kanderson

\\server\name

PSCRIPT Page Separator

Reflecting Java into Scheme

Kenneth R. Anderson[1], Timothy J. Hickey[2]

[1]BBN Technologies, Cambridge, MA, USA KAnderson@bbn.com,
[2]Brandeis University, Waltham, MA, USA tim@cs.brandeis.edu

Abstract. We describe our experience with SILK, a Scheme dialect written in Java. SILK grazes symbiotically on Java's reflective layer, enriching itself with Java's classes and their behavior. This is done with three procedures. `(constructor)` and `(method)`provide access to a specific Java constructor or method respectively. `(import)` allows the entire behavior of a class to be imported easily. `(import)` converts Java methods into generic functions that take methods written in Java or Scheme. In return, SILK provides Java applications with an interactive development and debugging environment that can be used to develop graphical applications from the convenience of your web browser. Also, because SILK has introspective access into Java, it can also be used for compile time metaobject scripting. For example, it can generate a new class using an old one as a template.

1 Introduction

Java's reflective layer provides access to metaobjects that reflect primitive, class, array, field, constructor, and method objects. There are obvious limitations to its reflective capabilities. For example, one can only affect the behavior of the running program by invoking methods and accessing fields of objects. One cannot define or redefine a new class or method. Also, one cannot subclass any metaobjects.

Despite these restrictions, Java's reflective capabilities are used successfully by many Java facilities, such as serialization, remote method invocation (RMI), and Java Beans. However, programming in base Java, is different from programming in the reflective layer. While Java is a statically typed language, the reflective layer is more dynamic. This suggests that a dynamic language, such as Scheme, could be ideal for programming the reflective layer. Scheme could be used as a dynamic language at runtime, and as a dynamic metalanguage at compile time.

Here we describe SILK (Scheme in about 50 K), a compact Scheme dialect written in Java. Because Java is reflective, considerable power can be gained relatively easily. For example, only two functions, `(constructor)` and `(method)`, need to be added to Scheme to provide complete access to the underlying Java.

Following Common Lisp, SILK provides generic functions. These generic functions accept several method types:

– Java static methods, instance methods, and constructors
– Methods written in Scheme that dispatch on Java classes

By arranging for Java to do most of the work, method dispatch is efficient. For most generic functions, the method dispatch overhead is only one Java method invocation beyond the normal overhead of invoking a reflected Java method through Scheme.

The foreign function interface between Scheme and Java is virtually transparent. The function `(import <class name>)` makes the methods and final static fields of the Java class named `<class name>` accessible to Scheme.

SILK started out simply as a Scheme implementation in Java. However, as its use of reflection has grown, it has become a powerful tool for controlling and developing Java applications. The following sections describe what programming in SILK is like, focusing on issues related to using reflection and the implementation of generic functions.

As this paper is concerned with Scheme, and Java reflection, we begin with a very brief introduction to the essential features of these topics. Anyone familiar with these topics can simply skip to the next section. Anyone with some knowledge of object-oriented programming and reflection should be able to follow this paper easily.

1.1 Java Reflection

Java is a statically typed object-oriented language, with a syntax similar to C. Java is a hybrid language with both primitive types, like `int` and `double`, and class based object types descending from the class `Object`. A Java program is built out of a set of classes. Java classes support single inheritance of implementation (extends) and multiple inheritance of interfaces (implements). An interface is a class that describes an object in terms of the protocol (set of methods) it most implement, but provides no implementation itself.

A class definition can contain other class definitions, static and instance fields, and static and instance methods.

A Java method has a signature, which is its name, and the types of each of its parameters. The signature is used at compile time to focus the choice of which method is to be invoked.

Java has two types of methods. An "instance method" has a distinguished first argument as well as possibly some parameters. When such a method is invoked, the dynamic type of this argument is used to select the appropriate method to invoke at runtime, while the declared types of the parameters are used to choose a signature at compile time.

A "static method" does not have a distinguished argument, but may have other parameters. It fills the role of a function in other languages. The choice of which static method to invoke is determined completely at compile time.

