

Lecture 17 – Compiling with Continuations

COSE212: Programming Languages

Jihyeok Park



2026 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**.
 - First-Class Continuations
 - KFAE – FAE with first-class continuations
 - Control Statements

- 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**.
 - First-Class Continuations
 - KFAE – FAE with first-class continuations
 - Control Statements
- In this lecture, let's learn **compiling with continuations**.

1. Compilers

2. Compiling with Continuations

- Continuation Passing Style

- Lambda Lifting

- Closure Conversion

- Alpha Renaming

- Transformation to Low-level IR

- Optimization of Low-level IR

1. Compilers

2. Compiling with Continuations

Continuation Passing Style

Lambda Lifting

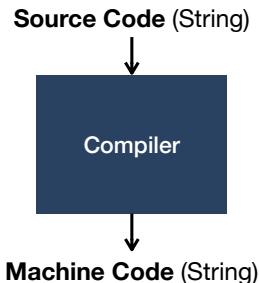
Closure Conversion

Alpha Renaming

Transformation to Low-level IR

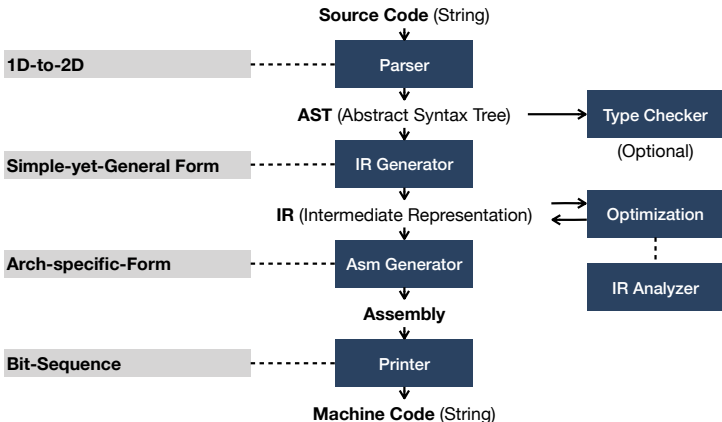
Optimization of Low-level IR

A **compiler** is a program that translates a program written in one language (the **source language**) into an equivalent program in another language (the **target language**).



Typically, the source language is a **high-level language** (e.g., Scala, Python, JavaScript, etc.) and the target language is a **low-level language** (e.g., JVM bytecode, LLVM IR, assembly, etc.).

The following figure shows a typical compilation process:



Let's focus on the **IR Generator** to learn how to compile with functional languages with continuations into a **low-level IR**.

How to compile our **functional languages** into a **low-level IR**?

```
/* FAE */  
val twice = f => {  
  a => f(f(a))  
};  
twice({  
  b => b * 2 + 1  
})(3) + 5
```

⇒

```
/* IR */  
F1:  
    mov r4, r3  
    jmp r2  
F2:  
    mov r4, F1  
    jmp r2  
F3:  
    mov r1, F2  
    jmp r2  
F4:  
    mul r1, r1, 2  
    add r1, r1, 1  
    jmp r4
```

```
F5:  
    add r1, r1, 5  
    jmp HALT  
F6:  
    mov r4, r1  
    mov r1, 3  
    mov r2, F4  
    mov r3, F5  
    jmp r4  
START:  
    mov r1, F4  
    mov r2, F6  
    jmp F3  
HALT:
```


How to compile our **functional languages** into a **low-level IR**?

```
/* FAE */  
val twice = f => {  
  a => f(f(a))  
};  
twice({  
  b => b * 2 + 1  
})(3) + 5
```

⇒

```
/* IR */  
F1:  
    mov r4, r3  
    jmp r2  
F2:  
    mov r4, F1  
    jmp r2  
F3:  
    mov r1, F2  
    jmp r2  
F4:  
    mul r1, r1, 2  
    add r1, r1, 1  
    jmp r4
```

```
F5:  
    add r1, r1, 5  
    jmp HALT  
F6:  
    mov r4, r1  
    mov r1, 3  
    mov r2, F4  
    mov r3, F5  
    jmp r4  
START:  
    mov r1, F4  
    mov r2, F6  
    jmp F3  
HALT:
```

Let's learn how to compile with **continuations**!

