5118014 Programming Language Theory

Ch 14. Lazy Evaluation

Shin Hong

Lazy Evaluation

- Lazy evaluation means delaying the evaluation of an expression until the result is required
  - lazy evaluation vs. eager evaluation
  - e.g., function arguments, variable definition

- In the Call-By-Name (CBN) semantics, each argument is passed as the expression itself
  - rather than the value of the expression

Examples

```
def square(x: Int): Int = {
  Thread.sleep(5000)  // models some expensive computation
  x * x
}
```

`x:=>Int` by-name parameter

```
def foo(x: Int, b: Boolean): Int =
  if (b) x else 0

val file = new java.io.File("a.txt")
foo(square(10), file.exists)
```

LFAE

- Change the eager semantics of function application into lazy
  - the argument must be passed as an expression, not value
- Introduce a new kind of values, expression-value

$$v ::= \cdots \mid (e, \sigma)$$

- Refine the semantics of function application, and add another form of evaluation, strict evaluation

$$\Downarrow \subseteq V \times V$$

Strict Evaluation (1/2)

- $v_1 \Downarrow v_2$ is true iff v_1 strictly evaluates to a normal value v_2

$$n \Downarrow n \quad \text{[Strict-Num]} \qquad \langle \lambda x.e, \sigma \rangle \Downarrow \langle \lambda x.e, \sigma \rangle \quad \text{[Strict-Clo]}$$

$$\frac{\sigma \vdash e \Rightarrow v_1 \qquad v_1 \Downarrow v_2}{(e, \sigma) \Downarrow v_2} \quad \text{[Strict-Expr]}$$

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow n_1 \qquad \sigma \vdash e_2 \Rightarrow v_2 \qquad v_2 \Downarrow n_2}{\sigma \vdash e_1 + e_2 \Rightarrow n_1 + n_2} \quad \text{[Add]}$$

Strict Evaluation (2/2)

- $v_1 \Downarrow v_2$ is true iff v_1 strictly evaluates to a normal value v_2

$$\frac{\sigma \vdash e_1 \Rightarrow v_1 \qquad v_1 \Downarrow \langle \lambda x.e, \sigma' \rangle \qquad \sigma'[x \mapsto (e_2, \sigma)] \vdash e \Rightarrow v}{\sigma \vdash e_1\ e_2 \Rightarrow v}\ [\textsc{App}]$$

CBV vs. CBN

- For FAE, if evaluating an expression in CBV results in a value, it yields the same value in CBN
  - this is true because a FAE expression does not have any side-effects

- There are expression that yields results only in CBN, not in CBV
  - e.g., a function application whose argument is a non-terminating expression

Interpreter

```
sealed trait Value
...
case class ExprV(e: Expr, env: Env) extends Value


def strict(v: Value): Value = v match {
  case ExprV(e, env) => strict(interp(e, env))
  case _ => v
}
```

```
def interp(e: Expr, env: Env): Value = e match {
  ...
  case Add(l, r) =>
    val NumV(n) = strict(interp(l, env))
    val NumV(m) = strict(interp(r, env))
    NumV(n + m)
  case Sub(l, r) =>
    val NumV(n) = strict(interp(l, env))
    val NumV(m) = strict(interp(r, env))
    NumV(n - m)
  case App(f, a) =>
    val CloV(x, b, fEnv) = strict(interp(f, env))
    interp(b, fEnv + (x -> ExprV(a, env)))
}
```

Call-by-Need: Motivation

- The current implementation of CBN make redundant calculation when a by-name parameter is evaluated multiple times
  - e.g.,

```
def square(x: Int): Int = {
  Thread.sleep(5000)  // models some expensive computation
  x * x
}

def bar(x: => Int, b: Boolean): Int =
  if (b) x + x else 0
```

Call-by-Need: Solution

- Store the value of an argument when an expression-value is first used, and afterward reuse the stored value
- In purely functional languages, the behaviors in call-by-need is identical to call-by-name

Call-by-Need: Implementation

```
case class ExprV(
  e: Expr, env: Env, var v: Option[Value]
) extends Value


def strict(v: Value): Value = v match {
  case ExprV(_, _, Some(cache)) => cache
  case ev @ ExprV(e, env, None) =>
    val cache = strict(interp(e, env))
    ev.v = Some(cache)
    cache
  case _ => v
}
```