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

1.7. Week 7: Objects as Desugarings

You will be implementing an interpreter for objects, based on chapter 1-4 of OOPLAI. The objective is to define objects by desugaring into the language of Week 5

1 Features to implement

This week's interpreter is based of the mutation interpreter for the core language of week 5, with the difference that the interpreter has support for string literals, string comparison, and string concatenation. The parser is given to you; your task is to implement a desugaring function which desugars objects into the language of week 5, and to extend the interpreter with support for strings. The rest of this section describes how objects are supposed to behave.

1.1 Object Creation

The language you will be implementing must include syntax for creating an object, by declaring its fields and methods. The following expression illustrates how to create an `object` with two `field` s (`x` and `y` , both with initial values `0`) and four `method` s (getters and setters for each of the two fields):

```
(object
  ((field x 0)
   (field y 0))
  ((method get-x () x)
   (method get-y () y)
   (method set-x (nx) (set x nx))
   (method set-y (ny) (set y ny))))
```

1.2 Object Scoping and Field Initialization

In the example above, the initial values of fields is given by the expression `0` , which we call the *field initialization expression*. All field names that an object defines must be in scope for every field initialization expression. Method names are not in scope for field initialization expressions, but `self` is; see below. Field initialization expressions can be arbitrary; see the grammar. The evaluation order of field initialization expressions is left to right.

1.3 Sending Messages to Objects

The methods of an object provide an interface for interacting with the state of the object, by sending objects messages. Objects interpret a message by applying the method by the same name as the message carries, or yields an error (`InterpException`) if the object does not know how to interpret the message. The following expression creates an `object` , bound to `point` , and passes the messages `set-x` and then `get-x` to the object, to yield the result `42` :

```
(let ((point
        (object
          ((field x 0)
           (field y 0))
          ((method get-x () x)
           (method get-y () y)
           (method set-x (nx) (set x nx))
           (method set-y (ny) (set y ny))))))
  (seq (msg point set-x 42)
       (msg point get-x)))
```

1.4 Object Encapsulation

Objects encapsulate state.
The only way to interact with the state that an object encapsulates is via the interface given by the methods of the object. The following expression yields an error (`InterpException`), because the object does not know how to interpret the message `x` :

```
(let ((point
        (object
          ((field x 0)
           (field y 0))
          ((method get-x () x)
           (method get-y () y)
           (method set-x (nx) (set x nx))
           (method set-y (ny) (set y ny))))))
  (msg point x))
```

The only kinds of messages that an object can interpret are given by the methods of the object. Field data is encapsulated.

1.5 Self

Objects can be self-referential; that is, methods and field initialization expressions of an object can reference the object that they reside in via a `self` identifier. The following expression creates a price object with a value field, and three methods: (1) a getter method; (2) a setter method; and (3) a method that takes another object as argument, compares the state of the value fields for the argument and current object, and returns the cheaper of the two objects; possibly the current object, referenced via the `self` identifier:

```
(let ((point1
        (object
          ((field val 1))
          ((method get-value () val)
           (method set-value (nv) (set val nv))
           (method compare (p)
             (if (num< (msg p get-value) (msg self get-value))
               p
               self)))))
      (point2
        (object
          ((field val 2))
          ((method get-value () val)
           (method set-value (nv) (set val nv))
           (method compare (p)
             (if (num< (msg p get-value) (msg self get-value))
               p
               self))))))
  (msg (msg point1 compare point2) get-value))
```

The program above returns `1` .

The `self` identifier is in scope for both field initialization expressions and method bodies. However, `self` is only initialized *after* all fields have been initialized. So the following program yields an interpretation error:

```
(object
  ((field me (msg self forty-two)))
  ((method forty-two () 42)))
```

Whereas the following program returns the self:

```
(msg (object ((field me (lambda () self)))
             ((method m () (me))))
     m)
```

1.6 Message Forwarding

(This feature is not mandatory to pass the assignment with a minimum grade.)

