Io:
A Wisp of a Language

Luke Travis
CPTR 316: Programming Languages
March 1, 2015

Abstract

This paper is an introduction to the Io programming language. Io is a fairly new language, having only been conceived in 2002.[1] Inspired by the likes of Smalltalk and LISP, it is a language which values minimalism and transparency above all else. Specifically, three aspects of Io stand out: everything returns a message, all objects are prototypes, and anything can be altered.

Overview

Io (always spelled with a capital "i" and a lowercase "o") was initially created by Steve Dekorte as an exercise to help him understand compilers better.[2] The project has since evolved into a community effort, however it is important to remember that as far as languages go it is still in it's infancy (as evidenced by the fact that it has yet to have a 1.0 release and builds are instead identified by their date)[3].

As a result of Io's origin it is written in C and is thus highly portable. Io is object oriented and interpreted. A pleasant side effect of this last aspect is that Io ships with a command line interpreter which is what the examples in this paper will draw upon.

Messages

[1] Tate, Bruce. Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. Raleigh, NC: Pragmatic Bookshelf ;, 2010.

[2] Idib.

[3] This author's build and the one used in this paper is Io 20110905.

The constructs of the language should be fairly familiar to anyone with some experience in most of today's mainstream languages. Basic "calculator" usage is quite easy:

```
Io> 1 + 1
==> 2
Io> 200 / 199
==> 1.0050251256281406
Io> 200 / 0
==> inf
Io> 0.8 sin
==> 0.7173560908995228
```

From these few examples you should notice a couple of interesting things. The first is that integer division is not the default. This is because numbers in Io are implemented as double precision floating point numbers in C code. This is very similar to how Javascript handles it's numbers too.

The second thing that stood out was probably that division by zero did not result in an error as it does in most programming languages, but instead returned `inf` for infinity.

The third interesting aspect of this example is that the `sin` function was called *after* our number. Is Io then a stack based language? No, this is instead a result of the fact that everything in Io returns a message. For instance, if simply `0.8` was entered, then the interpreter would also reply with `0.8`. What's happening in our `0.8 sin` statement is that the interpreter first interprets `0.8` and then sends its result to `sin`, thus resulting in the evaluation of the function sin(0.8). You'll see this message passing happening a lot as we take a look at Io but here's one last example with the tried-and-true "Hello world." program:

```
Io> "Hello world." print
Hello world.==> Hello world.
```

Here it's even more evident what is going on. You can see that `println` is both performing its function of printing a string to output *and* returning this very same string as a message. In fact we could even operate another procedure on this message:

```
Io> "Hello world." print sort
Hello world.==>  .Hdellloorw
```

Before we go any further I should also point out that Io has *three* different means of assignment. The = operator that we're used to from many other languages is only used when redefining an existing slot (you can think of these as variables). The `:=` operator, which is also often used for assignment in other languages, will act like the previous = operator and will also create a slot if it doesn't exist yet. The last operator is `::=`. This operator in turn will also behave like the preceding `:=` operator, but will also create a setter (a setter is a function used to set a value without directly accessing the value yourself, we'll show one in use momentarily). Having three assignment operators like this may seem confusing, if you really wanted to you could get away with solely using `:=`. However, utilizing all three lends a greater degree of both convenience and readability to your Io programs. To make sure that the differences between these three operators is clear, here is some example output:

```
Io> dog := "Max"
==> Max
Io> dog = "Sam"
==> Sam
Io> dog
```

```
==> Sam
Io> cat := Object clone do(name ::= "Fluffy")
==>  Object_0x7fca23541bc0:
  name              = "Fluffy"
  setName           = method(...)

Io> cat setName("Fuzzy")
==>  Object_0x7fca23541bc0:
  name              = "Fuzzy"
  setName           = method(...)

Io> cat name
==> Fuzzy
```

You'll notice that something weird about cloning was going on when we moved from dog to cat names and this is the topic of our next section.

