Lecture 14

Data Abstraction

Objects, inheritance, prototypes

Ras Bodik
Shaon Barman
Thibaud Hottelier

Hack Your Language!

CS164: Introduction to Programming Languages and Compilers, Spring 2012

UC Berkeley

Announcement

Ras will hold a review tomorrow 11-12 in the Woz

topics: parsing and coroutines; bring your questions

Project Proposals due Sunday

I will post remaining slides tonight

Where are we?

Our new constructs concerned **control abstraction:**

→ hiding complex (changes) to program control flow under suitable programming constructs

- lazy iterators, built on coroutines
- backtracking in regexes, built with coroutines
- search for a proof tree, hidden in the Prolog interpreter

There must also be **data abstraction**. A few examples:

Objects
Algebraic Data Types
Dictionaries
Relational Database

(even elementary data types like ints / floats)

Objects (review from CS61B)

Why objects?

abstraction: hide implementation under encapsulation

Why inheritance?

reuse: specialization of an object's behavior reuses its code

Our Design Rationale

We want to support objects

What is the minimum base language to support objects?

Our language already supports closures

which are similar in that they carry state *and* code

Can we build objects from this existing mechanism?

rather than adding support for objects into base language?

Single-Method Approach

We have seen closure-based objects already

Where did we use closures as objects?

iterators

Iterators are single-method objects

- on each call, iterators return the next element and "advance" their iterator state

Use of single-method object

```
d = newObject(0)
print d("get")     --> 0
d("set", 10)
print d("get")     --> 10
```

Multi-method object represented as a closure

```
function newObject (value)
    function (action, v) {
        if (action == "get“) {
            value
        } else if (action == "set“)  {
            value = v
        } else {
            error("invalid action")
}   }   }
```

Questions: how do we support inheritance, privacy?

Objects as tables

Recall tables

Create a table

```
{}
{ key1 = value1, key2 = value2 }
```

Add a key-value pair to table (or overwrite a k/w pair)

```
t = {}
t[key] = value
```

Read a value given a key

```
x = t[key]
```

Object as a table of attributes

```
Account = {balance = 0}

Account[“withdraw”] = function(v) {
    Account[“balance”] = Account[“balance”] - v
}

Account[“withdraw”](100.00)
```

What syntactic sugar we add to clean this up?

Account.withdraw → Account["withdraw"]

Sugar design

we discussed the choice of :

E ::= E . ID

or

E ::= E . E

Syntactic sugar

`p.f  →  p["f"]  →  get(p, "f")`

Careful: we need to distinguish between reading p.f

translated to get

and writing into p.f

translated to put

Object as a table of attributes, revisited

```
Account = {balance = 0}

function Account.withdraw (v) {
    Account.balance = Account.balance - v
}
Account.withdraw(100.00)

a = Account
Account = nil

a.withdraw(100.00)    -- ERROR!
```

refers to object via variable Account!

trickier error than a=nil or a.withdraw = nil

Introduce `self`

```
Account = {balance = 0}
function Account.withdraw (self, v) {
   self.balance = self.balance - v
}

a1 = Account
Account = nil
a1.withdraw(a1, 100.00)   -- OK

a2 = {balance=0, withdraw = Account.withdraw}
a2.withdraw(a2, 260.00)
```

The colon notation

```
function Account:withdraw (v) {
    self.balance = self.balance - v
}

a:withdraw(100.00)
```

How to desugar?

do we introduce a rule

E ::= E:E or

E:ID or

E:ID(ARGs) ?

Rewriting E:ID()

work out your SDT rules here

Discussion

What is the inefficiency of our current objects?

too much space wasted by each object carrying its objects and fields that are constant across many objects

