Compiling Java to PLT Scheme

Kathryn E. Gray Matthew Flatt
Univeristy of Utah

Abstract

Our experimental compiler translates Java to PLT Scheme; it enables the use of Java libraries within Scheme programs, and it makes our Scheme programming tools available when programming with Java. With our system, a programmer can extend and use classes from either language, and Java programmers can employ other Scheme data by placing it in a class using the Java native interface.

PLT Scheme's class-system, implemented with macros, provides a natural target for Java classes, which facilitates interoperability between the two languages, and PLT Scheme's `module` maintains Java security restrictions in Scheme programs. Additionally, `module`'s restrictions provide a deeper understanding of a Java compilation unit and make Java's implicit compilation units explicit.

1 Why Compile Java to Scheme?

Scheme implementations that compile to Java (or JVM bytecode) benefit from the extensive infrastructure available for Java programs, including optimizing just-in-time compilers, JVM debugging tools, and an impressive roster of Java-based libraries. For PLT Scheme, we have inverted the equation, compiling Java to Scheme. We thus obtain a Java implementation with access to PLT Scheme's libraries and facilities—especially the DrScheme environment and its teaching modes [5], which is the primary motivation for our effort [9].

By compiling Java to Scheme, we also gain access to the many libraries implemented in Java, as long as we can bridge the gap between Java and Scheme. In many ways, the translation is the same for Java-to-Scheme compilation as it is for Scheme-to-Java, but the trade-offs are somewhat different. In particular, libraries that contain native calls are no problem for Scheme-to-Java compilation, but Java-to-Scheme must provide special support for native methods. In contrast, a Scheme compilation model with expressive macros accommodates Java code more easily than Java's model of compilation accommodates Scheme.

Our Java-to-Scheme compiler remains a work in progress. Even so, we have gained experience that may be useful to future implementors of Java-to-Scheme compilers.

In the long run, we expect that many useful Java libraries will fit into our implementation, as PLT Scheme provides a significant set of libraries of its own. For example, we expect that the subset of AWT used by Swing can be mapped onto PLT Scheme's GUI primitives, thus enabling Swing-based applications to run in PLT Scheme. In other words, we believe that many Java libraries can be made to run by re-implementing certain "native" Java methods in PLT's native language, Scheme.

In this report, we describe

- a strategy for compiling Java classes to PLT Scheme, which exploits a macro-implemented extension of Scheme for object-oriented programming;
- the interaction of the strategy with PLT Scheme's module system;
- how we define the translation of run-time values between Java and Scheme; and
- open problems that we have not yet addressed.

Before introducing our compilation strategy, we begin with a description of two libraries that we would like to embed in Scheme, and the strategy of doing so. One is relatively easy to support, and the other is more difficult.

2 Java Libraries

Kyle Siegrist's probability and statistics library (PSOL) [16] provides various mathematical procedures. The library also deploys a graphical interface to experiment with the procedures, but the interface is not necessary to use the library's primary functionality.

Terence Parr's ANTLR [13] provides parsing technology that permits grammar inheritance.

These two libraries are representative of the kinds of code and dependencies found in non-graphical Java libraries. We estimate that roughly 10 to 15% of Java libraries have requirements similar to PSOL, while 30 to 40% are similar to ANTLR. The remaining libraries tend to depend on graphic capabilities.

Naturally, both packages rely on the basic components of Java's class system, including classes and interfaces, overloading, and static (as well as instance) members, as well as loops and arrays. Both libraries also depend on some of Java's core libraries, including `Object`, `String`, and `Throwable`.

PSOL depends on Java's `Math` library and numeric wrapper classes. The latter provides the ability to use primitive values (such as `1.2`) as objects, as well as functionality over numbers. These libraries rely on native methods with existing counterparts in PLT Scheme.

ANTLR requires several language features that we do not yet support: nested classes (commonly known as inner classes), the `switch` statement, and reflection. The first two, nested classes and `switch`, simply have not been implemented yet in our system, but there are no technical challenges. Reflection is more difficult than the other two, though, and we discuss this problem in Section 5.5.2. ANTLR further relies on utility libraries and `IO`, which in turn rely on nested classes and reflection. Several `IO` classes rely on native methods that have counterparts in PLT Scheme.

Neither PSOL nor ANTLR immediately ran in our current implementation. In the case of PSOL, we easily implemented the relevant `Math` methods and wrapper classes, so that PSOL now runs in non-graphical mode. We expect that implementing the `IO` methods needed for ANTLR will be similarly easy, but reflection support is a much larger obstacle. Support for graphical PSOL is completely out of reach in the short term.

3 Classes and Objects in PLT Scheme

PLT Scheme was designed from the start to support GUI programming with class-based, object-oriented constructs. Originally, the class implementation was built into the interpreter's core, and each object was implemented as a record of closures. Our current system is more Java-like, in that an object is a record of fields plus a per-class method table, and it is implemented outside the core by a collection of (an extended) `syntax-case` macros.

