5118014 Programming Language Theory

Ch 15. Continuation

Shin Hong

Motivation

- Many real-world programming languages support control diverter
  - basically, an expression is evaluated only after all its sub-expressions are evaluated
  - control diverters such as `return`, `break`, `continue`, `goto`, `throw` alters the flow of evaluations in various ways for convenient programming

- MFAE has no feature of control diverter, and the big-step semantics is not suitable of expressing such control diverters
  - a control diverter is intended to make a side-effect on the status of execution

Example: numOfWordsInFile

```
def numOfWordsInFile(name: String, word: String): Int = {
  val content =
    if (cached(name))
      getCache(name)
    else if (exists(name))
      read(name)
    else
      return -1
  numOfWords(content, word)
}
```

```
def numOfWordsInFile(name: String, word: String): Int = {
  if (cached(name))
    numOfWords(getCache(name), word)
  else if (exists(name))
    numOfWords(read(name), word)
  else
    -1
}
```

Approach

- We can decompose an evaluation of an expression into multiple steps
  - Ex. (1 + 2) + (3 + 4)

1. Evaluate the expression 1 to get the integer value 1.
2. Evaluate the expression 2 to get the integer value 2.
3. Add the integer value 1 to the integer value 2 to get the integer value 3.
4. Evaluate the expression 3 to get the integer value 3.
5. Evaluate the expression 4 to get the integer value 4.
6. Add the integer value 3 to the integer value 4 to get the integer value 7.
7. Add the integer value 3 to the integer value 7 to get the integer value 10.

- Control the execution order of the steps
- Define the semantics by specifying the order of steps

Redex and Continuation

- Decompose an expression into a redex and a continuation
  - a continuation is a function mapping a value to a value at a program state, which captures the next computation at a certain state of the program execution
  - e.g., (1 + 2) + (3 + 4) → 1 and (□ + 2) + (3 + 4), or 1 and $\lambda v.(v + 2) + (3 + 4)$

- Evaluating an expression is first evaluating a redex and then applying the redex value to the continuation

Continuation-passing Style (CPS) Programming

- CPS is a style of programming that passes and uses the remaining computation when an expression is evaluated
  - with the given continuation, an expression evaluation can determine the subsequent computation

- Using continuation, we can define an evaluation function (i.e., `interp`) to make every function call as a tail call

```scala
def factorial(n: Int): Int =
  if (n <= 1)
    1
  else
    n * factorial(n - 1)
```

```scala
type Cont = Int => Int

def factorialCps(n: Int, k: Cont): Int =
  if (n <= 1)
    k(1)
  else
    factorialCps(n - 1, x => k(n * x))
```

```scala
factorialCps(1, (v => v))
factorialCps(3, (v => v))
```

Interpreter of FAE

```
def interp(e: Expr, env: Env): Value = e match {
  case Num(n) => NumV(n)
  case Add(l, r) =>
    val v1 = interp(l, env)
    val v2 = interp(r, env)
    val NumV(n) = v1
    val NumV(m) = v2
    NumV(n + m)
  case Sub(l, r) =>
    val v1 = interp(l, env)
    val v2 = interp(r, env)
    val NumV(n) = v1
    val NumV(m) = v2
    NumV(n - m)
  case Id(x) => env(x)
  case Fun(x, b) => CloV(x, b, env)
  case App(f, a) =>
    val fv = interp(f, env)
    val av = interp(a, env)
    val CloV(x, b, fEnv) = fv
    interp(b, fEnv + (x -> av))
}
```

Interpreter of FAE in CPS

```
type Cont = Value => Value
```

```
def interpCps(e: Expr, env: Env, k: Cont): Value = e match {
  ...
}
```

```
interpCps(e, Map(), (v => v))
```

Interpreter of FAE in CPS

- `case Num(n) => k(NumV(n))`
- `case Id(x) => k(env(x))`
- `case Fun(x, b) => k(CloV(x, b, env))`
- `case Add(l, r) =>`

```
    interpCps(l, env, v1 =>
      interpCps(r, env, v2 => {
        val NumV(n) = v1
        val NumV(m) = v2
        k(NumV(n + m))
      })
    )
```

