RJava User and Developer Manual

Yi Lin
yi.lin@anu.edu.au

November 6, 2013

Abstract

RJava is a restricted subset of the Java language with low-level extensions that allow access to hardware and operating system. RJava utilizes the same syntax as Java, and consequently inherits benefits from Java such as type safety, various software engineering tools and productivity. Futhermore, by restrictions, RJava is a fully static language with closed world assumption. Thus it requires a much more succinct runtime, and is well suitable for aggressive static compilation and optimizations. RJava is designed to be an implementation language for virtual machine construction (and more broadly for system programming).

This manual describes the language and its current implementation– the RJava Compiler (RJC). It is intended for RJava users and developers who are willing to contribute. This manual will be maintained to keep pace with the RJC code base.

Contents

1 RJava Basics

This section describes about RJava's restrictions and extensions, which make RJava differentiate from the vanilla Java.

What is RJava?

> Generally speaking, unless what is *restricted* and what is *extended*, the other part of RJava remains the same as Java (syntactically and semantically) . Besides, the restrictions and the extensions are also compliant with Java *syntax*.

Thus, Java compilers and static analyzers can parse RJava code (they are syntactically same). Java runtimes can execute RJava programs as well; However, Java runtimes will not be able to recognize RJava specific restrictions and extension, thus executing RJava with Java runtimes may result in wrong results, which is not encouraged.

RJava is also compliant with existing Java editors and IDEs. When using Java IDE to edit RJava code, extensions and restrictions need to be imported and also try avoid using execution from IDE.

1.1 RJava Restrictions

One important feature about RJava is that all its restrictions are formalized and any RJava code need to declare the restrictions it complies with. The benefit is that both the developers and the RJava compilers will be able to see the declaration and ensure the restrictions are met.

1.1.1 Restriction Rules and Rulesets

RJava restrictions are defined as Java annotations, and they are annotated by @`RestrictionRule (org.rjava.restriction.RestrictionRule)`. The following is how @`NoException` restriction is defined:

```
@RestrictionRule
public @interface NoException{
}
```

Restriction rules can be used to annotate classes or methods, in the same way as Java annotations are used. However, it is *not* encouraged to directly use restriction rules. The preferred way is to form a `@RestrictionRuleset` by a set of chosen `@RestrictionRule`, and annotate a certain scope of code with such `@RestrictionRuleset`.

One example is the MMTk scope and its restriction `@MMTk`. The following ruleset indicates that any code within `@MMTk` needs to obey `@RJavaCore` ruleset (any RJava code need to include this ruleset, it will be described in next subsection) as well as two additional restrictions (`@NoRuntimeAllocation` and `@Uninterruptible`). The declaration is as below:

```
@RestrictionRuleset

@RJavaCore
@NoRuntimeAllocation
@Uninterruptible
public @interface MMTk{
}
```

1.1.2 Restriction Model

Restrictions follow those rules:

- Restriction rules and restriction rulesets apply to scopes of classes and methods to declare restrictions, *not* to fields.
- Restriction rules and rulesets may apply to another restriction ruleset, to indicate that the latter ruleset *includes* those rules and rulesets. This may happen iteratively.
- Any RJava class needs to be restricted by `@RJavaCore`, or restricted by a ruleset that (iteratively) includes `@RJavaCore`.
- A scope restricted by a restriction ruleset is restricted by every single restriction rule within that ruleset.
- If a class is restricted by Rule A, all its methods are restricted by Rule A unless the method is restricted otherwise (particularly `@Uninterruptible` vs. `@Interruptible`).
- Restrictions on a class do *not* affect its child classes. Restrictions on a method do *not* affect its overriding methods.

- Restrictions on a class *affect* its nested classes. As a result, those restrictions also *affect* the methods of the nested class.

1.1.3 RJavaCore Ruleset

@`RJavaCore` declares the basic restrictions that any RJava-compliant code need to follow. It declares as:

```
@RestrictionRuleset

@NoDynamicLoading
@NoReflection
@NoException
@NoCastOnMagicType
@NoExplicitLibrary
@NoEnum
public @interface RJavaCore {
}
```

Specifically,

- @`NoDynamicLoading`: the code is not allowed to dynamically load classes. To forbid this, `java.lang.Class.forName()` and `java.lang.Classloader.loadClass()` are not allowed. Also inheriting from java.lang.Classloader is also not allowed.