3.1 Class Constructs

A class is created with the keyword `class`, and the resulting form is much like Java's. It creates a new class given a superclass and a collection of method overrides, new methods, and new fields. Syntactically, `class` consists of an expression for the superclass followed by a sequence of declarations.

The following is a partial grammar for `class` clauses:

```
expr       =  ...
           |  (class super-expr clause ...)
clause     =  (field init-decl ...)
           |  expr
           |  (init init-decl ...)
           |  (public name ...)
           |  (override name ...)
           |  (private name ...)
           |  (define name method)
           |  (inherit name ...)
init-decl  =  name
           |  (name init-expr)
method     =  (lambda formals expr0 expr ...)
```

Instead of a constructor, a class contains `field` declarations and other *expr*s to evaluate each time the class is instantiated. An `init` introduces a keyword-based initialization argument (possibly with a default value) to be supplied when the class is instantiated. The `public` form names the new public methods defined in the class, `override` names the overriding methods defined in the class, `private` names private methods. Each method definition looks like a function definition using `define` and a name declared as `public`, `override`, or `private`. Macros such as `define/public` (not shown in the grammar) combine the declaration and definition into a single form.

The `inherit` form names methods to be inherited from the superclass. Inherited methods must be declared explicitly because classes are first-class values in PLT Scheme, and a class's superclass is designated by an arbitrary expression. The advantage of first-class classes is that a mixin can be implemented by placing a `class` form inside a `lambda` [6]. Obviously, Java code contains no mixin declarations, but as we note later in Section 7.2, mixins provide a convenient solution to certain interoperability problems.

The built-in class `object%` serves as the root of the class hierarchy, and by convention, `%` is used at the end of an identifier that is bound to a class. As in Java, the `class` system supports only single-inheritance. A class explicitly invokes the body expressions of its superclass using `super-new`.

Within a `class` body, fields and methods of the class can be accessed directly, and fields can be modified using `set!`. Initialization arguments can be accessed only from field initializers and other expressions, and not in method bodies. For example, a stack class can be implemented as follows:

```
(define stack%
  (class object%
    (public push pop)
    (init starting-item)
    (field (content (list starting-item)))
    (define push
      (lambda (v)
        (set! content (cons v content))))
    (define pop
      (lambda ()
        (let ((v (car content)))
          (set! content (cdr content))
          v)))
    (super-new)))
```

The `new` form instantiates a class. Its syntax is

```
(new class-expr (init-name expr) ...)
     ;  ⇒ an object
```

where each *`init-name`* corresponds to an `init` declaration in the class produced by *`class-expr`*. For example, a stack instance (that initially contains a `5`) is created as follows:

```
(define a-stack (new stack% (starting-item 5)))
```

The `send` form invokes a method of an instance,

```
(send obj-expr method-name arg-expr ...)
      ; ⇒ method result
```

where *method-name* corresponds to a `public` method declaration in *obj-expr*'s class (or one of its super classes). For example, `send` is used to push and pop items of *a-stack*:

```
(send a-stack push 17)
(send a-stack pop) ; ⇒ 5
```

Each execution of `send` involves a hash-table lookup for *method-name*; to avoid this overhead for a specific class, a programmer can obtain a generic function using `generic` and apply it with `send-generic`:

```
(generic class-expr method-name); ⇒ a generic
(send-generic obj-expr generic-expr
       arg-expr ...) ;  ⇒ method result
```

In accessing fields, the forms `class-field-accessor` and `class-field-mutator` produce procedures that take an instance of the given class and get or set its field.

```
(class-field-accessor class-expr field-name)
(class-field-mutator class-expr field-name)
```

where *field-name* corresponds to a `field` declaration in the class.

By default, a `field`, `init`, or method name has global scope, in the same sense as a symbol. By using global scope for member names, classes can be easily passed among modules and procedures (for mixins or other purposes).

A name can be declared to have lexical scope using `define-local-member-name`:

```
(define-local-member-name name ...)
```

When a *name* is so declared and used with `init`, `field`, or `public`, then it is only accessible through `override`, `inherit`, `new`, `send`, `generic`, `class-field-accessor`, and `class-field-mutator` in the scope of the declaration. At PLT Scheme's module level, local member names can be imported and exported, just like macros. At run-time, the values produced by `generic`, `class-field-accessor`, and `class-field-mutator` can be used to communicate a method or field to arbitrary code.

