The IO Programming Language

An Introduction

Tom Van Cutsem
Io is...

- A small prototype-based language
- A server-side scripting language
- Inspired by:

  - Smalltalk
    - Everything is an object
  - LISP
    - Code is a runtime inspectable/modifyable tree
  - Self
  - NewtonScript
  - Act1
  - Delegation
  - Small, Embeddable
  - Lua
Io: Some Facts

• Steve Dekorte, 2002 (www.iolanguage.com)
• Open Source, all platforms (even Symbian!)
• Interpreted, Virtual Machine is
  – ANSI C compatible (except for coroutines)
  – Very compact (~10K lines of code)
  – Incremental GC comparable to mark-and-sweep GC
  – Reasonably fast (cfr. Python, Perl, Ruby)
• Concurrency based on actors and implemented through coroutines
C bindings

• Easy embedding within a C program
• Multi-state embedding
• Bindings with C libraries easily incorporated:
  – Sockets
  – XML/HTML parsing
  – Regular expressions, encryption, compression
  – SQLite embedded transactional database
  – OpenGL bindings
  – Multimedia support
  – …
Simplicity!

- Tries to be the 🍏 of programming languages: things should “just work”

- **Prototypes**
- **Blocks with assignable scope**
- **Messages**
- **Objects**
- **Methods**
- **Variables**
- **Functions**
- **Namespaces**
- **Protocols**
- **Threads**
- **Locks**
- **Actors**
Sample Code: Basics

Hello World
"Hello World\n" print

Factorial
factorial := method(n,
  if (n == 1,
    return 1,
    return n * factorial(n - 1))
)

Control Flow v1
for(a,1,10,
  write(a))

Control Flow v2
10 repeatTimes(
  write("hello"))

Control Flow v3
block(a>0) whileTrue(
  a:=a-1 print)
Sample Code: Data structures

- Built-in Maps, Lists and Linked lists

**List Example**

```plaintext
l := List clone
l add(2 sqrt)
l push("foo")
l foreach(k,v,writeln(k,"->",v)) =>
0->1.414214
1->foo
l atPut(0, "Hello " .. "World")
```

**In-line Lists**

```plaintext
list(2 sqrt, "foo")
```
Sample Code: Objects

Account

Account := Object clone
Account balance := 0
Account deposit := method(v, balance := balance + v)
Account withdraw := method(v, balance := balance - v)
Account show := method(
   write("Account balance: ", balance, "\n")
)
myAccount := Account clone
myAccount deposit(10)
myAccount show

Extending primitives

Number double := method(self * 2)
1 double
=> 2

Singleton

MyObject := Object clone
MyObject clone := method(return self)
Delegation

Person := Object clone
Person name := “John Doe”
Person init := method(write(“new person created”))

jan := Person clone
jan name := “Jan” // leaves Person’s name unchanged!

Shallow copies
Super sends

Person := Object clone
Person name := "Jane Doe"
Person title := method(write(name))

Doctor := Person clone
Doctor title := method(write("Dr. ")); resend

Overriding
“Comb” Inheritance

Io's multiple inheritance

Typical multiple inheritance

↑ : Proto slot links

← : Parent slot links
Assignment

• Assignment is achieved through message passing
  • $o \ x := v$ is translated to $o \ \text{setSlot}("x",v)$
  • $o \ x = v$ is translated to $o \ \text{updateSlot}("x",v)$
First-class methods

• Selecting a method slot automatically activates it (cfr. Self)
• getSlot returns first-class reference to a method/block:
  
  ```
  dogSpeakMethod := Dog getSlot("speak")
  ```
• Methods do not encapsulate a scope: they can simply be introduced in other objects
  
  ```
  BarkingBird speak := getSlot("dogSpeakMethod")
  ```
OO Method Activation

• Similar to Self
• Upon method activation, a “locals” object is created with ao. the following slots:
  – proto: the message receiver
  – self: the message receiver
  – sender: locals object of the caller
  – thisMessage: reification of processed message
• Receiverless message sends are sent to the “locals” object (allows access to local variables)
Person `printOn(stream)`

**OO Method Activation (2)**

- **Person** `proto`, `name "..."`, `printOn`
- **Sender** `proto`, `Printer`
- **Message** `proto`, `args`, `name`, `printOn`, `List`
- **Locals** `self`, `sender`, `thisMsg`, `stream`
Blocks

- Identical to methods, but lexically scoped
  
  ```plaintext
  Pi := 3.14159
  addPiTo := block(v, v+Pi)
  list(1,2,3) translate(idx,val,addPiTo(val))
  ```

- The scope of a block always points to the “locals” object in which it was created

- Methods are just blocks whose scope is assignable: its scope is always re-set to the message receiver upon invocation
Blocks vs Methods

\[ x := 5 \]
\[ b := \text{block}(v, v + x) \]
\[ m := \text{method}(v, v + x) \]

Test := Object clone do (  
  x := 1,  
  accept := method(f, f(2))  
)

\( b(2) \)
\( \Rightarrow 7 \)
Test accept(getSlot(“b”))
\( \Rightarrow 7 \)
\( m(2) \)
\( \Rightarrow 7 \)
Test accept(getSlot(“m”))
\( \Rightarrow 3 \)
Catching exceptions

```
try(SomeObject someMessage)
  catch(FooError, e, 
      fooLogger log(e))
  catch(BarError, e, 
      barLogger log(e))
```

Catching exceptions

```
try(UserError "illegal action")
  catch(SystemError, e, 
      gui showDialog(a UserException); Nop
  catch(UserError, e, 
      gui showDialog(e))
```
Concurrency: Coroutines

Coroutines

```amazon
o1 := Object clone
o1 test := method(for(n, 1, 3, n print; yield))
o2 := o1 clone
o1 @test; o2 @test // @ = async message send
while(activeCoroCount > 1, yield)
=>
112233
```

Transparent Futures

```amazon
result := o @msg // returns a future
result := o @@msg // returns Nil
```
Metaprogramming

Quoted Expressions

```ruby
firstClassMessage := message( putPixel(x,y,color) )
Screen doMessage(firstClassMessage)
firstClassMessage argAt(0) asString
=> "x"
```

Objects as dictionaries

```ruby
Person := Object clone do(
  name := "Jan";
  age := 18
)
Person foreach(slotNam, slotVal,
  writeln(slotNam, " - ", slotVal))
=>
age - 18
name - Jan
"proto" - Object(...)
```
**Method Arity**

- Actuals without corresponding formals are not evaluated
- Formals without corresponding actuals are set to `Nil`

```ruby
test := method( "body" );
test( 1/0 )
=> "body"
identity := method(x, x);
identity
=> Nil
```
Reifying a message

• `thisMessage` denotes reification of message that triggered the current method

```plaintext
myAdd := method(
    args := thisMessage argsEvaluatedIn(sender);
    count := 0;
    args foreach(k,v, count += v);
    count)
```

```plaintext
Variable argument lists
```

```plaintext
myif := method(
    if (sender doMessage(thisMessage argAt(0)),
        sender doMessage(thisMessage argAt(1)),
        sender doMessage(thisMessage argAt(2)))
    )
myif(1 == 1, "ok", 1/0)
```

Lazy argument evaluation!
Conclusions

• Simple pure prototype-based language
• Syntax: everything is a message, semantics: everything is an object
• Subclassing and object instantiation replaced by cloning
• Metaprogramming facilities allow for language extensions
• Lots of libraries thanks to simple C binding mechanism
- the Lobby is the root of the Io namespace
- The black arrows show the order of a slot lookup starting in an instance of the Object prototype
- lookups ignore already traversed paths