SILK – a playful blend of Scheme and Java

Kenneth R. Anderson, BBN Technologies, Cambridge, MA
Timothy J. Hickey*, Brandeis University, Waltham, MA
Peter Norvig, NASA Ames Research Center, Moffett Field, CA

If we left out the prompt, we could write a complete Lisp interpreter using just four symbols:

```
(loop (print (eval (read))))
```

Consider what we would have to do to write a Lisp (or Java) interpreter in Java [after [6] p. 176].

Abstract

SILK (Scheme in *about* 50 K) is a compact Scheme implemented in Java. It is currently in its fourth implementation. The first version implemented a nearly R4RS Scheme in Java, but its access to Java was awkward. The current version has altered SILK's syntax and semantics slightly to better integrate with Java. This has simplified its implementation, and made SILK an effective Java scripting language, while preserving its Scheme flavor. SILK applications, applets and servlets are surprisingly compact because they take advantage of Scheme's expressiveness. Because SILK is interactive and has reflective access to Java, it provides a view into a Java application that few Java programmers have seen. For example, it easily reveals a security issue involving inner classes. SILK is an Open Source project at `http://silk.sourceforge.net`.

1 Introduction

SILK is a dialect of Scheme which has been designed as a scripting language for Java. It is an implementation of a large subset of R4RS[4] Scheme, implemented in Java, and extended with a simple and elegant mechanism for directly accessing Java. SILK fails to adhere to the minimal R4RS specification in three ways

- it only partially implements Continuations, as it adopts the try/catch/finally exception handling mechanism rather than the more powerful call/cc mechanism of SCHEME,
- strings are immutable, as SILK adopts the underlying Java type hierarchy with a few new classes (Pair, Symbol, Procedure, Closure, ...).
- SILK adopts the Java conventions for the syntax and semantics of numbers.

These deviations were made so as to simplify the use of SILK as a scripting language for Java.

Java, is a popular language to implement other languages in. For example, [25] identifies about 130 implementations for a few dozen languages which have been implemented in Java over the past five years. Currently, 15 of those are Lisp-related languages, with eight of them focused on Scheme. There are many reasons for this preponderance of Lisp/Scheme implementations; chiefly, Java is a simple object oriented language with syntax like C, single inheritance like Smalltalk, and garbage collection like Scheme. Java provides standard libraries that support web programming. It has the goal of being portable, with the motto "Write once, run anywhere". Java also has a noisy bandwagon, that can be hard to ignore.

Not only is Java a good target implementation language, but because Java provides no syntactic extension mechanism, Java applications can benefit from extension mini-languages. Scheme-like extension languages, such as SILK, can be especially effective because Scheme is a fertile soil for such mini-languages [2].

SILK started as a small Scheme implementation with limited access to Java. Each method and constructor had to be individually pulled across the Scheme/Java barrier. SILK is currently in its fourth implementation and Java meta-objects (classes, fields, methods and constructors) can be manipulated directly using a simple syntactic extension. SILK matches the Scheme type system to existing Java types, where it can. Also, SILK's syntax has been extended to allow for constants of all Java numeric types. This reduces the overhead of invoking Java methods from SILK, and reduces the ambiguity of dynamic method invocation.

The result is a Scheme dialect with direct, almost transparent, access to its implementation language, Java. In a C implementation of Scheme, such as Guile [16] or STK [14], after providing a Scheme implementation, one also provides extensions such as a Foreign Function Interface, a library of tools, such as SLIB[20], and an object oriented extension, such as GOOPS[15] or STKLOS[14]. In SILK, all of these features are obtained automatically via the simple Scheme/Java interface.

In the remainder of the paper we describe the main features of SILK programming, and the main features of SILK's implementation.

*This research was supported by NSF grant ITR-0082393

Type	Rule	Example
Class	*className* `".class"`	`> Dimension.class` `class java.awt.Dimension`
Constructor	*className* `"."`	`> (define d (Dimension.  100 200))` `java.awt.Dimension[width=100,height=200]`
Instance Method	`"."` *methodName*	`> (.getWidth d)` `100.0`
Instance Field	`"."` *fieldName* `"$"`	`> (.width$ d)` `100`
Static Method	*className* `"."` *methodName*	`> (System.getProperty "java.version")` `"1.2.2"`

Static Field	*className* `"."` *fieldName* `"$"`	`> (.println System.out$ "Hello Sailor!")`
		`Hello Sailor!`
		`#null`

Inner Class	*className* `"$"` *innerClassName*	`(import "java.awt.geom.*")`
		`> (Point2D$Double.  1.0 2.0)`
		`Point2D.Double[1.0, 2.0]`

Packageless Class	`"$"` *className*	`> ($Hello.main (list->array String.class '()))` `"Hello world!"`

Figure 1: Java Reflector Variable Syntax Rules

2 The Scheme/Java Interface in SILK

The key feature of SILK is that it allows Java classes, constructors, fields, and methods to be accessed directly through a variable constructed by adding a "." or "$" to its Java name according to the rules in Figure 1. We call these variables Java reflector variables.