For example, we can use `define-local-member-name` to make the *content* field private[1] to the scope of the *stack%* declaration:

```
(define-local-member-name content)
(define stack% (class ...))
(define stack-content
  (class-field-accessor stack% content))
(define (empty-stack? s)
  (null? (stack-content s)))
```

To support interfaces, PLT Scheme offers an `interface` form, plus a `class*` variant of `class` that includes a sequence of expressions for interfaces. An interface consists of a collection of method names to be implemented by a class, and like a class, it is a first-class object. As in Java, an interface can extend multiple interfaces.

[1]The class form also supports private field declarations, but we omit them for brevity.

```
expr  =  ···
      |  (class* super-expr
           (interface-expr ...)
           clause ...)
      |  (interface (super-expr ...) name ...)
```

The `generic` form accepts an interface in place of a class, but the current implementation of generics offers no performance advantage for interfaces.

3.2 PLT Scheme vs. Java

If PLT Scheme's class system were not already Java-like, we would have implemented new forms via macros to support Java-to-Scheme compilation. This layering allows us to develop and test the core class system using our existing infrastructure for Scheme, including debugging and test support.

PLT Scheme's class system does not include inner classes or static methods, so the Java-to-Scheme step transforms those Java constructs specially. Static methods are easily converted to procedures, and inner classes have strange scoping rules that seem better handled before shifting to macro-based expansion. Similarly, Java's many namespaces are transferred into Scheme's single namespace by the compiler, rather than by macros. In other words, we use macros to implement the parts of the compiler that fit naturally with lexical scope and local expansion, and we perform other tasks in the compiler.

4 Compilation Model

A single Java source file typically contains one `public` class (or interface). Often, the file itself corresponds to a compilation unit, so that one `.java` file can be compiled to one `.class` (or, in our case, to one `.scm` file).

In general however, reference cycles can occur among `.java` files, as long as they do not lead to inheritance cycles. Thus, the compilation unit corresponds to several mutually dependent `.java` files. For example, one class may refer to a field of another class, and compiling this reference requires information about the structure of the referenced class. In contrast, merely using a class as an identifier's type does not necessarily require information about the class, especially if the identifier is not used.

More concretely, the code in Figure 1 corresponds to three source files, one for each class. Compiling `Empty` requires knowledge of the superclass `List`, while compiling `List` requires knowledge of `Empty` for the constructor call. Similarly, `List` refers to `Cons` and `Cons` refers to `List`. Thus the three classes must all be compiled at the same time. This kind of cyclic reference appears frequently in real Java code.

Java's packages are orthogonal to compilation units because a group of mutually dependent `.java` files might span several Java `packages`. Furthermore, a mutually dependent group of files rarely includes all files for a package, so forcing a compilation unit to be larger than a package would lead to needlessly large compilation units. Finally, in most settings, a Java package can be extended by arbitrary files that simply declare membership in the package, which would cause an entire package to recompile unnecessarily.

To a first approximation, our Java-to-Scheme compiler produces a single Scheme `module` for each collection of mutually dependent Java sources, where `module` is the unit of compilation for PLT Scheme code [7]. Each class used by, but not a member of, the dependent group is `require`-ed into the `module`. The Java specification [8] requires that each class be initialized and available prior to its first use, which the `require` statement ensures.

The `module` is also a unit of organization at the Scheme level, and for interoperability, we would like to maintain the organization of the Java library in the Scheme program. Thus, our Java-to-Scheme compiler actually produces $N+1$ modules for N mutually dependent Java sources: one that combines the Java code into a compilation unit, and then one for each source file to re-export the parts of the compilation unit that are specific to the source.[2] Thus Scheme and Java programmer alike import each class individually. For example, compiling Figure 1 results in four modules: A composite module that contains the code of all three classes and exports all definitions, a `List` module that re-exports `List` and `main`, an `Empty` module that re-exports `Empty`, and a `Cons` module that re-exports `Cons` and field-relevant information.

In practice, we find that groups of mutually dependent files are small, so that the resulting compilation units are manageable. This is no coincidence, since any Java compiler would have to deal with the group as a whole. In other words, this notion of compilation unit is not really specific to our compiler. Rather, having an explicit notion of a compilation unit in our target language has forced us to understand precisely what compilation units are in Java, and to reflect those units in our compiler's result.

Currently, our compiler produces an additional file when generating Scheme from Java code. The extra file contains Java signature information, such as the types and names of fields and methods in a class, which the compiler needs to process additional compilation units. Other Java compilers typically store and access this information in a `.class` directly, and in a future version of our compiler, we intend to explore storing this compile-time information in a `module` in much the same way that compile-time macros are stored in `module`s.

5 Compilation Details

Our compiler begins by parsing Java code using a LEX-/YACC-style parser generator. Source tokens are quickly converted into location-preserving syntax identifiers, as used in macros. Thus, as the generated Scheme code is processed by the Scheme compiler, source information from the original Java program can be preserved during Scheme compilation. This source-location information is used mainly by DrScheme tools or for reporting run-time errors.

