On the Applicability of Io, a Prototype-Based Programming Language

Darren Broemmer
CS210 Summer 2007

Abstract

Over the last decade, Java has become a commonly used programming language because it addresses a number of prevalent, timely issues including multi-platform support, a focus on networking and web applications, and a simplification of the programming model through automatic memory management and the use of standard libraries. Just as Java built on features of existing languages such as C++ and attempted to address their weaknesses, the Io programming language has features and constructs that place it at the intersection of emerging programming needs and trends. With an ever increasing focus on rapidly developing quality software that adapts to changing requirements, a number of techniques are increasingly being used to manage complexity, achieve software reuse, and effectively implement the separation of concerns. These techniques include agile methodologies, domain-specific languages, scripting languages, meta-programming, and aspect-oriented programming. Io addresses many of the goals behind these approaches through its focus on simplicity, prototype-based development, and ingrained support for dynamic programming. However, Io and other prototype-based object-oriented languages have not yet reached a critical mass. Note that this excludes JavaScript, a widely used prototype-based language which does not have full object-oriented support. Many observations made of the Io language will be true for the majority of prototype-based object oriented languages, however Io was chosen due to its multi-platform support and focus on simplicity and dynamic programming.

This paper examines the suitability of Io for different types of applications, including the development of enterprise applications as well as evolving and exploratory systems. In order to give perspective, this paper will compare the use of Io to Java in terms of language features, constructs, and an evaluation of the same program written in both languages. The example program used is a simple bottoms-up natural language parser that then converts the parsed sentences into executable code in that programming language. This example provides for the comparison of an algorithm-based component (natural language processing) as well as a meta-programming component (dynamic code generation, modification, and execution).

Table of Contents

1. Introduction

Any software program written in a Turing complete language can be implemented in another Turing complete language, however different programming languages have features better suited for particular types of problems. Object-oriented languages have provided a marked improvement in software reuse and developer productivity, however the overall success rate of software development projects is still remarkably low compared to other more mature manufacturing disciplines [5]. Additionally, the demand for software in today's economy and the rate of change are both extremely high. Thus, enterprise applications are often lacking rich domain models critical in building quality, maintainable software implementations that closely represent their real world concerns [4]. Prototypes provide for object-oriented programming without static class definitions and a higher degree of flexibility better suited for evolving requirements. They also provide a mechanism to handle the many exception cases inevitable in building most applications, such as the use of dynamically added attributes. Objects in prototype languages have slots which can hold data or code, and both can typically be modified at runtime.

Io is an interpreted prototype-based language that addresses some of these software development issues through a focus on simplicity, the ability to expressively encapsulate real-world concerns, and the flexibility provided by ingrained meta-programming capabilities. As a programming language, it is at the intersection of current development needs, but it currently lacks tool support and a development community needed in order to become an enterprise class programming language. With the proper amount of support, Io could be applied to a large class of software problems as long as performance is not a primary concern given that the language's simplicity and flexibility comes at the cost of reduced performance. Additionally, support for mainstream enterprise databases is currently lacking, but this is an issue that would likely be addressed with increased developer support.

2. Related Work

Although there are limited references to the Io programming language, there is a significant body of work related to prototype-based languages in general and how they compare to class-based object-oriented languages. Two primary arguments [2] given in favor of prototype-based object oriented programming are (Dony, Malenfant, Cointe 1992):

- It is easier for people to comprehend concrete examples before generalizing concepts into abstract definitions.
- Classes add unnecessary constraints by preventing the customization of individual object instances as well as the inheritance of member data values.

Although the first issue tends to be more of a philosophical or process-oriented distinction, the second issue regarding class constraints deals with a potential limitation in the available programming constructs that support the abstraction and encapsulation of concerns in software. The need for this flexibility in software abstractions is supported by work done in the field of Prototype theory [7] which argues that there are no completely correct taxonomies. This is because some items in a category are more prototypical than others, and essentially there are always exceptions to the rule. This theory also includes the concept of the "distance" between outliers and true prototypes within a category implying that there are very fine distinctions in categories. Thus, static class definitions are limiting in the sense that it is difficult to model the outliers and the exceptions. In a prototype-based language, you can simply prototype an existing object and add data or behaviors to that particular instance, and any object can form the prototype for a set of new objects.

