Михаэль Франц
Томас Кистлер

Есть ли у Java альтернативы?

Michael Franz, Thomas Kistler (1998) Does Java Have Alternatives? // University of California, Irvine.
Р. Богатырев, перевод с англ.

На первый взгляд, Java по отношению к переносимому ПО, распространяемому через Интернет, занимает положение стандарта де-факто. Причем положение достаточно неприступное. В то же время, как это ни удивительно, альтернативу для Java-платформы не так уж и сложно обеспечить за счет использования механизма подключаемых модулей (plug-in), поддерживаемого большинством коммерческих веб-браузеров.

В настоящее время мы разрабатываем разностороннюю инфраструктуру для мигрирующих программных компонентов (mobile software components). Это долгосрочная исследовательская работа, и она напрямую не связана ни с Java, ни с World Wide Web. Однако в качестве демонстрации возможностей технологии не так давно мы начали небольшой проект Juice,[1)] целью которого является расширение сферы действия нашей платформы для мигрирующего кода в мир коммерческого Internet.

[1)] Прим. ред.: Juice — по-англ. "сок"; игра слов – Java ассоциируется с горячим кофе, а Juice — это холодный (яблочный) сок. Яблочный привкус обусловлен тем, что Михаэль Франц — старый поклонник компьютеров фирмы Apple.

Juice реализован в виде подключаемого модуля, генерирующего на лету машинный код. И хотя наш формат представления и архитектура исполняющей системы принципиально отличаются от технологии Java, как только соответствующий подключаемый модуль Juice будет установлен на ПК с Windows или с Mac OS, пользователь уже не сможет отличить аплеты, написанные на Java, от аплетов, написанных на Juice. Два этих вида могут мирно сосуществовать в рамках одной и той же веб-страницы.

Это означает, что Java может дополняться альтернативными технологиями (или даже полностью ими заменяться). При использовании технологии динамической генерации кода формат представления кода становится менее важным: может поддерживаться одновременно сколь угодно много таких форматов. Так что будущие разработчики смогут выбирать широкий диапазон платформ, причем использовать сразу несколько диалектов Java.

1. Введение

В рекордно короткие сроки технология Java компании Sun Microsystems стала почти синонимом переносимого ПО, способного распространяться через Интернет. Господствующие высоты Java предопределены тем, что встроенный механизм поддержки формата представления кода (*виртуальная Java-машина*, JVM) теперь не только является составной частью практически каждого вида браузеров, но и даже претендует на размещение в ядре различных ОС. Интересно отметить, что несмотря на то, что Java принята большинством интернет-сообщества в качестве стандарта де-факто для представления исполняемого кода, все же имеется удивительно простой способ предоставить альтернативную платформу даже в рамках нынешних коммерческих браузеров.

Мы реализовали такую альтернативу для Java-платформы и назвали ее Juice. Juice представляет собой дальнейшее развитие более ранней исследовательской работы, выполненной первым из авторов данной статьи и посвященной переносимому ПО и динамической генерации кода [3,4,5].[2)] Наша нынешняя работа имеет две важных особенности. Во-первых, схема переносимости, принятая в Juice, технологически является более совершенной, нежели подход Java. Она может указать путь для будущих архитектур, построенных на принципе миграции кода. Во-вторых, само лишь существование Juice уже наглядно демонстрирует, что Java может быть дополнена альтернативными технологиями. Причем усилия, которые на это нужно затратить, столь малы, что большинству людей в это просто трудно поверить. Как только модуль Juice установлен на компьютере, пользователю нет надобности заботиться о том, на чем базируются

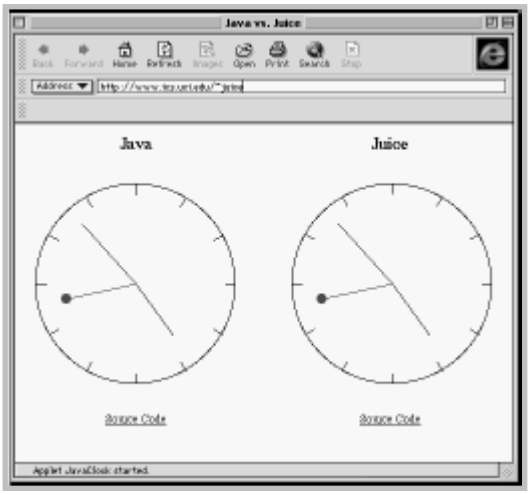
используемые им переносимые программы — на Java или на Juice. Это развенчивает широко распространенный миф о том, что для достижения переносимости исполняемый код обязательно должен быть представлен в Java-формате. К сожалению, данный миф имеет негативные последствия и способен породить неверные технологические решения.

