Lecture 16 – First-Class Continuations

COSE212: Programming Languages

Jihyeok Park

PLRG

2024 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
  - Small-step operational (reduction) semantics of FAE

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
  - Small-step operational (reduction) semantics of FAE

- In this lecture, we will learn **first-class continuations** and how to define the **control flow changes** in a program using them.

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
  - Small-step operational (reduction) semantics of FAE
- In this lecture, we will learn **first-class continuations** and how to define the **control flow changes** in a program using them.
- **KFAE** – FAE with **first-class continuations**
  - Interpreter and Reduction semantics

Contents

Contents

Recall: First-Class Citizen

In a programming language, an entity is said to be **first-class citizen** if it is treated as a **value**.

Recall: First-Class Citizen

In a programming language, an entity is said to be **first-class citizen** if it is treated as a **value**. In other words, it can be

1. **assigned** to a **variable**,
2. **passed** as an **argument** to a function, and
3. **returned** from a function.

Recall: First-Class Citizen

In a programming language, an entity is said to be **first-class citizen** if it is treated as a **value**. In other words, it can be

1. **assigned** to a **variable**,
2. **passed** as an **argument** to a function, and
3. **returned** from a function.

For example, Scala supports **first-class functions**.

```scala
def inc(n: Int): Int = n + 1
// 1. We can assign a function to a variable.
val f: Int => Int = inc
// 2. We can pass a function as an argument to a function.
List(1, 2, 3).map(inc)                    // List(2, 3, 4)
// 3. We can return a function from a function.
def addN(n: Int): Int => Int = m => n + m
val add3: Int => Int = addN(3)
add3(5)                                   // 3 + 5 = 8
```

First-class Continuations

Similarly, a **first-class continuation** means that a continuation is treated as a **value**.

First-class Continuations

Similarly, a **first-class continuation** means that a continuation is treated as a **value**. For example, `Racket` supports **first-class continuations**.

First-class Continuations

Similarly, a **first-class continuation** means that a continuation is treated as a **value**. For example, `Racket` supports **first-class continuations**.

In `Racket`, we can 1) **capture** the continuation using `let/cc` and 2) **change** the program's **control flow** using the captured continuation.

First-class Continuations

Similarly, a **first-class continuation** means that a continuation is treated as a **value**. For example, `Racket` supports **first-class continuations**.

In `Racket`, we can 1) **capture** the continuation using `let/cc` and 2) **change** the program's **control flow** using the captured continuation.

For example, let's change the control flow of the following program:

```
; Racket
(* 2 (+ 3 5))
```

(Note that `Racket` uses the prefix notation (e.g., `(+ 1 2)`) instead of the infix notation (e.g., `1 + 2`).)

First-class Continuations

Similarly, a **first-class continuation** means that a continuation is treated as a **value**. For example, Racket supports **first-class continuations**.

In Racket, we can 1) **capture** the continuation using let/cc and 2) **change** the program's **control flow** using the captured continuation.

For example, let's change the control flow of the following program:

```
; Racket
(* 2 (+ 3 5))
```

(Note that Racket uses the prefix notation (e.g., (+ 1 2)) instead of the infix notation (e.g., 1 + 2).)

by using the let/cc as follows:

```
; Racket
(* 2 (let/cc k (+ 3 (k 5))))    ; first-class continuation with `let/cc`
```

First-class Continuations

Similarly, a **first-class continuation** means that a continuation is treated as a **value**. For example, `Racket` supports **first-class continuations**.

In `Racket`, we can 1) **capture** the continuation using `let/cc` and 2) **change** the program's **control flow** using the captured continuation.

For example, let's change the control flow of the following program:

```
; Racket
(* 2 (+ 3 5))
```

(Note that `Racket` uses the prefix notation (e.g., `(+ 1 2)`) instead of the infix notation (e.g., `1 + 2`).)

by using the `let/cc` as follows:

```
; Racket
(* 2 (let/cc k (+ 3 (k 5))))    ; first-class continuation with `let/cc`
```

Let's compare the evaluation of the two expressions.