As our primary motivation for this work (pedagogic Java subsets) requires control over all error messages reported from the compiler, we chose to compile Java source instead of Java bytecode. While this limits the libraries available to our system, in the future we can use existing bytecode interpreting libraries to alleviate this limitation.

Java and PLT Scheme both strictly enforce an evaluation order on their programs. Coincidentally, both enforce the same ordering on function arguments and nested expressions. Therefore, those Java constructs which differ from Scheme only in syntax have a straightforward translation. For example,

```
abstract class List {
  abstract int length();

  static void main() {
    Test.test(new Empty().length(), 0);
    Test.test(new Cons(1,
                       new Empty()).length(),
              1);
  }
}

class Empty extends List {
  int length() { return 0; }
}

class Cons extends List {
  int car;
  List cdr;
  Cons( int c, List cdr ) {
    this.car = c;
    this.cdr = cdr;
  }
  int length() { return 1 + cdr.length(); }
}
```

Figure 1. A Cyclic Java program

```
int a = varA + varB, b = varA - varB;
if (a+b <= 2)
 res = a;
else
 res = b;
```

translates into

```
(let ((a (+ varA varB))
      (b (- varA varB)))
  (if (<= (+ a b) 2)
      (set! res a)
      (set! res b)))
```

wrapped with the appropriate source location and other information. Indeed, the majority of Java's statements and expressions translate as expected.

Currently, mathematical operations directly use standard Scheme operations where possible. Thus, unlike the Java specification, numbers do not have a limited range and will automatically become bignums. In the future, our compiler will use mathematical operations that overflow as in the Java specification.

5.1 Classes

A Java class can contain fields, methods, nested classes (and interfaces), and additional code segments, each of which can be static. Our Scheme class is similar, except that it does not support static members. Nevertheless, a static member closely corresponds to a Scheme function, value, or expression within a restricted namespace, i.e., a `module`, so static Java members are compiled to these scheme forms.

[2]If a class is not a member of any dependency cycle, then the compiler produces only one module.

An instance of a class is created with the `new` form described in Section 3.1. As noted in that section, PLT Scheme's `new` triggers the evaluation of the expressions in the top level of the class body. These expressions serve the same purpose as a single Java constructor. However, a Java class can contain multiple constructors, preventing a direct translation from a Java constructor to a sequence of top-level expressions. Instead, we translate Java constructors as normal methods in the Scheme class, and we translate a Java `new` expression into a Scheme `new` followed by a call to a constructor method. This behavior adheres to the guidelines for class instantiation provided by Java's specification [8].

5.2 Fields & Methods

Non-`static` Java fields translate into Scheme `field` declarations. A `static` Java field, meanwhile, translates into a Scheme top-level definition. Thus, the fields

```
static int avgLength;
int car;
```

within the class `Cons` become, roughly

```
(define avgLength 0)
```

and

```
(define Cons
  (class ···
    (field (car 0)) ···))
```

However, the above translation does not connect the variable `avgLength` to the containing class `Cons`. If multiple classes within a compilation unit contain a static field `avgLength`, the definitions would conflict. For non-static fields, Scheme classes do not allow subclasses to shadow field names again potentially allowing conflicts. Additionally, to avoid conflicts between Java's distinct namespaces for fields, methods, and classes, we append a *`~f`* to the name. Therefore, we combine *`avgLength`* with the class name and *`~f`*, forming the result as *`Cons-avgLength~f`*, and `car` becomes *`Cons-car~f`*. Note that Scheme programmers using this name effectively indicate the field's class.

Compilation generates a mutator function for both of these fields, plus an accessor function for the instance (non-`static`) field. Since the `module` form prohibits mutating an imported identifier, the mutator function *`Cons-avgLength-set!`* provides the only means of modifying the static field's value. If the static field is `final`, this mutator is not exported. Also, instance field mutators are not generated when they are `final`. Thus, even without compile-time checking, Scheme programmers cannot violate Java's `final` semantics.

Similarly, instance methods translate into Scheme methods and static methods into function definitions with the class name appended, but the name must be further mangled to support overloading. For example, the class `List` in Figure 2 contains two methods named `max`, one with zero arguments, the other expecting one integer. The method `max(int)` translates into *`max-int`*, and `max` translates into *`max`*. This mangling is consistent with the Java bytecode language, where a method name is a composite of the name and the types of the arguments. Also, since "`-`" may not appear in a Java name, our convention cannot introduce a collision with any other methods in the source.[3]

[3] We do not add a *`-m`* to method names, because *`~f`* distinguishes fields from methods, and method and class names must be distinguished already at the Java source level.

```
abstract class List {
 abstract int max();
 abstract int max(int min);
}
```

Figure 2. Overloaded methods

As mentioned in Section 5.1, constructors are compiled as methods, which we identify with special names. The constructor for `Cons` in Figure 1 translates into *`Cons-int-List-constructor`*. The *`-constructor`* suffix is not technically necessary to avoid conflicts, but it clarifies that the method corresponds to a constructor.

