PNPACK: Computing with probabilities in Java

Stuart Russell[1]
Lewis Stiller[1,2]
Othar Hansson[3]

1. Department of Computer Science, University of California at Berkeley, Berkeley, CA 94720. {russell,stiller}@cs.berkeley.edu.

2. Lewis Stiller is currently at Franz Inc., 1995 University Avenue, Berkeley, CA, 94704. stiller@franz.com.

3. Thinkbank, Inc, 1678 Shattuck Avenue, Suite 320, Berkeley, CA 94709. othar@thinkbank.com.

Abstract: This paper outlines several benefits and disadvantages of Java as the implementation language for a probabilistic inference package. Our focus is the linguistic features that make Java appropriate or inappropriate for this task.

1.0 Introduction

This paper describes our experiences and impressions of Java's suitability for use PNPACK, a library for computing with complex probabilistic models. We focus on the expressivity of the language and only touch on issues like development environments which, while quite important, are rapidly changing.

After reviewing the domain of application of our package, we will describe some of the features that caused us to consider Java for the development and some difficulties we encountered while using it.

The main purpose of this paper is to share experiences and impressions with other workers considering using Java for large-scale scientific codes. We think that the problems we encountered will be broadly applicable.

1.1 Probability background

The central concept in probability theory is that of a *random variable*, that is, an aspect of the world whose exact state is unknown. For example, this might be the *Disease* from which a patient is suffering, the *Weather* in some geographical region, or the *CrudeOilPrice* at some future time. Variables range over a set of values that can be discrete (e.g., *Disease*) or continuous (e.g., *CrudeOilPrice*). A *probability distribution* $P(X)$ for a variable X specifies the probability for each possible value. *Evidence* specifies observed values for a set of variables. The most typical inference problem involves computing the conditional probability distribution $P(X|E)$ for a variable X given some evidence E.

A complex situation is modeled by a set of random variables. Any particular configuration is described by an assignment of values to all the random variables. A joint probability distribution for a set of variables specifies a probability for each possible complete assignment. The joint probability distribution contains all the information required to answer any query about the domain.

Since the number of possible complete assignments to a set of variables is exponential in the number of variables, or infinite in the case of continuous variables, the key problem is to find compact representations of the joint probability distribution that nonetheless reflect the actual properties of the domain and to find efficient algorithms for manipulating these representations to solve problems. *Structured probability models* provide a variety of techniques for doing this.

We now sketch one type of structured probabilistic model: discrete Bayes nets.

Discrete Bayes nets, sometimes called probabilistic networks or belief nets, derive their power from the ability to represent conditional independence among variables, which allows them to take advantage of the locality of causal influences. Intuitively, a variable is independent of its indirect causal influences given its direct causal influences.

Formally, a probabilistic network is defined by a directed acyclic graph together with a conductorial probability table associated with each node. Each node represents a random variable. The conditional probability table associated with variable X specifies the conditional distribution $P(X|Parents(X))$. The arcs encode probabilistic dependence in the sense that each variable must be conditionally independent of its non-descendants in the graph, given its parents. This constraint implies that the network provides a complete representation of the joint distribution through the following equation:

$$P(x_1 \ldots x_n) = \prod_{i=1}^{n} P(x_i | Parents(X_i))$$

where $P(x_1 \ldots x_n)$ is the probability of a particular combination of values for $x_1 \ldots x_n$. Provided each node has a bounded number of parents, the total amount of information is now only linear in the number of variables (Pearl 1991.

Once a network has been constructed, inference algorithms operate on it to calculate probabilities for query variables given values for evidence variables. It is important to note that the distinction between evidence and query variables is entirely flexible—any variable can be set and any variable can be queried. Exact inference algorithms typically operate by transforming the underlying directed graph into a simpler structure, for example into a tree, and then performing inference on the simpler structure directly from the defining relations.