- @`NoReflection`: the code is not allowed to use any reflection-based feature. To forbid this, any method that may return `java.lang.Class` is not allowed. @`NoExplicitLibrary` already forbids the use of `java.lang.reflect` package. Futhermore, `java.lang.Object.getClass()` is not allowed (`.class` syntax is also not allowed, since on bytecode level, it gets translated into `Object.getClass()`.)

- @`NoException`: the code is not allowed to throw exceptions. To forbit this, `throws` in method declarations and also `throw` statement are not allowed. Catching blocks are allowed in order to maintain correct syntax when using library methods, however, there will be *no* exception thrown and catching blocks will *not* get executed in any case. It is preferred to write such code as below:

```
Object lock = new Object();
```

```
try {
  lock.wait();
} catch (InterruptedException ignore) {}
```

- `@NoCastOnMagicType`: the code is not allowed to do any type casting when *at least one* side is RJava magic types. RJava magic types reside in `org.vmmagic.unboxed` package, including 5 unboxed magic types (`Address`, `Extent`, `ObjectReference`, `Offset`, `Word`) and their array counterparts. Allowed type casting for magic types can be done via provided methods in those classes (more will be discussed in the section about extensions).

- `@NoExplicitLibrary`: the code is not allowed to import any java library. However, classes in the `java.lang.*` package are implicitly imported to any Java code, and they are entangled with Java syntax. RJava *allows* the use of `java.lang.*`. Current RJava Compiler has only implemented a subset of the `java.lang.*` package, more complete implementation will be done in the future.

- `@NoEnum`: the code is not allowed to use enumerate type. Enumerate type is forbidden since enabling full features of Java `enum` would highly involve with dynamic behaviors and extensive use of library methods from `java.lang.Enum`. We designed RJava to be a *fully static* language, thus we forbid the use of enumerate type.

These core restrictions define the RJava language. Any valid RJava code needs to declare their compliance with `@RJavaCore`.

By these restrictions, RJava 1) allows closed world assumption, which helps aggressive static compilation, 2) supports little dynamic behaviors and bares very succinct execution runtime, which makes RJava possible to run with limited hardware resources, and 3) is a simple yet still expressive language.

1.1.4 Other Predefined Restrictions

RJava defines the above *restriction model*. RJava encourages its users to utilize this model and, if favorable, add their own restriction rules and rulesets to more precisely describe restrictions on their own scope. However, RJava predefined a set of restrictions. Some are already mentioned in the sections above. These predefined restrictions are subject to change, check `org.rjava.restriction.rules` package in the code base for the latest information.

1.2 RJava Extensions

In order to undertake system programming task, RJava introduces extensions to allow efficient access to hardware and operating systems. RJava extensions include 1) the `org.vmmagic` package (by Daniel et al.) that allows memory/address representation and operations, and 2) the `org.rjava.osext` package that provides access to operating systems (including some system calls).

1.2.1 org.vmmagic

The `org.vmmagic` package provides 5 unboxed magic types to describe pointer-alike types and their operations. Though implementation may vary, 'unboxed' means those types are not normal RJava objects, and they are more like primitive types though there are methods declared for each type. These magic types share the same length as the pointer length on the target machine:

- `ObjectReference`: can be cast from and to an RJava object. *No* arithmetic, load/store or comparison operations are supported. `ObjectReference` can be cast to `Address` for further operations (unsafe).

- `Address`: used as a pointer-alike type. Arithmetic, load/store and comparison operations are supported.

- `Extent`: used to describe size in bytes (unsigned positive value) Arithmetic and comparison operations are supported.

- `Offset`: used to describe offset in bytes (signed). Arithmetic and comparison oeprations are supported.

- `Word`: used as a pointer-sized integer. Arithmetic, comparison and bitwised operations are supported. Can be cast from and to the other 4 magic types.

The above unboxed magic types each have an array counterpart, namely `ObjectReferenceArray`, `AddressArray`, `ExtentArray`, `OffsetArray`, `WordArray`. They provide `create()`, `get()/set()` and `length()` operations. Arrays of those magic types should *only* be created by using these types.

Besides magic types, `org.vmmagic` provides 'pragma' for supplying information to the compiler. Useful 'pragmas' include @`Inline`, @`NoInline`, @`NoBoundsCheck`, @`NoNullCheck`, etc. Some 'pragmas' from `org.vmmagic` are now considered as an RJava @`RestrictionRule` such as @`Uninterruptible`, and some are very specific to Java and not appliable for RJava thus will be deleted. This part is still a draft, and subject to change.