1. Compilers

2. Compiling with Continuations

- Continuation Passing Style

- Lambda Lifting

- Closure Conversion

- Alpha Renaming

- Transformation to Low-level IR

- Optimization of Low-level IR

Recall: Continuation-Passing Style (CPS)

We learned that **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.

We learned that **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.

For example, consider the following Scala code 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
```

We can rewrite it in **continuation-passing style** as follows:

```
type Cont = Int => Int  
def sumCPS(n: Int, k: Cont): Int =  
  if (n <= 1) k(1)  
  else sumCPS(n - 1, x => k(x + n))  
sumCPS(3, x => x * 5)      // (1 + 2 + 3) * 5 = 30
```

Continuation-Passing Style (CPS)

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val twice = f => {  
  a => f(f(a))  
};  
twice({  
  b => b * 2 + 1  
})(3) + 5
```

Continuation-Passing Style (CPS)

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val HALT = x => x;  
val twice = f => {  
  a => f(f(a))  
};  
HALT(twice({  
  b => b * 2 + 1  
})(3) + 5)
```

Let's transform the `twice` function into CPS.

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
  k1(a => f(f(a)))  
};  
twice({  
  b => b * 2 + 1  
}, x1 => HALT(x1(3) + 5))
```

Let's transform the $a \Rightarrow f(f(a))$ function into CPS.

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
  k1((a, k2) => k2(f(f(a))))  
};  
twice({  
  b => b * 2 + 1  
}, x1 => x1(3, x2 => HALT(x2 + 5)))
```

Let's transform the body of `x2 => HALT(x2 + 5)` into CPS using the syntactic sugar for `val`.

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => k2(f(f(a))))  
};  
twice({  
    b => b * 2 + 1  
}, x1 => x1(3, x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
})))
```

Let's transform the $b \Rightarrow b * 2 + 1$ function into CPS.

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
twice({  
    (b, k3) => k3(b * 2 + 1)  
}, x1 => x1(3, x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
})))
```

Let's transform the body of $(b, k3) \Rightarrow k3(b * 2 + 1)$ into CPS using the syntactic sugar for `val`.

Let's apply the CPS transformation to our running example.

(Assume that FAE is extended with multiple parameters.)

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
  k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
twice((b, k3) => {  
  val x5 = b * 2;  
  val x6 = x5 + 1;  
  k3(x6)  
}, x1 => x1(3, x2 => {  
  val x3 = x2 + 5;  
  HALT(x3)  
})))
```

This is the CPS version of our running example.

A **lambda lifting** transformation lifts nested functions to top-level functions.

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
twice((b, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6)  
}, x1 => x1(3, x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
})))
```

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
twice((b, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6)  
}, x1 => x1(3, x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
})))
```

First, let's lift the $(b, k3) \Rightarrow \dots$ function to top-level.

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
val x7 = (b, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6)  
};  
twice(x7, x1 => x1(3, x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
}))
```

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
  k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
val x7 = (b, k3) => {  
  val x5 = b * 2;  
  val x6 = x5 + 1;  
  k3(x6)  
};  
twice(x7, x1 => x1(3, x2 => {  
  val x3 = x2 + 5;  
  HALT(x3)  
})))
```

Next, let's lift the `x2 => ...` function to top-level.

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
val x7 = (b, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6)  
};  
val C1 = x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
};  
twice(x7, x1 => x1(3, C1))
```

We use the name C_k to denote that the function is a **continuation**.

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
val x7 = (b, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6)  
};  
val C1 = x2 => {  
    val x3 = x2 + 5;  
    HALT(x3)  
};  
twice(x7, x1 => x1(3, C1))
```

Let's lift the `x1 => ...` function to top-level.

Let's apply the **lambda lifting** transformation to our running example.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
  k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
val x7 = (b, k3) => {  
  val x5 = b * 2;  
  val x6 = x5 + 1;  
  k3(x6)  
};  
val C1 = x2 => {  
  val x3 = x2 + 5;  
  HALT(x3)  
};  
val C2 = x1 => x1(3, C1);  
twice(x7, C2)
```

We cannot lift the $(a, k2) \Rightarrow \dots$ and $x4 \Rightarrow \dots$ functions because f is their **captured variable** from the `twice` function.

