2)              // step 3
  }(3)                      // step 2
}(1)                        // step 1
```

Continuations

1. Evaluate `1`. (Result: `1`)
2. Evaluate `3`. (Result: `3`)
3. Add the results of step ❶ and ❷. (Result: `1 + 3 = 4`)
4. Evaluate `5`. (Result: `5`)
5. Multiply the results of step ❸ and ❹. (Result: `4 * 5 = 20`)

```
/* FAE */
val x1 = 1;                // step 1
val x2 = 3;                // step 2
val x3 = x1 + x2;          // step 3
val x4 = 5;                // step 4
val x5 = x3 * x4;          // step 5
x5                         // no more steps (continuation of step 5)
```

by using the syntactic sugar for variable definitions (`val`).

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

Contents

Continuation-Passing Style (CPS)

So far, we implement functions in **direct style**, where the result of a function is **returned** to the caller.

For example, the following Scala `sum` function is written in **direct style**:

```scala
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n

sum(3) * 5              // (1 + 2 + 3) * 5 = 30
```

Continuation-Passing Style (CPS)

So far, we implement functions in **direct style**, where the result of a function is **returned** to the caller.

For example, the following Scala `sum` function is written in **direct style**:

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n

sum(3) * 5              // (1 + 2 + 3) * 5 = 30
```

Continuation-passing style (CPS) is a style of programming that passes the continuation as an explicit parameter to a function and calls it to give the result to the continuation.

Continuation-Passing Style (CPS)

So far, we implement functions in **direct style**, where the result of a function is **returned** to the caller.

For example, the following Scala `sum` function is written in **direct style**:

```scala
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n

sum(3) * 5              // (1 + 2 + 3) * 5 = 30
```

Continuation-passing style (CPS) is a style of programming that passes the continuation as an explicit parameter to a function and calls it to give the result to the continuation. Let's rewrite the `sum` function in CPS:

```scala
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int = ???

sumCPS(3, x => x * 5)   // (1 + 2 + 3) * 5 = 30
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int = ???
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int = k(sum(n))
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int = k(sum(n))
```

It is not the correct implementation of `sum` in CPS because it depends on the original `sum` function.

Let's replace `sum(n)` with the body of `sum`.

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int = k(
  if (n <= 1) 1
  else sum(n - 1) + n
)
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int = k(
  if (n <= 1) 1
  else sum(n - 1) + n
)
```

Let's utilize the following equivalence:

```
e0(if (e1) e2 else e3)     ==     if (e1) e0(e2) else e0(e3)
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int =
  if (n <= 1) k(1)
  else k(sum(n - 1) + n)
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int =
  if (n <= 1) k(1)
  else k(sum(n - 1) + n)
```

But, it still depends on the original `sum` function.

Let's utilize the following equivalence:

```
k(sum(n - 1) + n)     ==     (x => k(x + n))(sum(n - 1))
                      ==     sumCPS(n - 1, x => k(x + n))
```

because `x => k(x + n)` is the continuation of `sum(n - 1)`.

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int =
  if (n <= 1) k(1)
  else sumCPS(n - 1, x => k(x + n))
```

Continuation-Passing Style (CPS)

```
def sum(n: Int): Int =
  if (n <= 1) 1
  else sum(n - 1) + n
```

Let's rewrite the `sum` function in CPS:

```
type Cont = Int => Int
def sumCPS(n: Int, k: Cont): Int =
  if (n <= 1) k(1)
  else sumCPS(n - 1, x => k(x + n))
```

If all functions are written in CPS, a program satisfies the properties:

- Every function takes a continuation as an explicit parameter.
- A continuation is used at most once in a function body.
- Every function call is in a tail position. (tail-call optimization)
- Every function ends with a function call.

Contents

1. Continuations

2. Continuation-Passing Style (CPS)

3. Interpreter of FAE in CPS
    - Addition and Multiplication
    - Function Application

Interpreter of FAE in CPS

The original interpreter of FAE is written in **direct style**, and continuations of the evaluation of expressions are **implicitly** represented.

```
def interp(expr: Expr, env: Env): Value = expr match
  case Num(n)    => NumV(n)
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
  case Id(x)     => env.getOrElse(x, error(s"free identifier: $x"))
  case Fun(p, b) => CloV(p, b, env)
  case App(f, a) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(a, env)))
    case v                => error(s"not a function: ${v.str}")
```

Interpreter of FAE in CPS

The original interpreter of FAE is written in **direct style**, and continuations of the evaluation of expressions are **implicitly** represented.

```
def interp(expr: Expr, env: Env): Value = expr match
  case Num(n)    => NumV(n)
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
  case Id(x)     => env.getOrElse(x, error(s"free identifier: $x"))
  case Fun(p, b) => CloV(p, b, env)
  case App(f, a) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(a, env)))
    case v                => error(s"not a function: ${v.str}")
```

To **explicitly** represent continuations of the evaluation of each expression in the interpreter of FAE, we need to modify the interpreter in **CPS**:

```
type Cont = Value => Value
def interpCPS(expr: Expr, env: Env, k: Cont): Value = ???
```

Interpreter of FAE in CPS

The original interpreter of FAE is written in **direct style**, and continuations of the evaluation of expressions are **implicitly** represented.

```
def interp(expr: Expr, env: Env): Value = expr match
  case Num(n)    => NumV(n)
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
  case Id(x)     => env.getOrElse(x, error(s"free identifier: $x"))
  case Fun(p, b) => CloV(p, b, env)
  case App(f, a) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(a, env)))
    case v                => error(s"not a function: ${v.str}")
```

To **explicitly** represent continuations of the evaluation of each expression in the interpreter of FAE, we need to modify the interpreter in **CPS**:

```
type Cont = Value => Value
def interpCPS(expr: Expr, env: Env, k: Cont): Value =
  k(interp(expr, env))
```

Interpreter of FAE in CPS

The original interpreter of FAE is written in **direct style**, and continuations of the evaluation of expressions are **implicitly** represented.