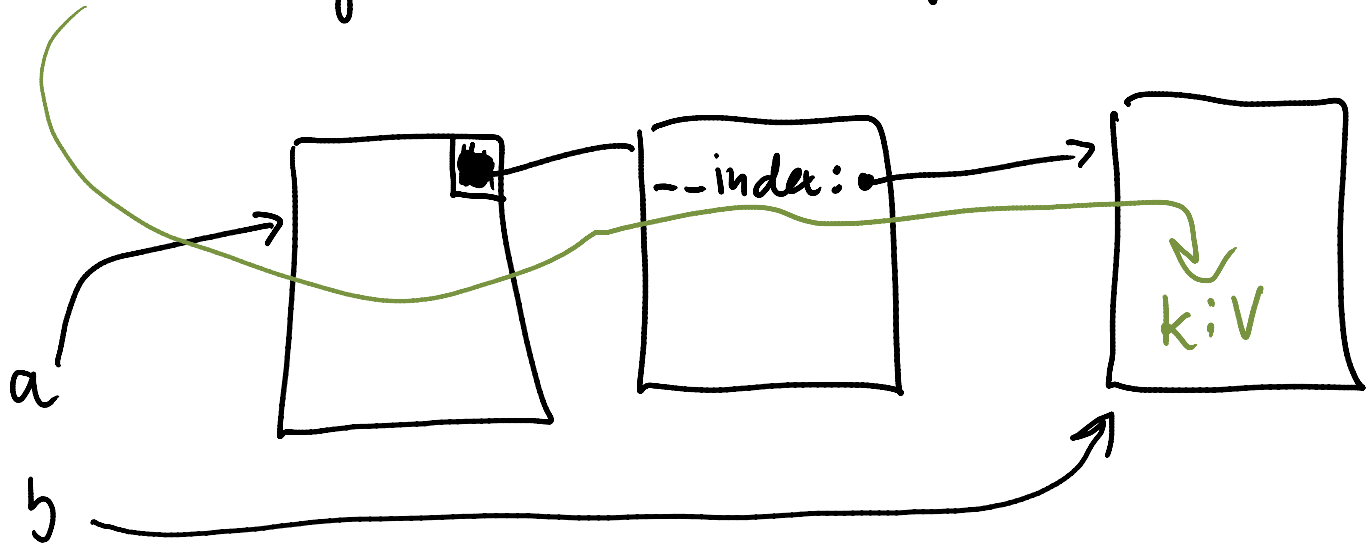
Meta-Methods

The __index metamethod

When a lookup of a field fails, interpreter consults the __index field:

```
setmetatable(a, {__index = b})
```

