The Io Programming Language

1.0

Contents

Introduction

Simplicity is the essence of happiness.
- Cedric Bledsoe

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 for 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.

Overview

- pure object language
- small interpreter (~10K lines)
- open source BSD license
- small memory footprint (between 64K-200K depending on the platform)
- reasonably fast (comparable to Python, Perl, Ruby)
- incremental garbage collector, weak links supported
- differential prototype-based object model
- strong, dynamic typing
- exceptions
- ANSI C implementation
- embeddable
- multi-state (multiple independent VMs can run in the same application)
- actor-based concurrency using coroutines/light weight thread
- 64 bit clean
- completely message-oriented, even assignments
- modifiable message trees instead of bytecodes
- C implementation written in OO style
- decompilable methods
- concurrency via actors

Perspective

Why Another Language?

Personal experience with dynamic, object oriented languages suggested that conceptual simplification leads to greater flexibility, power, and productivity and this seemed like a path worth exploring.

Design Goals

Io's goal is to be a language that is:

- easy to use
  - conceptually simple and consistent
  - easily embedded
- useful
  - multi-platform
  - capable of desktop, server and embedded scripting applications

To these ends, the following rules of thumb have been adopt;

Design Guidelines

It Just Works

You don't need to be a system administrator to install Io or need to set environment variables to use it. Io applications don't require installers and are not path dependent. As much as possible, things should "just work" out of the box.

Bindings Are Good

Many language communities view code outside the language as something to be avoided. Io embraces the idea of using C bindings for performance sensitive features (graphics, sound, encryption, array processing, etc) while maintaining multi-platform support by encouraging the use of platform independent or multi-platform C libraries (OpenGL, PortAudio, etc).

In order to provide a common base on which to build applications, the Io project also maintains official bindings for networking, graphics, sound, etc. that are included in the full distribution.

Objects are Good

When possible, bindings should provide an object oriented interface instead of mimicking low-level C APIs. Also, concrete design is favored over the abstract. Dozens of classes should not be required to do a simple operations.

Getting Started

Installing

To build, run:

```
make
```

Binaries will be placed in the binaries subfolder. Problems can be posted to the Io mailing list. To install:

```
make install
```

To test:

```
make test
```

Running Scripts

An example of running a script:

```
./binaries/io vm/_sampleCode/HelloWorld.io
```

There is no main() function or object that gets executed first in Io. Scripts are executed when compiled.

Interactive Mode

Running:

```
./binaries/io
```

will open the Io interpreter prompt.

You can evaluate code by entering it directly. Example:

```
Io> "Hello world!" println
==> Hello world!
```

Statements are evaluated in the context of the Lobby:

```
Io> print
[printout of lobby contents]
```

doFile

A script can be run from the interactive mode using the doFile method:

```
doFile(“scriptName.io”)
```

The evaluation context of doFile is the receiver, which in this case would be the lobby. To evaluate the script in the context of some other object, simply send the doFile message to it:

```
someObject doFile(“scriptName.io”)
```

Command Line Arguments

Example of printing out command line arguments:

```
args foreach(k, v, write("'", v, "'\n"))
```

launchPath

The Lobby "launchPath" slot is set to the location on the initial source file that is executed.

Syntax

Less is more.
- Ludwig Mies van der Rohe

Overview

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" | ";"
```

Expressions

Io code is composed of expressions which themselves are composed of messages. For performance reasons, String and Number literal messages have their results cached in their message objects.

Messages

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

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

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)
```

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.

Operators

An operator is just a message whose name contains no alphanumeric characters (other than ":", "_", "" or ".") or is one of the following words: or, and, return. Example:

```
1 + 2
```

This just gets compiled into the normal message:

```
1 +(2)
```

Which is the form you can use if you need to do grouping:

```
1 +(2 * 4)
```

Standard operators follow C's precedence order, so:

```
1 + 2 * 3 + 4
```

Is parsed as:

```
1 +(2 *(3)) +(4)
```

User defined operators (that don't have a standard operator name) are performed left to right.

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.

which will update the slot if present in the inheritance path and raise an exception otherwise. In the Locals prototype (which is cloned to create method locals) updateSlot is overridden so it calls updateSlot on the object if the slot is not found in the Locals. In this way, self is made implicit within methods.