The particular method or constructor to be used is determined by the SILK interpreter at runtime using the name of the method/constructor and the types of its arguments. This is discussed in more detail in the implementation section below. For an instance field, SILK returns an accessor function that takes an instance as its argument.

```
> (define d (java.awt.Dimension. 1 2))
java.awt.Dimension[width=1,height=2]
> (.height$ d)
2
> (.height$ d 10)
2
> d
java.awt.Dimension[width=1,height=10]
>
```

Static fields are treated as global variables so they can be assigned to using `(set!)`:

```
(set! Frog.count$ (+ Frog.count$ 1))
```

If SILK can't find a Java object with the appropriate name, the variable is treated as a normal Scheme variable.

To allow a less verbose access to Java, SILK provides an `(import)` procedure which can be used to simplify the naming of classes, much as the `import` statement does in Java, by dropping the package prefix.

```
;; Import a single class.
> (import "java.awt.Dimension")
#t

;; Import all the classes in a package.
> (import "java.util.*")
#t
```

Once a class or package has been imported, a class can be named by its name relative to the `(import)`

```
> (class "Dimension")
class java.awt.Dimension

> (class "Hashtable")
class java.util.Hashtable
```

Imports accumulate throughout the SILK session, and ambiguous class names cause an error. One can always specify a class unambiguously by giving the full package prefix for the class, or in the case of packageless classes by prepending a "$" to the class name.

2.1 Using Java reflection

SILK's implementation makes extensive use of reflection to implement dynamic method invocation, as described below. SILK programs can also make effective use of reflection. For example, a `(describe)` procedure that can describe any object produces output as shown in Figure 3 requires only 14 lines of code (as shown in Figure 2).

The static method `(AccessibleObject.setAccessible)` is a JDK 1.2 feature that allows an application to control access to reflective information. Here, the code allows accessibility to any field, even private ones. This is essential for the success of `(describe)`.

A simple inspector window that extends this idea to use JTables requires about 50 lines of code.

2.2 Wrapper classes

SILK uses Java reflection to provide access to any Java classes in the classpath. As of Java 1.3, the Java reflection package does not provide a general mechanism for constructing new classes, but it does allow one to construct classes implementing a set of interfaces provided at runtime.[1]

[1] One can always try the brute force approach of compiling the class to byte code and loading it on the fly, but this is not feasible in some embedded applications such as unsigned applets.

```
(define (describe x)
  (define (describe-fields x superclass)
    (let ((fs (.getDeclaredFields superclass)))
      (AccessibleObject.setAccessible fs #t)
      (iterate fs
        (lambda (f)
          (if (not
                 (Modifier.isStatic
                   (.getModifiers f)))
               (display
                 (string-append
                    "  " (.getName f) ": "
                   (.get f x) "\n")))))
     (let ((superclass (.getSuperclass superclass)))
       (if (not (isNull superclass))
           (describe-fields x superclass)))))
  (display (string-append x
    "\n is an instance of " (.getName (.getClass x))
                          "\n"))
  (describe-fields x (.getClass x)))
```

Figure 2: SILK code for `describe`

```
> (define h (Hashtable. 10))
{}
> (.put h "Fred" 1)
#null
> (.put h "Mary" 2)
#null
> (describe h)
{Mary=2, Fred=1}
 is an instance of java.util.Hashtable
  table: #(#null #null #null #null #null
          Fred=1 #null #null #null Mary=2)
  count: 2
  threshold: 7
  loadFactor: 0.75
  modCount: 2
  keySet: #null
  entrySet:
   java.util.Collections$SynchronizedSet@460583
  values: #null
#f
```

Figure 3: Example of use of the describe method

This restriction causes problems in pure SILK programming. For example, it is not possible to override the paint method of a window using reflection alone. The easy way out of this problem is to write a wrapper class in Java in which the desired behavior can be specified by a Scheme procedure. For example, the following wrapper class extends the paint method using a SILK procedure, `handler`, which is invoked with support from the SI (Simple Interface) class.

```
package mylib;
import silk.*;
public class MyCanvas
  extends java.awt.Canvas {
  private silk.Procedure handler;
  public MyCanvas(silk.Procedure handler) {
    MyCanvas(); this.handler=handler;}
  public void paint(java.awt.Graphics g) {
   if (handler != null)
       SI.call(handler,g); }}
```

This can be used as follows to create a window with a logo.

```
> (define (maketest)
    (define w (java.awt.Frame. "test"))
    (define c (mylib.MyCanvas. (lambda(g)
        (.setColor g (java.awt.Color. 0xff0000))
        (.setFont g (java.awt.Font. "Helvetica"
                        java.awt.Font.BOLD$ 24))
        (.drawString g "Silk" 50 50))))
    (.setBackground w (java.awt.Color. 0xffffff))
    (.add w c)
    (.resize w 300 220)
    (.show w)
    w)
```