```
/* FAE */
val HALT = x => x;
val twice = (f, k1) => {
  k1((a, k2) => f(a, x4 => f(x4, k2)))
};
val x7 = (b, k3) => {
  val x5 = b * 2;
  val x6 = x5 + 1;
  k3(x6)
};
val C1 = x2 => {
  val x3 = x2 + 5;
  HALT(x3)
};
val C2 = x1 => x1(3, C1);
twice(x7, C2)
```

Similarly, $k2$ in the $x4 \Rightarrow \dots$ function is also a **captured variable** from the $(a, k2) \Rightarrow \dots$ function.

```
/* FAE */
val HALT = x => x;
val twice = (f, k1) => {
  k1((a, k2) => f(a, x4 => f(x4, k2)))
};
val x7 = (b, k3) => {
  val x5 = b * 2;
  val x6 = x5 + 1;
  k3(x6)
};
val C1 = x2 => {
  val x3 = x2 + 5;
  HALT(x3)
};
val C2 = x1 => x1(3, C1);
twice(x7, C2)
```

To resolve this problem, we need to perform **closure conversion** by passing the captured variables as arguments to the function.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
  k1((a, k2) => f(a, x4 => f(x4, k2)))  
};  
val x7 = (b, k3) => {  
  val x5 = b * 2;  
  val x6 = x5 + 1;  
  k3(x6)  
};  
val C1 = x2 => {  
  val x3 = x2 + 5;  
  HALT(x3)  
};  
val C2 = x1 => x1(3, C1);  
twice(x7, C2)
```

There are diverse **closure conversion** algorithms, but we skip their details in this course. If we perform one of them, the result is as follows.

```
/* FAE */  
val HALT = x => x;  
val twice = (f, k1) => {  
    k1((a, f1, k2) => f1(a, f1, k2, (x4, f2, k4) => f2(x4, f2, k4, k4)))  
};  
val x7 = (b, f3, k5, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6, f3, k5)  
};  
val C1 = (x2, f4, k6) => {  
    val x3 = x2 + 5;  
    HALT(x3)  
};  
val C2 = x1 => x1(3, x7, C1);  
twice(x7, C2)
```

Finally, we can perform **lambda lifting** transformation for remaining functions as follows:

```
/* FAE */  
val HALT = x => x;  
val C3 = (x4, f2, k4) => {  
    f2(x4, f2, k4, k4)  
};  
val C4 = (a, f1, k2) => {  
    f1(a, f1, k2, C3)  
};  
val twice = (f, k1) => {  
    k1(C4)  
};
```

```
val x7 = (b, f3, k5, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6, f3, k5)  
};  
val C1 = (x2, f4, k6) => {  
    val x3 = x2 + 5;  
    HALT(x3)  
};  
val C2 = x1 => {  
    x1(3, x7, C1)  
};  
twice(x7, C2)
```

Now, our transformed code satisfies the following conditions.

- 1 Every function is in the **top-level scope**.
- 2 Every function call is in **tail position**.
- 3 Every function always **ends with function call**.

```
/* FAE */  
val HALT = x => x;  
val C3 = (x4, f2, k4) => {  
    f2(x4, f2, k4, k4)  
};  
val C4 = (a, f1, k2) => {  
    f1(a, f1, k2, C3)  
};  
val twice = (f, k1) => {  
    k1(C4)  
};
```

```
val x7 = (b, f3, k5, k3) => {  
    val x5 = b * 2;  
    val x6 = x5 + 1;  
    k3(x6, f3, k5)  
};  
val C1 = (x2, f4, k6) => {  
    val x3 = x2 + 5;  
    HALT(x3)  
};  
val C2 = x1 => {  
    x1(3, x7, C1)  
};  
twice(x7, C2)
```


To easily convert the code into the **low-level IR**, we need to perform **alpha renaming** to make every variable name unique and in a consistent manner (Fk: k -th function, xk: k -th parameter).

```
/* FAE */  
val HALT = x => x;  
val F1 = (x1, x2, x3) => {  
    x2(x1, x2, x3, x3)  
};  
val F2 = (x1, x2, x3) => {  
    x2(x1, x2, x3, F1)  
};  
val F3 = (x1, x2) => {  
    x2(F2)  
};
```

```
val F4 = (x1, x2, x3, x4) => {  
    val x5 = x1 * 2;  
    val x6 = x5 + 1;  
    x4(x6, x2, x3)  
};  
val F5 = (x1, x2, x3) => {  
    val x4 = x1 + 5;  
    HALT(x4)  
};  
val F6 = x1 => {  
    x1(3, F4, F5)  
};  
F3(F4, F6)
```

Now, we can easily convert the code into the **low-level IR**.

