Does Java Have Alternatives?

Michael Franz and Thomas Kistler
Department of Information and Computer Science
University of California
Irvine, CA 92697-3425
+1 (714) 824-4825
{franz, kistler}@ics.uci.edu

ABSTRACT

At first sight, Java's position as the de-facto standard for portable software distributed across the Internet seems virtually unassailable. Interestingly enough, however, it is surprisingly simple to provide alternatives to the Java platform, using the plug-in mechanism supported by the major commercial World Wide Web browsers.

We are currently developing a comprehensive infrastructure for mobile software components. This is a long-term research activity and not directly related to Java and the World Wide Web. However, purely as a technology demonstration, we have recently started a small spin-off project called "Juice" with the intent of extending our experimental mobile-code platform into the realm of the commercial Internet.

Juice is implemented in the form of a browser plug-in that generates native code on-the-fly. Although our software distribution format and run-time architecture are fundamentally different from Java's, once that the appropriate Juice plug-in has been installed on a Windows PC or a Macintosh computer, end-users can no longer distinguish between applets that are based on Java and those that are based on Juice. The two kinds of applets can even coexist on the same web-page.

This, however, means that Java can in principle be complemented by alternative technologies (or even gradually be displaced by something better) with far fewer complications than most people seem to assume. As dynamic code generation technology matures further, it will become less important which code-distribution format has the largest "market share"; many such formats can be supported concurrently. Future executable-content developers may well be able to choose from a wide range of platforms, probably including several dialects of Java itself.

Keywords

Mobile code technologies, plug-in browser extensions, on-the-fly code generation, Java, Juice, Oberon.

1 INTRODUCTION

In the short time since its launch, Sun Microsystems's *Java* technology has become almost synonymous with portable software that can be distributed across the Internet. Java's pre-eminent position is reinforced by the fact that built-in support for its distribution format, the *Java Virtual Machine* (JVM), is now not only part of practically every World Wide Web browser, but is starting to appear even within operating systems. Yet in spite of the de-facto adoption of Java by most of the Internet community as the standard platform for encoding executable content (at least for the time being), it remains surprisingly simple to provide alternatives to this platform, even within the context of commercial browser software.

We have created such an alternative to the Java platform and named it "Juice". Juice is an extension of the first author's earlier research on portable code and on-the-fly code generation [3, 4, 5][1]. Our current work is significant on two accounts: First, Juice's portability scheme is technologically more advanced than Java's and may lead the way to future mobile-code architectures. Second, the mere existence of Juice demonstrates that Java can be complemented by alternative technologies with far less effort than most people seem to assume. In fact, once that Juice has been installed on a machine, end-users need not be concerned at all whether the portable software they are using is based on Juice or on Java. In light of this, we are surprised by the widespread belief in the myth that, in order to be portable, all executable content must necessarily be encoded in Java. In the short term, this myth may lead to some ill-founded technology decisions.

Just as with Java, there are three major components to the Juice technology: 1) a source language and an API in which Juice applets are programmed, 2) an architecture-neutral distribution format, and 3) an environment for executing Juice applets, which in the current implementation of Juice is supplied in the form of a

---

[1] note that this earlier work on mobile code predates Java by several years

browser plug-in that *generates native code on-the-fly*. On all three accounts, Juice differs considerably from Java, yet from the web-browsing end-user's perspective, there is no obvious difference between Java and Juice applets.

In the following, we will introduce the three components of the Juice platform: source language, distribution format, and dynamic-compilation environment. We will then use a simple example for presenting Juice and Java side-by-side, arguing that the choice of a particular mobile-code solution may simply be a matter of personal taste, rather than a technological necessity. Luckily, it is the applet developer that needs to make this choice; the end user need not know any of it as multiple mobile-code technologies, such as Java and Juice, can happily coexist, even on the same web page (Figure 1).

2 PROGRAMMING JUICE APPLETS

Juice applets are programmed in the language *Oberon* [16], a direct successor of Pascal and Modula-2 that was defined by Niklaus Wirth (Pascal's original creator) in 1988. Oberon is surprisingly close to Java in spirit; like Java, Oberon is based on the principles of simplicity and safety. Oberon enforces type-safety by mandating array-bounds checking and prohibiting pointer arithmetic, it automates memory management through the provision of garbage collection, and provides source-level modularization facilities along with dynamic loading. Superficially, but not entirely untrue, one might argue that Oberon is a subset of Java with Pascal syntax, except that

Oberon was defined several years before Java.