One of Io's primary goals is simplicity [11]. With regards to the simplicity and the applicability of prototype languages, Taivalsaari [9] observes (1996) that:

> When compared to class-based languages, prototype-based languages are conceptually simpler, and have many other characteristics that make them suitable especially to the development of evolving, exploratory and distributed software systems.

Io also has a focus on being a powerful and dynamic language with engrained support for meta-programming. There is a significant body of work in the area of meta-programming in general, however little with regards to Io specifically. Neverov and Roe (2005) provide a nice summary of approaches to meta-programming including partial evaluation, staged languages, template meta-programming, and code generation libraries [6]. The latter two techniques are perhaps more widely used. Templates provide a fairly straightforward mechanism for generating code within code, however code generation libraries offer greater flexibility.

3. An Analysis of the Io Programming Language

The Io User Manual states the following [11]:

> Io's purpose is to refocus attention on expressiveness by exploring higher level dynamic programming features with greater levels of runtime flexibility and simplified programming syntax and semantics.

By nature, prototype languages are more dynamic than class-based object oriented languages because the programmer does not need to statically define each feature of every object within the system. The ability to dynamically prototype objects at runtime complements the ability to dynamically add data and methods to slots of individual object instances. Thus, although the primary abstraction mechanism in Io is objects created through prototypes, Io is largely focused on dynamic programming including the ingrained ability to dynamically create, modify, execute, and reflect on code. Although the majority of prototype languages allow object slots to have code (i.e. methods) or data, this approach is further enabled by the simplification of code constructs in Io such that everything is an object and all expressions are messages sent to objects. For example, Io does have not language statements, so control flow is achieved through the evaluation of methods defined on the encompassing environment object called the Lobby.

3.1 An Overview of Io

Io is an interpreted language created in 2002 by Steve Dekorte. Like scripting languages, Io is not compiled into a permanent binary state, it uses a virtual machine (VM) to directly interpret and execute Io source code. Internally, the Io VM maintains an in-memory tree representation of the code which allows for runtime modifications. Io has full object oriented support using a prototype-based model. In a manner somewhat analogous to functional programming languages, all expressions are messages sent to objects as function calls. However, an object's slots can contain state information (i.e. data) or code (i.e. methods). As a part of its focus on dynamic programming, Io also supports dynamic typing.

Io is primarily focused on three areas that strive to improve upon other commonly used languages:

- Simplicity
- Prototype-based programming model
- Ingrained support for dynamic programming, or meta-programming

3.2 Io and Emerging Programming Trends

Given these three areas of focus, the Io programming language is in alignment with a number of mainstream enterprise development technologies and techniques. The flexibility and rapid development associated with Agile development and scripting languages has become extremely popular of late, especially development using Ruby. Likewise, there is a large movement towards embedding scripting languages in Java as seen in JSR 223: Scripting for the Java Platform [12]. Dynamic programming techniques are increasingly being used to build higher-level reusable components as well as Domain Specific Languages (DSL).

Io's focus on flexibility and expressiveness positions it as a language that can meet many of the goals of these diverse efforts. Rather than being an addition to the language, meta-programming constructs are ingrained from the ground up with a certain simplicity not found, for example, in Java reflection APIs. The Io code to dynamically execute a named method is as follows from the Io sample code web page [11]:

```
anObject perform("SomeMethodName", arg1, arg2)
```

The corresponding code in Java is roughly equivalent to the following:

```
Class [] argClasses = { arg1.getClass(), arg2.getClass() };
Method someMethod = anObject.getClass().getMethod(
                              "SomeMethodName, argClasses);
someMethod.invoke(anObject, new Object[] { arg1, arg2 } );
```

From this simple example, the elegance and simplicity of Io is readily apparent. As another example, consider adding a method to an existing Account object in Io.

```
Account deposit := method(v, balance = balance + v)
```

To be able to modify code like this at runtime in Java requires the use or addition of a scripting language to the environment, such as Beanshell. In the case of Beanshell, there are still limitations in that dynamic Beanshell objects are separately scoped within an Interpreter object, and method calls are required to make these objects available to standard Java code.

