Concepts of Programming Languages

Course: CSE2120 Edition: 2019-2020

Available from January 26, 2020 until July 3, 2020

Your Enrollment

You're taking this course for a grade

- Your Course Dossier
- Your Submissions

Unenroll

Course Information

- Home
- All editions
- News archive
- Course rules
- Lecture notes
- Assignments

Course staff

Lecturers

- Casper Poulsen
- Eelco Visser

Assistants

- Ali Al-Kaswan
- Yana Angelova
- Wesley Baartman
- Kirti Biharie
- Philippos Boon Alexaki
- Luc Everse
- Boris Janssen
- Rembrandt Klazinga
- Mirco Kroon
- Chris Lemaire
- Sterre Lutz
- Yaniv Oren
- Wouter Polet
- Thijs Raijmakers
- Jim van Vliet
- Yoshi van den Akker
- Paul van der Stel
- Eric van der Toorn

1.8. Week 8: Lazy Evaluation

In this week's assignments, you will be implementing an interpreter with *lazy evaluation*, based on chapter 17.1 of the book.
The interpreter is for a language with built-in support for lists and recursion (i.e., recursion is no longer treated as syntactic sugar). The language does not have built-in support for mutation.

1 Features to Implement

This week's interpreter is a variation of the environment-based interpreter from week 4. You can probably reuse and adapt some of the code from that interpreter, but be aware of the differences.

1.1 Lazy Evaluation

Implement lazy evaluation using thunks as described in the book.
You should use the classes summarized below.

Your interpreter should use *memoization*: once a thunked computation has been interpreted, the interpreter memoizes the value so that each thunk is never interpreted more than once.

Choose sensible *strictness points*. For example, evaluation of expressions in the function position of a function application expression should be strict. You should implement a `strict` function that takes a value as input and, if the input value is a thunk, evaluates the expression to a value and returns the result. Evaluation of expressions in the argument positions of a function application expression should be lazy.

For example, an expression `(f e1 e2 e3)` should:
1. evaluate the expression `f` to a function closure `v`;
2. construct a thunk value for each `e1`, `e2`, and `e3`; and
3. evaluate the body of the function closure `v` using the environment of the closure, updated with `Bind`ings for each argument thunk.

Your language should also be extended with a `force` construct that adds first-class support for strictness to your otherwise lazy language. An expression `(force e)` should be treated as a strictness point that eagerly and *recursively* coerces all thunks into values.

1.2 Recursion as a Construct

Implement first-class support for recursion in your language.

The semantics of `letrec` is similar to the `letrec` construct from week 5. That is, it should use the "create, update, use" approach from the book.

You should implement `letrec` by exploiting Scala's support for mutation: that is, you can mutate the `Bind` objects in environments directly in Scala, instead of returning a new store representing the mutated memory state.

You should use `UninitializedV` for environment locations that have been created but not yet updated.

`letrec` should interpret binding expressions lazily (call-by-need).

1.3 Lists and Laziness

Like in previous weeks, your language should support linked lists, using `nil` for the empty list, and `cons` for constructing a single link of a list.

1.3.1 Lazy Interpretation of `cons`

Interpreting a `(cons e1 e2)` expression should thunk each of its two argument expressions, where `e1` is an expression that computes the head of the list, and `e2` is an expression that computes the tail of a list.

1.3.2 Strictness

The `strict` function should evaluate thunks to values. You should use this function to execute previously delayed computations when needed.

1.3.3 Force

Lists are composed of (possibly thunked) computations. Ensure that your implementation of `force` recursively applies strictness to list elements.

`force` should be implemented such that the result of the following program is the value `ConsV(NumV(1), NilV())`:

```
(force ((lambda (x y) (cons x y)) 1 nil))
```

2 Examples

2.1 Stream of Ones

The following expression constructs a lazily computed infinite list of `1` s and returns the first 10 of these:

```
(letrec ((ones (cons 1 ones))
         (take
           (lambda (n xs)
             (if (num= n 0)
               nil
               (cons (head xs) (take (- n 1) (tail xs)))))))
  (force (take 10 ones)))
```

The `take` function constructs a finite list with the first `n` element of a list `xs` (or fails if `xs` is shorter than `n`).