Another case in which wrapper classes are required is in the use of the Arrays.sort method, which expects a parameter of interface type `java.util.Comparator` to compare elements of the array. This can be handled by writing the `SchemeComparator` wrapper class shown below. The Comparator interface requires the methods compare(), equals() and hashCode(). SchemeComparator is an implementation of this interface that takes a Scheme procedure, proc, that provides the behavior for the compare() method. The other methods are written in Java. Static methods from the SI (Simple Interface) class provide convenient access from the Java to the Scheme side of the SILK application. SI.call() calls the procedure, proc, with two arguments. From the Scheme side of SILK, one can do the sorting by wrapping the comparison procedure using the `SchemeComparator` constructor:

```
import silk.*;
public class SchemeComparator
implements java.util.Comparator {
  private Procedure proc;
  public SchemeComparator(Procedure proc)
   { this.proc = proc; }
  public int compare(Object a, Object b) {
    return ((Number) SI.call(proc, a, b)).intValue(); }
  public boolean equals(Object that) {
    return this == that ||
      this.getClass() == that.getClass() &&
      this.proc == ((SchemeComparator) that).proc; }
  public int hashCode() { return proc.hashCode(); }
}

/* example of use
> (define compare
    (lambda (a b)
      (if (< a b) -1
          (if (< b a) 1
              0))))
{silk.Closure ??[2] (a b)}
> (let ((x #(7 4 7 3 1 2 7)))
    (Arrays.sort x
       (SchemeComparator. compare))
    x)
#(1 2 3 4 7 7 7)
*/
```

SILK uses the wrapping technique to implement the class silk.Listener which allows Scheme to provide the behavior required by the 39 Swing and AWT Listener classes of Java

1.2. Here's a simple example of its use: where (Listener. P) creates a class that implements all of the Swing and AWT listener interfaces.

```
(import "javax.swing.*") (import "java.awt.*")
(define (demo)
  (let ((f (JFrame. "Example"))
        (b (JButton. "Press Me")))
    (.addActionListener
        b
       (Listener. (lambda (e)
          (.println System.out$ "Yow!"))))
    (.add (.getContentPane f) b
          BorderLayout.CENTER$ )
    (.pack f) (.show f)))
```

In each case, the arguments to the interface method are put into a list which is then passed to the handler procedure P which computes the return value, if any.

3 SILK as an embedded language

In this section, we show how to embed SILK in a Java application. In particular, we show how wrapper classes allow SILK to provide applets and servlets written in Scheme.

3.1 SILK Applets

The strategy for SILK applets is to define a wrapper class silk.SchemeApplet extending java.applet.Applet in which the applet "interface" methods `init, start, stop, destroy` are implemented using scheme closures. Moreover, the scheme code which defines these closures and the names of the scheme procedures which implement each of these methods is obtained from the applet parameters. For example, to include a simple Scheme interpreter applet on a web page, one adds the following applet element to the web page:

```
<applet code=silk.SchemeApplet archive=silk.jar
   height=300 width=100%>
  <param name="prog" value="minieval.silk">
  <param name="init" value="rep_loop">
</applet>
```

where `silk.jar` is the jar file containing the SILK runtime classes, and `minieval.silk` is a file containing the following Read-Eval-Print-Loop GUI code:

```
(import "java.awt.*")
(import "java.io.*")
(define (rep_loop thisApplet)
  (define t (TextField. " " 40))
  (define ta (TextArea. 20 40))
  (.setBackground t (Color. 255 200 155))
  (.setLayout thisApplet (BorderLayout.))
  (.add thisApplet "North" t)
  (.add thisApplet "Center" ta)
  (.addActionListener t (Listener11. (lambda(e)
    (.setText ta (.toString
      (tryCatch
        (eval (read (InputPort.
          (StringReader. (.getText t)))))
        (lambda(e) e))))))))
```

3.2 SILK Servlets

The wrapper class approach works equally well for servlets. In this case, we write a Java class in which the "interface" procedures for HttpServlets are handled by Scheme closures stored in public instance fields of the object. The most important interface fields are:

- `doGet(request,response)`
- `doPost(request,response)`
- `destroy(request,response)`

The first two are called when the servlet receives a "request" to create a "response" using either the Get or Post method, respectively. The latter is called when the servlet container is retiring the servlet, either to better manage system resources or in preparation for stopping the server. To deploy such an Scheme servlet in the Tomcat servlet container[10] one needs to modify the web.xml file of the web application, which specifies how URL's are mapped to servlets.

For example, to create a simple servlet "localtime.sss" to return the current local time one must add the following elements to the web.xml file:

```
<servlet>
 <servlet-name> localtime </servlet-name>
 <servlet-class> demo.schemeservlet.SchemeServlet
 </servlet-class>
 <init-param>
  <param-name> code </param-name>
  <param-value> WEB-INF/scheme/localtime.sss
  </param-value>
 </init-param>
</servlet>
<servlet-mapping>
  <servlet-name> localtime </servlet-name>
  <url-pattern> /servlet/localtime </url-pattern>
</servlet-mapping>
```

and one creates a file "scheme/localtime.sss" containing the following Scheme expression, where the instance field `do_get` provides the `doGet()` behavior as a closure, if it is non-null.

```
(lambda(httpservlet)
 (.do_get$ httpservlet
  (lambda (request response)
   (let ((out (.getWriter response)))
         (.setContentType response "text/html")
     (.println out (string-append
      "<html>
       <head><title>LocalTime</title></head>
       <body>
       <h1> The current local time is</h1>"
       (java.util.Date.)
      "</body></html>")))))
 (.do_post$ httpservlet
   (lambda (request response)
     (.doGet httpservlet request response))))
```

The Servlet container creates a SchemeServlet object when the appropriate URL servlet-mapping rule fires. The SchemeServlet object `S` reads the expression `E` stored in the file `localtime.sss` and will evaluate `(E S)`. This allows the expression `E` to set the `do_get` and `do_post` fields of `S`.