F1:

```
mov x1, a1
mov x2, a2
mov x3, a3
mov a1, x1
mov a2, x2
mov a3, x3
mov a4, x3
jmp x2
```

F2:

```
mov x1, a1
mov x2, a2
mov x3, a3
mov a1, x1
mov a2, x2
mov a3, x3
mov a4, F1
jmp x2
```

F3:

```
mov x1, a1
mov x2, a2
mov a1, F2
jmp x2
```

F4:

```
mov x1, a1
mov x2, a2
mov x3, a3
mov x4, a4
mul x5, x1, 2
add x6, x5, 1
mov a1, x6
mov a2, x2
mov a3, x3
jmp x4
```

F5:

```
mov x1, a1
mov x2, a2
mov x3, a3
add x4, x1, 5
mov a1, x4
jmp HALT
```

F6:

```
mov x1, a1
mov a1, 3
mov a2, F4
mov a3, F5
jmp x1
```

START:

```
mov a1, F4
mov a2, F6
jmp F3
```

HALT:

The following lines of code are actually **unnecessary**:

F1:

```
mov x1, a1
```

```
mov x2, a2
```

```
mov x3, a3
```

```
mov a1, x1
```

```
mov a2, x2
```

```
mov a3, x3
```

```
mov a4, x3
```

```
jmp x2
```

F2:

```
mov x1, a1
```

```
mov x2, a2
```

```
mov x3, a3
```

```
mov a1, x1
```

```
mov a2, x2
```

```
mov a3, x3
```

```
mov a4, F1
```

```
jmp x2
```

F3:

```
mov x1, a1
```

```
mov x2, a2
```

```
mov a1, F2
```

```
jmp x2
```

F4:

```
mov x1, a1
```

```
mov x2, a2
```

```
mov x3, a3
```

```
mov x4, a4
```

```
mul x5, x1, 2
```

```
add x6, x5, 1
```

```
mov a1, x6
```

```
mov a2, x2
```

```
mov a3, x3
```

```
jmp x4
```

F5:

```
mov x1, a1
```

```
mov x2, a2
```

```
mov x3, a3
```

```
add x4, x1, 5
```

```
mov a1, x4
```

```
jmp HALT
```

F6:

```
mov x1, a1
```

```
mov a1, 3
```

```
mov a2, F4
```

```
mov a3, F5
```

```
jmp x1
```

START:

```
mov a1, F4
```

```
mov a2, F6
```

```
jmp F3
```

HALT:

After removing all unnecessary lines of code and assign registers based on the **graph coloring** algorithm, we get the following code:

```
/* IR */  
F1:  
    mov r4, r3  
    jmp r2  
F2:  
    mov r4, F1  
    jmp r2  
F3:  
    mov r1, F2  
    jmp r2  
F4:  
    mul r1, r1, 2  
    add r1, r1, 1  
    jmp r4
```

```
F5:  
    add r1, r1, 5  
    jmp HALT  
F6:  
    mov r4, r1  
    mov r1, 3  
    mov r2, F4  
    mov r3, F5  
    jmp r4  
START:  
    mov r1, F4  
    mov r2, F6  
    jmp F3  
HALT:
```

1. Compilers

2. Compiling with Continuations

- Continuation Passing Style

- Lambda Lifting

- Closure Conversion

- Alpha Renaming

- Transformation to Low-level IR

- Optimization of Low-level IR

The MiniPython language is a toy language inspired by Python.¹

<https://github.com/ku-plrg-classroom/docs/tree/main/cose212/mini-python>

- Please see above document on GitHub:
 - ① Implement `step` function.
 - ② Implement `locals` function.
- It is just a project, and you **don't need to submit** anything.
- However, some exam questions might be related to this project, so we **strongly recommend** solving it by yourself.
- The given test cases are **not sufficient**. Please write your own test cases to test your implementation.

¹<https://www.python.org/>

- Type Systems

Jihyeok Park

`jihyeok_park@korea.ac.kr`

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