First-class Continuations

The original expression is evaluated in the following order:

```
; Racket
(* 2 (+ 3 5))
```

1. Evaluate 2. (Result: 2)
2. Evaluate 3. (Result: 3)
3. Evaluate 5. (Result: 5)
4. Add the results of step 2 and 3. (Result: `3 + 5 = 8`)
5. Multiply the results of step 1 and 2 – 4. (Result: `2 * 8 = 16`)

First-class Continuations

PLRG

The original expression is evaluated in the following order:

```
; Racket
(* 2 (+ 3 5))
```

1. Evaluate 2. (Result: 2)
2. Evaluate 3. (Result: 3)
3. Evaluate 5. (Result: 5)
4. Add the results of step 2 and 3. (Result: `3 + 5 = 8`)
5. Multiply the results of step 1 and 2 – 4. (Result: `2 * 8 = 16`)

What is the continuation of the expression `(+ 3 5)`?

First-class Continuations

PLRG

The original expression is evaluated in the following order:

```
; Racket
(* 2 (+ 3 5))
```

1. Evaluate 2. (Result: 2)
2. Evaluate 3. (Result: 3)
3. Evaluate 5. (Result: 5)
4. Add the results of step 2 and 3. (Result: `3 + 5 = 8`)
5. Multiply the results of step 1 and 2 – 4. (Result: `2 * 8 = 16`)

What is the continuation of the expression `(+ 3 5)`? Step 5.

First-class Continuations

The original expression is evaluated in the following order:

```
; Racket
(* 2 (+ 3 5))
```

1. Evaluate 2. (Result: 2)
2. Evaluate 3. (Result: 3)
3. Evaluate 5. (Result: 5)
4. Add the results of step 2 and 3. (Result: `3 + 5 = 8`)
5. Multiply the results of step 1 and 2 – 4. (Result: `2 * 8 = 16`)

What is the continuation of the expression `(+ 3 5)`? Step 5.

What is the continuation of the expression 5?

First-class Continuations

The original expression is evaluated in the following order:

```
; Racket
(* 2 (+ 3 5))
```

1. Evaluate 2. (Result: 2)
2. Evaluate 3. (Result: 3)
3. Evaluate 5. (Result: 5)
4. Add the results of step 2 and 3. (Result: `3 + 5 = 8`)
5. Multiply the results of step 1 and 2 – 4. (Result: `2 * 8 = 16`)

What is the continuation of the expression `(+ 3 5)`? Step 5.

What is the continuation of the expression 5? Steps 4 and 5.

First-class Continuations

The original expression is evaluated in the following order:

```
; Racket
(* 2 (+ 3 5))
```

1. Evaluate 2. (Result: 2)
2. Evaluate 3. (Result: 3)
3. Evaluate 5. (Result: 5)
4. Add the results of step 2 and 3. (Result: `3 + 5 = 8`)
5. Multiply the results of step 1 and 2 – 4. (Result: `2 * 8 = 16`)

What is the continuation of the expression `(+ 3 5)`? Step 5.

What is the continuation of the expression 5? Steps 4 and 5.

Let's 1) **capture** the continuation of `(+ 3 5)` (i.e., 5) using `let/cc` and 2) **change** the **control flow** after evaluating 5 by using it as the continuation of 5 instead of the original one (i.e., 4 and 5).

