TECHNISCHE UNIVERSITÄT MÜNCHEN
FAKULTÄT FÜR INFORMATIK

Programming Languages

Prototypes

Dr. Michael Petter
Winter 2019/20

Outline

Prototype based programming

1. Basic language features
2. Structured data
3. Code reusage
4. Imitating Object Orientation

“Why bother with modelling types for my quick hack?”

Motivation – Polemic

Bothersome features

- Specifying types for singletons
- Getting generic types right inspite of co- and contra-variance
- Subjugate language-imposed inheritance to (mostly) avoid redundancy

Prototype based programming

- Start by creating examples
- Only very basic concepts
- Introduce complexity only by need
- Shape language features yourself!

“Let’s go back to basic concepts – *Lua*”

Basic Language Features

- Chunks being sequences of statements.
- Global variables implicitely defined

```
s = 0;
i = 1              -- Single line comment
p = i+s p=42       --[[ Multiline
comment --]]
s = l
```

Basic Types and Values

- Dynamical types – no type definitions
- Each value carries its type
- `type()` returns a string representation of a value's type

```
a = true
type(a)                 -- boolean
type("42"+0)            -- number
type("Petter "..1)       -- string
type(type)              -- function
type(nil)               -- nil
type([[<html><body>pretty long string</body>
</html>
]])                     -- string
a = 42
type(a)                 -- number
```

Functions for Code

✓ First class citizens

```
function prettyprint(title, name, age)
  return title.." "..name..", born in "..(2018-age)
end

a = prettyprint
a("Dr.","Petter",42)

prettyprint = function (title, name, age)
  return name..", "..title
end
```

Introducing Structure

- only one complex data type
- indexing via arbitrary values *except nil* ($\rightsquigarrow$ Runtime Error)
- arbitrary large and dynamically growing/shrinking

```
a = {}                    -- create empty table
k = 42
a[k] = 3.14159            -- entry 3.14159 at key 42
a["k"] = k                -- entry 42 at key "k"
a[k] = nil                -- deleted entry at key 42
print(a.k)                -- syntactic sugar for a["k"]
```

Table Lifecycle

- created from scratch
- modification is persistent
- assignment with reference-semantics
- garbage collection

```
a = {}                  -- create empty table
a.k = 42
b = a                   -- b refers to same as a
b["k"] = "k"            -- entry "k" at key "k"
print(a.k)              -- yields "k"
a = nil
print(b.k)              -- still "k"
b = nil
print(b.k)              -- nil now
```

“So far nothing special – let’s compose types”

Table Behaviour

Metatables

- are *ordinary tables*, used as collections of special functions
- Naming conventions for special functions
- Connect to a table via `setmetatable`, retrieve via `getmetatable`
- Changes behaviour of tables

```
meta = {}                          -- create as plain empty table
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
a = { prefix="Dr.",name="Petter"} -- create Michael
setmetatable(a,meta)               -- install metatable for a
print(a)                           -- print "Dr. Petter"
```

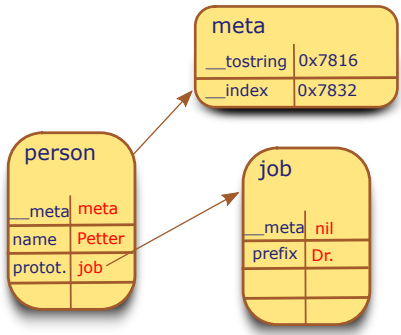
- Overload operators like `__add,__mul,__sub,__div,__pow,__concat,__unm`
- Overload comparators like `__eq,__lt,__le`

Delegation

- ⚠ reserved key `__index` determines *handling* of failed name lookups
- convention for signature: receiver table and key as parameters
- if dispatching to another table ⇝ *Delegation*

```
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key)
  return tbl.prototype[key]
end
job = { prefix="Dr." }
person = { name="Petter",prototype=job } -- create Michael
setmetatable(person,meta)                -- install metatable
print(person)                            -- print "Dr. Petter"
```

Delegation



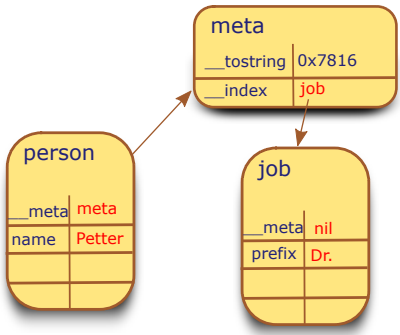

```lua
function meta.__tostring(person)   -- 0x7816
  return person.prefix .. " " .. person.name
end
function meta.__index(tbl, key) -- 0x7832
  return tbl.prototype[key]
end
```

Delegation 2

⇝ Conveniently, `__index` does not need to be a function

```
meta = {}
function meta.__tostring(person)
  return person.prefix .. " " .. person.name
end
job = { prefix="Dr." }
meta.__index = job                          -- delegate to job
person = { name="Petter" }                   -- create Michael
setmetatable(person,meta)                   -- install metatable
print(person)                               -- print "Dr. Petter"
```

Delegation 2




```lua
function meta.__tostring(person)   -- 0x7816
  return person.prefix .. " " .. person.name
end
```

Delegation 3

- `__newindex` handles unresolved updates
- frequently used to implement protection of objects

```
meta = {}
function meta.__newindex(abl,key,val)
  if (key == "title" and tbl.name=="Guttenberg") then
    error("No title for You, sir!")
  else
    tbl.data[key]=val
  end
end
function meta.__tostring(tbl)
  return (tbl.title or "") .. table.name
end
person={ data={} }              -- create person's data
meta.__index = person.data
setmetatable(person,meta)
person.name = "Guttenberg"      -- name KT
person.title = "Dr."            -- try to give him Dr.
```