A `private` Java member *does not* translate to a `private` Scheme member, because `static` Java members are not part of the Scheme class, but Java allows them to access all of the class's members. We protect `private` members from outside access by making the member name local to a module with `define-local-member-name`; the Java-to-Scheme compiler ensures that all accesses within a compilation unit are legal. Our compiler does not currently preserve protection for `protected` and package members.

5.3 Statements

Most Java statements (and expressions) translate directly into Scheme. The primary exceptions are `return`, `break`, `continue`, and `switch`, which implement statement jumps. For all except `switch`,[4] we implement these jumps with `let/cc`:[5]

```
(define-syntax let/cc
  (syntax-rules ()
    ((let/cc k expr ...)
     (call-with-current-continuation
       (lambda (k) expr ...)))))
```

A `return` translates into an invocation of a continuation that was captured at the beginning of the method. For example, the method `length` from `Empty` in Figure 1 becomes

```
(define/public length
  (lambda ()
    (let/cc return-k
      (return-k 0))))
```

The statements `break` and `continue` terminate and restart a `for`, `while`, or `do` loop, respectively. To implement these, we capture suitable continuations outside and inside the loop, such that

```
while(true) {
  if (x == 0)
    break;
  else if (x == 5)
    continue;
  x++;
}
```

becomes

[4] We have not implemented `switch`.

[5] We actually use `let/ec`, which captures an escape-only continuation.

```
(let/cc break-k
  (let loop ()
   (let/cc continue-k
     (when #t
           (if (= x 0)
               (break-k)
               (if (= x 5)
                   (continue-k)
                   (set! x (+ x 1))))
           (loop)))))
```

As it happens, `let/cc` is expensive in PLT Scheme. We plan to apply a source-to-source optimizer to our Java-to-Scheme compiler's output to eliminate these `let/cc` patterns, putting each statement in a separate `letrec`-bound function and chaining them. Although we could avoid `let/cc` in the output of our Java-to-Scheme compiler, it is easier to translate most Java statements directly to Scheme, and then work with Scheme code to optimize.

5.4 Native Methods

Most Java implementations use C to provide native support. Our system, naturally, uses Scheme as the native language. When our compiler encounters a class using `native` methods, such as

```
class Time {
 static native long getSeconds(long since);
 native long getLifetime();
}
```

the resulting `module` for `Time requires` a Scheme module *`Time-native-methods`* which must provide a function for each native method. The name of the native method must be the Scheme version of the name, with *`-native`* appended at the end. Thus a native function for `getSeconds` should be named *`Time-getSeconds-long-native`* and `getLifetime` should be *`getLifetime-native`*.

Within the compiled code, a stub method is generated for each native method in the class, which calls the Scheme native function. When `getSeconds` is called, its argument is passed to *`Time-getSeconds-long-native`* by the stub, along with the class value, relevant accessors and mutators, and generics for private methods. An instance method, such as `getLifetime`, additionally receives `this` as its first argument.

5.5 Constructs in Development

We have not completed support of `switch`, labeled statements, nested classes, and reflection. The first two are straightforward, and we discuss our design of the other two further in this section. Our partial implementation of nested classes suggests that this design is close to final.

5.5.1 Nested Classes

In Java, a nested class may either be `static` or an instance class, also known as an inner class. An inner class can appear within statement blocks or after `new` (i.e. an anonymous inner class).

Static nested classes are equivalent to top-level classes that have the same scope as their containing class, with the restriction that they may not contain inner classes. These can be accessed without directly accessing the containing class. When compiled to Java bytecodes, nested classes are lifted out and result in separate `.class` files. We equivalently lift a nested class, and provide a separate module for external access. We treat a nested class and its container as members of a cycle, placing both in the same module.

Inner classes are also compiled to separate classes. Unlike static nested classes, they may not be accessed except through an instance of their containing class. A separate module is therefore not provided, and construction may only occur through a method within the containing class.

The name of a nested class is the concatenation of the containing class's name with the class's own name. Class `B` in

```
class A {
  class B {
  }
}
```

is accessed as `A.B`. For anonymous inner classes, we intend to follow the bytecode strategy: the class will be given a name at compile-time, the containing class name appended with a call to `gensym`, and then lifted as other nested classes.

5.5.2 Reflection

Java supports multiple forms of reflection: examining and interacting with classes and objects specified at runtime; dynamically extending classes; and modifying the means of class loading and compilation. The first one can be supported either with macros or generating methods during compilation to provide the data. We do not yet know how the second will be supported, or what support for the third would mean within our system.

