Io, a Small Programming Language

Steve Dekorte
www.dekorte.com
steve@dekorte.com

ABSTRACT

Io is small prototype-based programming language. The ideas in Io are mostly inspired by Smalltalk[1] (all values are objects), Self[2] (prototype-based), NewtonScript[3] (differential inheritance), Act1[4] (actors and futures for concurrency), LISP[5] (code is a runtime inspectable / modifiable tree) and Lua[6] (small, embeddable).

Io offers a more flexible language with more scalable concurrency in a smaller, simpler package than traditional languages and is well suited for use as both scripting and embedding within larger projects. Io is implemented in C and it's actor based concurrency model is built on coroutines and asynchronous i/o. It supports exceptions, incremental garbage collection and weak links. Io has bindings for many multiplatform libraries including Sockets, OpenGL, FreeType, PortAudio and others as well as some modules for transparent distributed objects and a user interface toolkit written in Io. This presentation will include an overview of the language and demos of some multiplatform desktop applications written with it.

Categories and Subject Descriptors

D.3.0 [**PROGRAMMING LANGUAGES**]: General

General Terms

Languages.

Keywords

Languages, Object Oriented, Prototype-based, Actors, Coroutines, Lazy Evaluation

1. OBJECTS

In Io, everything is an object (including the locals storage of a block) and all actions are messages (including assignment). The system can more generally be described as expressions composed only of messages being evaluated in contexts (objects) with some lookup rules about connected contexts.

Copyright is held by the author/owner(s).
OOPSLA05, October 1620, 2005, San Diego, California, USA.
ACM 1-59593-193-7/05/0010.

Objects are composed of slots, which are key/value pairs, and a list of protos.When an object receives a message it looks for a matching slot, if not found, the lookup continues depth first recursively in its protos. Lookup loops are detected and avoided. If the matching slot contains a method, it is executed, if it contains any other type of value it returns the value. Io has no globals.

1.1 Differential Inheritance

New objects are made by cloning existing ones. A clone is an empty object that has the parent in it's list of protos. A new instance's init slot will be activated and this gives the object a chance to initialize itself. Like NewtonScript[3], slots in Io are create-on-write.

1.2 Expressions

Message arguments are passed as expressions and evaluated by the receiver. Selective evaluation of these arguments is used to implement control flow.

```
for(i, 1, 10, i println)
a := if(b == 0, c + 1, d)
```

And do other useful things:

```
glChunk := method(
    glPushMatrix
    sender doMessage(thisMessage argAt(0))
    glPopMatrix
)

glChunk(
    glTranslated(1,2,3)
    glRectd(0,0,10,10)
)
```

Likewise, dynamic evaluation can be used with enumeration without the need to wrap the expression in a block. Examples:

```
people select(person, person age < 30)
names := people map(person, person name)
```

Expressions can also be manipulated at runtime, which has any number of interesting uses.

1.3 Blocks with Assignable Scope

Io supports both methods (object scoped blocks) and lexically scoped blocks by having a single Block primitive that has an assignable scope. If a block has no scope assigned, it acts like a method. If the block's scope is set the the scope in which it was defined, it acts like a lexically scoped block.

2. SYNTAX

Io has no keywords or statements. Everything is an expression composed entirely of messages. The informal BNF description:

```
exp ::= { message | terminator }
message ::= symbol [arguments]
arguments ::= "(" [exp [ { exp "," } ]] ")"
symbol ::= identifier | number | string
terminator ::= "\n" | ";"
```

There is also some syntax sugar for operators (including assignment), which are handled by an Io macro executed on the expression after it is compiled into a message tree. Some sample source code:

```
Account := Object clone
Account balance := 0
Account deposit := method(amount,
    balance = balance + amount
)

account := Account clone
account deposit(10.00)
account balance println
```

Like Self[2], Io's syntax does not distinguish between accessing a slot containing a method from one containing a variable.

2.1 Assignment

Io has two assignment messages, ":=" and "=".

```
a := 1 // compiles to setSlot("a", 1)
```

which creates the slot in the current context. And:

```
a = 1 // compiles to updateSlot("a", 1)
```

which sets the slot if it is found in the lookup path or raises an exception otherwise. By overloading updateSlot and forward in the Locals prototype, self is made implicit in methods.

3. CONCURRENCY

Actors implemented with coroutines are used for concurrency. Any object can be sent an asynchronous message by placing a @ before the message name. This immediately returns a transparent future object which becomes the return value when it is ready. If a future is accessed before the result is ready, the accessing coroutine is unscheduled until the result is ready. If such a wait would result in a deadlock, an exception is raised.

When an object receives an asynchronous message it puts the message in its queue and, if it doesn't already have one, starts a coroutine to process the queue. An object processing a message queue is called an "actor". Queued messages are processed sequentially in a first-in-first-out order. Control can be yielded to other actors by calling yield. It's also possible to pause and resume an actor, which unschedules or reschedules it's coroutine.

Blocking operations such as reading on a socket will automatically unschedule the calling coroutine until the data is ready or a timeout or error has occurred.

4. IMPLEMENTATION

Coroutines are implemented via C stack manipulation. Memory management is handled by a non-moving, incremental tri-color collector. Expressions are compiled to a message tree and evaluation is done by walking the tree. Performance is comparable to popular bytecode based scripting languages such as Python. More information can be found at http://www.iolanguage.com/.

5. REFERENCES

[1] Goldberg, A et al. *Smalltalk-80: The Language and Its Implementation*, Addison-Wesley, 1983.

[2] Ungar, D and Smith, RB. *Self: The Power of Simplicity*, OOPSLA, 1987.

[3] Smith, W. *Class-based NewtonScript Programmin*g, PIE Developers magazine, Jan 1994.

[4] Lieberman, H. *Concurrent Object-Oriented Programming in Act 1*, MIT AI Lab, 1987.

[5] McCarthy, J et al. *LISP I programmer's manual, MIT Press*, 1960.

[6] Ierusalimschy, R, et al. *Lua: an extensible extension language*, John Wiley & Sons, 1996.