4 SILK as a Scripting Language

Beckman argues that scripting languages are inevitable [2]. In the 80's Jon Bentley popularized the idea of Little Languages [3]. Such a language can be used to describe in a compact way, one aspect of your project, graphical user interface layout for example. Beckman argues that this decoupling of aspects is essential, because the only other option is to keep changing all the source code. He also argues that little languages often grow to become more complete languages, adding control structure, classes, etc. TCL and Visual Basic are unfortunate examples of this trend. Beckman further argues that Scheme is an excellent choice for a little language, because it is also a complete language in which other extension languages can be easily embedded.

SILK is a fertile soil for such mini-languages. As an example, consider an HTML generation mini-language based on two procedures (tag) and (tag-seq). Both return a string. The function (directory-listing) generates an HTML page where each file in the directory is represented as a row in a table with a hyperlink to the file.

```
(define (directory-listing file)
  ;; Directory listing of file.
  (define (yesify x) (if x "yes" "no"))
  (define (row f)
      (tag 'tr
        (tag 'td
           (let ((x (.getName f)))
             (if (.isDirectory f)
                 (tag `(a (href ,(.toURL f))) x)
                 x)))
        (tag 'td (.length f))
        (tag 'td (Date. (.lastModified f)))
        (tag 'td (yesify (.canRead f)))
        (tag 'td (yesify (.canWrite f)))))
  (tag-seq
   (tag 'head (tag 'title "Directory"))
   (tag 'body
     (tag `(table (border 1))
       (tag `caption
         (tag 'em
            "Directory listing of " file))
         (tag 'tr
           (map (lambda (x) (tag 'th x))
               '(Name Length "Last Modified"
                 Readable Writeable)))
         (map* row (.listFiles file))
  ))))
```

This definition has the same shape as the page it is defining. Namely a head, followed by a body consisting of a table. The table has a caption, and its first row is a sequence of a headers followed by one row per file. It would be hard to get such a compact representation using either a Java Servlet, or Java Server Pages.

4.1 SILK Scheme Server Pages

The tag mini-language is quite helpful when writing SILK servlets as described above which must generate html. For example, a file browser servlet can be implemented in SILK by wrapping the appropriate scheme servlet code around the directory-listing function. To simplify the process of "wrapping servlet code" around an expression, we have implemented a simple SILK servlet (sssp.sss – the SILK Scheme Server Pages servlet) which is invoked whenever a file with the suffix .sssp is specified by the client. This servlet reads the file as a scheme expression, evaluates it in a context in which request, response, and httpservlet are bound to their current values, and then writes the resulting expression to the output stream of the response. This is similar to LAML[7] and LSP[13], but is much more tightly bound to Java. For example, to rewrite the localtime servlet from Section 3.2 as an SILK Scheme Server Page, one creates a file named localtime.sssp in the scheme webapp directory of the Tomcat server [10] and stores the following scheme expression in the file:

```
(tag 'html (tag 'head (tag 'title "Local Time"))
 (tag '(body (bgcolor white))
   (tag 'h1 "Current Local Time is")
   (java.util.Date.)))
```

4.2 JLIB – a GUI-building language

The JLIB mini-language allows one to implement simple Graphical User Interfaces in a declarative style. There are several key ideas behind JLIB

- each GUI component is represented by a Scheme procedure whose arguments describe the contents and properties of that component. The procedure uses the type of each argument to determine the default interpretation of that argument.
- Actions are expressed as closures.
- The labelled components (textfield, button, choice, label, textarea) are viewed I/O devices on which strings and expressions can be read or written
- Components can be given local names using a "tagger" which either stores a component in a hash table (as in (t NAME OBJ)) or looks up the object from its local name (as in (t NAME)).
- Special syntax is provided for commonly used objects and containers (e.g. colors, fonts, layout managers).

For example, the following code implements a simple guessing game. Note the initial line loads in the compiled form of the JLIB.

```
(jlib.JLIB.load)
(define (guesswin)
 (define guess
  (let ((ans ()) (t (maketagger)))
   (window "Guessing game" yellow
    (menubar
      (menu "File"
        (menuitem "New Game" (action (lambda(e)
          (set! ans (round (* 100 (Math.random))))
          (writestring (t "prompt")
            "Guess a number between 0 and 99"))))
        (menuitem "Quit" (action (lambda(e)
          (.hide guess))))))
    (border
     (north  (label "Silk Guessing Game"
                    (HelveticaBold 18)))
     (center (row
       (t "prompt"
        (label "Welcome to the Silk Guessing Game"))
       (t "response"
        (textfield "" 10 (action (lambda(e)
         (let ((g (readexpr (t "response"))))
          (writestring (t "prompt") g " is "
           (cond ((= g ans) "correct!")
                 ((< g ans) "too low")
```

```
                  ((> g ans) "too high"))))
          (writestring (t "response") ""))))))))))
 (.pack guess) (.show guess))
```

JLIB has been successfully used to teach programming to non-science students at the college level over the past four years [17].