a[k] finds the (k,v) pair in b

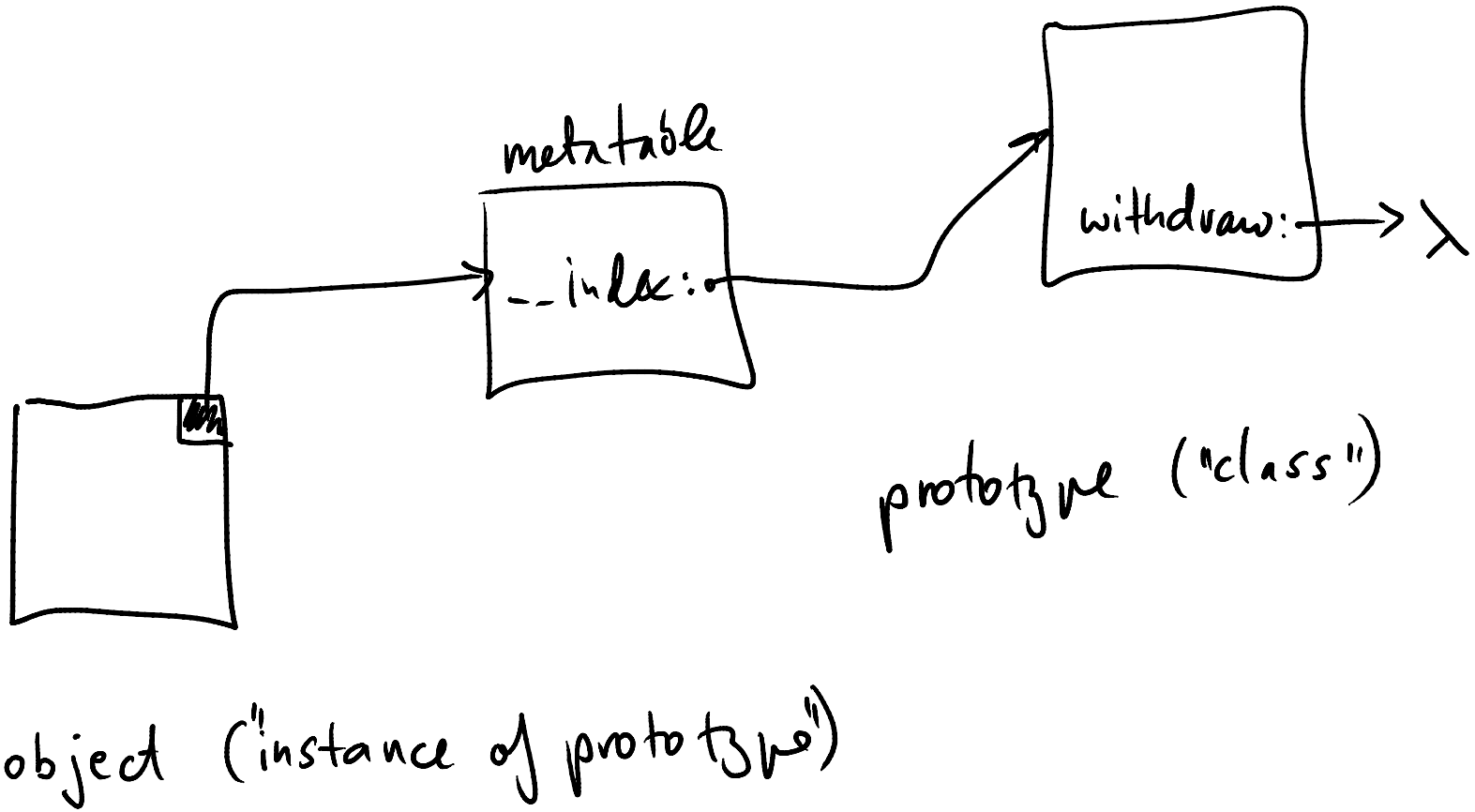


Prototypes

poor man's classes

What runtime setup do we want?

A prototype is an object that behaves like a class



Create an object

```
function Account:new (o) {
    -- create object if user does not provide one
    o = o or {}
    setmetatable(o,self)
    self.__index = self
    o
}

a = Account:new({balance = 0})
a:deposit(100.00)
```

we are doubling the prototype as the metatable

DRAW THE OBJECT CONFIGURATION

Note about cs164 projects

We may decide not to use metatables, just the __index field. The code

```
function Account:new (o) {
    o = o or {}
    setmetatable(o,self)
    self.__index = self
    o
}
```

Would become

```
function Account:new (o) {
    o = o or {}
    o.__index = self
    o                    }
```

Inheritance

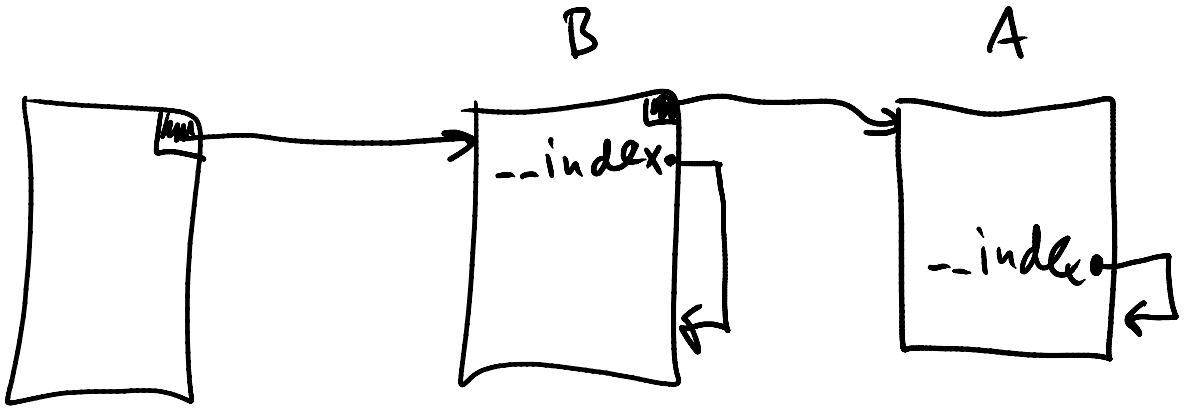
Inheritance allows reuse of code …

… by specializing existing class (prototype)

How to accomplish this with a little "code wiring"?