2.2 Natural Numbers

The following expression constructs an infinite series containing all of the natural numbers, and takes the first 10:

```
(letrec ((mk-nats (lambda (n) (cons n (mk-nats (+ n 1)))))
         (nats (mk-nats 0))
         (take
           (lambda (n xs)
             (if (num= n 0)
               nil
               (cons (head xs) (take (- n 1) (tail xs)))))))
  (force (take 10 nats)))
```

An alternative approach is to use a `zip-with` function and the `ones` stream from earlier:

```
(letrec ((ones (cons 1 ones))

         (nats (cons 0 (zip-with plus ones nats)))

         (plus (lambda (n m) (+ n m)))

         (zip-with
           (lambda (f xs ys)
             (if (is-nil xs)
               nil
               (cons
                 (f (head xs) (head ys))
                 (zip-with f (tail xs) (tail ys))))))

         (take
           (lambda (n xs)
             (if (num= n 0)
               nil
               (cons (head xs) (take (- n 1) (tail xs)))))))

  (force (take 10 nats)))
```

2.3 Factorial Numbers

We can reuse the `zip-with` function to construct the infinite series of factorial numbers:

```
(letrec ((ones (cons 1 ones))

         (nats (cons 0 (zip-with plus ones nats)))

         (facs (cons 1 (zip-with times (tail nats) facs)))

         (plus (lambda (n m) (+ n m)))
         (times (lambda (n m) (* n m)))

         (zip-with
           (lambda (f xs ys)
             (if (and (is-nil xs) (is-nil ys))
               nil
               (cons
                 (f (head xs) (head ys))
                 (zip-with f (tail xs) (tail ys))))))

         (take
           (lambda (n xs)
             (if (num= n 0)
               nil
               (cons (head xs) (take (- n 1) (tail xs)))))))

  (force (take 10 facs)))
```

3 Grammar

```
module lazy-functions

imports Common

context-free syntax

  Expr.NumExt       = INT      // integer literals
  Expr.TrueExt      = [true]
  Expr.FalseExt     = [false]
  Expr.IdExt        = ID

  Expr.UnOpExt      = [([UnOp] [Expr])]
  Expr.BinOpExt     = [([BinOp] [Expr] [Expr])]

  UnOp.MIN          = [-]
  UnOp.NOT          = [not]
  UnOp.FORCE        = [force]
  UnOp.HEAD         = [head]
  UnOp.TAIL         = [tail]
  UnOp.ISNIL        = [is-nil]
  UnOp.ISLIST       = [is-list]

  BinOp.PLUS        = [+]
  BinOp.MULT        = [*]
  BinOp.MINUS       = [-]
  BinOp.AND         = [and]
  BinOp.OR          = [or]
  BinOp.NUMEQ       = [num=]
  BinOp.NUMLT       = [num<]
  BinOp.NUMGT       = [num>]
  BinOp.CONS        = [cons]

  Expr.IfExt        = [(if [Expr] [Expr] [Expr])]

  Expr.FdExt        = [(lambda ([ID*]) [Expr])]
  Expr.AppExt       = [([Expr] [Expr*])]
  Expr.LetExt       = [(let ([LetBind+]) [Expr])]
  Expr.LetRecExt    = [(letrec ([LetBind+]) [Expr])]

  Expr.NilExt       = [nil]
  Expr.ListExt      = [(list [Expr*])]

  LetBind.LetBind   = [([ID] [Expr])]
```