First-class Continuations

We can change the program's control flow as follows:

```
; Racket
(* 2 (let/cc k (+ 3 (k 5))))    ; first-class continuation with `let/cc`
```

1	Evaluate 2.	(Result: 2)
2	Let `k` be the continuation of 2 – 7.	(`k` is Step 8)
3	Evaluate 3.	(Result: 3)
4	Evaluate `k`.	(Result: Step 8)
5	Evaluate 5.	(Result: 5)
6	Call the result of step 4 with that of 5.	(**Replace Cont.**)
7	Add the results of step 3 and 4 – 6.	(Unreachable)
8	Multiply the results of step 1 and 2 – 7.	(Result: `2 * 5 = 10`)

First-class Continuations

We can change the program's control flow as follows:

```
; Racket
(* 2 (let/cc k (+ 3 (k 5))))   ; first-class continuation with `let/cc`
```

1	Evaluate 2.	(Result: 2)
2	Let `k` be the continuation of 2 – 7.	(`k` is Step 8)
3	Evaluate 3.	(Result: 3)
4	Evaluate `k`.	(Result: Step 8)
5	Evaluate 5.	(Result: 5)
6	Call the result of step 4 with that of 5.	(**Replace Cont.**)
7	Add the results of step 3 and 4 – 6.	(Unreachable)
8	Multiply the results of step 1 and 2 – 7.	(Result: `2 * 5 = 10`)

It means that

- Step 2 defines the continuation of steps 2 – 7 as a value in `k`.
- Step 6 replaces the continuation of step 5 with `k`.

First-class Continuations

Some functional languages support **first-class continuations**.

- Racket

```
(* 2 (let/cc k (+ 3 (k 5))))                ; 2 * 5 = 10
```

- Ruby

```
2 * (callcc { |k| 3 + k.call(5)})           # 2 * 5 = 10
```

- Haskell

```
do
  x <- callCC $ \k -> do
    y <- k 5
    return $ 3 + y
  return $ 2 * x                            -- 2 * 5 = 10
```

- ...

Contents

KFAE – FAE with First-Class Continuations

Now, let's extend FAE into KFAE with a new keyword `vcc` to capture the **first-class continuations**.

```
/* KFAE */
2 * { vcc k; 3 + k(5) }
```

KFAE – FAE with First-Class Continuations

PLRG

Now, let's extend FAE into KFAE with a new keyword `vcc` to capture the **first-class continuations**.

```
/* KFAE */
2 * { vcc k; 3 + k(5) }
```

Here is another example of KFAE:

```
/* KFAE */
{
  vcc done;
  val f = {
    vcc exit;
    2 * done(1 + {
      vcc k;
      exit(k)
    })
  };
  f(3) * 5
}
```

Concrete/Abstract Syntax

For KFAE, we need to extend **expressions** of FAE with

1. **first-class continuations** (`vcc`)

Concrete/Abstract Syntax

For KFAE, we need to extend **expressions** of FAE with

1. **first-class continuations** (`vcc`)

We can extend the **concrete syntax** of FAE as follows:

```
// expressions
<expr> ::= ... | "vcc" <id> ";" <expr>
```

Concrete/Abstract Syntax

For KFAE, we need to extend **expressions** of FAE with

1. **first-class continuations** (vcc)

We can extend the **concrete syntax** of FAE as follows:

```
// expressions
<expr> ::= ... | "vcc" <id> ";" <expr>
```

and the **abstract syntax** of FAE as follows:

$$\text{Expressions} \quad \mathbb{E} \ni e ::= \ldots \mid \texttt{vcc}\ x;\ e \qquad (\texttt{Vcc})$$

```
enum Expr:
  ...
  // first-class continuations
  case Vcc(name: String, body: Expr)
```

Contents

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */

2 * { vcc k; 3 + k(5) }
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) }
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) } // 2 * 5 = 10
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) } // 2 * 5 = 10
```

```
/* KFAE */
{
  vcc done;
  val f = {
    vcc exit;
    2 * done(1 + {
      vcc k;
      exit(k)
    })
  };
  f(3) * 5
}
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) } // 2 * 5 = 10
```

```
/* KFAE */
{
  vcc done;        // done = x => x
  val f = {
    vcc exit;
    2 * done(1 + {
      vcc k;
      exit(k)
    })
  };
  f(3) * 5
}
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) } // 2 * 5 = 10
```