1.2.2 org.rjava.osext

The `osext` extension allows access to operating systems. Currently `osext` only includes a minimum set of methods for implementing a memory manager (MMTk), and it *will* expand during further development.

- `OSConcurrency`: concurrency/threading related methods.
  - `void mutexLock/Unlock(Object lock)`: provides an alternative to using synchronization on an RJava object.
  - `void threadSuspend/Resume(Thread t)`: suspending an RJava thread (unsafe)
- `OSMemory`: memory related methods.
  - `Address malloc(int size)`: allocates raw memory
  - `Address mmap(Address start, Extent length, int protection, int flags, int fd, Offset offset)`: calls mmap system call
  - `int mprotect(Address start, Extent length, int prot)`: calls mprotect system call
  - `Address memset(Address start, int c, Extent length)`: calls memset system call
- `OSNative`: other methods that calls to native code
  - `int errno()`: a wrap of errno() in errno.h
  - `String strErrno()`: a warp of strerror()
  - `double random()`: generates a random double between 0 and 1

1.3 Relation between RJava and MMTk/JikesRVM

The RJava project was motivated by trying improve the portability of Memory Management ToolKit (MMTk). MMTk started with JikesRVM project, and serves as its memory manager. The same as the rest part of JikesRVM, MMTk is also written in a variant of Java (ad-hoc restrictions with `org.vmmagic` extensions). Though MMTk was designed to be a portable language-agnostic memory manager, its portability was never a success.

One of the main reasons that constrain its portability is the portability of the language it is written in. On one hand, such Java variant is specially tailored, and requires support from its hosting runtime. It is no longer a 'write once run everywhere' Java program, and it cannot execute on a stock Java VM. Whoever wants to host MMTk needs to find a way to execute the Java variant that MMTk is written in. On the other hand, MMTk is written in such an Java variant, and there is an inneglectable performance cost to integrate MMTk(Java) with a hosting runtime written in C/C++ which is the most common case. Past experiences (VMKit, Rotor, GHC) of porting MMTk took an approach of ahead-of-time (AOT) compiling MMTk to native codes, which overcame these two obstacles. However, *none* of their approaches is general or reusable, and each of them took a great effort. Furthermore, in term of performance, *none* of the experiences was reported as a success (meanwhile MMTk hosted by JikesRVM achieves excellent performance).

Thus the idea of formalizing the Java variant that MMTk is written in and providing an AOT compiler to lower such language for effective integration with low-level C/C++ code becomes a promising solution to MMTk portability issues. This is one part of the motivations.

Another important motivation that evolves the RJava project is the clear benefits from using a higher-level language for virtual machine implementation. Compared with traditional approach of using C/C++, higher-level languages provide benefits in safety and productivity, both of which are valuable in VM construction. However, existing experiences of using high-level languages for VM construction (including JikesRVM and MMTk) all formed their own variant of such high-level languages, which not only results in inability of code reusing but also practically introduces metacircularity issues. As we already found out, metacircularity should be orthogonal to using high-level languages; however, in practice so far, whenever high-level languages are used in virtual machine construction, the chosen implementation language is always the target language of the VM. Thus this project not only wants to settle the reusability/portability issues but also wants to create a general implementation language that can deliver high-level language benefits without falling into metacircular traps.

In short, RJava originates from the ad-hoc coding pattern used in MMTk, and absorbs the `org.vmmagic` package which is also used in MMTk. However, it evolves to be a more general implementation languages to deliver higher-level language benefits as well as good performance.

In next section, we wil describe the usage of the RJava Compiler, current implementation of RJava.

2 RJava Compiler Tools

The RJava Compiler (RJC) is an ahead-of-time compiler for RJava. It parses RJava programs, checks restriction compliance and translates into the target C code. Then the C backends could further compile the generated code into binary or other form of instructions (e.g. LLVM IR).

2.1 Building RJC

The RJava Compiler is written in Java. An ant build file is provided for automatic building.

1. Go to RJC root directory ($root).

2. Use
   ```
   ant -f mybuild.xml
   ```
   After the building succeeds, in $root/build, the following files can be found:

   - components/: a copy of external Java archives that RJC uses;
   - rjava_ext/ : a copy of RJava extensions source file;
   - rjava_rt/ : a copy of RJava runtime source file;
   - rjc.jar: the executable RJava Compiler archive.