Aspect-Oriented Programming (AOP) is another technique used in programming for encapsulating concerns that cut across an inheritance hierarchy. This is one of the few emerging areas in which Io does not offer true language support. Although messages can be dynamically introspected and proxied to other objects or methods, Io does not have a language construct to define aspects that wrap a method call with before and after behavior. Of the emerging trends discussed, AOP is perhaps the earliest on the adoption curve, however aspects provide a powerful encapsulation mechanism worthy of consideration for any language focused on expressiveness. This would be a powerful addition to the Io language, yet at the same time, its inclusion in the language likely has only a minor affect on applicability.

3.3 The Applicability of Io

Consider for a moment two generalized categories of software development at opposite ends of the spectrum:

- Applications implemented using common class-based object oriented languages, such as Java or C++, with fairly rigorous analysis and design methodologies
- Prototypes or exploratory development using Agile processes and dynamic scripting languages.

In reality, these two categories of development do not need to be done using different programming languages. Io provides features that make it adequate as a general purpose object-oriented language. Prototypes have the ability to emulate classes and instances if their usage is limited to only prototyping the reference implementation and not changing their code. At the same time, the flexibility and dynamic programming capabilities allow it to be used to model numerous exceptions that occur in most projects. The fact that it is an interpreted language helps lend itself to Agile and exploratory development as well.

Essentially, Io and other prototype-based languages provide for a more flexible implementation of object oriented programming. As an example, a common requirement for business applications is the concept of dynamically added attributes. This requirement can often be found in re-engineering efforts for applications that have a primary focus on making an up-front investment to make the new system more flexible down the road. Because stakeholders are often concerned about either omitting something or the occurrence of an unforeseen evolving business requirement, the concept of dynamically-added attributes on certain domain objects is levied. Typically, there is also a requirement to be able to apply business rules to these new attributes as well. In cases like these, prototypes work extremely well because existing objects can be prototyped and one-off object instances can dynamically be modified to have the new property along with corresponding code. This ability would be very useful in many enterprise applications and fairly straightforward to implement in Io whereas in a language like Java, it would normally require a fair amount of additional application logic. However, a flexible database implementation would be required in either case to support persistence requirements. In this area, Io is far behind in that it currently only advertises support for a few transactional embedded databases. This limitation would currently eliminate Io from consideration for most enterprise applications.

However, even if prototype-based languages are technically superior to class-based languages, the question of whether the current technologies are "good enough" [10] comes into question. In order to increase adoption, there needs to be enough momentum to cause developers and project managers to move away from established technologies. However, this comes at the cost of additional risk to the project due to the learning curve. Additionally, methodology issues need to be addressed such as the impact of prototype-based languages on typical UML artifacts. For the most part, UML can be used to represent prototype implementations with the exception of static class diagrams that have slightly different interpretations when it comes to prototypes.

4. A Comparison of Java and Io

By many indications and surveys [13], Java is one of the main programming languages used today. Therefore, a comparison of Io to Java helps illustrate the viability of using Io on a new development project. Whereas Io is a lesser known prototype-based language with ingrained dynamic programming support, Java is a mainstream object-oriented language which has only recently been in the process of adding support for meta-programming through JSR 223. The two languages are compared based on the following criteria:

- Expressiveness (Abstraction mechanisms for the separation of concerns)
- Flexibility (Dynamic programming, or meta-programming, capabilities)
- Simplicity
- Performance