```
/* KFAE */
{
  vcc done;         // done = x => x
  val f = {
    vcc exit;       // exit = y => val f = y; f(3) * 5
    2 * done(1 + {
      vcc k;
      exit(k)
    })
  };
  f(3) * 5
}
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) } // 2 * 5 = 10
```

```
/* KFAE */
{
  vcc done;        // done = x => x
  val f = {
    vcc exit;      // exit = y => val f = y; f(3) * 5
    2 * done(1 + {
      vcc k;       // k    = z => val f = { 2 * done(1 + z) }; f(3) * 5
      exit(k)
    })
  };
  f(3) * 5
}
```

KFAE – FAE with First-Class Continuations

Then, what is the expected result of the following KFAE expressions?

```
/* KFAE */
// k is a continuation can be represented as `x => 2 * x`
2 * { vcc k; 3 + k(5) } // 2 * 5 = 10
```

```
/* KFAE */
{
  vcc done;          // done = x => x
  val f = {
    vcc exit;        // exit = y => val f = y; f(3) * 5
    2 * done(1 + {
      vcc k;         // k    = z => val f = { 2 * done(1 + z) }; f(3) * 5
      exit(k)
    })
  };
  f(3) * 5
}                    // 4 (= 1 + 3)
```

Recall: Interpreter and Reduction Sem. for FAE

In the previous lecture, we have defined the **first-order representation** of **continuations** with **value stack**:

```
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]
```

$$
\begin{array}{llll}
\text{Continuations} & \mathbb{K} \ni \kappa ::= & \square & (\texttt{EmptyK}) \\
 & & \mid (\sigma \vdash e) :: \kappa & (\texttt{EvalK}) \\
 & & \mid (+) :: \kappa & (\texttt{AddK}) \\
 & & \mid (*) :: \kappa & (\texttt{MulK}) \\
 & & \mid (@) :: \kappa & (\texttt{AppK}) \\
\text{Value Stacks} & \mathbb{S} \ni s ::= & \blacksquare \mid v :: s & (\texttt{List[Value]})
\end{array}
$$

Recall: Interpreter and Reduction Sem. for FAE

Then, we have defined the **reduction relation** $\rightarrow \in (\mathbb{K} \times \mathbb{S}) \times (\mathbb{K} \times \mathbb{S})$ between **states** consisting of pairs of **continuations** and **value stacks**:

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

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

Recall: Interpreter and Reduction Sem. for FAE

Then, we have defined the **reduction relation** $\rightarrow \in (\mathbb{K} \times \mathbb{S}) \times (\mathbb{K} \times \mathbb{S})$ between **states** consisting of pairs of **continuations** and **value stacks**:

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

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

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

```
def eval(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
```

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

Interpreter and Reduction Semantics for KFAE

Now, let's extend the interpreter and reduction semantics for FAE to KFAE by adding the **first-class continuations**.

First, we need to extend the values of FAE with **continuation values** consisting of pairs of continuations and value stacks:

```
// values
enum Value:
  case NumV(number: BigInt)
  case CloV(param: String, body: Expr, env: Env)
  case ContV(cont: Cont, stack: Stack)
```

$$
\begin{array}{llll}
\text{Values} & \mathbb{V} \ni v ::= & n & (\texttt{NumV}) \\
& & \mid \langle \lambda x.e, \sigma \rangle & (\texttt{CloV}) \\
& & \mid \langle \kappa \mid\mid s \rangle & (\texttt{ContV})
\end{array}
$$

Then, let's fill out the missing cases in the `reduce` function and reduction rules for $\rightarrow$ in the reduction semantics of KFAE.

First-Class Continuations