Prototypes

Like most of our more recent programming languages, Io is object oriented. However, unlike most of the mainstream OO languages Io is prototype based. What does this mean? Well the simple answer is that Io lacks classes, everything is an instance. For example, in C++ you might declare a class "Person" and then create instances of that class with the procedure "new". Io doesn't do things that way, instead a "Person" (and a couple of "people") would be created like this:

```
Io> Person := Object clone
==>  Person_0x7fdc41492740:
  type              = "Person"
Io> Steve := Person clone
==>  Steve_0x7fdc43009050:
  type              = "Steve"
Io> Adam := Person clone
==>  Adam_0x7fdc4300ec00:
  type              = "Adam"
```

What's happening here is that Io is using `Object` as a prototype for `Person`, and then using `Person` as a prototype for our individual people. As the `clone` procedure hints, both our `Person` class and the people we've created with it are clones of the root object prototype (aptly named): `Object`. Now here's some interesting effects of having all prototypes as existing objects as well, if a method is called that an object does not posses than it will search it's prototypes for that method as well. Observe what happens when we change `Person` after altering one of our people:

```
Io> Steve gender := "female"
==> female
Io> Person gender := "male"
==> male
Io> Steve gender
==> female
Io> Adam gender
==> male
```

As you can see, even though these "instances" were created before giving `Person` the `gender` slot, they still inherit it if they don't already have said slot defined.

At this point you may have started to wonder how multiple inheritance could work if every object is a (possibly modified) copy of a predecessor. How Io actually handles this is by having each object maintain a list of prototypes it inherits from. When you first create an object through cloning, the original object is added to this list. You can easily add more objects to this list like this:

```
Io> Adam appendProto(Animal clone)
==>  Adam_0x7fdc41594f90:
  type             = "Adam"
```

And now `Adam` has both `Person` and `Animal` as it's prototypes. But what if we want to look under the hood and verify that these are the prototypes that `Adam` has? This brings us to possibly the most inviting and interesting aspect of Io: its transparency.

Transparency

Transparency is being able to look into a program or a piece of code and see what's actually there and what all the individual pieces are doing. With many of the commonly used languages today this is either difficult or impossible, and very few which do have this capability have it there "out of the box", usually some kind of module is required. This is Io's strong suit. You'll recall that I've been referring to what would normally be called variables as "slots". A slot is an object, just like everything else in Io, which is attached to another object. Let's see an example without actually creating anything, just using the vanilla Io code:

```
Io> List slotNames
==> list(capacity, with, mapInPlace, reduce, sortByKey, push, rest,
containsIdenticalTo, swapIndices, removeAt, asMap, at, detect,
difference, sort, sortBy, reverseReduce, append, sortKey, empty,
removeSeq, asJson, uniqueCount, join, second, slice, size,
reverseForeach, containsAll, min, copy, indexOf, asSimpleString,
intersect, select, unique, isNotEmpty, flatten, isEmpty, insertAt,
max, first, atInsert, contains, groupBy, asMessage, prepend, setSize,
asString, appendSeq, insertAfter, remove, justSerialized, atPut, last,
foreach, map, third, reverseInPlace, union, sum, containsAny,
preallocateToSize, sliceInPlace, insertBefore, removeAll, reverse,
itemCopy, selectInPlace, appendIfAbsent, average, pop, asEncodedList,
sortInPlaceBy, ListCursor, sortInPlace, exSlice, removeFirst,
fromEncodedList, cursor, removeLast, mapFromKey)
```

That's a lot of information! So what are we looking at? We are looking at all of the slots that are in the object `List`. Well that's obvious but what do these slots actually consist of? Io can check that too:

```
Io> List getSlot("join")
==> # /tmp/io-ZPGZpr/io-2013.12.04/libs/iovm/io/A3_List.io:28
method(sep,
    result := Sequence clone
    if(sep,
        max := self size - 1
        self foreach(idx, value,
            result appendSeq(value)
            if(idx != max, result appendSeq(sep))
        )
    ,
        self foreach(value, result appendSeq(value))
    )
    result
)
```