Oberon is a much smaller language than Java, having been designed almost as the "essence of a programming language". For example, Oberon provides no language-level support for concurrency. While we agree with Ted Lewis [7] that Java's concurrency scheme falls disappointingly short of the existing state-of-the-art ante, Oberon offers no built-in support for concurrency at all. For the project described here, we have not attempted to change the Oberon language and have therefore only studied applets that can be constructed from the set of language features in the intersection of Oberon and Java. However, we note that this is only an incidental effect of our choice of Oberon as a source language and in no way limits our claim about Java's substitutability in principle. Moreover, we note that some current optimizing translators for Java also exclude the concurrency capabilities of the Java language [11], because support of threads has a performance penalty associated with it [14].

A Juice-applet development tool-kit is now a standard part of the Oberon software distribution from ETH Zurich and UC Irvine [13] for Apple Macintosh and Microsoft Windows. Besides providing a full implementation of *Oberon System 3* [6], it supplies a set of Juice-specific APIs along with a compatibility-box recreating the environment of a browser plug-in within the Oberon environment. Hence, Juice applets under construction can be tested interactively without having to exit the
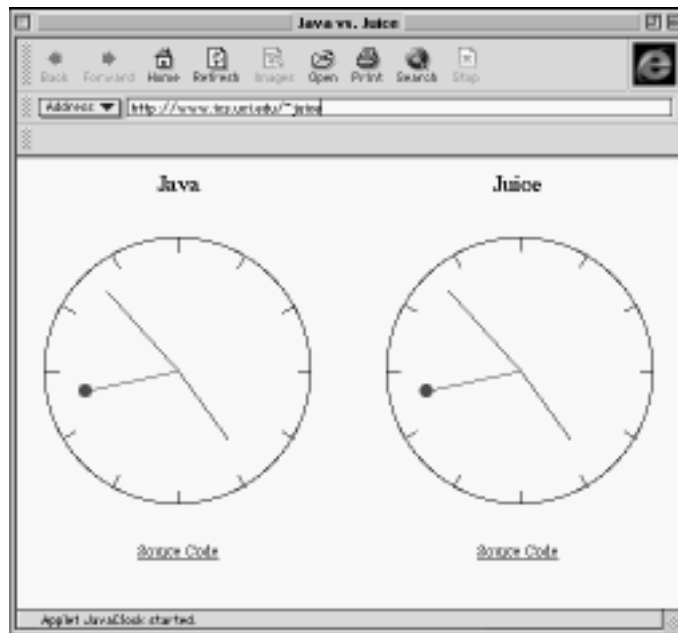

Figure 1: Java Applet (left) vs. Juice Applet (right)

development environment.

3 DISTRIBUTING JUICE APPLETS

Juice's code distribution format, which we call "slim binaries" [2], is based on the Ph.D. thesis work of the first author [4]. The slim binary format differs considerably from virtual-machine representations such as p-Code [12] or Java byte-codes [8] in that it does not resemble executable code. Instead, it is based on a tree-shaped program representation as is typically generated transiently in optimizing compilers. This intermediate tree-representation is then compressed by merging isomorphic sub-trees, using a variant of Welch's classic LZW algorithm [15] that has been specifically adapted towards compressing program trees. Being a dedicated algorithm, it achieves remarkable information densities (Figure 2).

Using a tree-based intermediate code-representation has the disadvantage that it is not well suited for interpretation. While representations such as p-Code and Java byte-codes permit random access, i.e. one can jump 20 instructions forward in the code and resume interpretation, this is not possible with slim binaries. Each symbol in a slim-binary-encoded program can be interpreted only in the context of all the symbols that precede it. Because of this characteristic, our implementations have eschewed interpretation of the intermediate form from the very beginning and have incorporated on-the-fly code generators [4, 5].