```
def reduce(k: Cont, s: Stack): (Cont, Stack) = (k, s) match
  case (EvalK(env, expr, k), s) => expr match
    ...
    case Vcc(x, b) => (EvalK(env + (x -> ContV(k, s)), b, k), s)
```

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

$$\texttt{Vcc} \quad \langle (\sigma \vdash \texttt{vcc}\ x;\ e) :: \kappa \mid\mid s \rangle \quad \rightarrow \quad \langle (\sigma[x \mapsto \langle \kappa \mid\mid s \rangle] \vdash e) :: \kappa \mid\mid s \rangle$$

It defines a new immutable binding x in the environment σ that maps to a **continuation value** $\langle \kappa \mid\mid s \rangle$, and then evaluates the body expression e in the extended environment $\sigma[x \mapsto \langle \kappa \mid\mid s \rangle]$.

Function Application

```scala
def reduce(k: Cont, s: Stack): (Cont, Stack) = (k, s) match
  case (EvalK(env, expr, k), s) => expr match
    ...
    case App(f, e) => (EvalK(env, f, EvalK(env, e, AppK(k))), s)
  ...
  case (AppK(k), a :: f :: s) => f match
    case CloV(p, b, fenv) => (EvalK(fenv + (p -> a), b, k), s)
    case ContV(k1, s1)    => (k1, a :: s1)
    case v                => error(s"not a function: ${v.str}")
```

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

$$\begin{array}{llcl}
\texttt{App}_1 & \langle (\sigma \vdash e_1(e_2)) :: \kappa \mid\mid s \rangle & \rightarrow & \langle (\sigma \vdash e_1) :: (\sigma \vdash e_2) :: (@) :: \kappa \mid\mid s \rangle \\
\texttt{App}_{2,\lambda} & \langle (@) :: \kappa \mid\mid v_2 :: \langle \lambda x.e, \sigma \rangle :: s \rangle & \rightarrow & \langle (\sigma[x \mapsto v_2] \vdash e) :: \kappa \mid\mid s \rangle \\
\texttt{App}_{2,\kappa} & \langle (@) :: \kappa \mid\mid v_2 :: \langle \kappa' \mid\mid s' \rangle :: s \rangle & \rightarrow & \langle \kappa' \mid\mid v_2 :: s' \rangle
\end{array}$$

The new $\texttt{App}_{2,\kappa}$ rule handles when the function expression evaluates to a continuation value $\langle \kappa' \mid\mid s' \rangle$. It changes the control flow to the continuation κ' with the given argument value v_2 and the value stack s'.

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$\langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square \qquad\qquad || \blacksquare \qquad\qquad \rangle$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rl}
& \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square \quad || \ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square \quad || \ \blacksquare \ \rangle
\end{array}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rlll}
& \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ 2 :: \blacksquare & \rangle
\end{array}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rlll}
 & \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \|\ 2 :: \blacksquare & \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \ \|\ s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rll}
 & \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \mid\mid \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \mid\mid \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \mid\mid 2 :: \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \mid\mid 2 :: \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \mid\mid 2 :: \blacksquare \quad \rangle
\end{array}
$$

$$
\text{where} \left\{
\begin{array}{l}
\sigma_0 = [k \mapsto \langle \kappa_0 \mid\mid s_0 \rangle] \\
\kappa_0 = (*) :: \square \\
s_0 = 2 :: \blacksquare
\end{array}
\right.
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ 2 :: \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \|\ 2 :: \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 2 :: \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare \quad \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \ \|\ s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rll}
 & \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \mid\mid\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \mid\mid\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \mid\mid\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \mid\mid\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \mid\mid\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \mid\mid\ 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_0 \vdash k) :: (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \mid\mid\ 3 :: 2 :: \blacksquare \ \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \mid\mid s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \|\ \blacksquare \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ \blacksquare \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ 2 :: \blacksquare \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \|\ 2 :: \blacksquare \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 2 :: \blacksquare \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_0 \vdash k) :: (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare \rangle \\
\xrightarrow{(\texttt{Id})} & \langle\ (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \|\ \langle \kappa_0 \ \|\ s_0 \rangle :: 3 :: 2 :: \blacksquare \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \ \|\ s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \|\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_0 \vdash k) :: (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Id})} & \langle\ (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \|\ \langle \kappa_0 \ \|\ s_0 \rangle :: 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (@) :: (+) :: (*) :: \square & \|\ 5 :: \langle \kappa_0 \ \|\ s_0 \rangle :: 3 :: 2 :: \blacksquare \ \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \ \|\ s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rlll}