The `java.lang` and `java.lang.reflect` packages provide metaclasses that reflect the class and its components. To invoke a method, one must first ask the class that defined the method for the method's metaobject, using `getMethod()`, which takes the methods name, and an array of argument types as parameters. The method can be invoked by using its `invoke()` method which takes a parameter which is an object array of the arguments. Java provides a set of wrapper classes that let one convert a primitive type to an object. Thus for example, an `int` can be represented as an instance of class `Integer`. Here's an example of using the reflection layer to construct a hash table of size 10 and then invoke a `put ()` method on it:

```
package elf;
import java.lang.reflect.*;
import java.util.Hashtable;

public class Reflect {
  public static void main(String[] args) {
    try {
      // Hashtable ht = new Hashtable(10);
      Class[] types1 = new Class[] { Integer.TYPE };
      Constructor c =
        Hashtable.class.getConstructor(types1);
      Object[] args1 = new Object[] {new Integer(10)};
      Hashtable ht = (Hashtable) c.newInstance(args1);

      // ht.put("Three", new Integer(3))
      Class[] types2 = new Class[] {Object.class,
                                    Object.class };
      Method m = Hashtable.class.getMethod("put",
                                           types2);
      Object[] args2 =
        new Object[] { "Three", new Integer(3) };
      m.invoke(ht, args2);

      System.out.println(ht);  // Prints: {Three=3}
    } catch (Exception e) { e.printStackTrace(); }}}
```

1.2 Scheme

Scheme is a dynamically typed language that uses simple Lisp syntax. A Scheme program is built out of a sequence of expressions. Each expression is evaluated in turn.

Scheme provides a set of primitive types. The following exhibit shows the Scheme type name and its Java implementation used in SILK:

Scheme	**Java**
boolean	Boolean
symbol	silk.Symbol
char	Character
vector	Object[]
pair	silk.Pair

```
procedure    silk.Procedure
number       Number
  exact      Integer
  inexact    Double
string       char[]
port
 inputport   silk.InputPort
 outputport  java.io.PrintWriter
```

A Scheme procedure takes arguments of any type. Type predicates can be used to identify the type of an object at runtime.

While Scheme would not be considered "object oriented", the Scheme community has developed SLIB [1], a standard library that provides more complex data structures built from these types, including several object oriented extensions.

The power of Scheme comes from such features as:

1. A compact and clear definition, requiring only 50 pages, including its denotational semantic, example, references, and index.
2. A procedure can construct and return a new procedure, as you would expect from a functional language.
3. The syntax is so simple that new minilanguages can extend the language easily using functions or Scheme's macro facility, `(define-syntax)`.

Both Java and Scheme provide garbage collection.

2 SILK's Access to Java

SILK offers two distinct interface to Java which are discussed in the following sections.

2.1 Primitive Access to Java is Easy

Originally, SILK provided two primitive procedures for accessing Java behavior:

```
(constructor CLASSNAME ARGTYPE1 ...)
(method METHODNAME CLASSNAME ARGTYPE1 ...)
```

The `(constructor)` procedure is given a specification of a Java constructor, in terms of a class name and a list of argument type names, and returns a procedure implementing that constructor. The `(method)` is given a specification of a Java method, in terms of a method name, class name, and a list of argument type names, and returns a procedure implementing that method.

So, for example, we can use the `java.math.BigInteger` package to see if the number, 12345678987654321, is probably prime (with a probability of error of less than 1 part in 2^{-10} when the result is `#t`):

```
> (define isProbablePrime
    (method "isProbablePrime" "java.math.BigInteger"
            "int"))
isProbablePrime
```

```
> (define BigInteger
    (constructor "java.math.BigInteger"
                 "java.lang.String"))
BigInteger
> (isProbablePrime (BigInteger "12345678987654321") 10)
#f
```

It is useful to have additional access to Java's reflection layer. Here we define the procedure `(class)` that returns a class object given its full name string:

```
> (define class (method "forName" "java.lang.Class"
                        "java.lang.String"))
class
> (define HT (class "java.util.Hashtable"))
HT
> HT
class java.util.Hashtable
```

Here we define a procedure, `(get-static-value)` that given a class and a string naming a static field, returns the value of the corresponding static field. We then ask for the value of the `TYPE` field of the class `Void`. In Java, that would be simply `Void.TYPE`.

```
>(define get-field
    (method "getField" "java.lang.Class"
            "java.lang.String"))
get-field
>(define get-field-value
    (method "get" "java.lang.reflect.Field"
                  "java.lang.Object"))
get-field-value
>(define (get-static-value class field-name)
   (get-field-value (get-field class field-name) '()))
get-static-value
>(get-static-value (class "java.lang.Void") "TYPE")
void
>
```

2.2 But, Procedures Aren't Generic Enough

This approach was used extensively for a year. However, two problems became apparent:

1. Procedures are not generic. Procedures must be carefully named apart to avoid name conflicts. There are many potential conflicts, such as:
    - classes `java.util.Hashtable` and `java.lang.reflect.Field` both provide a `put` method.
    - class `java.lang.Object` provides a `toString` method, while `java.lang.reflect.Modifier` provides a static `toString` method.