To **explicitly** represent continuations of the evaluation of each expression in the interpreter of FAE, we need to modify the interpreter in **CPS**:

```scala
type Cont = Value => Value
def interpCPS(expr: Expr, env: Env, k: Cont): Value = k(expr match
  case Num(n)    => NumV(n)
  case Add(l, r) => numAdd(interp(l, env), interp(r, env))
  case Mul(l, r) => numMul(interp(l, env), interp(r, env))
  case Id(x)     => env.getOrElse(x, error(s"free identifier: $x"))
  case Fun(p, b) => CloV(p, b, env)
  case App(f, a) => interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(a, env)))
    case v                => error(s"not a function: ${v.str}")
)
```

Interpreter of FAE in CPS

The original interpreter of FAE is written in **direct style**, and continuations of the evaluation of expressions are **implicitly** represented.

To **explicitly** represent continuations of the evaluation of each expression in the interpreter of FAE, we need to modify the interpreter in **CPS**:

```
type Cont = Value => Value
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
  case Num(n)    => k(NumV(n))
  case Add(l, r) => k(numAdd(interp(l, env), interp(r, env)))
  case Mul(l, r) => k(numMul(interp(l, env), interp(r, env)))
  case Id(x)     => k(env.getOrElse(x, error(s"free identifier: $x")))
  case Fun(p, b) => k(CloV(p, b, env))
  case App(f, a) => k(interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(a, env)))
    case v                => error(s"not a function: ${v.str}")
  )
```

Interpreter of FAE in CPS

The original interpreter of FAE is written in **direct style**, and continuations of the evaluation of expressions are **implicitly** represented.

To **explicitly** represent continuations of the evaluation of each expression in the interpreter of FAE, we need to modify the interpreter in **CPS**:

```
type Cont = Value => Value
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
  case Num(n)    => k(NumV(n))
  case Add(l, r) => k(numAdd(interp(l, env), interp(r, env)))
  case Mul(l, r) => k(numMul(interp(l, env), interp(r, env)))
  case Id(x)     => k(env.getOrElse(x, error(s"free identifier: $x")))
  case Fun(p, b) => k(CloV(p, b, env))
  case App(f, a) => k(interp(f, env) match
    case CloV(p, b, fenv) => interp(b, fenv + (p -> interp(a, env)))
    case v                => error(s"not a function: ${v.str}")
  )
```

Let's modify the `Add`, `Mul`, and `App` cases because they still use the original `interp` function.

Addition and Multiplication

```
type Cont = Value => Value
def interpCPS(expr: Expr, env: Env, k: Cont): Value = expr match
  ...
  case Add(l, r) =>
    k(numAdd(interp(l, env), interp(r, env)))
  ...
```

The current evaluation part is `interp(l, env)`.

Its continuation is `lv => k(numAdd(lv, interp(r, env)))`.

Addition and Multiplication

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

Let's rewrite it by passing the continuation into `interpCPS`.

Addition and Multiplication

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

Similarly, the current evaluation part is `interp(r, env)`.

Its continuation is `rv => k(numAdd(lv, rv))`.

Addition and Multiplication

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

Let's rewrite it by passing the continuation into `interpCPS`.

Addition and Multiplication

```
type Cont = Value => Value
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))                  // cont. of `interp(r, env)`
      })
    })
  ...
```

Addition and Multiplication

```
type Cont = Value => Value
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))
      })
    })
  case Mul(l, r) =>
    interpCPS(l, env, {
      lv => interpCPS(r, env, {
        rv => k(numMul(lv, rv))
      })
    })
  ...
```

Function Application

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

In a similar way, we can rewrite function application case.

The current evaluation part is `interp(f, env)`.

Function Application

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

Function Application

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

Let's move the continuation invocation `k(...)` into the inside of the `match` expression.

Function Application

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

We do not need to wrap `error(...)` with `k` because it does not return a value but throws an exception.

Now, the current evaluation part is `interp(a, env)`.

Function Application

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

Function Application

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

Now, the current evaluation part is `interp(b, fenv + (p -> av))`.

Function Application

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

Function Application

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

Summary

1. Continuations

2. Continuation-Passing Style (CPS)

3. Interpreter of FAE in CPS
    - Addition and Multiplication
    - Function Application

Claim of Midterm Exam

- The score for the midterm exam will be uploaded to **Blackboard** by tomorrow before noon.
- The claim hours are scheduled as follows:
  - **10/29 (Tue.)** 15:00-17:00
  - **10/30 (Wed.)** 15:00-17:00
  - **Place:** Room 609A, Science Library Bldg

Next Lecture

- Continuations (2)

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