& \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & \|\ 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & \|\ 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_0 \vdash k) :: (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \|\ 3 :: 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Id})} & \langle\ (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & \|\ \langle \kappa_0 \ \|\ s_0 \rangle :: 3 :: 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (@) :: (+) :: (*) :: \square & \|\ 5 :: \langle \kappa_0 \ \|\ s_0 \rangle :: 3 :: 2 :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_{2,\kappa})} & \langle\ (*) :: \square & \|\ 5 :: 2 :: \blacksquare & \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \ \|\ s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Example 1

Let's interpret the expression $2 * (\texttt{vcc}\ k;\ (3 + k(5)))$:

$$
\begin{array}{rll}
 & \langle\ (\varnothing \vdash 2 * (\texttt{vcc}\ k;\ (3 + k(5)))) :: \square & ||\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\varnothing \vdash 2) :: (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & ||\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\varnothing \vdash (\texttt{vcc}\ k;\ (3 + k(5)))) :: (*) :: \square & ||\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_0 \vdash (3 + k(5))) :: (*) :: \square & ||\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Add}_1)} & \langle\ (\sigma_0 \vdash 3) :: (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & ||\ 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\sigma_0 \vdash k(5)) :: (+) :: (*) :: \square & ||\ 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_0 \vdash k) :: (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & ||\ 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Id})} & \langle\ (\sigma_0 \vdash 5) :: (@) :: (+) :: (*) :: \square & ||\ \langle \kappa_0 \ ||\ s_0 \rangle :: 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (@) :: (+) :: (*) :: \square & ||\ 5 :: \langle \kappa_0 \ ||\ s_0 \rangle :: 3 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_{2,\kappa})} & \langle\ (*) :: \square & ||\ 5 :: 2 :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_2)} & \langle\ \square & ||\ 10 :: \blacksquare \ \rangle
\end{array}
$$

$$
\text{where} \begin{cases} \sigma_0 = [k \mapsto \langle \kappa_0 \ ||\ s_0 \rangle] \\ \kappa_0 = (*) :: \square \\ s_0 = 2 :: \blacksquare \end{cases}
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$\langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square \qquad || \ \blacksquare \qquad \rangle$$

where $\left\{ \right.$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \|\ \blacksquare \quad \rangle \\
\overset{(\texttt{App}_1)}{\rightarrow} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \blacksquare \quad \rangle
\end{array}
$$

where $\Bigg\{$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rlll}
 & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \|\ \blacksquare & \rangle \\
\overset{(\texttt{App}_1)}{\to} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \blacksquare & \rangle \\
\overset{(\texttt{Fun})}{\to} & \langle\ (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare & \rangle
\end{array}
$$

$$
\text{where} \left\{ \begin{array}{lcl} e_0 & = & \texttt{vcc}\ r;\ r(x + 1) * 2 \\ & & \end{array} \right.
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rlll}
 & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\texttt{@}) :: \square & \|\ 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare & \rangle
\end{array}
$$

$$
\text{where} \left\{ \begin{array}{lcl} e_0 & = & \texttt{vcc}\ r;\ r(x + 1) * 2 \\ & & \end{array} \right.
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x \texttt{ + } 1) \texttt{ * } 2))(3)$:

$$
\begin{array}{rlll}
 & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x \texttt{ + } 1) \texttt{ * } 2))(3)) :: \square & \| \ \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x \texttt{ + } 1) \texttt{ * } 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \| \ \blacksquare & \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \| \ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\texttt{@}) :: \square & \| \ 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare & \rangle \\
\xrightarrow{(\texttt{App}_{2,\lambda})} & \langle\ (\sigma_0 \vdash \texttt{vcc}\ r;\ r(x \texttt{ + } 1) \texttt{ * } 2) :: \square & \| \ \blacksquare & \rangle
\end{array}
$$

$$
\text{where} \begin{cases} e_0 = \texttt{vcc}\ r;\ r(x \texttt{ + } 1) \texttt{ * } 2 \\ \sigma_0 = [x \mapsto 3] \end{cases}
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rll}
 & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & ||\ \blacksquare\ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & ||\ \blacksquare\ \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (\texttt{@}) :: \square & ||\ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare\ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\texttt{@}) :: \square & ||\ 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare\ \rangle \\