Expressiveness in a programming language is fundamental to a basic goal of computer scientists in software development which is the separation of concerns [1]. If the software domain model closely represents the real world concerns which it implements, the probability of producing a quality, maintainable software product is greatly increased. Although code expressions in Io are somewhat simpler than their Java counterparts, as illustrated in the reflection examples, there are not significant differences overall in terms of abstraction mechanisms between the two languages. Each is a pure object-oriented language, although as previously noted, prototypes allow for a more flexible implementation of object oriented development. Io also allows for multiple inheritance, although Java interfaces provide some of the same benefits. In fact, interfaces are a powerful programming construct used to separate implementation from behavior, and this is something that was most likely omitted from Io due to the dynamic nature of adding, modifying, and invoking methods in object slots. Again, this shows a slight diversion between development processes that are more agile and those with additional rigor in the analysis and design phase. Interfaces are also critical for use as contracts between components, subsystems, and systems on software integration projects. Although Io has some primitive objects with networking support, there is minimal related work to demonstrate the use of Io in system integration efforts.

On the basis of flexibility, Io is clearly superior as previously discussed based on the meta-programming capabilities built into the language. However, it is also important to consider that static type checking during compilation is often used to catch many programming errors up front. During the development of the sample application, there were a couple of occasions where a variable was misspelled, and rather than causing a compile error immediately, a new slot was created on the object and a bug was introduced which took a while to correctly diagnose. However, this criticism can generally be applied when comparing any statically typed language with a scripting language. Io does make a tradeoff decision to sacrifice somewhat in terms of performance for flexibility. More details are given on this topic in section 5 on the sample application comparison.

With regards to simplicity, Io appears to have an advantage due to the fact that there are no primitives, everything is an object, and all code expressions are messages sent to objects. Although the subject of specific APIs is a fairly subjective topic, it can be argued that some standard Java libraries are also more complex to use. The `java.util.Calendar` and `java.util.Date` classes are often the subject of such criticism.

Finally, an essential question for each development effort is the practicality of using a lesser known language like Io? The lack of an extensive user community with open-source libraries, components, and tools support is a critical issue for Io when compared to Java development. Development with Java has drastically improved over the years through the innovation that has occurred with open source libraries. Without the availability of libraries and frameworks such as Spring and Struts, the level of effort required to develop Java applications is significantly higher. Additionally, Java EE 5 (Java Enterprise Edition) provides a number of infrastructure and component services, such as messaging, security, and container-managed transactions that are not readily available in Io. These issues present difficulties for the adoption of Io in enterprise development environments.

Table 1 shows a comparison of basic language features between Java and Io.

Feature	Java	Io
Packaging Mechanism	Source code separated into packages and compiled classes into jar files.	No true packaging mechanism. Source directories can be specified to search for prototype definitions.
Abstraction Mechanisms	Objects via instances of static class definitions; Interfaces	Objects via prototyping other objects; No interface construct.
Multiple Inheritance	No, although interfaces provide some of the benefits	Yes, multiple prototypes are supported, including the addition of new prototypes at runtime.
Garbage Collection	Automatic	Automatic
Primitives	Basic data type primitives. There are also corresponding classes.	Everything is an object, including basic data types such as numbers. Methods can be added to these objects also.
Dynamic Programming	Read and invocation	Read, write, and invocation
GUI	Swing and AWT libraries	Flux addon

Table 1: Language Feature Comparison

5. A Sample Application

The sample application provides a restricted, or controlled, natural language interface using a dictionary and a simple context-free grammar to parse sentences into a parse tree. From the resulting parse tree, code can be generated. Because the dynamic programming capabilities of both languages have been addressed in previous sections, the focus of this section is on the language parser component. The implementation of this component in both languages is used to examine differences including a performance comparison parsing the same sentence. A banking domain is used for this example and the input sentence, emulating a simplified user interface at an ATM, is as follows:

```
   I want to transfer from checking account to savings account.
```

The output from the Java version of the parser is shown below. The output notation is as follows:

```
startIndex – endIndex   phrase/word type  (token)
```