[2)] Прим. ред. — Данная работа опередила Java на несколько лет.

Для технологии Juice, как и для Java, существуют три составляющих:

- язык программирования и API-интерфейс, с помощью которого реализуются аплеты;
- нейтральный по отношению к архитектуре формат представления кода;
- среда для выполнения Juice-аплетов, которая в нынешней реализации Juice поставляется в виде подключаемого модуля, *генерирующего машинный код на лету* (on-the-fly).

По каждой из этих составляющих Juice значительно отличается от Java, хотя с точки зрения пользователя браузера нет явной разницы между аплетами Java и Juice.[3)]



[3)] Прим.ред. — На самом деле это не совсем так: у Juice есть преимущество — при инициализации Juice-аплетов за счет использования прозрачного цвета вообще нельзя увидеть зону отрисовки, тогда как у Java она обычно заливается серым цветом.

В последующих разделах статьи мы познакомим читателя с тремя составляющими Juice-платформы: языком программирования, форматом представления кода и средой динамической компиляции. Далее мы разберем небольшой пример, демонстрирующий реализацию с помощью Juice- и Java-аплетов одной и той же задачи. Вы увидите, что выбор конкретного решения связан скорее с вопросами вкуса, нежели технологической необходимости. К счастью, проблема подобного выбора стоит только перед разработчиком аплетов — пользователю нет надобности знать о том, сколько разных технологий (таких как Java и Juice) мирно сосуществуют в рамках одной и той же веб-страницы (рис.1).

2. Программирование Juice-аплетов

Juice-аплеты программируются на языке *Oberon* [16][4)], являющемся прямым преемником языков Паскаль и Modula-2, которые также были созданы Никлаусом Виртом (Niklaus Wirth). Oberon (разработанный в 1988 г.) по своему духу удивительно близок языку Java: он также опирается на принципы простоты и надежности. Строгая типизация достигается в Oberon за счет обязательной проверки выхода индекса массива за границы диапазона и за счет запрета адресной арифметики. Для автоматизации управления памятью используется механизм сборки мусора, кроме того, предоставляются средства ведения модульности на уровне исходного текста с обеспечением динамической загрузки модулей. Если взять крайне грубое приближение (не являющееся все же корректным), то Oberon можно рассматривать как подмножество Java с синтаксисом Паскаля. Правда здесь не следует забывать о том, что Oberon был реализован на несколько лет раньше Java.

[4)] Прим. ред. — Точнее говоря, на языке Oberon-2, созданном Никлаусом Виртом и Ханспетером Мессенбоком (Hanspeter Moessenboeck) и являющимся диалектом языка Oberon.

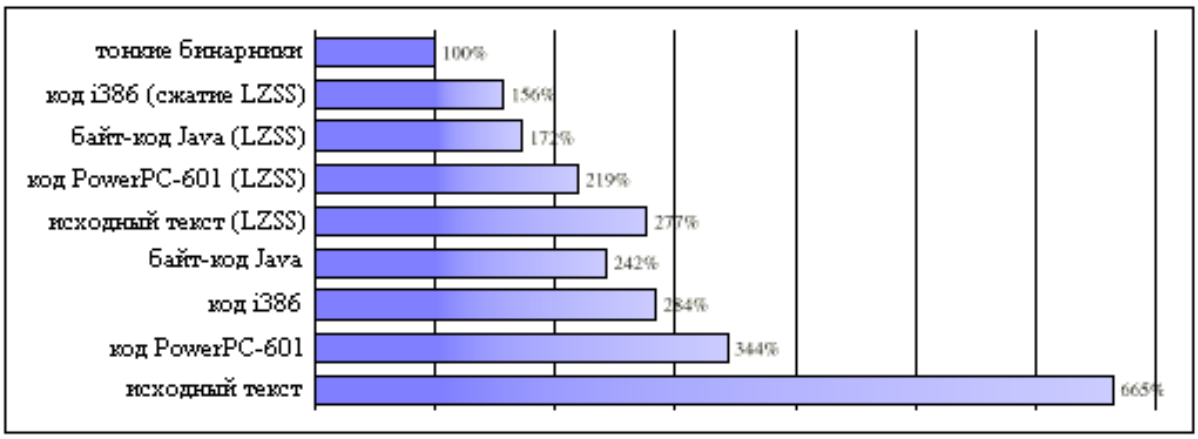
Oberon — гораздо более компактный язык, чем Java. Ведь он задумывался как "квинтэссенция" языка программирования. Вот почему Oberon не имеет встроенной поддержки параллельного программирования. В этом аспекте мы согласны с Тедом Льюисом (Ted Lewis) [7], который считает, что схема, принятая в Java, является неудовлетворительной. В рамках нашего проекта мы не пытались вносить собственные изменения в язык Oberon, а потому изучали лишь те аплеты, которые можно построить совместно с помощью Java и Juice. Здесь хотелось бы отметить, что это единственный побочный эффект нашего выбора, павшего на Oberon, и что нет принципиальных ограничений на взаимозаменяемость по отношению к Java. Более того, ряд