On the other hand, reading a slim binary in our system re-creates the original tree data-structure, which is not only an almost ideal input for an optimizing code-generator, but it also makes code verification relatively easy. The slim binary format preserves structural information such as control flow and variable scope that is lost in the transition to linear representations such as Java byte-codes. In order to perform code generation with advanced optimizations from a byte-code representation, a time-consuming pre-processing step is needed to re-create the lost structural information. This is not necessary with slim binaries. This argument applies not only with respect to code optimization, but also for *code verification*: analyzing a mobile program for violation of type and scoping rules is much simpler when the program has a tree-based representation than it is with a linear byte-code sequence.

Our present implementation of slim binaries is in so far restrictive as it supports exactly one source language, Oberon. In this respect, our system presently doesn't do much better than abstract-machine-based portability schemes in which the instruction set of the virtual machine is explicitly crafted to support a particular source language. While we do not foresee any difficulties in encoding syntax trees for other languages, possibly even using the identical format, the suitability for other languages has yet to be established by an actual implementation.

We have therefore recently started a follow-up project with the aim of constructing a compiler that takes Java as its input, but generates slim binaries instead of Java byte-codes as its output. This tool will not only enable a more direct comparison of our code representation and our dynamic compilation architecture with their respective Java counterparts, but it will also aid the wider discussion of platform-independent mobile-code solutions by disengaging the question of source languages from the separate issue of finding suitable distribution formats.

4 EXECUTING JUICE APPLETS

The only part of Juice that is visible to end-users is a set of platform-specific plug-ins for *Netscape Navigator* and *Microsoft Internet Explorer*. Once the appropriate Juice plug-in has been installed on a user's machine, the user can then view and execute Juice content in the same manner as Java applets. Hence, after installation of the plug-in, users can no longer distinguish between Java and
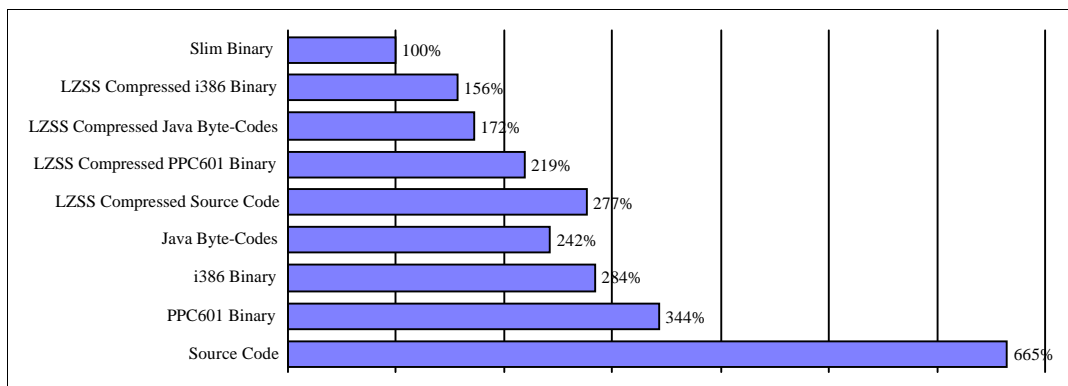

Figure 2: Relative Size of Representative Program Suite in Various Formats

Juice applets, other than by disabling either format manually.

The Juice plug-in contains a dynamic code-generator that translates from the slim binary representation into the native code of the respective target architecture (PowerPC or Intel 80x86). This translation occurs before the applet is started, but is fast enough not to be noticed under normal circumstances. In contrast, most just-in-time compilers for Java translate individual methods as they are called rather than the whole applet at once. Translating the whole applet at once usually results in better code quality since it permits inter-procedural optimizations to be exploited.

Due to the greater compactness of slim binaries in comparison to Java byte-codes, less time has to be spent on the transmission of Juice applets. The time saved can then be used to offset the cost of code generation. As the first author showed in his 1994 dissertation [4], using the slim binary representation can reduce I/O by so much that it fully compensates for the additional effort of code generation. The dissertation originally concerned itself with object code read directly from a disk medium over a fast bus; in the context of networks (which often have transfer rates far slower than secondary storage), this work gains added significance. Further, the speed of processors is rising faster than the speed of I/O, so that hardware technology is evolving in favor of increased density of code-representation formats.