- methods can be overloaded, so for example, class `java.lang.StringBuffer` provides 10 methods named `append`.

2. For each Java method or constructor one needs, it must be named and defined separately. Even if only a few methods of a class are used, this can be a fair amount of work.

2.3 (import) Lifts Java Classes into SILK Wholesale

To overcome this problem, an (import) function was added used to import the static and instance methods of a class. The goal was to make this similar to the import statement in Java. Here we import Hashtable:

```
> import "java.util.Hashtable")
importing java.util.Hashtable in 161 ms.
#t
>
```

The result of the `(import)` is that:

- The global variable `Hashtable.class` is given the value of the class `Hashtable.`
- Each public instance method applicable to `Hashtable` is made a generic function. This includes inherited methods, such as `clone()`, and `toString()` that are inherited from `java.lang.Object`. Generic functions, whose name conflicts with an existing Scheme procedure, have "#" add to the name as a suffix. Such conflicts include `load#`, `substring#`, `length#`, `apply#`, `list#`, `append#`, and `print#`.
- Generic functions are also made for each static method, but they must be named apart from instance methods. They are given a name that looks like `Class.method`. See the example uses of such methods below.
- A generic constructor function named after the class, `Hashtable`, is defined. Thus one does not need to say "new", just use the name of the class, `Hashtable`. This approach is similar to Haskell or ML. It makes constructing nested objects a bit more compact. It also fits well with Scheme's syntax.
- Each global constant is represented in Java as a `public static final` field is assigned to a global variable of the form `Class.field`.

We can immediately start using Hashtables:

```
>(define ht (Hashtable 20))
ht
>ht
{}
>(put ht 'clone 1)
()
>ht
{clone=1}
```

```
>(put ht 'zone 2)
()
>ht
{clone=1, zone=2}
>(get ht 'clone)
1
>
```

The procedure `(import)` creates generic functions. For example, `(get)` is a generic function with three methods:

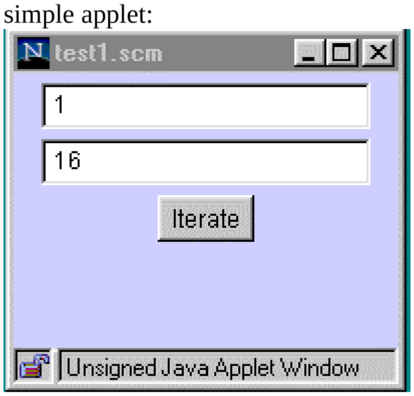
```
> get
{silk.Generic get[3]}
> (for-each print (methods get))
{InstanceMethod Object Map.get(Object)}
{InstanceMethod Object silk.GlobalEnv.get(Symbol)}
{InstanceMethod Object Field.get(Object)}
#t
>
```

Here's an example of using a static method:

```
> (import "java.lang.Float")
importing java.lang.Float in 81 ms.
#t
> (Float.parseFloat "17.42")
17.42
```

2.4 Here's an Example of Using (import)

To show what programming with `(import)` and generic functions is like, here's a simple applet:



To use it, type a number into the second textfield. The top textfield contains the current estimate of its square root. Each time the "iterate" button is pressed, an iteration of Newton's method is performed to better approximate the square root.

Here's the code. The EasyWin class is borrowed from JLIB.

```
(import "java.lang.Double")
```

```
(import "java.awt.Color")
(import "java.awt.Button")
(import "java.awt.TextField")
(import "jlib.EasyWin")

(define (test1)
  ;; Construct a test window.
  (let ((win (EasyWin "test1.scm" this-interpreter))
        (g (TextField "1" 20))
        (x (TextField "16" 20))
        (go (Button "Iterate")))
    (define (action e)           ; Define call back.
      (setText g
               (toString
                (f
                 (Double.valueOf (getText g))
                 (Double.valueOf (getText x))
                 ))))
    (define (f g x)              ; Newton's method.
      (/ (+ g (/ x g)) 2.0))
    (resize win 200 200)         ; Size the window.
    (add win g)                  ; Add the components.
    (add win x)
    (add win go)
    (addActionCallback win action)
    (setBackground win (Color 200 200 255))
    (show win)))

(test1)                          ; Try it out.
```

