Outline

Prototype based programming
• Basic language features
• Structured data
• Code reuse
• 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
- Massaging language imposed inheritance to by chance dodge redundancy

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

“Let’s try to use only basic concepts – Lua”

Basic language features
- Chunks being sequences of statements.
- Global variables implicitly defined

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

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
-- boolean
type(a)     -- number
type("x")  -- string
type("Simon ".1) -- function
type(nil)   -- nil
```
Functions for code

- First class citizens

```javascript
function prettyprint(title, name, age)
  return title.".".name..", born in "(2014-age)
end
```

Introducing structure

- only one complex data type
- indexing via arbitrary values: `except nil` (~ Runtime Error)
- arbitrary large and dynamically growing/shrinking

```javascript
a = {} -- create empty table/object
k = 42
a[k] = 3.14159 -- entry 3.14159 at key 42
a["honeydew"] = k -- entry 42 at key "honeydew"
```

Lifecycle

- creation from scratch
- modification persistent
- assignment with reference-semantics
- garbage collection

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

Introducing structure

- only one complex data type
- indexing via arbitrary values: `except nil` (~ Runtime Error)
- arbitrary large and dynamically growing/shrinking

```javascript
a = {} -- create empty table/object
k = 42
a[k] = 3.14159 -- entry 3.14159 at key 42
a["honeydew"] = k -- entry 42 at key "honeydew"
```

Lifecycle

- creation from scratch
- modification persistent
- assignment with reference-semantics
- garbage collection

```javascript
a = {} -- create empty table/object
a.k = 42 -- b refers to same as a
b = a -- entry "honeydew" at key "k"
```

Introducing structure

- only one complex data type
- indexing via arbitrary values: `except nil` (~ Runtime Error)
- arbitrary large and dynamically growing/shrinking

```javascript
a = {} -- create empty table/object
k = 42
a[k] = 3.14159 -- entry 3.14159 at key 42
a["honeydew"] = k -- entry 42 at key "honeydew"
```

Lifecycle

- creation from scratch
- modification persistent
- assignment with reference-semantics
- garbage collection
“So far nothing special – let’s compose types”

**Table Behaviour**

- Change behaviour of tables
- Tables as collections of special functions
- Name conventions for special functions
- Access to metatable via `getmetatable` and `setmetatable`

```lua
meta = {}

function meta.__tostring(person)
    return person.prefix .. " " .. person.name
end

a = { prefix="Dr.",name="Simon"} -- create Axel
setmetatable(a,meta) -- install metatable for a
print(a) -- print "Dr. "Simon"
```

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

---

**Delegation**

⚠️ Forward name resolution to another table

```lua
meta = {}

function meta.__tostring(person)
    return person.prefix .. " " .. person.name
end

function meta.__index(table, key)
    return table.prototype[key]
end

job = { prefix="Dr." }
person = { name="Simon",prototype=job } -- create Axel
setmetatable(person,meta) -- install metatable
print(person) -- print "Dr. Simon"
```

---

**Delegation**

⚠️ Forward name resolution to another table

```lua
meta = {}

function meta.__tostring(person)
    return person.prefix .. " " .. person.name
end

function meta.__index(table, key)
    return table.prototype[key]
end

job = { prefix="Dr." }
person = { name="Simon",prototype=job } -- create Axel
setmetatable(person,meta) -- install metatable
print(person) -- print "Dr. Simon"
```
Delegation

```plaintext
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. " } -- delegate to job
person = { name="Simon" } -- create Axel
```setmetatable(person,meta) -- installmetatable
print(person) -- print "Dr. Simon"

Delegation 2

```plaintext
Delegation 3

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

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

Account = { balance=0 }
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

giro = { balance = 0 }
setmetatable(giro,Account)   -- delegate from giro to Account
Account.withdraw(giro,10)
giro.withdraw(giro,10)       -- withdraw independently

print(Account:tostring())
print(giro:tostring())

Introducing “classes”

- Particular objects used as classes
- self for accessing own object

Account = { }
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,self)        -- Account is metatable
  self.__index=self
  self.__tostring = Account.tostring
  return template
end

Account = Account:new({balance=10}) -- create instance
print(Account)
Account.withdraw(10)
print(giro)
Introducing identity

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

Introducing "classes"

- Particular objects used as classes
- self for accessing own object

```plaintext
Account = { }
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, self) -- Account is metatable
    self.__index = self
    self.__tostring = Account.toString
    return template
end
giro = Account:new({balance=10}) -- create instance
print(giro)
giro:withdraw(10)
```

Inheriting functionality

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

```plaintext
ACCOUNT = Account:new({balance=0, limit=100})
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(giro)
print(specialgiro)
```

Multiple Inheritance

- Delegation leads to chain-like inheritance

```plaintext
function createClass (parent1, parent2)
    local c = {}
    -- new class
    setmetatable(c, {__index =
        function (t, k)
            return parent1[k] -- in both parents
        end
    })
    c.__index = c
    -- c is metatable of instances
    function c:new (o)
        return o or {}
    end
    -- constructor for this class
end
```

function createClass (parent1, parent2)
    local c = {}
    -- new class
    setmetatable(c, {__index =
        function (t, k)
            return parent1[k] -- in both parents
        end
    })
    c.__index = c
    -- c is metatable of instances
    function c:new (o)
        return o or {}
    end
    -- constructor for this class
end
```
Multiple Inheritance

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

ResearchingDoctor = createClass(Doctor,Researcher)
axel = ResearchingDoctor:new({ name="Axel Simon" })
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;
    GCEntity *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

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

Roberto Ierusalimschy.  
*Programming in Lua, Third Edition.*  
ISBN 859037985X.

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes Filho.  
Lua—an extensible extension language.  

Roberto Ierusalimschy, Luiz Henrique de Figueiredo, and Waldemar Celes.  
The implementation of Lua 5.0.  