It is useful to be able to forward messages to another object. Implement a syntactic form for an object that does *method delegation*. That is, if the object receives a message that it does not know how to interpret, it forwards the message to a delegatee object. The delegatee object is given at object creation time. For example, the following expression creates a `point2d` object and a `point3d` , and `point3d` forwards messages to a `point2d` object. Evaluating the expression returns `3` .

```
(letrec ((point2d
            (object
              ((field x 0)
               (field y 0))
              ((method get-x () x)
               (method get-y () y)
               (method set-x (nx) (set x nx))
               (method set-y (ny) (set y ny)))))
         (point3d
            (object-del point2d
              ((field z 0))
              ((method get-z () z)
               (method set-z (nz) (set z nz))))))
  (seq (msg point3d set-x 3)
       (msg point3d get-x)))
```

Note that `object` s must have at least one field, whereas `object-del` s can have zero or more (see the grammar below).

1.7 Late Binding of Self

(This feature is not mandatory to pass the assignment with a minimum grade.)

In order to reuse functionality between objects, it is useful to support *late binding of self* (as also explained in OOPLAI). For example, consider the following expression:

```
(letrec ((vehicle-factory
            (lambda ()
              (object
                ((field position 1))
                ((method speed-factor () 1)
                 (method get-position () position)
                 (method move () (set position (* position
                                                  (msg self speed-factor)))))))))
         (car
            (object-del
              (vehicle-factory)
              ()
              ((method speed-factor () 2))))
         (bicycle
            (object-del
              (vehicle-factory)
              ()
              ((method speed-factor () 4)))))
  (seq
    (msg car move)
    (seq
      (msg bicycle move)
      (cons
        (msg car get-position)
        (cons
          (msg bicycle get-position)
          nil)))))
```

The following expression binds `vehicle-factory` to an *object factory*; i.e., a function which returns a "fresh" vehicle object each time it is called, where each vehicle object has a method `move` which calls a method `speed-factor` to decide how far a given vehicle can move within a given time frame. The expression above also defines two objects `bicycle` and `car` which both implement their own `speed-factor` methods, and delegate to vehicle instances produced by `vehicle-factory` , thereby reusing the `move` method for calculating how far a `bicycle` and a `car` can travel in a given time frame.

The expression returns the list `(cons 4 (cons 2 nil))` .

`do-seq` Expressions

(This feature is not mandatory to pass the assignment with a minimum grade.)

`do-seq` expressions take a sequence of expressions and evaluates them in left-to-right order. It evaluates to the value of the last expression in the sequence. `do-seq` expressions should always have at least one subexpression. (The parser enforces this.)

For example, the following code evaluates to `2` :

```
(let ((x 0))
  (do-seq
    (set x 1)
    (set x (+ x x))
    x))
```

2 Desugaring

You are given a parser for the language given by the grammar below, and you are given an interpreter for the core expression language summarized further below. Your job is to implement the desugarer from the abstract syntax into the core expression language.

Objects should desugar into closures that encapsulate the state of fields, and that implement the semantics of field access, field initialization, and method calling summarized in the assignment notes above.

Message expressions such as `(msg obj m e1 ... en)` , where `e1...en` are the argument expressions of the message `m` to be passed to the object `obj` , should desugar into a core language method invocation akin to an expression like `((obj "m") <...>)` . Here, the `<...>` includes the argument expressions `e1...en` , and (if you implement object delegation) the notion of self for the method. In other words, objects should desugar into functions, following OOPLAI.

3 Grammar

```
module mutation

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.HEAD         = [head]
  UnOp.TAIL         = [tail]
  UnOp.ISNIL        = [is-nil]
  UnOp.ISLIST       = [is-list]

  UnOp.BOX          = [box]
  UnOp.UNBOX        = [unbox]

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

  BinOp.SETBOX      = [setbox]
  BinOp.SEQ         = [seq]

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

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

  Expr.FdExt        = [(lambda ([ID*]) [Expr])]
  Expr.AppExt       = [([Expr] [Expr*])]
  Expr.LetExt       = [(let ([LetBind+]) [Expr])]
  Expr.Set          = [(set [ID] [Expr])]
  LetBind.LetBindExt = [([ID] [Expr])]

  Expr.RecLamExt    = [(rec-lam [ID] ([ID]) [Expr])]

  Expr.LetRecExt     = [(letrec ([LetBind+]) [Expr])]

  Expr.StringExt    = STRING // string literal values, marked by double quotes; i.e., "somestring"
  Expr.ObjectExt    = [(object ([FieldExt+]) ([MethodExt+]))]
  Expr.ObjectDelExt = [(object-del [Expr] ([FieldExt*]) ([MethodExt+]))]
  Expr.MsgExt       = [(msg [Expr] [ID] [Expr*])]
  Expr.DoSeqExt     = [(do-seq [Expr+])]

  Object.FieldExt   = [(field [ID] [Expr])]
  Object.MethodExt  = [(method [ID] ([ID*]) [Expr])]
```