Object Oriented Programming

⚠ so far no concept for multiple *objects*

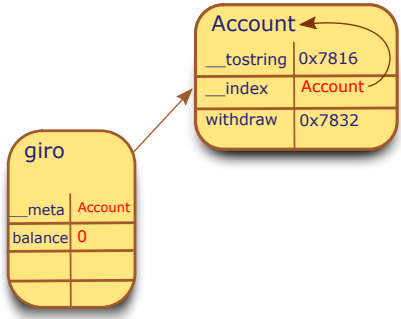
```
Account = { balance=0 }
function Account.withdraw (val)
 Account.balance=Account.balance-val
end
function Account.__tostring()
   return "Balance is "..Account.balance
end
setmetatable(Account,Account)
Account.withdraw(10)
print(Account)
```

Introducing Identity

- Concept of an object's *own identity* via parameter
- Programming aware of multiple instances
- Share code between instances

```
function Account.withdraw (acc, val)
 acc.balance=acc.balance-val
end
function Account.tostring(acc)
   return "Balance is "..acc.balance
end
Account.__index=Account          -- share Account's functions
mikes = { balance = 0 }
daves = { balance = 0 }
setmetatable(mikes,Account)      -- delegate from mikes to Account
setmetatable(daves,Account)      -- del. from daves to Account
Account.withdraw(mikes,10)
mikes.withdraw(mikes,10)         -- withdraw independently
mikes:withdraw(10)
print(daves:tostring() .. " " .. mikes:tostring())
```

Introducing Identity




```
function Account.withdraw (acc, val)
 acc.balance=acc.balance-val
end
function Account.tostring(acc)
   return "Balance is "..acc.balance
end
```

Introducing “Classes”

- Particular tables *used* like classes
- *self* table for accessing object-relative attributes
- connection via creator function *new* (like a constructor)

```
function Account:withdraw (val)
 self.balance=self.balance-val
end
function Account:tostring()
   return "Balance is "..self.balance
end
function Account:new(template)
 template = template or {balance=0}   -- initialize
 setmetatable(template,{__index=self})-- delegate to Account
 getmetatable(template).__tostring = Account.tostring
 return template
end
giro = Account:new({balance=10})      -- create instance
giro:withdraw(10)
print(giro)
```

Inheriting Functionality

- Differential description possible in child class style
- Easily creating particular singletons

```
LimitedAccount = { }
setmetatable(LimitedAccount,{__index=Account})
function LimitedAccount:new()
  instance = { balance=0,limit=100 }
  setmetatable(instance,{__index=self})
end
function LimitedAccount:withdraw(val)
  if (self.balance+self.limit < val) then
     error("Limit exceeded")
  end
  Account.withdraw(self,val)
end
specialgiro = LimitedAccount:new()
specialgiro:withdraw(90)
print(specialgiro)
```

Multiple Inheritance

⇝ Delegation leads to chain-like inheritance

```
function createClass (parent1,parent2)
  local c = {}                    -- new class, child of p1&p2
  setmetatable(c, {__index =
    function (t, k)               -- search for each name
      local v = parent1[k]        -- in both parents
      if v then return v end
      return parent2[k]
    end}
  )
  c.__index = c                   -- c is prototype of instances
  function c:new (o)              -- constructor for this class
    o = o or {}
    setmetatable(o, c)            -- c is also metatable
    return o
  end
  return c                        -- finally return c
end
```

Multiple Inheritance

```
Doctor     = { postfix="Dr. "}
Researcher = { prefix=" ,Ph.D."}

ResearchingDoctor = createClass(Doctor,Researcher)
axel = ResearchingDoctor:new( { name="Michael Petter" } )
print(axel.prefix..axel.name..axel.postfix)
```

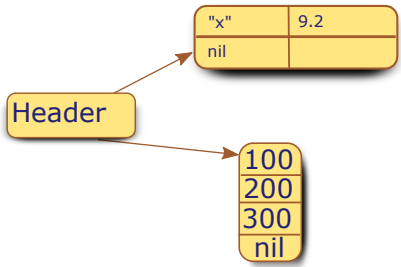
⇝ The special case of dual-inheritance can be extended to comprise multiple inheritance

Implementation of Lua

```
typedef struct {
  int type_id;
  Value v;
} TObject;
```

```
typedef union {
  void *p;
  int b;
  lua_number n;
  GCObject *gc;
} Value;
```

- Datatypes are simple values (Type+union of different flavours)
- Tables at low-level fork into Hashmaps with pairs and an integer-indexed array part



Further Topics in Lua

- Coroutines
- Closures
- Bytecode & Lua-VM

Lessons Learned

Lessons Learned

1. Abandoning fixed inheritance yields ease/speed in development
2. Also leads to horrible runtime errors
3. Object-orientation and multiple-inheritance as special cases of delegation
4. Minimal featureset eases implementation of compiler/interpreter
5. Room for static analyses to find bugs ahead of time

Further Reading...

[1] R. Ierusalimschy.
Programming in Lua, Third Edition.
Lua.Org, 2013.

[2] R. Ierusalimschy, L. H. de Figueiredo, and W. Celes.
The implementation of lua 5.0.
Journal of Universal Computer Science, 2005.

[3] R. Ierusalimschy, L. H. de Figueiredo, and W. C. Filho.
Lua-an extensible extension language.
Softw., Pract. Exper., 1996.