4.3 SILK as a Debugging Tool

Because SILK applications are interactive, Java objects can be manipulated during debugging and software development. This provides a unique view into Java applications.

For example, the solution to the following puzzle is easily revealed:

```
/** Can you write a class that can
    view or alter the secret? **/
public final class Holder {
  private int secret;
  public Holder(int secret)
  { this.secret = secret;}
  public final boolean isHappy()
  { return secret > 25; }
  private final Object booster() {
    return new Object() {
      private final void boost() { secret++; }}; }
  private static boolean isHolder(Object x) {
    return x.getClass() == Holder.class; }
}
```

By normal Java semantics, the secret should not be accessible. However, if one uses reflection to list the methods declared on the class we find something interesting (where the `iterate` procedure is discussed in section 5.4.

```
> (iterate (.getDeclaredMethods Holder.class) print)
static java.lang.Class Holder.class$(java.lang.String)
static int Holder.access$0(Holder)
static void Holder.access$1(Holder,int)
private final java.lang.Object Holder.booster()
public final boolean Holder.isHappy()
private static boolean Holder.isHolder(java.lang.Object)
#f
```

Recall that an internal "$" in a Java reflector variable signifies an inner classes. The two **Holder.access$?** methods are unexpected additions to the Class. They are added by the Java compiler to allow access to the secret by the inner class. This is how inner classes were added to JDK 1.1 without changing the JDK 1.0.2 Virtual Machine. Thus any private field that is referenced by an inner class is available to a Class in the same package.

5 Implementation

Before discussing implementation details we provide a brief history of the versions of SILK to help motivate the design decisions chosen. The initial version of SILK was written in about 20 hours with about 650 lines of code[22]. The primary goals were to develop a Lisp that was small, fast to load (even over the web), easy to understand and modify, and that could interface to Java. SILK expanded to about 50KB of Java byte code over the next few months as it was extended to pass all of the tests in Aubrey Jaffer's online r4rstest.scm [19] test suite which tests Scheme compliance with the R4RS standard. This version also had two procedures `(constructor)` and `(method)` that could be used to construct SILK procedures that invoked a Java constructor or method. The SILK 2.0 version compiled Scheme syntactic expressions into Code objects that could be more efficiently evaluated. The SILK 3.0 version added generic functions [9]. The procedure `(import)` would make all the methods of a class accessible as generic functions [1]. While this approach provided easy access to Java, it had two problems:

- Since it imported all the methods of a Class rather than just the ones that would be used, some unnecessary work was always done. This increased startup time.
- Java method names could collide with non-generic SILK primitives. So these methods were renamed by adding a "#" suffix.

The SILK 4.0 version started as a reimplementation of the SILK kernel in which the main evaluation loop was optimized by first analyzing each expression to be evaluated (performing variable lookup, macroexpansion, etc.) and then executing the analyzed code. It had roughly the capabilities of SILK 1.0 but was an order of magnitude faster. This version became the basis of current development. Complete support for Java's numeric types, and a new implementation of generic functions, that is language independent were added.

5.1 Object type hierarchy

An important issue for a Java implementation of Scheme is how the Scheme type hierarchy should be implemented as Java classes. A reasonable object-oriented way is to implement the Scheme types is as a class hierarch under Object, as is done by Scheme package [18] and HotScheme [12]. This way, most of the 150 or so Scheme primitives can be implemented as instance methods. While this simplifies the implementation, it requires wrapper objects that take more space and make crossing the Scheme/Java frontier more costly since each Scheme type must be converted in either direction.

The first implementations of SILK implemented a Scheme type with its "closest" Java type. For example, the Scheme types `(char?)` and `(string?)` were implemented as a Java types `Character` and `char[]`, respectively. This allowed Scheme semantics that strings are mutable, while in Java they are not. Also, since a Java String is immutable, it was converted to a Scheme symbol (also immutable) when a Java method returned it to Scheme.

This lead to several complications;

- Any string returned from Java to SILK was interned as a symbol. This lead to excessive memory use in some applications.
- If you actually wanted to get a string from Java you had to do extra work like `(symbol->string (.toString x))`.
- Since a `char[]` is not used that often in Java, it was converted to a `String` when passed to Java, which added overhead. It also made dynamic method lookup more ambiguous.

The current solution to these problems is to implement the Scheme (string?) type as the Java class `String`. This requires Scheme's strings to be immutable, so that the primitive procedure `(string-set!)` no longer works.

Numeric Types Originally, Scheme's `(number?)` type was implemented by Java classes `Integer` and `Double`. This was reasonable for many Scheme programs. However, Java would occasionally expect or return a `Number` of a different type. After that, SILK arithmetic might or might not work. For example, to compare the last modified date of two files you had to carefully convert a `Long` into a `Double` by first converting it into a `String`:

```
(define (last-modified file)
  (Double. (.toString (.lastModified file))))
```

The current solution to this problem is to implement all Java numeric types. SILK syntax adapts the Java syntax for numbers so that `Long` and `Float` constants can be specified as `3L` or `3.14F`. Moreover SILK arithmetic follows the same conversion rules as Java.