\xrightarrow{(\texttt{App}_{2,\lambda})} & \langle\ (\sigma_0 \vdash \texttt{vcc}\ r;\ r(x + 1) * 2) :: \square & ||\ \blacksquare\ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_1 \vdash r(x + 1) * 2) :: \square & ||\ \blacksquare\ \rangle
\end{array}
$$

$$
\text{where} \left\{ \begin{array}{lcl}
e_0 & = & \texttt{vcc}\ r;\ r(x + 1) * 2 \\
\sigma_0 & = & [x \mapsto 3] \\
\sigma_1 & = & \sigma_0[r \mapsto \langle \square \ ||\ \blacksquare \rangle]
\end{array} \right.
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (@) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (@) :: \square & \|\ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (@) :: \square & \|\ 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_{2,\lambda})} & \langle\ (\sigma_0 \vdash \texttt{vcc}\ r;\ r(x + 1) * 2) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_1 \vdash r(x + 1) * 2) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\sigma_1 \vdash r(x + 1)) :: (\sigma_1 \vdash 2) :: (@) :: (*) :: \square & \|\ \blacksquare \ \rangle
\end{array}
$$

$$
\text{where} \begin{cases} e_0 & = & \texttt{vcc}\ r;\ r(x + 1) * 2 \\ \sigma_0 & = & [x \mapsto 3] \\ \sigma_1 & = & \sigma_0[r \mapsto \langle \square \ \|\ \blacksquare \rangle] \end{cases}
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rll}
 & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \mid\mid \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\texttt{@}) :: \square & \mid\mid 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_{2,\lambda})} & \langle\ (\sigma_0 \vdash \texttt{vcc}\ r;\ r(x + 1) * 2) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_1 \vdash r(x + 1) * 2) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\sigma_1 \vdash r(x + 1)) :: (\sigma_1 \vdash 2) :: (\texttt{@}) :: (*) :: \square & \mid\mid \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_1 \vdash r) :: (\sigma_1 \vdash x + 1) :: (\texttt{@}) :: (\sigma_1 \vdash 2) :: (*) :: \square & \mid\mid \blacksquare \ \rangle \\
\dots & &
\end{array}
$$

$$
\text{where} \begin{cases} e_0 & = \quad \texttt{vcc}\ r;\ r(x + 1) * 2 \\ \sigma_0 & = \quad [x \mapsto 3] \\ \sigma_1 & = \quad \sigma_0[r \mapsto \langle \square \mid\mid \blacksquare \rangle] \end{cases}
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (\texttt{@}) :: \square & \|\ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (\texttt{@}) :: \square & \|\ 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_{2,\lambda})} & \langle\ (\sigma_0 \vdash \texttt{vcc}\ r;\ r(x + 1) * 2) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_1 \vdash r(x + 1) * 2) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\sigma_1 \vdash r(x + 1)) :: (\sigma_1 \vdash 2) :: (\texttt{@}) :: (*) :: \square & \|\ \blacksquare \quad \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_1 \vdash r) :: (\sigma_1 \vdash x + 1) :: (\texttt{@}) :: (\sigma_1 \vdash 2) :: (*) :: \square & \|\ \blacksquare \quad \rangle \\
\cdots & & \\
\rightarrow^* & \langle\ (\texttt{@}) :: (\sigma_1 \vdash 2) :: (*) :: \square & \|\ 4 :: \langle \square \ \|\ \blacksquare \rangle :: \blacksquare \quad \rangle
\end{array}
$$

$$
\text{where} \begin{cases} e_0 &= \texttt{vcc}\ r;\ r(x + 1) * 2 \\ \sigma_0 &= [x \mapsto 3] \\ \sigma_1 &= \sigma_0[r \mapsto \langle \square \ \|\ \blacksquare \rangle] \end{cases}
$$

Example 2

Let's interpret the expression $(\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)$:

$$
\begin{array}{rll}
& \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))(3)) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\varnothing \vdash (\lambda x.(\texttt{vcc}\ r;\ r(x + 1) * 2))) :: (\varnothing \vdash 3) :: (@) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Fun})} & \langle\ (\varnothing \vdash 3) :: (@) :: \square & \|\ \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Num})} & \langle\ (@) :: \square & \|\ 3 :: \langle \lambda x.e_0, \varnothing \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_{2,\lambda})} & \langle\ (\sigma_0 \vdash \texttt{vcc}\ r;\ r(x + 1) * 2) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Vcc})} & \langle\ (\sigma_1 \vdash r(x + 1) * 2) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{Mul}_1)} & \langle\ (\sigma_1 \vdash r(x + 1)) :: (\sigma_1 \vdash 2) :: (@) :: (*) :: \square & \|\ \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_1)} & \langle\ (\sigma_1 \vdash r) :: (\sigma_1 \vdash x + 1) :: (@) :: (\sigma_1 \vdash 2) :: (*) :: \square & \|\ \blacksquare \ \rangle \\
\dots & & \\
\rightarrow^* & \langle\ (@) :: (\sigma_1 \vdash 2) :: (*) :: \square & \|\ 4 :: \langle \square \ \| \ \blacksquare \rangle :: \blacksquare \ \rangle \\
\xrightarrow{(\texttt{App}_{2,\kappa})} & \langle\ \square & \|\ 4 :: \blacksquare \ \rangle
\end{array}
$$

$$
\text{where} \begin{cases}
e_0 & = & \texttt{vcc}\ r;\ r(x + 1) * 2 \\
\sigma_0 & = & [x \mapsto 3] \\
\sigma_1 & = & \sigma_0[r \mapsto \langle \square \ \| \ \blacksquare \rangle]
\end{cases}
$$

Contents

Control Statements

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

Control Statements

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
```