Let's draw the desired run-time organization:

Assume class A, subclass B, and b an instance of B



Must set this up in the constructor

Tasks that we need to perform

Define a class

```
Account = {balance = 0}
function Account:new (o) {
    o = o or {}
    setmetatable(o, sel)
    self.__index = self
    o
}
function Account:deposit (v) {
    self.balance = self.balance + v  }
function Account:withdraw (v) {
   if (v > self.balance) {
         error"insufficient funds" }
   self.balance = self.balance - v
}
```

Create subclass of Account

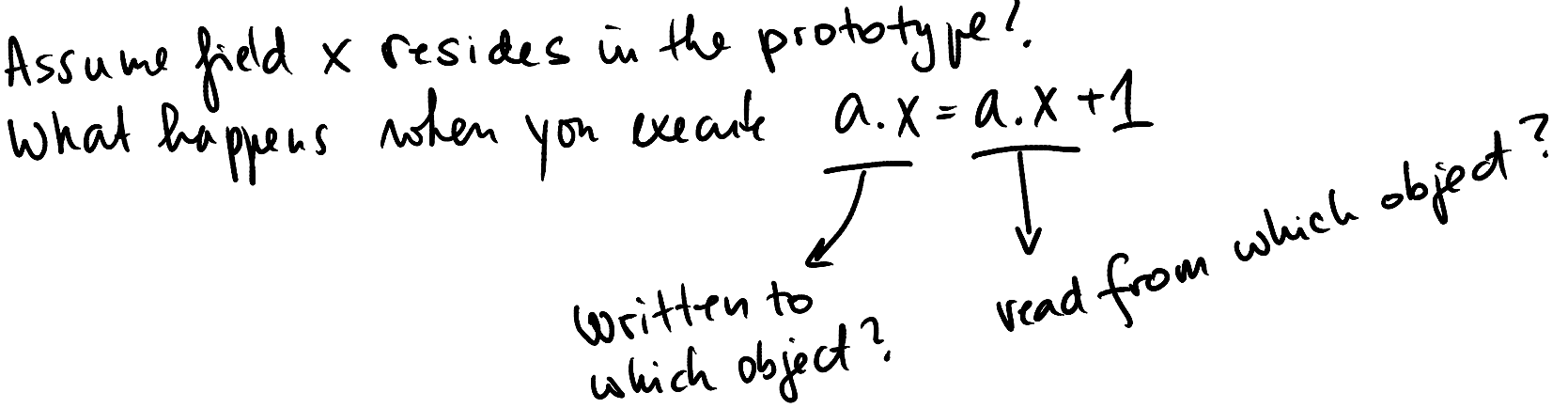
Again, draw the configuration of objects

```
SpecialAccount = Account:new()
s = SpecialAccount:new({limit=1000.00})
s:deposit(100.00)

function SpecialAccount:withdraw (v)
      if (v - self.balance >= self:getLimit()) {
        error"insufficient funds"
      }
      self.balance = self.balance - v
}

function SpecialAccount:getLimit () {
      self.limit or 0
}
```

Discussion of prototype-based inheritance

Notice the sharing:

- constant-value object attributes (fields) remain stored in the prototype until they are assigned.
- After assignment, the object stores the attribute rather than finding it in the prototype



Multiple Inheritance

read about it in PiL

“Privacy”

protecting the implementation

Our goal

Support large programmer teams.

Bad scenario:

- programmer A implements an object O
- programmer B uses O relying on internal details of O
- programmer A changes how O is implemented
- the program now crashes on customer's machine

How do OO languages address this problem?

- private fields

Language Design Excercise

Your task: design an analogue of private fields

Lua/164 supports meta-programming

- it should allow building your own private fields

Object is a table of methods

```
function newAccount (initialBalance)
    def self = {balance = initialBalance}

    def withdraw (v) {
        self.balance = self.balance – v }
    def deposit (v) {
        self.balance = self.balance + v }
    def getBalance () { self.balance }

    {
        withdraw = withdraw,
        deposit = deposit,
        getBalance = getBalance
}   }
```

Use of this object

```
acc1 = newAccount(100.00)
acc1.withdraw(40.00)
print acc1.getBalance()      --> 60
```

Discussion of Table of methods approach

This approach supports private data

- Users of the object cannot access the balance except via objects methods.

Why is this useful?

- implementation is hidden in functions and can be swapped
- because the client of this object is not reading its fields

How can we extend the object with private methods?

We can safely change the implementation

```
function newAccount (initialBalance)
    def self = {
        balance = initialBalance,
        LIM = 1000,
    }
    def extra() {
        if (self.balance > self.LIM)
        { self.balance * 0.1 } else { 0 }
    }
    def getBalance () { self.balance + extra() }
    // as before
    {  /* methods */ }
}
```

More discussion

Can the table-of-methods objects be extendedto support inheritance?

Reading

Required:

- Chapter 16 in PiL