Character constants Scheme and Java have different print representations for character constants. For example, the character "c" is represented as `#\c` in Scheme and 'c' in Java. In SILK we accommodate Java by allowing both `#\c` and #'c' syntax and allowing the user to specify which syntax to use for output.

Null Java, as in C and C++, can return the object `null` to represent the lack of a return value of any particular type. While this is similar to Common Lisp's notion of `nil`, there is no equivalent in Scheme.

The first versions of SILK equated Java `null` with the Scheme empty pair, `'()`. The current version of SILK prints Java null as "#null" which is distinct from '(). This means that to check for a `#null` being returned from Java into the variable `x` one must say: `(eq? x #null)` rather than `(null? x)`.

5.2 Primitive procedures

A procedure object must be written for most Scheme primitive procedures (about 150). One way to do this use a separate class, or an anonymous inner class. However, even a tiny inner class produces about a 500 byte class file which can make the resulting .jar file fairly large. Our approach is to use a single class, silk.Primitive, that implements the primitives. Instances of the class are distinguished by an `int` that is used in a case statement to dispatch to the right primitive code. Scheme automatically generates this class. We have also shown how to implement all of the Scheme primitives directly in SILK using only the Java reflector variables, but this approach results in somewhat slower code because Java's reflection method invocation is used.

5.3 Dynamic method invocation

A key feature of SILK is that Java methods can be invoked dynamically at runtime. This is done by following, as much as possible, the Java method selection semantics. However, while Java does part of the selection at compile time, and the rest at runtime, SILK does the full selection at runtime, based on the runtime types of the arguments.

There are some complications to this approach [8, 5, 1], such as:

- A null argument provides no type information. Though this does not seem to happen often in practice.
- Automatic type conversion, such as type widening makes method selection more complicated.

SILK's current approach is to not perform any type widening during method selection. This keeps method lookup unambiguous.

Our approach to dynamic method invocation is language independent. SILK determines at read time if a generic call site refers to an instance or static method invocation. In the static case, the lookup is done based first on the method name, and then on the types of the arguments. In the instance case, the lookup is done based first on the target argument type, then the name of the method, and finally the types of the remaining arguments.

For 80% of the static methods and 70% of the instance methods the lookup leads to a single method that is invoked directly [1]. Otherwise each potential method is searched by comparing the types of its arguments, to determine the most specific method to be invoked.

5.4 Generics

The implementation of dynamic invocation requires Silk to examine the types of the arguments to determine the most closely matching Java method or constructor. Silk also provides a similar facility for Scheme procedures.

The form `(define-method)` can be used to define a method as part of a generic function. For example, here we define a generic `(iterate)` that iterates over any Java collection, list, or array:

```
(import "java.lang.reflect.*")
(define (identity x) x)
(define (make-iterator more? next seq)
  (lambda (items action)
    (let loop ((items items))
      (if (more? items)
          (begin (action (next items))
                 (loop (seq items)))))))
(let  ((it (make-iterator .hasMoreElements
                 .nextElement identity)))
 (define-method
    (iterate (items Enumeration) action)
    (it items action)))
(let ((it (make-iterator .hasNext
                         .next identity)))
 (define-method
    (iterate (items Iterator) action)
    (it items action)))
(let   ((it (make-iterator pair? car cdr)))
  (define-method
    (iterate (items silk.Pair) action)
    (it items action)))
(define-method
  (iterate (items Map) action)
  (iterate (.values items) action))
(define-method
  (iterate (items Collection) action)
  (iterate (.iterator items) action))
(define-method (iterate (items Object)action)
  (if (.isArray (.getClass items))
      (iterate-array items action)
      (error "Don't know how to iterator over " items)))
```

Here we use a functional programming style of abstracting the "iterator pattern" into the procedure `(make-iterator)`. In Java, an array is a subclass of `Object`, so the `Object` version of the `(iterate)` method must check for an array to iterate over its elements. The procedure `(define-method)` is similar to Common Lisp's `(defmethod)` except that

- An argument specializer is a Java class.
- Generic functions take a fixed number of arguments.
- There is no `(call-next-method)`.

5.5 Threads

SILK provides access to Java's threaded execution model by making the class Procedure, the superclass of all Scheme procedures, implement the Runnable interface. Thus Scheme procedures can be invoked directly in separate threads. Here's an example that starts a thread that simply sleeps for 10 seconds (10,000 ms) and then prints the date.

```
> (begin
  (.start (Thread. (lambda ()
                     (Thread.sleep 10000L)
                     (print (Date.)))))
  (print (Date.)))
Mon Aug 14 21:44:24 EDT 2000
Mon Aug 14 21:44:24 EDT 2000
> Mon Aug 14 21:44:34 EDT 2000
```

5.6 Exceptions

SILK also adopts the Java exception handling model. The approach taken is to catch all Java exceptions and rethrow them as RuntimeExceptions. SILK provides a macro `(tryCatch Expr Proc)` which evaluates the Expr and, if an exception `e` is thrown, returns the value `(Proc e)`. For example,

```
> (define (newdiv x y)
    (tryCatch (/ x y) (lambda(e) 0)))
newdiv
> (newdiv 1 0)
0
>
```

6 Interfacing Scheme using the Proxy class

JDK 1.3 added the classes java.lang.reflect.InvocationHandler and java.lang.reflect.Proxy that can be used to define a class that implements an interface dynamically. Essentially, a proxy class implements a set of Interfaces by calling the invoke() method of an InvocationHandler when a method is invoked on the proxy.