The main thrust of our continuing research is focused on improving code quality. Our implementations so far are all based on a well-established family of compiler back-ends originating at ETH Zurich that produce high quality code comparable to that of straightforward commercial compilers [1]. On some newer RISC architectures, however, these back-ends cannot fully compete with highly optimizing compilers. Of further concern to our particular application of load-time code generation is the fact that optimizers for certain RISC architectures may have vastly different run-time characteristics than the compilers we have been using so far.

Consequently, we are now pursuing a two-tier strategy of code generation. Rather than compiling every module exactly once when it is loaded and then leaving it alone, we use a background process executing only during idle cycles that keeps compiling the already loaded modules over and over. Since this is strictly a re-compilation of already functioning modules, and since it occurs completely in the background, this process can be as slow as it needs to be, allowing the use of far more aggressive, albeit slower, optimization techniques than would be tolerable in an "interactive" context. When background code-generation has completed, the code-images of the re-generated modules are substituted for their older counterparts in situ, without disrupting the ongoing program execution.

Periodic re-optimization of already executing code allows to fine-tune the code-generator's output beyond the level generally achievable by static compilation. Not only does it enable run-time profiling data from the current execution to drive the next iteration of code optimization, but it also makes it possible to cross-optimize application programs and their dynamically loaded extensions and libraries. We are currently experimenting with global optimization techniques that were pioneered by incremental compilers and link-time optimizers. Among them are register allocation and code inlining across module boundaries, global instruction scheduling, and global cache optimization. Run-time extensible systems present new challenges to these old problems, since no closed analysis is possible due to the fact that further modules can be dynamically linked to the already executing system at any time.

5 A DIRECT COMPARISON OF JUICE AND JAVA PROGRAMMING

The easiest way of demonstrating the different "flavors" of programming in Juice vs. programming in Java is by presenting actual source texts. In Figure 3 we present, side by side, the source of a simple applet displaying the current time in analog form (as shown in Figure 1), encoded using Java (left) and Juice (right). These sources and the resulting executable applets can also be found on our World Wide Web site.

6 CONCLUSION AND OUTLOOK

Portable, executable content need not necessarily be tied to Java technology. In this paper, we have presented an alternative to Java called *Juice*. Our implementation uses the existing plug-in mechanism of the major commercial World Wide Web browsers, demonstrating that the plug-in mechanism is suitable for supporting alternative mobile-code solutions even in the case where on-the-fly code generation is desired.

Our implementation also shows that alternative mobile code solutions can remain completely transparent to end-users once that an appropriate plug-in has been installed. Hence, the eventual migration path from Java to a successor standard at the end of Java's life-cycle will probably be much less painful than most people anticipate now.

In fact, the plug-in mechanism opens the door for potentially many different Java alternatives that could be introduced over time, *gradually* reducing Java's pre-eminence. Besides the Juice solution described here, a strong initial candidate to win market share from Java might be Lucent's *Inferno* [9] (assuming that Inferno could

be packaged as a plug-in), but other contenders will surely appear. Note that each plug-in itself can be distributed across the Internet, authenticated by a code-signing mechanism, simplifying the logistics of supporting several competing code-formats concurrently.

It is also possible, and even probable, that the Java standard itself will fragment into several dialects. For example, Microsoft is incorporating an API into its version of Java and its *Internet Explorer* browser that differs from the developments at Sun Microsystems, Java's original creator. There may come a point at which the differences between the various sets of libraries become irreconcilable, leading to mutually incompatible versions of Java. This difference could be hidden from end-users using the same approach that we have taken with Juice.

We believe that dynamic code generation technology is reaching a level of maturity that it will soon diminish the relative importance of "market share" of any particular code distribution format. In order to be commercially successful, distribution formats will have to mimic Java in providing architecture neutrality and safety, but further considerations such as code density will surely gain in importance. For example, some future distribution formats may be targeted towards particular application domains. In this larger context, the current enthusiasm surrounding Java may soon appear to have been somewhat overblown.

REFERENCES

[1] M. Brandis, R. Crelier, M. Franz, and J. Templ; "The Oberon System Family"; Software-Practice and Experience, 25:12, 1331-1366; 1995.