`where phrase/word type is  {S = sentence | NP = noun phrase | VP = verb phrase | PP = prepositional phrase | J = adjective | I = infinitive }`

```
NLP Parsing: I want to transfer from checking account to savings account
NounPhrases
-----------
0-0  NP (i)

1-1  NP (want)

5-9  NP (account)
5-5  J   (checking)
7-9  PP   (to)
8-9  NP     (account)
8-8  J       (savings)

8-9  NP (account)
8-8  J   (savings)

VerbPhrases
-----------
1-9  VP (want)
2-3  VP   (transfer)
2-2  I     (to)
4-6  PP   (from)
5-9  NP     (account)
5-5  J       (checking)
7-9  PP       (to)
8-9  NP         (account)
8-8  J           (savings)
7-9  PP   (to)
8-9  NP     (account)
8-8  J       (savings)

PrepositionalPhrases
-----------
4-6  PP (from)
5-9  NP   (account)
5-5  J     (checking)
7-9  PP     (to)
8-9  NP       (account)
8-8  J         (savings)

7-9  PP (to)
8-9  NP   (account)
8-8  J     (savings)

Sentences
-----------
0-9  S (i want transfer to   from account checking  to account savings
to account savings    )
0-0  NP   (i)
1-9  VP   (want)
2-3  VP     (transfer)
2-2  I       (to)
4-6  PP     (from)
5-9  NP       (account)
5-5  J         (checking)
7-9  PP         (to)
8-9  NP           (account)
```

```
8-8  J             (savings)
7-9  PP     (to)
8-9  NP       (account)
8-8  J         (savings)

Parsed sentence:
i want transfer to   from account checking  to account savings      to
account savings
Time to parse (ms): 34
```

Given that all other initialization logic is completed, the timings for this parsing logic for 5 sample runs are shown in Table 2. All times are in milliseconds.

Run	Java Execution Time (ms)	Io Execution Time (ms)
1	24	165
2	17	167
3	64	207
4	68	168
5	34	165
Average	**41.4**	**174.4**
Median	**34**	**167**

Table 2: Language Performance Comparison

On average, the Java version of the controlled language parser is 4.21 times faster than the Io version. Although Java is significantly faster, Io still performs reasonably well. Again, the issue of whether this is good enough for the requirements at hand is the key question.

In terms of code comparison, there is a key difference in using Io that class-based object oriented programmers need to consider. An object that is prototyped shares the member data of the prototype unless its slot with the same name is filled. This caused an issue in development with the `NLPNode` object used to hold the parsed tree structure of the sentence. In the Java implementation, `NLPNode` has a `childElements` collection member variable which is initialized in the constructor. However, in Io, the `NLPNode` object is prototyped, and unless you initialize the `childElements` property in the prototyped object, it will share the exact same data as the original. Thus, bugs occurred during the development of the Io version because initially any child added to an `NLPNode` object was added to all `NLPNode` objects for this reason. In order to emulate class-based programming and avoid this issue, a Factory pattern [4] can be used to encapsulate the instance creation and initialization logic.

The `NaturalLanguageParser parse` method highlights a few differences between the two implementations. A portion of the Io version of this method is shown below:

```
parse := method(Sequence input,

      wordList := List clone

      tokens := input split
      tokens foreach (i, v,

            // Determine each word type
            nlpNode := NLPNode clone
            nlpNode childElements = List clone
            nlpNode nodeValue := v
            nlpNode start := i
            nlpNode end := i
            theWordEntry := dictionary at(v)
            if (theWordEntry == nil,
                Exception raise(nlpNode nodeValue ..
                                    " is not in dictionary."),
                nlpNode types := theWordEntry wordTypes)
            wordList append(nlpNode)
      )

      // Parse out the noun, verb, and prepositional phrases
      parsedNounPhrases := parser getNounPhrases(wordList)
      parsedVerbPhrases := parser getVerbPhrases(wordList)
      parsedPrepPhrases := parser getPrepositionalPhrases(wordList,
                                          parsedNounPhrases)
      mergePrepositionalPhrases(parsedNounPhrases,
                          parsedVerbPhrases,
                          parsedPrepPhrases)

      // Display the results
      "NounPhrases" println
      "-----------" println
      parsedNounPhrases foreach (i, v, v displayString(0) " " println)

      // Display the rest of the results and determine sentence
      // …
   )
```

The `foreach` construct in Io provides a simple way to iterate over loops. Although Java recently added a similar construct, the Io `foreach` has a built in index variable which is a nice convenience. The string (or `Sequence`) split method is also a nice convenience and performs the same task in less lines of code than the Java equivalent using `StringTokenizer`. Notice that this implementation initializes all properties of the `NLPNode` prototype so that the previously mentioned issue with sharing member data does not occur.