нынешних оптимизирующих трансляторов для Java также игнорирует средства параллелизма данного языка [11], поскольку поддержка программных потоков (threads) приводит к серьезным накладным расходам [14].

Инструментарий для построения Juice-аплетов является неотъемлемой частью комплекта поставки среды Oberon для Apple Macintosh и Microsoft Windows, который предоставляется бесплатно Швейцарским федеральным технологическим институтом в Цюрихе (ETH Zurich) и Университетом Калифорнии в Ирвайне (University of California, Irvine) [13]. Наряду с обеспечением полной реализации системы *Oberon System 3* [6] поставляется и набор специфичных для Juice API-интерфейсов, а также среда, имитирующая в рамках системы Oberon поведение подключаемого к браузеру модуля Juice. Таким образом, Juice-аплеты можно отлаживать и тестировать интерактивно, без выхода из инструментальной системы.

3. Миграция Juice-аплетов

Формат представления кода Juice, который мы называем "тонкие бинарники" (slim binaries) [2], опирается на результаты диссертационной работы Михаэля Франца [4]. Формат тонких бинарников значительно отличается от представления, характерного для виртуальных машин, такого как P-код [12] или байт-код Java [8]. В нем не хранится исполняемый код. Вместо этого Juice-формат опирается на древовидное представление программы в том виде (синтаксические деревья), который обычно порождается оптимизирующими компиляторами. Это промежуточное представление затем упаковывается путем слияния изоморфных поддеревьев с использованием классического LZW-алгоритма (вариант Уэлша) [15], адаптированного специально для сжатия деревьев программного кода. За счет специфики он позволяет добиться значительной степени компактности кода (рис. 2).



Использование древовидного промежуточного кода имеет тот недостаток, что оно не пригодно для процесса интерпретации. В то же время такие форматы, как P-код и байт-код Java, позволяют использовать прямой доступ, например, позволяют переместиться на 20 инструкций и продолжить интерпретацию кода. В случае тонких бинарников это невозможно. Каждая лексема ("слово языка") в Juice-формате может интерпретироваться лишь в контексте всех тех лексем, которые ей предшествуют. По этой причине наша реализация избегает проведения интепретации промежуточного кода с самого его начала и использует встроенный механизм генерации кода на лету [4,5].

С другой стороны, при считывании тонких бинарников наша система восстанавливает первоначальную древовидную структуру, которая не только идеально подходит для проведения оптимизирующей генерации кода, но и также позволяет относительно просто проводить *верификацию кода*. Дело в том, что формат тонких бинарников сохраняет всю информацию о структуре (в частности, потоки управления и области видимости переменных), тогда как при переходе к линеаризированной форме (как в случае байт-кода Java) эта информация попросту теряется. Поэтому для того чтобы провести генерацию машинного кода из байт-кода Java с привлечением передовых методов оптимизации, необходимо затратить немало времени на восстановление утраченной информации. В случае Juice-кода этого делать не надо. То же самое справедливо и в отношении верификации кода: гораздо проще осуществлять анализ

мигрирующей программы на предмет нарушения правил типизации и областей видимости, если она имеет древовидное представление, а не носит вид линеаризированного байт-кода.

Наша нынешняя реализация тонких бинарников несколько ограничена, поскольку поддерживает только один язык — Oberon. В этом смысле наша система сейчас ничем не лучше схем переносимости, построенных на основе абстрактных машин, где набор инструкций виртуальной машины специально подбирается для поддержки конкретного языка программирования. В то же время мы не видим никаких принципиальных трудностей в плане построения упакованных синтаксических деревьев для других языков, быть может, даже с использованием в точности такого же Juice-формата. Ведь специфика языка задается лишь на стадии фактической реализации.

В настоящее время мы начали экспериментальный проект с целью построения такого компилятора, который использует Java в качестве входного языка, а на выходе генерирует не байт-код, а тонкие бинарники. Этот инструментарий позволит не только провести прямое сравнение нашего представления кода и нашей архитектуры динамической компиляции с тем, что имеется у Java. Появится веское основание для начала широкого обсуждения платформонезависимых решений для мигрирующего кода,— для дискуссий, где исходный язык программирования будет отделен от проблем поиска приемлемого формата представления кода.