From this we can see exactly what code is called when the `join` method for a list is used. Similarly it's not just the methods of objects we can view, we can also see the operators that Io uses:

```
Io> OperatorTable
==> OperatorTable_0x7fdfca477960:
Operators
  0   ? @ @@
  1   **
  2   % * /
  3   + -
  4   << >>
  5   < <= > >=
  6   != ==
  7   &
  8   ^
  9   |
  10  && and
  11  or ||
  12  ..
  13  %= &= *= += -= /= <<= >>= ^= |=
```

```
  14  return

Assign Operators
  ::= newSlot
  :=  setSlot
  =   updateSlot

To add a new operator: OperatorTable addOperator("+", 4) and implement
the + message.
To add a new assign operator: OperatorTable addAssignOperator("=",
"updateSlot") and implement the updateSlot message.
```

As you can see, Io makes it incredibly easy to see what the precedence of it's operators are and even tells you how create new ones. So what about modifying things though? What if you want to change the way that Io runs things? That's straight forward too. Unlike most other languages, nothing is sacred in Io, you can redefine anything. For instance if you can redefine the `clone` procedure for `Object` to return the number `42`. Now obviously this makes it impossible to create new objects based on `Object` which severely impairs any code you could write, but the option is still there. Let's see a more useful change, let's say that you no longer wanted division by `0` to return `inf` and instead wanted it to return `0`:

```
Io> Number oldDiv := Number getSlot("/")
==> Number_/()
Io> Number / := method(d, if(d == 0, 0, self oldDiv(d)))
==> method(d,
    if(d == 0, 0, self oldDiv(d))
)
Io> 4 / 2
==> 2
Io> 4 / 0
==> 0
```

Now we have redefined how the `/` operator works on zero division while still keeping it's functionality with all other denominators. Once again you may have

noticed something a bit interesting in this example: the way in which we applied the old method of division, we only used *one* argument. Remember how everything in Io is an object? Because of this behavior only one argument is needed for division because the division method is being called by one of the numbers it's self. As is often the case, the way we're doing math here involves a bit of syntactic sugar to obscure the fact that we're just calling functions. Yet again, Io let's you look under the hood at what's really going on in this process:

```
Io> 8 + 3 * 7 - 2 / -5
==> 29.3999999999999986
Io> message(8 + 3 * 7 - 2 / -5)
==> 8 +(3 *(7)) -(2 /(-(5)))
```

Now we can see exactly how Io is reading in our mathematical expression. Note again how each function takes only one argument (even the negation function).

Conclusion

By now I hope you've gotten a much better feel for how elegantly Io can be. Its uncluttered approach to programming where "everything is an object and all actions are messages"[4] allows a much cleaner view of what your program is actually doing and makes it possible to alter any aspect of it's operation on the fly. While it is still a fledgling language, Io is already has it being used in many fields such as satellites, routers, video games, and animation due to it's ease of learning and powerful design. Though this was just a brief introduction, there is so much more Io

[4] Dekorte, Steve. "Io Programming Guide." Io Programming Guide. January 1, 2006. Accessed March 2, 2015. http://iolanguage.org/scm/io/docs/IoGuide.html#Objects-Prototypes.

can do; I hope this overview has piqued your curiosity and shown you the potential of a truly simple language.

Bibliography

Tate, Bruce. Seven Languages in Seven Weeks: A Pragmatic Guide to Learning Programming Languages. Raleigh, NC: Pragmatic Bookshelf ;, 2010.

Dekorte, Steve. "Io Programming Guide." Io Programming Guide. January 1, 2006. Accessed March 2, 2015. http://iolanguage.org/scm/io/docs/IoGuide.html#Objects-Prototypes.