Numbers

The following are valid number formats:

```
123
123.456
0.456
.456
123e-4
123e4
123.456e-7
123.456e2
```

Hex numbers are also supported (in any casing):

```
0x0
0x0F
0XeE
```

Strings

Strings can be defined surrounded by a single set of double quotes with escaped quotes(and other escape charaters) within.

```
s := "this is a \"test\".\nThis is only a test."
```

For strings with no escaped characters and/or spanning many lines, triple quotes can be used.

```
s := """this is a "test".
This is only a test."""
```

Comments

Comments of the //, /**/ and # style are supported. Examples:

```
a := b // add a comment to a line

/* comment out a group
a := 1
b := 2
*/
```

The "#" style is useful for unix scripts:

```
#!/usr/local/bin/io
```

That's it! You now know everything there is to know about Io's syntax. Control flow, objects, methods, exceptions are expressed with the above syntax.

Objects

In all other languages we've considered [Fortran, Algol60, Lisp, APL, Cobol, Pascal], a program consists of passive data-objects on the one hand and the executable program that manipulates these passive objects on the other. Object-oriented programs replace this bipartite structure with a homogeneous one: they consist of a set of data systems, each of which is capable of operating on itself. - David Gelernter and Suresh J Jag

Overview

Io's guiding design principle is simplicity through conceptual unification.

concept	***unifies***
prototypes	*objects, classes, namespaces, locals functions,*
blocks with assignable scope	*methods, closures*
messages	*operators, calls, assignment, variable accesses*

In Io, everything is an object, including the locals storage of a block and the namespace itself, and all actions are messages, including assignment. Objects are composed of a list of key/value pairs called slots, and an internal list of objects from which it inherits called protos. A slot's key is a symbol (a unique immutable sequence) and it's value can be any type of object.

Prototypes

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 which gives the object a chance to initialize itself. Like NewtonScript[3], slots in Io are create-on-write.

```
me := Person clone
```

To add an instance variable or method, simply set it:

```
myDog name := "rover"
myDog sit := method("I'm sitting\n" print)
```

When an object is cloned, its "init" slot will be called if it has one.

Inheritance

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 (at runtime) and avoided. If the matching slot contains an activatable object, such as a Block or CFunction, it is activated, if it contains any other type of value it returns the value. Io has no globals and the root object in the Io namespace is called the Lobby.

Since there are no classes, there's no difference between a subclass and an instance. Here's an example of creating a the equivalent of a subclass:

```
Io> Dog := Object clone
==> Object_0x4a7c0
```

The above code sets the Lobby slot "Dog" to a clone of the Object object. Notice it only contains a protos list contains a reference to Object. Dog is now essentially a subclass of Object. Instance variables and methods are inherited from the proto. If a slot is set, it creates a new slot in our object instead of changing the proto:

```
Io> Dog color := "red"
Io> Dog
==> Object_0x4a7c0:
  color := "red"
```

Multiple Inheritance

You can add any number of protos to an object's protos list. When responding to a message, the lookup mechanism does a depth first search of the proto chain.

Forward

If an object doesn't respond to a message, it will invoke its "forward" method if it has one. You can get the name of the message and it's arguments from the Message object in the "thisMessage" slot.

```
MyObject forward := method(
        write("sender = ", sender, "\n")
        write("message name = ", thisMessage name, "\n")
        args := thisMessage argsEvaluatedIn(sender)
        args foreach(i, v, write("arg", i, " = ", v, "\n") )
)
```

Resend

Sends the current message to the receiver's proto with the context of self. Example:

```
A := Object clone
A m := method(write("in A\n"))
B := A clone
B m := method(write("in B\n"); resend)
B m
```

will print:

```
in B
in A
```

For sending other messages to the receiver's proto, super is used.

Super

Sometimes it's necessary to send a message directly to a proto. Example:

```
Dog := Object clone
Dog bark := method(writeln("woof!"))

fido := Dog clone
fido bark := method(
        writeln("ruf!")
        super(bark)
)
```

Both resend and super are implemented in Io.

Locals

The first message in a block/method has its target set to the block locals. The locals object has its "proto" and "self" slots set to the target object of the message. So slot lookups look in the locals first, then get passed to the target object and then to its proto, etc until reaching the Lobby. In this way, an assignment with no target goes to the locals first:

```
a := 10
```

creates an "a" slot in the locals object and sets its value to 10. To set a slot in the target, get its handle by referencing the "self" slot of the locals first:

```
self a := 10
```

sets the "a" slot in the target.

Methods

A method is an anonymous function which, when called creates an Object to store it's locals and sets it's proto and self slots to the target of the message. The Lobby method method() can be used to create methods. Example:

```
method((2 + 2) print)
```

Using a method in an object example:

```
Dog := Object clone
Dog bark := method("woof!" print)
```

The above code creates a new "subclass" of Object named Dog and adds a bark slot containing a block that prints "woof!". Example of calling this method:

```
Dog bark
```

The default return value of a block is the return value of its last message.

Blocks

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 to the scope in which it was defined, it acts like a lexically scoped block.