4. Выполнение Juice-аплетов

Единственной частью Juice, которая видна конечному пользователю, является набор специфичных подключаемых модулей, предназначенных для таких браузеров, как *Netscape Navigator* и *Microsoft Internet Explorer*. Как только соответствующий подключаемый модуль установлен на компьютере, пользователь может видеть и выполнять Juice-аплеты точно так же, как и в случае Java. А потому после установки подключаемого модуля для пользователя теряются различия между Juice и Java (ручное отключение модуля, правда, требует несколько иных манипуляций).

Модуль Juice содержит динамический кодогенератор, который осуществляет трансляцию из формата тонких бинарников в машинный код соответствующей целевой платформы (PowerPC или Intel 80x86). Трансляция производится до того, как начинает свою работу аплет, но скорость ее такова, что в обычной ситуации процесс трансляции распознать невозможно. В отличие от этого подхода большиство JIT-компиляторов Java транслируют отдельные методы по мере их вызова, а не весь аплет целиком. А ведь единоразовая трансляция аплета за счет проведения межпроцедурной оптимизации обычно позволяет получить куда более качественный код.

Благодаря более высокой компактности тонких бинарников в сравнении с байт-кодом Java на передачу Juice-аплетов тратится меньше времени. Сэкономленный резерв может быть использован для снижения накладных расходов на генерацию кода. Как было показано в диссертационной работе Франца, выполненной в 1994 г. [4], использование формата тонких бинарников может настолько сократить затраты на ввод-вывод, что они полностью компенсируют дополнительные операции по отложенной генерации машинного кода. В диссертации прежде всего подвергались изучению вопросы считывания объектного кода непосредственно с внешних дисков (по сравнению с быстрой шиной данных). В случае использования сетей (скорость передачи данных в которых гораздо ниже, чем у внешних дисков) проведенные исследования приобретают дополнительную ценность. Кроме того, быстродействие процессоров растет куда быстрее, нежели скорость операций ввода-вывода, поэтому аппаратные технологии по-прежнему требуют крайне компактных форматов представления кода.

Основной акцент в наших исследованиях мы делаем на улучшении качества кода. Наши реализации до недавнего времени базировались на хорошо апробированном семействе кодогенераторов, первоначально разработанных в стенах ETH Zurich. Причем качество генерируемого ими кода вполне сравнимо с работой добротных коммерческих компиляторов [1]. Однако на некоторых новейших RISC-архитектурах они не могут в полной мере конкурировать с промышленными оптимизирующими компиляторами. В основу наших дальнейших работ по совершенствоанию генерации кода во время загрузки мы положили тот факт, что оптимизаторы для конкретной RISC-архитектуры могут иметь совершенно иные характеристики, чем те, которые свойственны используемым нами компиляторам.

Вот почему мы выбрали путь двухстадийной стратегии генерации кода. Вместо того чтобы ограничиваться одной компиляцией каждого модуля строго до его загрузки, мы используем фоновый процесс, который, работая во время циклов простоя процессора, вновь и вновь компилирует загруженные модули. Поскольку это является рекомпиляцией уже функционирующих модулей и поскольку все операции по оптимизации ведутся в фоновом режиме, есть возможность управлять скоростью компиляции и качеством генерируемого кода. Иными словами, техника компиляции подбирается в "интерактивном режиме". Когда фоновая генерация кода завершается, блок кода регенерированных модулей подставляется вместо своего предшественника причем без нарушения работы программы.

Периодическая повторная оптимизация уже исполняемого кода позволяет добиться более тонкой настройки выходных характеристик кодогенератора по сравнению с тем уровнем, который достижим для статической компиляции. Это не только позволяет проводить профилирование данных с учетом следующей итерации в ходе оптимизации кода, но и дает возможность обеспечивать кросс-оптимизацию прикладных программ и тех динамически загруженных библиотек, которыми они пользуются. В настоящее время мы проводим эксперименты с методами глобальной оптимизации, которые появились на заре инкрементальных компиляторов (incremental compilers) и оптимизаторов этапа компоновки (link-time optimizers). Помимо этого в зоне нашего внимания распределение регистров и вставка кода (inlining) на границах модулей, глобальная диспетчеризация инструкций и глобальная оптимизация кэша. Программные системы, расширяемые во время своей работы (run-time extensible systems), бросают новый вызов старым проблемам. Ведь провести законченный анализ такой системы невозможно, поскольку модули могут подстыковываться к работающей системе в любой момент времени.