Stochastic approximation algorithms using Monte Carlo simulation and variants of importance sampling have also been developed, and indeed for more general networks stochastic simulation is the only known way to evaluate a query.

We have described the simplest kind of network, a directed acyclic network with discrete nodes, and numerous systems, both in C and in Java, handle this case smoothly.

The representational difficulties start to arise when more general types of nodes are allowed, for example, continuous nodes or even other probabilistic networks. The conditional probability tables specifying the distribution of each node given its parents can be stored in a variety of formats. Finally, the graphs themselves can be more general then directed acyclic graphs: some edges can be undirected, the graph might be infinite or only implicitly represented; a variety of different semantics can be associated with different graph structures. Buntine (1994) describes a number of issues in designing software for graphical probability models.

The simplest kind of queries are those asking for the probability of evidence given its parents, the inference problem. In practice, however, the exact conditional probability tables or even the structure of the graph itself is unknown and must be *learned* from the data. Learning algorithms are typically much more computationally intensive than inference algorithms.

1.2 Goals of the PNPACK project

PNPACK is a library under development with the goal of facilitating computation with structured probability models. It is designed to support a variety of inference algorithms, learning algorithms, and distributions. Its broad scope distinguishes it from previous probabilistic network systems, which typically restrict the user to very restricted classes of distributions, as well from statistical packages, which typically provide very limited facilities for structured models.

Bayes nets are one example among many families of distribution representations. Other families we will handle in PNPACK include hidden Markov models, Markov random fields, mixture models, hierarchical models of experts, as well as a wide range of fundamental probability distributions from statistics. PNPACK aims to provide tools for workers in computational statistics, probabilistic AI, and many application areas such as speech processing, image analysis, and computational biology. We intend to make PNPACK freely distributable after the model of influential libraries like LAPACK.

The broad scope of our library and the interrelationships, the many types of algorithms and data structures to be represented, the various classes of users, and the need for a high-quality, integrated, graphical user-interface, all constrain the library architecture.

For example, the dynamic graph algorithms we use to manipulate the graphical structures of the probability distributions are best expressed in a garbage-collected language like Lisp. The BLAS are probably best expressed in a dialect of Fortran, such as Fortran 90 or High Performance Fortran. The data manipulation and I/O is most cleanly handled in C or C++. The representation of hierarchical mathematical structures is best handled in an object-oriented language like C++ or a functional language like SISAL or ML. Finally, the GUI could be written in Tcl/Tk or in a C++-library like Galaxy.

Unfortunately, supporting multiple languages and libraries is quite difficult for an academic, multiplatform project. In the next section, some of the rationale behind our choice of Java as the main implementation language is discussed.

2.0 Java benefits

The advantages of Java are well-publicized and we briefly describe some of particular relevance to PNPACK.

- Java dramatically simplifies the distribution process. Only one set of binaries needs to be maintained and distributed, licensing restrictions are quite mild, linking in new modules is straightforward.
- The package system is suited to the construction of large systems.
- The garbage collection simplifies the expression of the dynamic graph algorithms.
- Many libraries exist or are under development, and standard native code and graphics packages exist.
- Java is popular, widely available, and easy to learn.

3.0 Java lacunae

In the course of developing the PNPACK library, we observed that certain features of the Java language made it more difficult to use than it would otherwise be. We discuss some of these issues here, but we emphasize that this list is not intended as an indictment of Java; indeed, since Java was not designed for large-scale scientific applications, it clearly is not surprising that there are some difficulties in using it for such applications. Rather, we view this list primarily as a guide to programmers writing Java preprocessors, or other languages targeting the JVM.

3.1 Restricted overriding

Java enforces the curious restriction that the return type of an overridden method must coincide with the return type of the overriding method. It would be much more logical to follow the C++ convention—and, indeed, the convention of all other object-oriented languages with which we are familiar—and allow the return type of an overriding method to extend the return type of an overridden method ("covariant subtyping of return types in overridden methods," as it is called).