[2] M. Franz and T. Kistler; "Slim Binaries"; Communications of the ACM, 40:12, to appear; also available as Technical Report No. 96-24, Department of Information and Computer Science, University of California, Irvine; 1996.

[3] M. Franz and S. Ludwig; "Portability Redefined"; in Proceedings of the Second International Modula-2 *Conference*, Loughborough, England; 1991.

[4] M. Franz; Code-Generation On-the-Fly: A Key to Portable Software; Doctoral Dissertation No. 10497, ETH Zurich, simultaneously published by Verlag der Fachvereine, Zürich, ISBN 3-7281-2115-0; 1994.

[5] M. Franz; "Technological Steps toward a Software Component Industry"; in *Programming Languages and System Architectures*, Springer Lecture Notes in Computer Science, No. 782, 259-281; 1994.

[6] J. Gutknecht, "Oberon System 3: Vision of a Future Software Technology"; *Software–Concepts and Tools*, 15:1, 26-33, 1994.

[7] T. Lewis; "If Java is the Answer, What Was the Question?"; *IEEE Computer*, 30:3, 136&133-135, March 1997.

[8] T. Lindholm, F. Yellin, B. Joy, and K. Walrath; *The Java Virtual Machine Specification*; Addison-Wesley; 1996.

[9] Lucent Technologies Inc.; *Inferno*; http://plan9.bell-labs.com/inferno/.

[10] M. Franz and T. Kistler; *Juice*; http://www.ics.uci.edu/~juice.

[11] G. Muller, B. Moura, F. Bellard, Ch. Consel; "Harissa: A Flexible and Efficient Java Environment Mixing Bytecode and Compiled Code"; Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS), USENIX Association Press, 1-20; 1997.

[12] K. V. Nori, U. Amman, K. Jensen, H. H. Nägeli and Ch. Jacobi; Pascal-P Implementation Notes; in D.W. Barron, editor; Pascal: The Language and its Implementation; Wiley, Chichester; 1981.

[13] Institut für Computersysteme, ETH Zurich, and Department of Information and Computer Science, University of California at Irvine; *Oberon Software Distribution*; http://www-cs.inf.ethz.ch/Oberon.html or http://www.ics.uci. edu/~oberon.

[14] T. A. Proebsting, G. Townsend, P. Bridges, J. H. Hartman, T. Newsham, and S. A. Watterson; "Toba: Java For Applications – A Way Ahead of Time (WAT) Compiler"; Proceedings of the Third Conference on Object-Oriented Technologies and Systems (COOTS), USENIX Association Press, 41-53; 1997.

[15] T. A. Welch; "A Technique for High-Performance Data Compression"; *IEEE Computer*, 17:6, 8-19; 1984.