3 The Generic Function Protocol

Compared to Common Lisp, or even Tiny CLOS, the SILK generic function protocol is simple. Because the protocol is defined in terms of non generic Java methods, metacircularity is not an issue [3]. Here are the essential abstract classes and methods:

```
public abstract class Procedure extends SchemeUtils {
  // Apply the Procedure to a list of arguments.
  public abstract Object apply(Pair args, Engine eng);
}

public abstract class Generic extends Procedure {
  // Add a method to the generic.
  public abstract void addMethod(GenericMethod m);
}

public abstract class GenericMethod extends Procedure {
  // Two GenericMethod's match if they have equal
  // lists of parameterTypes.
```

```
  public abstract boolean match(GenericMethod x);

  // Is the method applicable to the list of arguments?
  public abstract boolean isApplicable(Pair args);

  // Returns the method that is more applicable.
  public abostract GenericMethod moreApplicableMethod
                                  (GenericMethod m2);
}
```

A `Procedure` is an object that can be applied to a list of arguments. It is the basis for function calling in SILK. Class `SchemeUtil` simply provides many convenient utilities (a common Java idiom). `Engine` is used to execute code fragments that allows tail call optimization and need not be considered further here.

A `Generic` is a `Procedure` defined in terms of a set of `GenericMethod's` The `addMethod()` method is used to add a `GenericMethod` to the set. No two `GenericMethod`'s in the set are allowed to `match()`.

There are currently four classes of `GenericMethod. ConstructorMethod, StaticMethod`, and `InstanceMethod` are wrapper classes for each of the corresponding Java metaobjects. A `SchemeMethod` is used to define methods whose behavior is written in Scheme.

Since `(import)` assigns constructors, static methods and instance methods to different generic functions, the methods in a generic function tend to be of only one type. However, a generic function can have any subclass of `GenericMethod.` This allows `Scheme methods` to be added to any `Generic`, for example.

3.1 Choosing the Applicable Method

To choose the applicable method, we follow Java semantics closely. This may seem surprising since Java chooses a method based on the dynamic type of its first argument, and the declared type of its other arguments. SILK simply makes this choice at runtime based on the types of arguments passed to the `Generic`.

For example, here are the methods for the `(list#)` Generic looks like after importing `java.io.file` and `javax.swing.JFrame`:

```
> list#
{silk.Generic list#[7]}
> (for-each print (methods list#))
{InstanceMethod String[] File.list(FilenameFilter)}
{InstanceMethod String[] File.list()}
{InstanceMethod void Component.list(PrintStream, int)}
{InstanceMethod void Component.list(PrintWriter, int)}
{InstanceMethod void Component.list()}
{InstanceMethod void Component.list(PrintWriter)}
{InstanceMethod void Component.list(PrintStream)}
```

To list the contents of a directory, the second method would be chosen:

```
> (define f (File "d:/java/jlib3/src/silk/"))
f
```

```
> (for-each* print (list# f))
Closure.java
Code.java
ConstructorMethod.java
...
>
```

The only complication with this approach is Scheme datatypes must be mapped to an appropriate Java type during method invocation. Currently, there are two issues.

1. SILK's numeric types include only `Integer` and `Double`. So, methods such as `java.util.Hashtable(int, float)` can't be invoked directly. One must first convert the required `float`, using `(Float 0.75)` for example. While SILK does not use `Float` or `Long` objects, once such objects are constructed it treats them as normal numbers.
2. Scheme symbols and strings (represented as Java `char[]`) are mapped to class `String` during method invocation. This can lead to an ambiguity, such as in the `append()` method of `java.lang.StringBuffer`:

```
>(import "java.lang.StringBuffer")
importing java.lang.StringBuffer in 70 ms.
#t
 >append#
{silk.Generic append#[10]}
 >(for-each print (methods append#))
{InstanceMethod StringBuffer.append(char[], int, int)}
{InstanceMethod StringBuffer.append(char[])}   ; ***
{InstanceMethod StringBuffer.append(boolean)}
{InstanceMethod StringBuffer.append(String)}   ; ***
{InstanceMethod StringBuffer.append(Object)}
{InstanceMethod StringBuffer.append(char)}
{InstanceMethod StringBuffer.append(long)}
{InstanceMethod StringBuffer.append(int)}
{InstanceMethod StringBuffer.append(float)}
{InstanceMethod StringBuffer.append(double)}
```

Since this `Generic` has methods on both `String` and `char[]`, SILK can't decide which to use. In such a case, the user must invoke a particular method using `(method)`.