The reason for the restriction on return types of methods is somewhat obscure. According to comments on the java-interest mailing list from 1995, the restriction appears to have been a workaround for a bug in an early version of the linker. However, this hypothesis is curious, since it would seem to have been more logical to fix the bug in the linker rather than to disable covariant subtyping of return types in overridden methods. One can speculate that it might simply be too difficult, administratively, to change the specification of the virtual machine.

In any case, we run into this restriction frequently. For example, suppose we have a `class Graph` whose vertices are instances of `class Node`, and suppose we would like to extend `class Graph` to `class StochasticGraph` whose vertices are of `class StochasticNode` and which implements additional methods appropriate to such a graph. If there is a method `Graph.find()` returning a `Node`, it is natural to want to extend this method to `StochasticGraph.find()` returning a `StochasticNode`. This cannot be done, however. Similar issues arise whenever we want to extend a class of collections of some class of `Object`s to hold only specific kinds of `Object`s, but with additional properties.

3.2 APIs

Because there are various inconsistencies in the Sun container APIs, it is difficult to write a new container API in a canonical manner. As Chris Laffra (1997) points out, there are more than 6 different and incompatible ways to get the number of elements of a container. For example, one uses `size()` to get the length of a `Vector` but `length()` to get the length of a `String`.

Similarly, `Vector` has a method `setElementAt(Object obj, int index)`, but `Hashtable` has a method `put(Object key, Object value)` and `StringBuffer` has a method `setCharAt(int index, char c)`.

Although these and similar inconsistencies are minor for users of the standard library, they make it difficult to decide on naming conventions for methods of large packages.

3.3 typedefs

Typedefs are used in languages like C to encapsulate machine-dependent types, to abbreviate complex combinations of types, and to document primitive types. Although the first two usages are subsumed in Java by its standardized primitive types on the one hand and by its class structure on the other, the lack of typedefs for primitive types forces us to use opaque method signatures that are difficult to read and difficult to modify. For example, we would like to use terms such as Probability instead of float or double in method signatures.

3.4 Fully dynamic dispatch

Although Java dynamically dispatches on overridden methods, it follows the C++ convention of dispatching to overloaded methods based on the compile-time type of the arguments, rather than the CLOS and Objective C convention of dispatching to overloaded methods based on the class of the arguments. Usually the dynamic dispatch behavior would be preferred in our applications, since we rarely know much about the run-time type of our objects. Static resolution is usually chosen for efficiency reasons, but since Java is slow anyway, we find the choice of static dispatching to be less convenient than dynamic dispatch.

3.5 Operator overloading

Operator overloading would be useful to us in several ways. First, it would substantially simplify and clarify mathematical expressions. Second, it would help us defer the decision of whether to implement a data type as a primitive (or array) or as a class by making it easier to switch between these representations. Third, as we saw above, operator overloading of brackets would permit a syntax that is easier to remember than the various inconsistent usages of “put” and “set” in the Java APIs.

A restricted form of operator overloading has been implemented in the JPP preprocessor by Nik Shaylor. A good discussion of operator overloading in Java and a proposal for adding limited overloading to Java are available from the Visual Numerics web site.

3.6 Parameterized types

A number of proposals to add parameterized types to Java have been implemented [Odersky 1997; Myers1997]. However, we feel that simply adding parameterized types is insufficient: one must also add a typedef capability. The interaction between typedefs and parameterized types has not been adequately addressed in some current proposals.

To see the problem, consider a parameterized type List<A>, where A is the name of a class. Since A itself might be a complex parameterized class, we would typically want to rename List<A>, something like typedef List<A> Nodes.

The only way to do this in Java is to define Nodes as a class extending List<A>:

```
class Nodes extends List<A>{...}.
```

However, this will not mesh gracefully with methods in List<A> like

```
List<A> append(List<A>),
```

since the returned value from an append() call could not be passed to another method expecting a Nodes argument.