Blocks

A block is the same as a method except it is lexically scoped. That is, variable lookups continue in the context of where the block was created instead of the target of the message which activated the block. A block can be created using the Object method block(). Example:

```
b := block(a, a + b)
```

Blocks vs. Methods

This is sometimes a source of confusion so it's worth explaining in detail. Both methods and blocks create an object to hold their locals when they are called. The difference is what the "proto" and "self" slots of that locals object are set to. In a method, those slots are set to the target of the message. In a block, they're set to the locals object where the block was created. So a failed variable lookup in a block's locals continue in the locals where it was created. And

a failed variable lookup in a method's locals continue in the object to which the message that activated it was sent.

Arguments

Methods can also be defined to take arguments. Example:

```
add := method(a, b, a + b)
```

The general form is:

method(<arg name 0>, <arg name 1>, ..., <do message>)

Variable Arguments

The thisMessage slot that is preset (see next section) in locals can be used to access the unevaluated argument messages. Example:

```
myif := method(
        (sender doMessage(thisMessage argAt(0))) ifTrue(
         sender doMessage(thisMessage argAt(1))) ifFalse(
         sender doMessage(thisMessage argAt(2)))
)

myif(foo == bar, write("true\n"), write("false\n"))
```

The doMessage() method evaluates the argument in the context of the receiver.

Preset Locals

The following local variable slots are set when a method is activated:

slot	***references***
target	*current object*
thisBlock	*current block*
thisMessage	*message used to call this block*
sender	*locals object of caller*
slotContext	*context in which slot was found*
self	*current object **

* when a block is activated, self is set to *the locals object where the block was created.*

Introspection

getSlot

To get the value of a block in a slot without activating it, use the "getSlot" method:

```
myMethod := Dog getSlot("bark")
```

Above, we've set the locals object "myMethod" slot to the bark method. It's important to remember that if you then want to pass the method value without activating it, you'll have to use the getSlot method:

```
otherObject newMethod := getSlot("myMethod")
```

Here, the target of the getSlot method is the locals object.

code

Methods/Block objects have arguments and messages methods for introspecting them. A useful convenience method is code, which returns a string representation of the source code of the method in a normalized form.

```
Io> method(a, a * 2) code
==> “method(a, a *(2))”
```

Control Flow

True, False and Nil

Comparison

Comparison operations (==, !=, >=, <=, >, <) return either the true or false singletons.

```
Io> 1 < 2
==> true
```

Conditions

If

The Lobby contains the condition and loop methods. A condition looks like:

if(<condition>, <do message>, <else do message>)

Example:

```
if(a == 10, "a is 10" print)
```

The else argument is optional. The condition is considered false if the condition expression evaluates to false or nil, and is considered true otherwise.
The result of the evaluated message is returned, so:

```
if(y < 10, x := y, x := 0)
```

is the same as:

```
x := if(y < 10, y, 0)
```

Conditions can also be used in this form(though not as efficiently):

```
if(y < 10) then(x := y) else(x := 2)
```

Else-if is also supported:

```
if(y < 10) then(x := y) elseif(y == 11) then(x := 0) else(x := 2)
```

Smalltalk style ifTrue, ifFalse, ifNil and ifNonNil methods are also supported:

```
(y < 10) ifTrue(x := y) ifFalse(x := 2)
```

Notice that the condition expression must have par

Loops

Loop

The loop method can be used for “infinite” loops:

```
loop("foo" println)
```

While

Like conditions, loops are just messages. while() takes the arguments:

while(<condition>, <do message>)

Example:

```
a := 1
while(a < 10,
        a print
        a = a + 1
)
```

For

for() takes the arguments:

for(<counter>, <start>, <end>, <do message>)

The start and end messages are only evaluated once, when the loop starts.

Example:

```
for(a, 0, 10,
        a print
)
```

To reverse the order of the loop, just reverse the start and end values:

```
for(a, 10, 0, a print)
```

Note: the first value will be the first value of the loop variable and the last will be the last value on the final pass through the loop. So a loop of 1 to 10 will loop 10 times and a loop of 0 to 10 will loop 11 times.
Example of using a block in a loop:

```
test := method(v, v print)
for(i, 1, 10, test(i))
```

Repeat

The Number repeat method is simpler and more efficient when a counter isn't needed.

```
3 repeat("foo" print)
==> foofoofoo
```

Break and Continue

The flow control operations break and continue are supported in loops. For example:

```
for(i, 1, 10,
        if(i == 3, continue)
        if(i == 7, break)
        i print
)
```

Would print:

```
12456
```

Return

Any part of a block can return immediately using the return method. Example:

```
Io> test := method(123 print; return "abc"; 456 print)
Io> test
123
==> abc
```

Concurrency

Coroutines

Coroutines are user level cooperative threads

Scheduler

Actors

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.