3.2 Use Only The Most General Method

To minimize the method lookup overhead, for Java instance methods we let Java's single argument dispatch do most of the work. To do that, we only store the most general Java methods in a generic function. For example, for the generic function `(toString)` we only need the `Object.toString()` method:

```
>(methods toString)
({InstanceMethod String Object.toString()})
>
```

We call such a method, a "most general method".

The feasibility of this approach was studied using the 494 classes reachable from the class `javax.swing.Jframe`. Here are some statistics:

Count	**What**
494	classes
458	public classes
52	Exception classes
176	static most general methods
2759	instance most general methods.
2935	total most general methods.

There were 134 generic functions that contain only static methods. 93% of them have two or fewer methods:

Count	**# Methods, Cumulative % and examples**	
110	1	82.1%
15	2	93.3%
6	3	
1	4	createPackedRaster
1	5	getKeyStroke
1	9	valueOf

The 2,759 most general instance methods fall into 1525 generic functions. 91% of these have three or fewer methods:

Count	**# Methods, Cumulative % and examples**	
1058	1	69.4%
255	2	86.1%
75	3	91.0%
39	4	
24	5	
23	6	
17	7	
10	8	
6	9	
1	10	
5	11	
1	12	
1	13	
2	16	
1	17	get
1	18	contains
3	20	clone insert print
1	22	println
1	24	remove
1	36	add

So, most generic functions have one or two methods so our approach favors such situations.

Methods are only added to a generic function when a class is imported. So, the above statistics reflect what you get if you imported all 494 classes. The number of actual methods is likely to be substantially lower than this. For example, when using only javax.swing.JFrame, (add) only has six methods, not 36.

3.3 A Few Discriminator States Are Adequate

Since the number of methods per generic function tends to be small, we focus on optimizing the method lookup for such cases. We use a discriminator function with a small number of states. The state of the discriminator is recomputed whenever a method is added to the generic function. The states are chosen based on the static statistical analysis above. In contrast, Common Lisp chooses discriminator states dynamically so performance is adapted to each run of an application [4]. Here is a description of the states:

1. NOMETHODS - No methods, an error occurs if invoked. The discriminator is in this state when the generic function is first constructed, before any methods have been added to it.
2. ONEMETHOD - One method simply invoke the method.
3. TWOMETHODINSTANCEFIXED - Two instance methods with the same number of arguments. Check the first argument of the first method. If it is appliable, apply it, otherwise apply the second method. This works because the types of the first arguments are disjoint because of the most general method requirement.
4. TWOMETHODNOTFIXED - Two methods of any type with different numbers of arguments. Choose the method based on the number of arguments.
5. GENERAL - Most general lookup. Compute the most applicable method based on all of the arguments of all methods.

For the most likely case of a generic function only having one or two methods, discrimination is little more than a switch jump and a subclass test or two. For cases where an error would occur, we simply invoke the wrong method and let Java signal the error.

Most of the cost of invoking a generic function is in crossing the Scheme/Java frontier to actually apply the method. Currently this involves converting a list of arguments from the Scheme side to an array of arguments on the Java side. String and symbol arguments are also converted to appropriate Java types. The return value must also be converted. For example, a boolean result must be inverted to either `#t` or `#f`.

3.4 Scheme Methods

Besides using Java methods in generic functions, it is useful to define methods directly in Scheme, using the (define-method) macro:

```
(define-method name ((arg class) ...)
  (form) ..)
```

Where class names a Java class.