SILK takes advantage of this by defining an Invocation-Hander that invokes a Scheme procedure to perform the actual method invocation.

```
package silk;

public class SchemeInvocationHandler
     implements InvocationHandler {
  Procedure proc;
  public SchemeInvocationHandler(Procedure proc)
   { this.proc = proc; }
  public Object invoke(
     Object proxy, Method method, Object[] args)
    throws Throwable {
    return SI.call(proc, proxy, method, args);
  }}
```

We can now define the procedure (Proxy) that takes an Class[] of interfaces, and a handler procedure and returns an object that implements the interfaces by invoking the handler:

```
(define (Proxy interface[] handler)
  (Proxy.newProxyInstance
   #null
   interface[]
   (SchemeInvocationHandler. handler)))
```

The handler takes three arguments, the proxy object, the Method to be invoked, and the Object[] of any arguments. Using this, we can define the Listener class described above dynamically, at runtime:

```
(define (Listener handler)
  (Proxy listener-interfaces
         (lambda (proxy method argv)
           (handler (vector-ref argv 0)))))
```

The handler argument is a procedure of one argument, an Event, and listener-interfaces is an array of type Object[] of listener interfaces.

6.1 Tracing Java method invocations

The Proxy class is particularly useful for wrapping behavior around each method invocation on a delegate object. We can capture this idiom easily in Scheme:

```
(define (delegate-to delegate handler)
  (Proxy
   (.getInterfaces (.getClass delegate))
   (lambda (proxy method argv)
     (handler delegate method argv))))
```

We can then define a specialization of it for tracing interface method invocations on an object:

```
(define (trace-handler delegate method argv)
  (print (list 'call: delegate (.getName method) argv))
  (let ((result (.invoke method delegate argv)))
    (print (list 'return: result))
    result))
(define (trace-object x) (delegate-to x trace-handler))
```

We can use it to trace operations on a Hashtable, for example:

```
> (define h (Hashtable. 10))
{}
> (define ph (trace-object h))
(call: {} "toString" #null)
(return: "{}")
{}
> (.put ph "red" 1)
(call: {} "put" #("red" 1))
(return: #null)
#null
> (.put ph "green" 2)
(call: {red=1} "put" #("green" 2))
(return: #null)
#null
> ph
(call: {green=2, red=1} "toString" #null)
(return: "{green=2, red=1}")
{green=2, red=1}
> (.get ph "green")
(call: {green=2, red=1} "get" #("green"))
(return: 2)
2
```

Implementation	Java files	Lines	Scheme files	Lines	Total Lines	Generics
SILK 1.0	12	1,905	0	0	1,905	No
SILK 2.0	20	2,778	0	0	2,778	No
SILK 3.0	28	3,508	5	510	4,018	Yes
Lisc 1.2.3	27	3,296	5	1,239	4,535	No
SILK 4.0	24	3,869	4	710	4,579	Yes
Skij [8]	27	2,523	44	2,844	5,367	Yes
HotScheme	114	5,674	0	0	5,674	No
Jaja [9]	66	5,760	2	4,088	9,848	No
Kawa [10]	273	16,629	14	708	17,337	No
Scheme package	252	13,249	39	1,345	14,594	No

Figure 4: Scheme implementation statistics

7 Related Work

There are several other Scheme implementations in Java we are aware of, that we briefly describe. Figure 4 shows statistics from these implementations.

SILK: The first four rows of the table show sizes for each version of SILK. In the current version, about 1,000 lines of Java, in four classes, are automatically generated from Scheme. This includes about 150 Scheme primitives and the generic Listener class.

Lisc: LISC [21] is a very lightweight, from scratch, fully functional scheme interpreter. It is very comparable to SILK, but tries to be more modular, faster, and complete, and provides a few useful enhancements outside the Scheme standard. LISC does not have dynamic access to Java, but does support static access to Java through Modules which are Java objects extending the Module class which can be loaded into the LISC interpreter.

Skij: Skij is a Scheme advertised as a scripting extension for Java [23, 8]. It is similar in capabilities to SILK and has extensive Java support including (peek) and (poke) for reading and writing slots, (invoke) and (invoke-static) for invoking methods, and (new) for constructing new instances of a Java class.

(new) (invoke) and (invoke-static) invoke the appropriate Java method using runtime looked up based on all of its arguments. This approach is similar to SILK's. However, SILK's generic functions also allow Scheme methods to be added.

Jaja: Jaja [24] is a Scheme based on the Christian Queinnecs wonderful book "Lisp in Small Pieces". It includes a Scheme to Java complier written in Scheme, because it requires only 1/3 the code of a Java version. Compared to SILK, Jaja is written in a more object oriented style. Like SILK, Jaja uses a single class (Jaja in Jaja, and U in SILK) to provide globals and utility functions. Unlike SILK, in Jaja, each Scheme type has one or more Java classes defined for it. Similar to SILK, in Jaja, the empty list () is represented as an instance of the class EmptyList. All Jaja objects are serializable.