2.2 RJava Helloworld

In this subsection, we show an example of how to compiling the simplest RJava hello world into executable.

1. Type in the source code as below, save it as 'HelloWorld.java'. For simplicity, we put the source code in the same directory as rjc.jar (under $root/build). Note that RJava uses the same file name extension as Java (.java).

   ```
   // HelloWorld.java
   import org.rjava.restriction.ruleset.RJavaCore;

   @RJavaCore
   public class HelloWorld {
   ```

```
  public static void main(String[] args) {
    System.out.println("Hello RJava");
  }
}
```

2. Compiling RJava into C.
   ```
   java -cp rjc.jar:rjava_rt/:. org.rjava.compiler.RJavaCompiler
   -rjava_ext rjava_ext/
   -rjava_rt rjava_rt/
   -soot_jdk components/soot/
   -dir .
   HelloWorld.java
   ```
   This will invoke RJava Compiler (the arguments passed to RJC will be discussed later, a script is also provided to simplify arguments needed). By default, the generated code locates in output directory of current directory. The RJava runtime sources will also be copied to the output directory, so they can be compiled along with the generated sources. And a GNU Makefile will be generated for compiling C into binary.

3. Compiling C into binary.
   ```
   cd output; make
   ```
   By default, the Makefile will use 'gcc' as the C backend to compile the generated C files. The output binary in this case will be 'HelloWorld' (named after the class where main method is found).

2.3 Full Command Line Options

The basic use of the RJava Compiler is

```
java -cp rjc.jar:rjava_rt/:. org.rjava.compiler.RJavaCompiler
[-option value] [source file(s)]...
```

Here lists all the command line options of the RJava Compiler.

Environment setting options

(Note: the'rjc' script will free you from setting these options manually.)

- **-rjava_ext PATH [REQUIRED]**
  set PATH as the directory for RJava extension source files. PATH

should contain org/rjava/osext, org/rjava/restriction and org/vmmagic. RJC requires this option to locate those extensions during parsing RJava sources.

- **-rjava_rt PATH [REQUIRED]**
  set PATH as the directory for RJava runtime source files. PATH should contain the implementation of `java.lang.*` and other runtime source files. RJC will copy the contents in this directory to the output directory, allowing the runtime to be compiled and linked with RJava programs.

- **-soot_jdk PATH [REQUIRED]**
  set PATH as the directory for the JDK jars that Soot needs. PATH should contain two JDK jars, jce.jar and rt.jar. RJC uses Soot to parse RJava source files, and Soot requires those two jars to locate Java's library.

Source file options

- **-dir PATH1:PATH2:... [REQUIRED]**
  set PATH as the classpath of RJava classes; several paths are separated by colon. RJC needs this option to correctly locate RJava files and recognize their packages. For example, org.yourcomp.HelloWorld should be located at PATH/org/yourcomp/HelloWorld.java.

- **SOURCE_FILE(S) [OPTIONAL]**
  name one SOURCE_FILE or several SOURCE_FILES to be compiled. RJC will also compile any classes referenced and used. When left blank, RJC will compile all the RJava files under the defined 'dir'.

- **-l SOURCE_LIST_FILE [OPTIONAL]**
  name a SOURCE_LIST_FILE. A SOURCE_LIST_FILE should be a pure ascii file, and have one source file name per line. RJC will compile all the denoted source files in the SOURCE_LIST_FILE.

Output options

- **-o OUTPUT [OPTIONAL]**
  the final binary will be named as OUTPUT. This option will affect the GNU Makefile that RJC generates. *By default*, OUTPUT is named

after the class name which the main method is located in. When RJC meets several main methods during compilation, OUTPUT will be set to the first class that contains a main method, and RJC will report warnings for the following encounters of main methods. When there is no main methods in the source files (usually when compiling a library), OUTPUT will be set as 'lib'.

- **-outdir OUTDIR [OPTIONAL]**
  the target files will be generated to OUTDIR. *By default*, OUTDIR is the 'output' directory under current working directory. This option is currently *ignored*.

- **-m [OPTIONAL]**
  setting this flag will mute console output of RJC. It is usually set when chaining a large number of compilations (e.g. unit testing). *By default*, this flag is unset.