[16] N. Wirth; "The Programming Language Oberon"; *Software-Practice and Experience*, 18:7, 671-690; 1988.

```java
import java.awt.*;
import java.util.*;

public class JavaClock
extends java.applet.Applet
implements Runnable
{

    Thread timer = null;

    public void run()
    {
        while (timer!=null) {
            repaint();
            try {Thread.sleep(1000);}
            catch(InterruptedException e) {return;}
        }
    }

    public void start()
    {
        if (timer == null) {
            timer = new Thread(this);
            timer.start();
        }
    }

    public void stop()
    {
        timer = null;
    }

    public int min(int a, int b)
    {
        if (a < b) return a;
        else return b;
    }

    public void arcline(Graphics g, int angle, int x, int y, int r1, int r2, boolean dot)
    {
        int x1, y1, x2, y2; double s, c, a;

        angle = (angle - 15) % 60;
        a = 2*Math.PI / 60 * angle;
        s = Math.sin(a); c = Math.cos(a);
        x1 = (int)(r1*c + 0.5);
        y1 = (int)(r1*s + 0.5);
        x2 = (int)(r2*c + 0.5);
        y2 = (int)(r2*s + 0.5);
        g.drawLine(x+x1, y+y1, x+x2, y+y2);
        if (dot) g.fillOval(x+x2-5, y+y2-5, 10, 10);
    }

    public void paint(Graphics g)
    {
        int r, r0, rs, rm, rh, x, y, i;

        r = min(size().width, size().height) / 2; r0 = 10*r / 11;
        rs = 8*r /11; rm = 9*r/11; rh = 7*r/11; x = r; y = r;
        g.setColor(Color.white);
        g.fillRect(0, 0, size().width, size().height);
        g.setColor(Color.black);
        g.drawOval(0, 0, 2*r, 2*r);
        for (i=0; i<60; i+=5) arcline(g, i, x, y, r0, r, false);

        Date now = new Date();
        arcline(g, now.getMinutes(), x, y, 0, rm, false);
        arcline(g, now.getHours()*5+now.getMinutes()/12, x, y, 0, rh, false);
        g.setColor(Color.red);
        arcline(g, now.getSeconds(), x, y, 0, rs, true);
    }

    public void update(Graphics g)
    {
        paint(g);
    }

}
```

```
MODULE JuiceClock;

    IMPORT
        Math := JuiceMath, Applets := JuiceApplets,
        Devices := JuiceDevices, Misc := JuiceMisc;

    TYPE
        Applet = POINTER TO AppletDesc;
        AppletDesc = RECORD (Applets.AppletDesc)
            hour, min, sec: INTEGER
        END;

    PROCEDURE Min(a, b: INTEGER): INTEGER;
    BEGIN
        IF a < b THEN RETURN a ELSE RETURN b END
    END Min;

    PROCEDURE ArcLine(angle, x, y, r1, r2: INTEGER; dot: BOOLEAN);
        VAR x1, y1, x2, y2: INTEGER; s,c,a : REAL;
    BEGIN angle := (angle-15) MOD 60;
        a := 2 * Math.pi / 60 * angle;
        s := Math.Sin(a); c := Math.Cos(a);
        x1 := SHORT(ENTIER(r1*c + 0.5));
        y1 := SHORT(ENTIER(r1*s + 0.5));
        x2 := SHORT(ENTIER(r2*c + 0.5));
        y2 := SHORT(ENTIER(r2*s + 0.5));
        Devices.Line(x+x1, y+y1, x+x2, y+y2);
        IF dot THEN Devices.FillOval(x+x2-5, y+y2-5, 10, 10) END
    END ArcLine;

    PROCEDURE Update (me: Applet);
        VAR r, r0, rs, rm, rh, x, y, i: INTEGER;
    BEGIN Devices.Setup(me.device);
        r := Min(me.device.w, me.device.h) DIV 2; r0 := 10*r DIV 11;
        rs := 8*r DIV 11; rm := 9*r DIV 11; rh := 7*r DIV 11; x := r; y := r;
        Devices.SetForeColor(Devices.white);
        Devices.FillRect(0, 0, me.device.w, me.device.h);
        Devices.SetForeColor(Devices.black);
        Devices.FrameOval(0, 0, 2*r, 2*r);
        i := 0; WHILE i < 60 DO ArcLine(i, x, y, r0, r, FALSE); INC(i, 5) END;

        Misc.GetTime(me.hour, me.min, me.sec);
        ArcLine(me.min, x, y, 0, rm, FALSE);
        ArcLine(me.hour * 5 + me.min DIV 12, x, y, 0, rh, FALSE);
        Devices.SetForeColor(Devices.red);
        ArcLine(me.sec, x, y, 0, rs, TRUE);
        Devices.Restore(me.device)
    END Update;

    PROCEDURE AppletHandler (me: Applets.Applet; VAR M: Applets.AppletMsg);
        VAR hour, min, sec: INTEGER;
    BEGIN
        WITH me: Applet DO
            WITH M: Applets.DisplayMsg DO
                IF M.id = Applets.update THEN Update(me)
                ELSE Applets.AppletHandler(me, M)
                END
            | M: Applets.IdleMsg DO Misc.GetTime(hour, min, sec);
                IF (hour # me.hour) OR (min # me.min) OR (sec # me.sec) THEN
                    Update(me)
                END
            ELSE Applets.AppletHandler(me, M)
            END
        END
    END AppletHandler;

    PROCEDURE NewApplet*;
        VAR a: Applet;
    BEGIN NEW(a); a.handle := AppletHandler; Applets.newApplet := a
    END NewApplet;

END JuiceClock.
```

Figure 3: Java Source Code (left) vs. Juice Source Code (right)