Kawa: Kawa [11] is an ambitious Scheme implementation. It includes a Scheme to Java byte code compiler. Each function becomes a Java class compiled and loaded at runtime.

Scheme package: In the Scheme package [18] the Scheme object Types are arranged in its own class hierarchy. Each primitive procedure is implemented as its own class. This requires over 250 class files, resulting in a 250 kilobyte .jar file.

HotScheme: HotScheme [12] also uses a Scheme object hierarchy and one class per primitive as the Scheme package does. The hierarchy is split so that all Scheme objects, except #f (false) inherit from the TRep (representation of true) class. Primitives come with their own documentation strings. The HotScheme home page is an interactive Scheme applet.

8 Conclusion

This paper describes SILK, a compact Java implementation of Scheme. During the four implemenations over the past three years, SILK has moved from being a mostly compliant Scheme to being a Scheme/Java hybrid language. SILK Scheme code has immediate access to any object or behavior written in Java. Also, by using a wrapper class, or JDK 1.3's Proxy class, a Java application can access behavior written in Scheme.

The Scheme side of SILK benefits by having access to Java's extensive library of classes. For example, it is easy for SILK to provide applets and servlets that can be written in Scheme.

The Java side of SILK benefits by having a scripting language that can be used for various tasks, such as GUI layout, as in JLIB. Because of SILK's access to Java's reflection, it is effective both as a runtime prototyping and debugging environment, as well as a compile-time metaprogramming scripting language.

Acknowledgements

We thank Tom Mitchell, Jeff Tustin, and Geoffrey Knauth for reading an early draft and providing other assistance.

References

[1] K.R. Anderson, T.J. Hickey, "Reflecting Java into Scheme", Lecture Notes In Computer Science v.1616, pp. 154-174, 1999.

[2] Brian Beckman, "A scheme for little languages in interactive graphics", Software Practice and Experience, 21(2), pp. 187-208, Feb, 1991.

[3] J.L. Bentley, "More Programming Pearls", Addison-Wesley, Reading, MA, 1988.

[4] William Clinger and Jonathan Rees, editors. "The revised^4 report on the algorithmic language Scheme." In ACM Lisp Pointers 4(3), pp. 1-55, 1991 `http://www-swiss.ai.mit.edu/~jaffer/r4rs_toc.html`

[5] Suchitra Gupta, Jeff Hartkopf, and Suresh Ramaswamy, "Covariant specialization in Java", Journal of Object Oriented Research, pp. 16-20, May 2000.

[6] Peter Norvig, "Paradigms of Artificial Intelligence Programming: Case Studies in Common Lisp" , Morgan Kaufman, 1992,
`http://www.norvig.com/paip.html`

[7] , Kurt Normark, "Programming World Wide Web pages in Scheme" ACM Sigplan Notices, 34(12), 1999
`http://www.cs.auc.dk/~normark`

[8] Michael Travers, Java Q & A, Dr. Dobb's Journal, pp. 103-112, Jan. 2000.

Website References

[9] Ken Anderson, A version of SILK with generic functions,
`http://openmap.bbn.com/~kanderso/silk/jlib`

[10] Apache.org, Tomcat: Java Servlets and JavaServer Pages Reference Implementation
`http://jakarta.apache.org`

[11] Per Bothner, Kawa the Java-based Scheme System,
`http://www.cygnus.com/~bothner/kawa.html`

[12] Gene Callahan and Brian Clark, 1997 HotScheme
`http://www.stgtech.com/HotScheme`

[13] Kaelin Colclasure and Sunil Mishra, Lisp Server Pages software
`http://lsp.everest.com`

[14] Eric Gallesio, STk Reference Manual
`http://kaolin.unice.fr/STK`

[15] GNU,GOOPS: The Guile Object Oriented Programming System,
`http://www.gnu.org/software/goops/goops.html`

[16] The Guile Project at GNU
`http://www.gnu.org/software/guile`

[17] Timothy J. Hickey, CS2a: Introduction to Computers, Brandeis University, Class Notes and webpage
`http://www.cs.brandeis.edu/~tim/Classes/Aut00/CS2a`

[18] Stephane Hillion, The scheme package,
`http://www-sop.inria.fr/koala/shillion/sp/index.html`

[19] Aubrey Jaffer, r4rstest.scm,
`ftp://ftp-swiss.ai.mit.edu/pub/scm/r4rstest.scm`

[20] Aubrey Jaffer, SLIB: The Portable Scheme Library, Version 2c5,
`http://www-swiss.ai.mit.edu/~jaffer/slib.html`

[21] Scott G. Miller, LISC
`ftp://ftp.gamora.org/pub/gamora/lisc`

[22] Peter Norvig, SILK: Scheme in Fifty K,
`http://www.norvig.com/SILK.html`

[23] Mike Travers, Skij, IBM alphaWorks archive,
`http://www.alphaworks.ibm.com/formula/Skij`

[24] Christian Queinnec, JaJa: Scheme in Java,
`http://www-spi.lip6.fr/~queinnec/WWW/Jaja.html`

[25] Robert Tolksdorf, Programming Languages for the Java Virtual Machine,
`http://grunge.cs.tu-berlin.de/~tolk/vmlanguages.html`