For example, here we define the generic function `(iterate collection action)` that maps the function action over the elements of collection:

```
(import "java.util.Iterator")
(import "java.util.Collection")
(import "java.util.Map")
(import "java.util.Vector")
(import "java.util.Hashtable")

(define-method iterate ((items java.util.Iterator)
                        action)
  (if (hasNext items)
      (begin (action (next items))
             (iterate items action))))

(define-method iterate ((items java.util.Collection)
                        action)
  (iterate (iterator items) action))

(define-method iterate ((items java.util.Map) action)
  (iterate (entrySet items) action))

(define-method iterate ((items silk.Pair) action)
  (action (car items))
  (let ((items (cdr items)))
    (if (pair? items) (iterate items action))))

(define-method iterate ((items java.lang.Object[])
                action)
  (let loop ((i 0)
         (L (vector-length items)))
    (if (< i L) (begin (action (vector-ref items i))
               (loop (+ i 1) L)))))
```

The `collection` argument can be an array, a Scheme list (of type `silk.Pair`), or any of the Java 1.2 collection types. This type of integration is not easy in Java because new instance methods cannot be added to existing classes. Here's an example use:

```
>(define h (Hashtable 50))
h
>(put h 'fred 3)
()
>(put h 'mary 4)
()
>(iterate h print)
fred=3
mary=4
()
>
```

Scheme methods are treated like Java static methods. Currently, there is no provision for `(call-next-method)`. One must invoke such a method directly using `(method)`.

Java classes can be defined directly in Scheme using a `(define-class)` macro, using the compiling technique described below. Such classes only have constructor and field accessor methods. Scheme methods can be added to the class using `(define-method)`.

4 Creating New Code From Old

It should be clear that SILK fulfills one of Scheme's important roles as an embedded scripting and prototyping language [5]. Perhaps a greater strength is that Scheme can be used as a compile time scripting language to generate new code, perhaps in another language, such as C or Java. Two examples of this are described in references [6] and [7] where Scheme is used to set up a complex numerical problem such as a complex computer visualization. Partial evaluation is then used to generate an efficient algorithm to compute the solution of the problem, in C.

Java development environments, such as Sun's Java Bean box, compile small glue classes automatically. This generated code usually follows a standard template and requires some introspection of an existing class. We can do the same thing in SILK. Essentially, Scheme becomes a macro language for Java.

For example, the normal Java runtime environment does not provide a way to trace individual methods. Here we sketch how to add such tracing capability. The basic idea is to generate a subclass of an existing class which allows its methods to be traced. If SILK has enough access to the Java application, an instance of this traceable class can be substituted into the application without changing any Java code. As part of this process, we would have a method, m2, say the method `Hashtable.put()`, and generate a traced method from it using `(gen-trace-method m2)`:

```
>m2
public synchronized Object Hashtable.put(Object,
                                          Object)
>(emit (gen-trace-method m2))
public synchronized java.lang.Object
    put(java.lang.Object a1, java.lang.Object a0) {
  if(trace) Trace.enter(this + ".put(" + a1 + ", " +
                        a0 + "}");
  java.lang.Object result = super.put(a1, a0);
  if(trace) Trace.exit(result);
  return result;
  }
#t
```

Here's some Scheme code that does this:

```
(define-method method-return-type
  (m java.lang.reflect.Method)
  (getName (getReturnType m)))

(define-method gen-trace-method
  (m java.lang.reflect.Method)
```

```
  '(,(gen-method-signature m)
    { ,(gen-trace-method-body m) }))

(define-method gen-method-signature
  (m java.lang.reflect.Method)
  '(,(Modifier.toString (getModifiers m))
    ,(method-return-type m)
    ,(getName m) "("
    ,(map-args
      (lambda (arg) '(,(arg-type arg) ,(arg-name arg)))
      ","
      (arg-types m))
    ")"))

(define-method gen-trace-method-body
  (m java.lang.reflect.Method)
  '(if "(" trace ")"
       Trace.enter "(" this +
       ,(quotify "." (getName m) "(")
       ,(map-args (lambda (arg) '(+ ,(arg-name arg)))
                  "+ \", \""
                  (arg-types m))
       + ,(quotify "}") ")" ";"
       ,(method-return-type m) result = super.
       ,(getName m) "("
       ,(map-args arg-name "," (arg-types m))
       ")" ";"
       if "(" trace ")" "Trace.exit(result)" ";"
       return result ";"))
```

This code generation system is extremely basic. `(emit)` takes a list structure, produced by `(gen-trace-method)`here, and formats it into readable Java code. Scheme's backquote macro characters, `'(, ,@)` are used to construct the necessary list structures. `(map-args)` is like `(map)` but adds a separator between each argument, to generate a comma separated argument list, for example.