3.7 Macros

Macros are particularly useful in iteration constructs. A macro allows us to iterate over the elements of an array or the elements of a collection using similar syntax. In the case of iteration over elements of a collection, macros allow us seamlessly to modify the syntax for iterators. In addition to this use, we currently use macros in three other settings: to simulate typedefs, to facilitate linear conversion of legacy Fortran code, and to simulate parameterized types. Because Java was not intended to be used with a macro preprocessor, we have experienced difficulties with this, however: macros tend not to work well with IDEs such as Sun's Java Workshop, they make debugging trickier, they interfere with the readability of the code, and they confuse javac's built-in make.

3.8 Libraries

There are several problems with the Java libraries. In the case of the libraries of data structures and for graphics, although there are many contenders, they tend to change very rapidly, and it is unclear which, if any, of the libraries will endure.

Data structure libraries tend to be hampered by the lack of parameterized types and by the need to wrap and unwrap primitive types. In JDK 1.0.2 these libraries also had to use various awkward conventions for handling closures and method pointers; typically various ancillary classes would need to be defined. Although the reflection and nested class APIs of JDK 1.1 should clean up the latter problems a bit, most data structures libraries do not yet use these features.

Graphics libraries are also in a confusing state, with Sun and other vendors releasing a number of such libraries. Because JDK 1.1 drastically changes the event model from that of JDK 1.0.2, the state of graphics libraries is even more confusing. In addition, the AWT 1.0.2 was not well specified, and numerous incompatibilities in implementations exist.

Numeric libraries have dual problem, so to speak: there are no substantial numeric libraries available at all, although there are a few very small libraries available. Although one can link to native code, we would prefer not to have to do this for portability reasons; currently we hand-translate code from Fortran source.

3.9 Performance

There is in Java a tension between using primitive and array types, which have direct virtual machine support, and using classes, which allow more flexibility and better naming. Overuse of classes forces additional layers of indirection to be processed when boxing and unboxing data types, and forces more objects to be created and deleted. Apart from the over time expense in all this, it tends to slow down garbage collection and to result in poor memory access patterns, even if the code is naively compiled to native machine code.

4.0 Conclusion

Java has benefits that are difficult to achieve with other languages. Nevertheless, our experience has been that the language itself currently lacks the expressivity and flexibility we would have preferred for PNPACK.

Our work suggests two areas for futher study:

- To what extent are the problems we experienced using Java for our application particular to our application, and to what extent do they apply to other applications?
- How can we design new languages that remedy the language problems discussed in this paper while retaining the benefits of Java?

5.0 References

1. Buntine, Wray. Operations for learning with graphical models. *Journal of Artificial Intelligence Research*. December, 1994
2. Cozman, Fabio. JavaBayes Version 0.3. http://www.cs.cmu.edu/~fgcozman/Research/JavaBayes/Home/.
3. Gosling, James; Bill Joy; Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996
4. Laffra, Chris. *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*. Prentice Hall: NJ, 1997.
5. Martin Odersky and Philip Wadler. Pizza into Java: Translating theory into practice. In *Proceedings of the 24th ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.
6. Motomura, Yoichi. Bayesian Network Simulator. http://www.etl.go.jp/etl/suri/motomura/BN/bn-java.html.
7. Myer, Andrew; Joseph A. Bank; Barbara Liskov;. Parameterized types for Java. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, Paris, France, January 1997.
8. Pearl, Judea. *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann: San Francisco, Second edition, 1991.
9. Nagarajan, Sreekanth. Bayesian Network Editor version 0.9. http://ee1.engr.orst.edu/~nagarasr/bayesnet/index.html.
10. Shaylor, Nik. JPP—A preprocessor for the Java language. Version 0.05. http://www.digiserve.com/nshaylor/jpp.html
11. Visual Numerics. Java Numerical Language Proposal. http://www.vni.com/products/wpd/njl/JNL/docs/intro.html.