5. Непосредственное сравнение программирования на Juice и Java

Самый простой способ продемонстрировать различный "вкус" Java и Juice — представить исходные тексты общего примера. На рис.3 вы можете увидеть две реализации простого аплета, задача которого — отображать текущее время в виде обычных (аналоговых) часов (см. также рис.1). Сначала вариант на Java, затем — на Juice.

Листинг 1. Аплет Clock. Реализация на 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);
      }
```

Листинг 2. Аплет Clock. Реализация на Juice

```
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.
```

6. Заключение и некоторые прогнозы

Переносимый исполняемый код совсем необязательно должен быть связан с технологией Java. В данной статье мы представили альтернативу Java, получившую название *Juice*. Наша реализация использует существующий механизм подключаемых модулей, который имеется во всех основных коммерческих веб-браузерах. Она наглядно демонстрирует, что данный механизм может оказаться весьма удачным для поддержки альтернативных решений, исповедающих миграцию кода, причем даже в том случае, когда желательно располагать динамической генерацией машинного кода. Наша реализация также показывает, что альтернативные подходы могут оставаться полностью прозрачными для конечных пользователей. Следовательно, возможный путь перехода от языка Java к его преемнику (поджидающего Java на закате его жизни) будет куда менее болезненным, чем это могло показаться большинству из нас.

Механизм подключаемых модулей открывает дверь для неограниченно большого числа различных альтернатив технологии Java, которые могут со временем появиться, *постепенно* ослабляя превосходство Java. Помимо представленных здесь решений Juice сильным кандидатом на победу в борьбе за рынок может оказаться технология *Inferno* [9], разработанная в компании Lucent Technologies (при условии, что Inferno может быть реализована в виде подключаемого

модуля). Наверняка появятся и другие претенденты. Стоит отметить, что каждый подобный модуль может распространяться через Интернет с использованием аутентификационного механизма подписи кода. При этом будет упрощаться логика одновременной поддержки нескольких конкурирующих форматов представления кода.

Также возможно и даже желательно, чтобы стандарт Java распался на несколько диалектов. Например, компания Microsoft встроила особый API-интерфейс в собственную реализацию Java и в свой браузер *Internet Explorer*, который отличается от разработок компании Sun Microsystems (где и была создана исходная версия Java). Это может привести к такой ситуации, когда различия между разными наборами библиотек станут неустранимыми, а следовательно появятся взаимно несовместимые версии Java. Такое различие может быть скрыто от пользователя с использованием того же самого подхода, который мы взяли на вооружение при разработке Juice.

Мы уверены, что технология динамической генерации кода достигнет такого уровня развития, когда относительная важность "рыночной стоимости" конкретного формата представления кода станет куда более скромной. Для того чтобы завоевать коммерческий успех, форматы представления должны подражать Java в контексте обеспечения надежности и архитектурной нейтральности. Но при этом на первый план начнут выходить такие вопросы, как компактность кода. Некоторые будущие форматы представления мигрирующего кода могут быть нацелены на специфические области применения. В свете этого нынешняя эйфория, окружающая Java, на поверку может оказаться непомерно раздутой.

Литература

1. M.Brandis, R.Crelier, M.Franz, J.Templ "The Oberon System Family" // Software-Practice and Experience, 1995, Vol.25, No.12, pp.1331-1366.
2. M.Franz, T.Kistler "Slim Binaries" // Communications of the ACM, Vol.40, No.12.
3. M.Franz, S.Ludwig "Portability Redefined" // 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, 1994.
5. M.Franz "Technological Steps toward a Software Component Industry" // Programming Languages and System Architectures // Springer Lecture Notes in Computer Science, No.782, pp.259-281, 1994.
6. J.Gutknecht "Oberon System 3: Vision of a Future Software Technology" // Software–Concepts and Tools, 1994, Vol.15, No.1, pp.26-33.
7. T.Lewis "If Java is the Answer, What Was the Question?" // IEEE Computer, 1997, Vol.30, No.3, pp.136&133-135.
8. T.Lindholm, F.Yellin, B.Joy, K.Walrath "The Java Virtual Machine Specification" // Addison-Wesley, 1996.
9. Lucent Technologies Inc. "Inferno", http://plan9.bell-labs.com/inferno/.
10. M.Franz, 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, 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, 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, 1984, Vol.17, No.6, pp.8-19.

16. N.Wirth "The Programming Language Oberon" // Software-Practice and Experience, 1988, Vol.18, No.7, pp.671-690.