Interpreter of FAE in CPS

- ```
  case App(f, a) =>
    interpCps(f, env, fv =>
      interpCps(a, env, av => {
        val CloV(x, b, fEnv) = fv
        interpCps(b, fEnv + (x -> av), k)
      })
    )
  ```
  ```

Small-step Operational Semantics

- Big-step semantics specify to which value an expression is evaluated, thus it is not suitable of specifying the order of evaluation steps
  - e.g.,

$$\frac{\sigma \vdash e_1 \Rightarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2}$$

- Small-step operational semantics defines a relation between states and states, and specify how an expression is evaluated to a value
  - e.g., (1 + 2) + (3 + 4) is reduced to 3 + (3 + 4), to 3 + 7, to 10

State and Reduction

- A state of FAE is a pair of a computation stack and a value stack
  - a state is denoted as $k||s$
  - $k \quad ::= \quad \square \mid \sigma \vdash e :: k \mid (+) :: k \mid (-) :: k \mid (@) :: k$
  - $s \quad ::= \quad \blacksquare \mid v :: s$

- Reduction is a relation over $K \times S \times K \times S$
  - $\rightarrow \ \subseteq \ K \times S \times K \times S$
  - $k_1||s_1 \rightarrow k_2||s_2$, $k_1||s_1 \rightarrow^* k_9||s_9$

Big-step and Small-step Operational Semantics

$$\forall \sigma.\forall e.\forall v.(\sigma \vdash e \Rightarrow v) \leftrightarrow (\sigma \vdash e :: \square \mid\mid \blacksquare \rightarrow^{*} \square \mid\mid v :: \blacksquare)$$

$$\forall \sigma.\forall e.\forall v.\forall k.\forall s.(\sigma \vdash e \Rightarrow v) \leftrightarrow (\sigma \vdash e :: k \mid\mid s \rightarrow^{*} k \mid\mid v :: s)$$

FAE Semantics in Small-step

```
case Num(n) => k(NumV(n))
```

$$\sigma \vdash n :: k \mid\mid s \rightarrow k \mid\mid n :: s \quad \text{[Red-Num]}$$

```
case Id(x) => k(env(x))
```

$$\sigma \vdash x :: k \mid\mid s \rightarrow k \mid\mid \sigma(x) :: s \quad \text{[Red-Id]}$$

```
case Fun(x, b) => k(CloV(x, b, env))
```

$$\sigma \vdash \lambda x.e :: k \mid\mid s \rightarrow k \mid\mid \langle \lambda x.e, \sigma \rangle :: s \quad \text{[Red-Fun]}$$

FAE Semantics in Small-step

```
case Add(l, r) =>
  interpCps(l, env, v1 =>
    interpCps(r, env, v2 =>
      k(add(v1, v2))
    )
  )
```

$$\sigma \vdash e_1 + e_2 :: k \;||\; s \rightarrow \sigma \vdash e_1 :: \sigma \vdash e_2 :: (+) :: k \;||\; s \quad \text{[Red-Add1]}$$

$$(+) :: k \;||\; n_2 :: n_1 :: s \rightarrow k \;||\; n_1 + n_2 :: s \quad \text{[Red-Add2]}$$

$$\begin{array}{lrcr} & \sigma \vdash e_1 + e_2 :: k & || & s \\ \rightarrow & \sigma \vdash e_1 :: \sigma \vdash e_2 :: (+) :: k & || & s \\ \rightarrow^* & \sigma \vdash e_2 :: (+) :: k & || & n_1 :: s \\ \rightarrow^* & (+) :: k & || & n_2 :: n_1 :: s \\ \rightarrow & k & || & n_1 + n_2 :: s \end{array}$$

Example

$$
\begin{array}{rrcr}
 & \emptyset \vdash (1+2)-(3+4) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 1+2 :: \emptyset \vdash 3+4 :: (-) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 1 :: \emptyset \vdash 2 :: (+) :: \emptyset \vdash 3+4 :: (-) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 2 :: (+) :: \emptyset \vdash 3+4 :: (-) :: \square & \| & 1 :: \blacksquare \\
\rightarrow & (+) :: \emptyset \vdash 3+4 :: (-) :: \square & \| & 2 :: 1 :: \blacksquare \\
\rightarrow & \emptyset \vdash 3+4 :: (-) :: \square & \| & 3 :: \blacksquare \\
\rightarrow & \emptyset \vdash 3 :: \emptyset \vdash 4 :: (+) :: (-) :: \square & \| & 3 :: \blacksquare \\
\rightarrow & \emptyset \vdash 4 :: (+) :: (-) :: \square & \| & 3 :: 3 :: \blacksquare \\
\rightarrow & (+) :: (-) :: \square & \| & 4 :: 3 :: 3 :: \blacksquare \\
\rightarrow & (-) :: \square & \| & 7 :: 3 :: \blacksquare \\
\rightarrow & \square & \| & -4 :: \blacksquare
\end{array}
$$

FAE Semantics in Small-step