Let's represent them using **first-class continuations**!

Control Statements

- `return` statement:

```
x => body
```

Control Statements

- `return` statement:

```
x => body
```

means

```
x => { vcc return;
  body // return(e) directly returns e to the caller
}
```

Control Statements

- `return` statement:

```
x => body
```

means

```
x => { vcc return;
  body // return(e) directly returns e to the caller
}
```

- `break` and `continue` statements:

```
while (cond) body
```

Control Statements

- `return` statement:

```
x => body
```

means

```
x => { vcc return;
  body // return(e) directly returns e to the caller
}
```

- `break` and `continue` statements:

```
while (cond) body
```

means

```
{ vcc break;
  while (cond) { vcc continue;
    body // continue(e)/break(e) jumps to the next/end of the loop
  }
}
```

Control Statements

We can represent other control statements similarly, but think for yourself!

- `exception` in `Python`

```
try:
    x = y / z
except ZeroDivisionError:
    x = 0
```

- `generator` in `JavaScript`

```
const foo = function* () { yield 'a'; yield 'b'; yield 'c'; };
let str = '';
for (const c of foo()) { str = str + c; }
str // 'abc'
```

- `coroutines` in `Kotlin`
- `async`/`await` in `C#`
- ...

Summary

Exercise #10

https://github.com/ku-plrg-classroom/docs/tree/main/cose212/kfae

- Please see above document on GitHub:
  - 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.

Next Lecture

- Compiling with Continuations

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