Garbage Collection

Even if an object is unreferenced, it will not be garbage collected until its message queue is processed.

Blocking

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.

Actors

Io uses actors and transparent futures for concurrency. An actor is an object with it's own coroutine which is uses to process it's queue of asynchronous messages. Any object can be sent an asynchronous message by placing a @ or @@ before the message name. (think of the "a" in @ as standing for "actor" or "asynchronous")

Example:

```
  result := self foo

// synchronous message

  futureResult := self @foo // async message, immediately return
a Future

  self @@foo // async message, immediately return nil
```

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 messages in its queue. Queued

messages are processed sequentially in a first-in-first-out order. Control can be yielded to other coroutines by calling "yield". Example:

```
obj1 := Object clone
obj1 test := method(for(n, 1, 3, n print; yield))
obj2 := obj1 clone
obj1 @@test; obj2 @@test
while(Scheduler activeActorCount > 1, yield)
```

This would print "112233".

Here's a more real world example:

```
HttpServer handleRequest := method(aSocket,
  handler := HttpRequestHandler clone
  handler @@handleRequest(aSocket)
)
```

Yield

An Object will automatically yield between processing each of its asynchronous messages. The yield method only needs to be called if a yield is required during an asynchronous message execution.

Pause and Resume

It's also possible to pause and resume an object. See the concurrency methods of the Object primitive for details and related methods.

Futures

Io's futures are transparent. That is, when the result is ready, they become the result. If a message is sent to a future (besides the two methods it implements), it waits until it turns into the result before processing the message. Transparent futures are powerfull because they allow programs minimize blocking while also freeing the programmer from managing the fine details of synchronization.

Auto Deadlock Detection

An advantage of using futures is that when a future requires a wait, it will check to see if pausing to wait for the result would cause a deadlock(it traverses the list of connected futures to do this). An "Io.Future.value" exception is raised if the action would cause a deadlock.

The @ and @@ Operators

The @ or @@ before an asynchronous message is just a normal operator message. So:

```
self @test
```

Gets parsed as(and can be written as):

```
self @(test)
```

Efficiency

Since coroutines are used instead of OS threads, tens of thousands of actors can be handled without significant performance issues. Also, the waits described above are not busy-waits.

The coroutine that is waiting is actually removed from the scheduler until the result it is waiting on is ready.

Exceptions

Raise

An exception can be raised by calling raise() on an exception proto.

exceptionProto raise(<description>)

There are three predefined children of the Exception proto: Error, Warning and Notification. Examples:

```
Exception raise("generic foo exception")
Warning raise("No defaults found, creating them")
Error raise("Not enough memory")
```

Try and Catch

To catch an exception, the try() method of the Object proto is used. try() will catch any exceptions that occur within it and return it. It returns nil if no exception is caught.

```
e := try(<doMessage>)
```

To catch a particular exception, the Exception catch() method can be used. Example:

```
e := try(
    // ...
)

e catch(Exception,
    writeln(e coroutine backtraceString)
)
```

The first argument to catch indicates which types of exceptions will be caught. catch() returns the exception if it doesn't match and nil if it does.

Pass

To re-raise an exception caught by try(), use the pass method. This is useful to pass the exception up to the next outer exception handler, usually after all catches failed to match the type of the current exception:

```
e := try(
        // ...
)

e catch(Error,
        // ...
) catch(Exception,
        // ...
) pass
```

Custom Exceptions

To implement your own exception types, simply clone an Exception:

```
MyErrorType := Error clone
```

Persistence

Store

store

load

Maps

Garbage Collection

Primitives

This document is not meant as a reference manual, but an overview of the base primitives and an bindings is provided here to give the user a feel for what is available and where to look in the reference documentation for further details.

Primitives

Primitives are objects built into Io whose methods are implemented in C and (except for the Object primitive) store some hidden data in their instances. For example, the Number primitive has a double precision floating point number as it's hidden data. All Io primitives inherit from the Object prototype and are mutable. That is, their methods can be changed. The reference docs contain more info on primitives.

Object

The ? Operator

Sometimes it's desirable to conditionally call a method only if it exists (to avoid raising an exception). Example:

```
if(obj getSlot("foo"), obj foo)
```

Putting a "?" before a message has the same effect:

```
obj ?foo
```

Sequence

List

Map

Block

File

Directory

Message

WeakLink

Debugger

SkipDB

Bindings

The full Io distribution comes with official bindings for much of the functionality that

Networking

Graphics

User Interface

Sound

Regular Expressions

Compression

Encryption

Blowfish

Digests

MD5
SHA1

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 Programming
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