4 Classes

4.1 Abstract Syntax

```
sealed abstract class ExprExt
case class TrueExt() extends ExprExt
case class FalseExt() extends ExprExt
case class NumExt(num: Int) extends ExprExt
case class BinOpExt(s: String, l: ExprExt, r: ExprExt) extends ExprExt
case class UnOpExt(s: String, e: ExprExt) extends ExprExt
case class IfExt(c: ExprExt, t: ExprExt, e: ExprExt) extends ExprExt
case class ListExt(l: List[ExprExt]) extends ExprExt
case class NilExt() extends ExprExt
case class AppExt(f: ExprExt, args: List[ExprExt]) extends ExprExt
case class IdExt(c: String) extends ExprExt
case class FdExt(params: List[String], body: ExprExt) extends ExprExt
case class LetExt(binds: List[LetBindExt], body: ExprExt) extends ExprExt
case class LetRecExt(binds: List[LetBindExt], body: ExprExt) extends ExprExt

case class LetBindExt(name: String, value: ExprExt)

object ExprExt {
  val binOps = Set("+", "*", "-", "and", "or", "num=", "num<", "num>", "cons")
  val unOps = Set("-", "not", "force", "head", "tail", "is-nil", "is-list")
  val reservedWords = binOps ++ unOps ++ Set("if", "lambda", "let", "true", "false", "nil", "list", "letrec")
}
```

4.2 Desugared Syntax

```
sealed abstract class ExprC
case class TrueC() extends ExprC
case class FalseC() extends ExprC
case class NumC(num: Int) extends ExprC
case class PlusC(l: ExprC, r: ExprC) extends ExprC
case class MultC(l: ExprC, r: ExprC) extends ExprC
case class IfC(c: ExprC, t: ExprC, e: ExprC) extends ExprC
case class EqNumC(l: ExprC, r: ExprC) extends ExprC
case class LtC(l: ExprC, r: ExprC) extends ExprC
case class AppC(f: ExprC, args: List[ExprC]) extends ExprC
case class IdC(c: String) extends ExprC
case class FdC(params: List[String], body: ExprC) extends ExprC
case class LetRecC(binds: List[LetBindC], body: ExprC) extends ExprC
case class ForceC(e: ExprC) extends ExprC
case class NilC() extends ExprC
case class ConsC(hd: ExprC, tl: ExprC) extends ExprC
case class HeadC(e: ExprC) extends ExprC
case class TailC(e: ExprC) extends ExprC
case class IsNilC(e: ExprC) extends ExprC
case class IsListC(e: ExprC) extends ExprC

case class LetBindC(name: String, value: ExprC)
```

4.3 Values

```
sealed abstract class Value
case class NumV(v: Int) extends Value
case class BoolV(v: Boolean) extends Value

import java.util.UUID.randomUUID // for generating a random ID and hash code

case class ClosV(f: FdC, env: List[Bind]) extends Value {
  override def toString: String = s"ClosV($f, <env>)"
  override def hashCode(): Int = randomUUID().hashCode()
}
case class ThunkV(var value: Either[(ExprC, List[Bind]), Value]) extends Value {
  override def toString: String = value match {
    case Left((e, _)) => s"ThunkV($e, <env>)"
    case Right(v) => s"ThunkV($v)"
  }
  override def hashCode(): Int = randomUUID().hashCode()
}
case class ConsV(head: Value, tail: Value) extends Value
case class NilV() extends Value
case class UninitializedV() extends Value
```

The `Either[(ExprC, List[Bind]), Value]` type is used to represent a thunk that *either* records an expression and the environment under which that expression is closed *or* a value.

The `var` annotation on the `value` argument of a `ThunkV` denotes a mutable field.

Since environments may be *cyclic*, the `ClosV` and `ThunkV` case classes are defined to override the `toString` and `hashCode` methods, which might otherwise attempt a divergent traversal of a cyclic environment.

4.4 Other

```
case class Bind(name: String, var value: Value)
type Environment = List[Bind]
```

Note that `Bind`ings are now mutable to allow the updates of `LetRec` values.

5 Exceptions

Specific exceptions should be created that *inherit* from the given abstract exceptions.
Creating specific exceptions for each case makes debugging a lot easier and are more informative.
Throw only exceptions derived from `ParseException` in the parser, `DesugarException` in the desugarer, and so on.

```
abstract class ParseException   extends RuntimeException
abstract class DesugarException extends RuntimeException
abstract class InterpException  extends RuntimeException
```