The corresponding Java code is shown below:

```
public NLPNode parse(String sentence) {

      AILogger.log(3, "NLP Parsing: " + sentence);

      ArrayList wordList = new ArrayList();

      // Parse out words
      StringTokenizer tokenizer =
            new StringTokenizer(sentence," ,.?;:");
      int wordCount = 0;
      while (tokenizer.hasMoreElements()) {
            String token = tokenizer.nextToken().toLowerCase();
            wordList.add(getWordNode(token,wordCount++));
      }

      // Collection of nounphrases and verbphrases
      TreeSet nounPhrases = getNounPhrases(wordList);

      TreeSet verbPhrases = getVerbPhrases(wordList);

      TreeSet prepPhrases = getPrepositionalPhrases(
                                          wordList,nounPhrases);

      mergePrepositionalPhrases(prepPhrases,nounPhrases,verbPhrases);

      // Display the results
      display("NounPhrases",nounPhrases);

      // Display the rest of the results and determine sentence
      // …
}

/**
 * Create NLPNode based on word types (noun, verb, etc.)
 */
public NLPNode getWordNode(String token, int wordCount) {

      String wordTypes = dictionary.getWordType(token);
      return new NLPNode(token,wordTypes,wordCount);
}
```

6. Conclusion

Io is an extremely flexible and simple programming language to use. It is well suited for agile and exploratory software development. However, the current lack of open source libraries, development tools, and component services most likely prevent it from being used in the majority of enterprise applications. In the future, with enough community support, Io can be applied to a vast array of software problems as it finds itself at the intersection of a number of emerging programming trends and capabilities.

7. References

[1] Edsger Dijkstra. On the Role of Scientific Thought. 1974
[2] Christophe Dony, Jacques Malenfant, Pierre Cointe. Prototype-Based Languages: From a New Taxonomy to Constructive Proposals and Their Validation. ACM SIGPLAN Notices, conference proceedings on Object-oriented programming
[3] Eric Evans. Domain-Driven Design: Tacking Complexity in the Heart of Software. Addison-Wesley, 2004.
[4] Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. Design Patterns: Elements of Reusable Object Oriented Software. Addison-Wesley, 1995.
[5] Jack Greenfield, Keith Short, Steve Cook, Stuart Kent. Software Factories: Assembling Applications with Patterns, Models, Frameworks, and Tools. Wiley, 2004.
[6] Gregory Neverov, Paul Roe. Towards a fully-reflective meta-programming language. Proceedings of the Twenty-eighth Australasian conference on Computer Science – Volume 38 ACSC 2005.
[7] Eleanor Rosch, Carolyn Mervis, Wayne Gray, David Johnson, Penny Boyes-Braem. Basic objects in natural categories. Cognitive Psychology 8, 1976.
[8] Randall Smith, Mark Lentezner, Walter Smith, Antero Taivalsaari, David Ungar. Prototype-Based Languages: Object Lessons from Class-Free Programming (Panel). ACM SIGPLAN Notices, Proceedings of the ninth annual conference on Object-oriented programming systems, languages, and applications OOPSLA 1994. systems, languages, and applications OOPSLA 1992.
[9] Antero Taivalsaari. Classes vs. Prototypes: Some Philosophical and Historical Observations. 1996
[10] Ed Yourdon. When Good-Enough Software is Best. IEEE Software, Vol 12, no. 3, 1995.
[11] The Io Language website. http://www.iolanguage.com
[12] Scripting in Java SE 6 (JSR 223) from the java.net website. https://scripting.dev.java.net/
[13] TIOBE Programming Index website. http://www.tiobe.com/tpci.htm