SILK can automatically compile Java files using:

```
(import "java.lang.String")
(import "java.lang.reflect.Array")
(import "sun.tools.javac.Main")
(import "java.lang.System")
(define (compile-file file)
  ;; Compile the *.java file, file,
  ;; using the current CLASSPATH.
  (let ((as (Array.newInstance String.class 3))
        (main (Main (get-field System.class 'out)
                    'silkc)))
    (vector-set! as 0 "-classpath")
    (vector-set! as 1 (System.getProperty
                       "java.class.path"))
    (vector-set! as 2 file)
    (compile main as)))
```

In this simple example, Java's Method metaclass was used directly to generate the traced code. A more realistic example would provide a compile time metaobject protocol that would be used to do code generation more formally. While this example is simple, it should be clear that SILK can be used as a useful compile time software development environment without a substantial amount of work.

5 Related Work

5.1 Other Scheme Implementations

There are three other Scheme implementations in Java we are aware of, which we briefly describe: The following exhibit shows statistics from these implementations.

Scheme implementation statistics

Implementation	java files	lines	Scheme files	lines	Generics
Silk 1.0	12	1905	0	0	No
Silk 2.0	20	2778	0	0	No
Generic Silk 2.0	28	3508	5	510	Yes
Skij [8]	27	2523	44	2844	Yes
Jaja [9]	66	5760	?	?	No
Kawa [10]	273	16629	14	708	No

Skij: Skij is a Scheme advertised as a scripting extension for Java. 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 is a Scheme based on the Christian Queinnec's wonderful book "Lisp in Small Pieces"[11]. It includes a Scheme to Java compiler 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 super class `(Jaja in Jaja, and SchemeUtils in Silk)` to provide globals and utilitiy functions. Unlike Silk, in Jaja, each Scheme type has one or more Java classes defined for it. Also, in Jaja, the empty list '`()` is represented as an instance of the class `EmptyList`, while in Silk it is represented by null. All Jaja objects are serializable.

Kawa: Kawa 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.

5.2 Generic Function Dispatch

In any language that uses generic functions, efficient method lookup is extremely important. Method lookup is basically a two-dimensional table lookup in a very sparse table. There has been extensive research recently on table compression methods [12] [13] [14][15].

While these techniques are quite interesting, they seemed too complex to implement for our tiny Scheme environment. Instead, we follow the approach taken in Common Lisp, and associate each generic function with a list of its methods. The trick then becomes, given a generic function and a set of arguments, choose the appropriate method.

In Common Lisp, [4] a generic function does method lookup using one of several strategies. The strategy used is based on the number of different methods the generic function has actually invoked. Backing up these strategies is a per generic function caching scheme. The advantage of such a dynamic approach is that it tailors itself to the running application.

In SILK, a generic function uses a static strategy based on the number and types of methods that have been added to the generic function, as described above. This approach seems to work well, but we have not had a chance to analyze this in any detail. However, other evidence suggests that a static approach such as ours is not unreasonable.

For example, STK[14] is a Scheme dialect that uses an object system based on Tiny CLOS [3]. Method lookup is done by searching for the appropriate method each time, without any caching. Our belief is that this approach works reasonably because most methods only have a few methods.

We have verified this with one typical Common Lisp application (Allegro CL 5.0 + CLIM). There were 1,568 generic functions and 952 classes. The following is a cumulative histogram of the number of methods per generic function.

#methods	Cumulative %
1	59.3
2	79.6
3	89.1
4	91.7
5	94.4
10	97.6
20	99.1
100	100.0

From this we see that 59% of the generic functions have one method. These are likely to be assessor methods. 89% of the methods have three or fewer methods. However, one generic function, `(print-object)` has 100 methods.

Thus we expect our static approach to be reasonable. However, adding a Scheme method to a generic function can often force a full lookup to be done. Thus as more Scheme methods are used over Java methods, exploring other approaches will become important. Queinnec [9] describes a method lookup approach that uses discrimination net. This approach looks promising for SILK.

6 Conclusion

There are many Scheme implementations, This is partly because it relatively easy to implement a reasonably efficient implementation, and techniques to build a high performance implementation are well understood. It is also partly because, while Scheme is a useful language in its own right, it has found and important role as a scripting language embedded in an application. Guile and STK are two examples of embeddable Schemes implemented in C [16].

Beckman argues that scripting languages are inevitable [5]. In the 80's Jon Bently popularized the idea of Little Languages [17]. Such a language can be used to describe in a compact way, one aspect of your project, graphical 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.