Note that [Expr+] denotes one or more of [Expr].

Also note that `object` s must have at least one field, whereas `object-del` s can have zero or more.

4 Classes

These classes should be used in your solution.

4.1 Abstract Syntax

The abstract syntax is postfixed with Ext for extended syntax. The language is the same as the language of Week 5, extended with:

- `StringExt` and binary operations `str=` and `str++` for string equality and string concatenation;
- `ObjectExt` for creating an object;
- `ObjectDelExt` for creating a delegating object;
- `MsgExt` for sending a message to an object.
- `DoSeqExt` for sequencing multiple expressions.

```
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, a: 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 SetExt(id: String, e: ExprExt) extends ExprExt
case class RecLamExt(name: String,
                     param: String,
                     body: ExprExt) extends ExprExt
case class LetRecExt(binds: List[LetBindExt],
                     body: ExprExt) extends ExprExt

case class StringExt(str: String) extends ExprExt
case class ObjectExt(fields: List[FieldExt], methods: List[MethodExt]) extends ExprExt
case class ObjectDelExt(del: ExprExt, fields: List[FieldExt], methods: List[MethodExt]) extends ExprExt
case class FieldExt(name: String, value: ExprExt)
case class MethodExt(name: String, args: List[String], body: ExprExt)
case class MsgExt(recvr: ExprExt, msg: String, args: List[ExprExt]) extends ExprExt
case class DoSeqExt(expr: List[ExprExt]) extends ExprExt

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

object ExprExt {
  val binOps = Set("+", "*", "-", "and", "or", "num=", "num<", "num>",
    "cons", "setbox", "seq", "str=", "str++")
  val unOps = Set("-", "not", "head", "tail", "is-nil", "is-list", "box", "unbox")
  val reservedWords = binOps ++ unOps ++ Set("list", "if", "lambda",
    "let", "true", "false", "rec-lam", "set", "letrec",
    "object", "field", "method", "msg", "self", "do-seq")
}
```

4.2 Desugared Syntax

The desugared syntax is postfixed with C for core syntax. The language is the same as the language of Week 5, extended with `StringC` for creating a string literal and `EqStrC` for comparing computed literals.

```
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 NilC() extends ExprC
case class ConsC(l: ExprC, r: 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 AppC(f: ExprC, as: List[ExprC]) extends ExprC
case class IdC(c: String) extends ExprC
case class FdC(params: List[String], body: ExprC) extends ExprC
case class UndefinedC() extends ExprC
case class BoxC(v: ExprC) extends ExprC
case class UnboxC(b: ExprC) extends ExprC
case class SetboxC(b: ExprC, v: ExprC) extends ExprC
case class SetC(v: String, b: ExprC) extends ExprC
case class SeqC(b1: ExprC, b2: ExprC) extends ExprC
case class UninitializedC() extends ExprC
case class EqStrC(l: ExprC, r: ExprC) extends ExprC
case class ConcStrC(l: ExprC, r: Expr) extends ExprC
case class StringC(str: String) extends ExprC
```

4.3 Values

```
abstract class Value
case class NumV(v: Int) extends Value
case class BoolV(v: Boolean) extends Value
case class NilV() extends Value
case class ConsV(hd: Value, tl: Value) extends Value
case class PointerClosV(f: FdC, env: List[Pointer]) extends Value
case class UninitializedV() extends Value
case class BoxV(l: Int) extends Value
case class StringV(name: String) extends Value
```

4.4 Other

```
case class Pointer(name: String, location: Int)
case class Cell(location: Int, value: Value)
```