The first form of reflection allows users to create new class instances with strings, inspect and modify fields, call methods, and inspect what fields and methods are available. The last of these is easily supported by generating the information during compilation and storing it in an appropriate method. The other functionality can be supported through Scheme functions.

6 Run-Time Support

Java provides two kinds of built-in data: primitive values, such as numbers and characters, and instances of predefined classes. The former translate directly into Scheme, and most of the latter (in `java.lang`) can be implemented in Java. For the remainder of the built-in classes, we define classes directly in Scheme.

6.1 Strings

Although the `String` class can be implemented in Java using an array of `char`s, we implement `String` in Scheme. This implementation allows a Scheme string to hold the characters of a Java string, thus facilitating interoperability. From the Scheme perspective, a Java `String` provides a *`get-mzscheme-string`* method to return an immutable Scheme string.

6.2 Arrays

A Java array cannot be a Scheme vector, because a Java array can be cast to and from `Object` and because assignments to the array indices must be checked (to ensure that only objects of a suitable type are placed into the array). For example, an array created to contain

List objects might be cast to Object[]. Assignments into the array must be checked to ensure that only List, Cons, and Empty objects appear in the array.

To allow casts and implement Java's restrictions, a Java array is an instance of a class that descends from Object. The class is entirely written in Scheme, and array content is implemented through a private vector. Access and mutation to the vector are handled by methods that perform the necessary checks.

6.3 Exceptions

PLT Scheme's exception system behaves much like Java's. A value can be raised as an exception using *raise*, which is like Java's throw, and an exception can be caught using with-handlers. The with-handlers form includes a predicate for the exception and a handler, which is analogous to Java's implicit instance test with catch and the body of the catch form. The body of a with-handlers form corresponds to the body of a try before catch. We implement Java's finally clause using dynamic-wind.

Unlike Java's throw, the PLT's *raise* accepts any value, not just instances of a throwable. Nevertheless, PLT tools work best when the raised value is an instance of the exn record. This record contains fields specifying the message, source location of the error, and tracing information.

Our implementation of the Throwable class connects Java exception objects to PLT Scheme exception records. A Throwable instance contains a PLT exception record, and when the Throwable is given to throw, the exception record is extracted and raised. This exception record is an extension of the base PLT exception record, with an added field referencing the Throwable instance. If a catch form catches the exception, the Throwable can be extracted.

Besides generally fostering interoperability, this re-use of PLT Scheme's exception system ensures that Java programs running within DrScheme get source highlighting and stack traces for errors, etc. All of Java's other built-in exception classes (which derive from Throwable) are compiled from source.

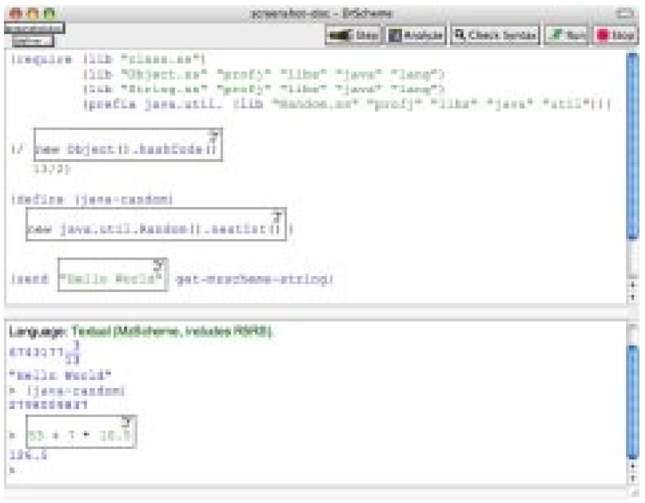
7 Interoperability

Java–Scheme interoperability is not seamless in our current implementation, but programs written in one language can already access libraries written in the other.

7.1 Java from Scheme

A compiled Java library is a module containing Scheme definitions, so that importing the library is therefore as simple as importing a Scheme library. Scheme programmers gain access to the class, (non-private) static members, field accessors, and nested classes of the Java code, and they can derive new classes and interfaces from the Java classes and interfaces. In general, they may treat bindings from Java code without regard to the original language, except to the degree that data types and protocols expose that language.