We have tried to show that SILK makes effective use of Java's reflective capabilities:

1. The implementation of its data types were carefully chosen to match those of Java closely. This minimizes the impedance mismatch between Scheme and Java. Currently the main remaining mismatch is that strings in Scheme are represented as `char[]` because strings are mutable in Scheme. Making Scheme strings immutable, as they are in Java would allow us to use the String class. This change would only require dropping the procedure `(string-set!)` from SILK.

2. Using `(import)` SILK has almost transparent access to Java. A Scheme implemented in another languages requires substantial foreign function interface to be written. While it is possible to automatically generate this interface from C headers files, for example, it is a fair amount of work.
3. Once SILK is available, it can be used for scripting in the base language, providing the benefits that Beckman suggests. For example, a SILK extension is used to layout Java Swing components. A graphical designer lays out the components and a Java programmer wires in the underlying behavior.
4. Because SILK has direct access to the reflection of any Java class, SILK can be used as a metalevel scripting language to generate new Java classes at compile or runtime. While other Java systems have similar capabilities, we've tried to show that the amount of work required in SILK is small.
5. Reflection could be used more aggressively in the implementation of the underlying Scheme interpreter itself. For example, the Scheme primitives could be implemented as Java classes and imported into a kernel language. The Scheme interpreter and Scheme-to-Java compiler, which are written in Java, could also be accessed via reflection, and even modified at runtime using a custom classloader.

Acknowledgments

The authors wish to thank Peter Norvig for developing the first version of SILK and for keeping it simple enough that we could easily use and extend it. We thank Geoffrey S. Knauth, Richard Shapiro, Tom Mitchell, Bruce Roberts, and Jeff Tustin for reviewing drafts of this paper. We thank Rusty Bobrow who thought SILK was simple enough that he suggested it to his daughter for programming a High School science project. And we'd like to thank Rusty's brother, Danny Bobrow for long ago teaching us that "Meta is Beta!" That two generations of one family use reflection, reflects well on reflection.

References

1. http://www-swiss.ai.mit.edu/~jaffer/SLIB.html
2. Aubrey Jaffer, r4rstest.scm, ftp://ftp-swiss.ai.mit.edu/pub/scm/r4rstest.scm.
3. Gregor Kiczales, Tiny CLOS, file://parcftp.xerox.com/pub/mops/tiny/
4. Gregor Kiczales and Luis Rodriguez, Efficient method dispatch in PCL, proceedings 1990 ACM Conference on LISP and Functional Programming, Nice, France, June 1990, 99-105.
5. Brian Beckman, A scheme for little languages in interactive graphics, Software Practice and Experience, 21, 2, p. 187-208, Feb, 1991.
6. A. Berlin and D. Weise, Compiling scientific code using partial evaluation. Technical Report CSL-TR 90-422, Artificial Intelligence Laboratory, Massachusetts Institute of Technology, 1990.
7. Clifford Beshers Steven Feiner, Generating efficient virtual worlds for visualization using partial evaluation and dynamic compilation, ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'97), p. 107-115.

8. Mike Travers, Skij, IBM alphaWorks archive, http://www.alphaworks.ibm.com/formula/Skij
9. Christian Queinnec, JaJa: Scheme in Java, http://www-spi.lip6.fr/~queinnec/WWW/Jaja.html
10. Per Bothner, Kawa the Java-based Scheme System, http://www.cygnus.com/~bothner/kawa.html
11. http://www-spi.lip6.fr/~queinnec/Papers/dispatch.ps.gz
12. Shih-Kun Huang and Deng-Jyi Chen, Efficient algorithms for method dispatch in object-oriented programming systems, Journal of Object-Oriented Programming, 5(5):43-54, September 1992.
13. E.Amiel, O. Gruber, and E. Simon, Optimmizing multi-method dispatch using compressed dispatch tables. In OOPSLA '94, October 1994.
14. Jan Vitek and R. Nigel Horspool, Taming message passing: Efficien method lookup for dynamically typed languages. ECOOP '94 - 8th European Conference on Object-Oriented Programming, Bologna (Italy), 1994.
15. Weimin Chen and Volker Turau, Multiple-dispatching base on automata, Theory and Practice of Object Systems, 1(1):41-60, 1995.
16. http://kaolin.unice.fr/STk/
17. J.L. Bentley, More Programming Pearls, Addison-Wesley, Reading, MA, 1988.