```
case App(f, a) =>
  interpCps(f, env, fv =>
    interpCps(a, env, av => {
      val CloV(x, b, fEnv) = fv
      interpCps(b, fEnv + (x -> av), k)
    })
  )
```

$$\sigma \vdash e_1\ e_2 :: k \mid\mid s \rightarrow \sigma \vdash e_1 :: \sigma \vdash e_2 :: (@) :: k \mid\mid s \quad [\text{Red-App1}]$$

$$(@) :: k \mid\mid v :: \langle \lambda x.e, \sigma \rangle :: s \rightarrow \sigma[x \mapsto v] \vdash e :: k \mid\mid s \quad [\text{Red-App2}]$$

$$\begin{array}{llcr}
 & \sigma \vdash e_1\ e_2 :: k & \mid\mid & s \\
\rightarrow & \sigma \vdash e_1 :: \sigma \vdash e_2 :: (@) :: k & \mid\mid & s \\
\rightarrow^* & \sigma \vdash e_2 :: (@) :: k & \mid\mid & \langle \lambda x.e, \sigma' \rangle :: s \\
\rightarrow^* & (@) :: k & \mid\mid & v_2 :: \langle \lambda x.e, \sigma' \rangle :: s \\
\rightarrow & \sigma'[x \mapsto v_2] \vdash e :: k & \mid\mid & s \\
\rightarrow^* & k & \mid\mid & v :: s
\end{array}$$

Example

$$
\begin{array}{lrcr}
 & \emptyset \vdash e\ 1\ 2 :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash e\ 1 :: \emptyset \vdash 2 :: (@) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash e :: \emptyset \vdash 1 :: (@) :: \emptyset \vdash 2 :: (@) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 1 :: (@) :: \emptyset \vdash 2 :: (@) :: \square & \| & \langle e, \emptyset \rangle :: \blacksquare \\
\rightarrow & (@) :: \emptyset \vdash 2 :: (@) :: \square & \| & 1 :: \langle e, \emptyset \rangle :: \blacksquare \\
\rightarrow & \sigma_1 \vdash \lambda \mathsf{y}.\mathsf{x} + \mathsf{y} :: \emptyset \vdash 2 :: (@) :: \square & \| & \blacksquare \\
\rightarrow & \emptyset \vdash 2 :: (@) :: \square & \| & \langle \lambda \mathsf{y}.\mathsf{x} + \mathsf{y}, \sigma_1 \rangle :: \blacksquare \\
\rightarrow & (@) :: \square & \| & 2 :: \langle \lambda \mathsf{y}.\mathsf{x} + \mathsf{y}, \sigma_1 \rangle :: \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{x} + \mathsf{y} :: \square & \| & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{x} :: \sigma_2 \vdash \mathsf{y} :: (+) :: \square & \| & \blacksquare \\
\rightarrow & \sigma_2 \vdash \mathsf{y} :: (+) :: \square & \| & 1 :: \blacksquare \\
\rightarrow & (+) :: \square & \| & 2 :: 1 :: \blacksquare \\
\rightarrow & \square & \| & 3 :: \blacksquare
\end{array}
$$

Example

- 1+$((\lambda x. 2 + x)$ 3)

```
def interpCps (e: Expr, env: Env, k: Cont): Value = {
  e match {
    case Num(n) => k(NumV(n))
    case Id(x) => k(env(x))
    case Add(l, r) =>
      interpCps(l, env, lv => {
        interpCps(r, env, rv => {
          val NumV(nl) = lv
          val NumV(nr) = rv
          k(NumV(nl + nr))
        })
      })
    case App(f, a) =>
      interpCps(f, env, fv => {
        interpCps(a, env, av => {
          val CloV(x, b, fenv) = fv
          interpCps(b, fEnv + (x -> av), k)
        })
      })
  }
}
```

```scala
def interp(e: Expr, env: Env): Value = e match {
  case Num(n) => NumV(n)
  case Add(l, r) =>
    val v1 = interp(l, env)
    val v2 = interp(r, env)
    val NumV(n) = v1
    val NumV(m) = v2
    NumV(n + m)
  case Sub(l, r) =>
    val v1 = interp(l, env)
    val v2 = interp(r, env)
    val NumV(n) = v1
    val NumV(m) = v2
    NumV(n - m)
  case Id(x) => env(x)
  case Fun(x, b) => CloV(x, b, env)
  case App(f, a) =>
    val fv = interp(f, env)
    val av = interp(a, env)
    val CloV(x, b, fEnv) = fv
    interp(b, fEnv + (x -> av))
}
```

```scala
def interp (k: List[Comp], s: List[Value]): Value = {
  val h::t = k

  h match {
    case (expr, env) => {
      expr match {
        case Num(n) => interp(t, NumV(n)::s)
        ...
        case Add(l,r) => interp((l,env)::(r,env)::Plus::k, s)
        case App(f,a) => interp((f,env)::(a,env)::At::k, s)
      }
    }
    case Plus => {
      val v2::v1::st = s
      val Num(n2) = v2
      val Num(n1) = v1
      interp(t, NumV(n1 + n2)::st)
    }
    ...
    case At => {
      val av::fv::st = s
      val CloV(x, b, fenv) = fv
      interp( (b, fenv + (x -> av))::k, st )
    }
  }
}
```

- 1+(($\lambda x. 2 + x$) 3)

```
def interp (k: List[Comp], s: List[Value]): Value = {
  val h::t = k

  h match {
    case (expr, env) => {
      expr match {
        case Num(n) => interp(t, NumV(n)::s)
        ...
        case Add(l,r) => interp((l,env)::(r,env)::Plus::k, s)
        case App(f,a) => interp((f,env)::(a,env)::At::k, s)
      }
    }
    case Plus => {
      val v2::v1::st = s
      val Num(n2) = v2
      val Num(n1) = v1
      interp(t, NumV(n1 + n2)::st)
    }
    ...
    case At => {
      val av::fv::st = s
      val CloV(x, b, fenv) = fv
      interp( (b, fenv + (x -> av))::k, st )
    }
  }
}
```