In particular, to interact with Java classes, a Scheme programmer must remember certain protocols regarding constructors and inner classes. As discussed in Section 5.1, the constructor must be called after an object is instantiated, which means that the programmer must explicitly invoke the constructor when instantiating or extending the class. Inner classes must not be instantiated directly with new, but instead instantiated through a method supplied by the containing class. (In all probability we can make inner classes module-local, and expose only an interface for instance tests, but we are uncertain whether this strategy will work with reflection.)



```
(require (lib "class.ss")
         (lib "Object.ss" "profj" "libs" "java" "lang")
         (lib "String.ss" "profj" "libs" "java" "lang")
         (prefix java.util. (lib "Random.ss" "profj" "libs" "java" "util")))

(/ new Object().hashCode()
   13/2)

(define (java-random)
  new java.util.Random().nextInt() )

(send "Hello World" get-mzscheme-string)
```

```
Language: Textual (MzScheme, includes R5RS).
6743177 1/13
"Hello World"
> (java-random)
2798258827
> 55 + 7 * 10.5
128.5
>
```

Figure 3. Java Box

As a practical matter, a Scheme programmer will think of a Java-implemented library in Java terms, and therefore must manually mangle member names, as discussed in Section 5.2. Mangled names can potentially be quite long. Consider the method equals, which takes an instance of Object. The mangled version is *equals-java.lang.Object*, to fully qualify which Object is meant. We are investigating ways to avoid this problem.

One strategy, which presently partially works within DrScheme, is to insert a graphical box representing a Java expression (see Figure 3), instead of plain text. The expression within the box contains Java instead of Scheme and results in a value. Assigning types to the arguments (to resolve overloading) remains an open problem, thus Scheme values cannot be accessed within a box.

Another remaining problem is that, while our compiler is designed to produce modules that are compatible with PLT Scheme's compilation model, the compilation manager itself does not know how to invoke the compiler (given a reference to a .java file). We are working on an extension of the compilation manager that locates a compiler based on a file's suffix. For now, manual compilation meets our immediate needs.

7.2 Scheme from Java

The native mechanism described in Section 5.4 provides a way to make Scheme functionality available to Java, but native is not a suitable mechanism for making a Scheme class available as a Java class. Instead, our compiler can use a Scheme class directly as a Java class, for instantiation, extension and overriding, or instance tests. At compile time, the compiler needs specific type information for the class, its fields, and its methods. This information is currently supplied in a separate file, with the extension .jinfo.

Every class in Java extends Object, but not every Scheme class does so. To resolve this mismatch, the compiler does not actually treat Object as a class. Instead:

- The core `Object` methods are implemented in a mixin, *`Object-mixin`*. Therefore, `Object` methods can be added to any Scheme class that does not already supply them, such as when a non-`Object` Scheme class is used in Java.
- Indeed the `Object` class used for instantiation or class extension in Java code is actually `(`*`Object-mixin`* `object%)`.
- `Object` instance tests are implemented through an interface, instead of a class. This works because `Object` has no fields (fortunately) so the class is never needed.

A `.jinfo` file indicates whether a Scheme class already extends `Object` or not, so that the compiler can introduce an application of *`Object-mixin`* as necessary. A Scheme class can explicitly extend a use of *`Object-mixin`* to override `Object` methods.

We used the native interface to quickly develop a pedagogic graphics library, based on existing Scheme functionality. Java programmers are presented with a canvas class, which supports drawing various geometric shapes in a window. This class can be subclassed with changes to its functionality. Internally, the Java class connects to a functional graphics interface over MrEd's graphics.

8 Performance

So far, we have invested little effort in optimizing the code that our compiler generates. As a result, Java programs executed through our compiler perform poorly compared to execution on a standard JVM. In fact, Java programs perform poorly even compared to equivalent programs written directly in Scheme. The current performance problems have many sources (including the use of continuations, as noted in Section 5.3), all of which we expect to eliminate in the near future. Ultimately, we expect performance from Java code that is comparable to that of PLT Scheme code.

9 Related work

The J2S compiler [3] compiles Java bytecodes into Scheme to achieve good performance of Java-only programs. This compiler additionally targets Intel X86 with its JBCC addition. J2S globally analyzes and optimizes the bytecode to enhance performance. Java classes compile into vectors containing method tables, where methods are implemented as top-level definitions. Instances of a class are also represented as vectors. Unlike our system, this compilation model does not facilitate conceptual interoperability between Scheme and Java programs. Native methods may be written in Scheme, C, C++, or assembly, which allows greater flexibility than with our system at the cost of potential loss of security. As with our system, J2S does not support reflection.

Several Scheme implementations compile to Java (either source or bytecode) [1, 2, 4, 12, 15]. All of these implementations address the interaction between Scheme and Java, but whereas we must address the problem of handling object-oriented features in Scheme, implementors of Scheme-to-Java implementors must devise means of handling closures, continuations, and other Scheme data within Java:

- JScheme [1, 2] compiles an almost-R^4RS Scheme to Java. Within Scheme, the programmer may use static methods and fields, create instances of classes and access its methods and fields, and implement existing interfaces. Scheme names containing certain characters are interpreted automatically as manglings of Java names. Java's reflection functionality is employed to select (based on the runtime type of the arguments) which method to call. This technique is slower than selecting the method statically, but requires less mangling.
- SISC [11] interprets R^5RS, with a Java class representing each kind of Scheme value. Closures are represented as Java instances containing an explicit environment. Various SISC methods provide interaction with Java [12]. As with JScheme the user may instantiate Java objects, access methods and fields, and implement an interface. When passing Scheme values into Java programs, they must be converted from Scheme objects into the values expected by Java, and vice-versa. To access Scheme from Java, the interpreter is invoked with appropriate pointers to the Scheme code.
- The Kawa [4] compiler takes R^5RS code to Java bytecode. Functions are represented as classes, and Scheme values are represented by Java implementations. Java static methods may be accessed through a special primitive function class. Values must be converted from Kawa specific representations into values expected by Java. In general, reflection is used to select the method called, but in some cases, the compiler can determine which overloaded method should be called and specifies it statically.
- In addition to a C back end, Bigloo [14, 15] also offers a bytecode back end. For this, functions are compiled into either loops, methods or classes (to support closures). Scheme programmers may access and extend Java classes.

PLT Scheme developers have worked on embedding other languages in Scheme, including Python [10], OCaml, and Standard ML. At present, the Java-to-Scheme compiler described here is the most complete.

10 Conclusion

Our strategy for compiling Java to Scheme is straightforward: we first develop macro-based extensions of Scheme that mirror Java's constructs, and then we translate Java code to the extended variant of Scheme. This strategy facilitates interoperability between the two languages. It also simplifies debugging of the compiler, since the compiler's output is human-readable, and the target macros can be developed and tested independently from the compiler.

For PLT Scheme, the main target constructs for the compiler are `module` and `class` (plus standard Scheme constructs, such as procedures). These forms preserve most of the safety and security properties of Java code, ensuring that the Java programmer's expected invariants hold when the code is used by a Scheme programmer. Scheme programmers must follow a few protocols when interacting with Java libraries, and manually include type information within method calls. However, we believe that future work will reduce these obstacles.

While still in development, our Java-to-Scheme compiler has deployed with PLT Scheme since version 205. We continue to add language constructs and interoperability features.

Acknowledgments

We would like to thank Mario Latendresse, whose Java to Java bytecode compiler, written in Scheme, provided the front-end for preliminary versions of this work. We would also like to thank Matthias Felleisein for comments on early drafts of this paper, and the reviewers for their many helpful comments.

11 References

[1] K. Anderson, T. Hickey, and P. Norvig. *JScheme User manual*, Apr. 2002. `jscheme.sourceforge.net/jscheme/doc/userman.html`.

[2] K. R. Anderson, T. J. Hickey, and P. Norvig. SILK - a playful blend of Scheme and Java. In *Proc. Workshop on Scheme and Functional Programming*, Sept. 2000.

[3] É. Bergeron. Compilation statique de Java. Master's thesis, Université de Montréal, 2002.

[4] P. Bothner. Kawa: Compiling Scheme to Java. In *Lisp Users Conference*, Nov. 1998.

[5] R. B. Findler, C. Flanagan, M. Flatt, S. Krishnamurthi, and M. Felleisen. DrScheme: A pedagogic programming environment for Scheme. In *Proc. International Symposium on Programming Languages: Implementations, Logics, and Programs*, pages 369–388, Sept. 1997.

[6] R. B. Findler and M. Flatt. Modular object-oriented programming with units and mixins. In *Proc. ACM International Conference on Functional Programming*, pages 94–104, Sept. 1998.

[7] M. Flatt. Composable and compilable macros: You want it when? In *Proc. ACM International Conference on Functional Programming*, pages 72–83, Oct. 2002.

[8] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification*. Addison-Wesley, 2000.

[9] K. E. Gray and M. Flatt. ProfessorJ: a gradual introduction to Java through language levels. In *Companion of the 18th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, pages 170–177, Oct. 2003.

[10] P. Meunier and D. Silva. From Python to PLT Scheme. In *Proc. Workshop on Scheme and Functional Programming*, Nov. 2003.

[11] S. G. Miller. SISC: A complete Scheme interpreter in Java. `sisc.sourceforge.net/sisc.pdf`, Feb. 2003.

[12] S. G. Miller and M. Radestock. *SISC for Seasoned Schemers*, 2003. `sisc.sourceforge.net/manual`.

[13] T. Parr. ANTLR parser generator and translator generator. `http://www.antlr.org/`.

[14] B. P. Serpette and M. Serrano. Compiling Scheme to JVM bytecode: A performance study. In *Proc. ACM International Conference on Functional Programming*, Oct. 2002.

[15] M. Serrano. *Bigloo: A "practical Scheme compiler" User Manual*, Apr. 2004. `www-sop.inria.fr/mimosa/fp/Bigloo/doc/bigloo.html`.

[16] K. Siegrist. Virtual laboratories in probability and statistics. `http://www.math.uah.edu/stat/`.