- **-dt [OPTIONAL]**
  setting this flag will facilitate debugging on target code. When C is the target, this flag will additionally set '-g' as gcc flags when generating GNU Makefile. *By default*, this flag is unset.

Target options

- **-m32 [OPTIONAL]**
  setting this flag will instruct the target code to be compiled for 32 bits address. When C is the target, this flag will set '-m32' as gcc flags when generating GNU Makefile. *By default*, this flag is unset.

- **-host_os OS [OPTIONAL]**
  the target code will be compiled for and execute on the named OS. Part of RJava runtime contains OS-dependent code. Currently only MAC OS X ('mac') and Linux ('linux') are supported. *By default*, host OS is 'mac'.

Target-specific options

The RJC allows different targets for RJava to compile into. Options that start with '-target' is considered as target-specific options, and will be processed by different target code generators. Note that current implementation only targets C.

Target-specific options for targeting C

- **-target:mm=VAL [OPTIONAL]**
  name VAL as the memory management scheme for the target code. Currently, available options are using hans-boehm conservative GC ('boehm'), and using default malloc ('malloc'). The latter makes the generated code leak memory since RJava does not provide mechanisms to explicitly free memory; however, it can be useful when allocated memory tends to be permanent. A third option is provided as 'boehm-prebuilt', which uses the static library built beforehand from hans-boehm GC (currently 4 versions of the static libraries exist, 32/64bits version for Linux/Mac). This option is intended to save the time of building hans-boehm GC from sources during testing. *By default*, 'boehm-prebuilt' is used.

2.4 The 'rjc' script

Since setting environment path options for the RJava Compiler is tedious and error-prone, and using RJC jar when current working directory is not the RJC root directory makes the situation even more frustrated, a 'rjc' perl script is provided under the root directory for improved usability.

The 'rjc' script will automatically set those environment paths, and invoke the RJava Compiler under $root/build directory. If the build directory does not exist, it will build the compiler first. All options other than environment setting options can be passed to the script, which will relay to the RJava Compiler. The script also works when current working directory is not the RJC root directory. Be sure to keep the 'rjc' script *unmoved* in the RJC root directory.

It is always preferred to use the 'rjc' script rather than directly using the RJC jar.

In next section, we will describe the implementation of the RJava Compiler, which is intended for RJava developers.

3 RJava Compiler Implementation

The RJava Compiler is an RJava ahead-of-time (AOT) compiler written in Java. Currently it targets only C, translating RJava into C. However, its design bares multiple targets, and more targets can be added in the future while reusing most of the compiler features.

The RJava Compiler achieves the following major tasks:

- utilizing Soot (McGill Sable Group) to parse RJava source files and generating Jimple AST
- performing RJava-level analysis and optimizations (leaving target-level optimizations to backends)
- generating target code and runtime

3.1 Detailed Workflow

This subsection describes the detailed procedure of translating RJava into target code. For easy matching the description here with the actual code, we also reference the key classes and methods.

1. **Processing arguments and forming compilation task**
   The main method of RJava Compiler (`org.rjava.compiler.RJavaCompiler.main()`) starts by processing the arguments and setting options/flags one by one. The source files to compile will be extracted and encapsulated as a CompilationTask (`org.rjava.compiler.CompilationTask`) no matter they are denoted as a file of source list, a set of source file names, or all sources under a certain directory. CompilationTask store source class names and the classpath of the classes (fed as the '-dir' argument).

2. **Instantiating compiler instance**
   A RJava Compiler instance will be instantiated, whose constructor will take the CompilationTask as its argument. After the instantiation of a `org.rjava.compiler.RJavaCompiler` instance (singleton pattern), `init()`, `compile()`, and `finish()` will be executed sequentially. Each of them will be discussed later.

3. **Initializing the compiler**
   The source code parsing (done by Soot) and all the static analyses happen at this step.

(a) **Initializing code generator**
CodeGenerator (`org.rjava.compiler.target.CodeGenerator`, abstract class) is the part to generate the target code. It needs to be initialized according to some arguments processed earlier (`CodeGenerator.init()`).

(b) **Initializing SemanticMap**
SemanticMap (`org.rjava.compiler.semantics.SemanticMap`) stores semantic information about current compiling program and different analyses. This step accomplishes the following tasks:

- **Parsing sources and generating AST**
  Soot is wrapped as `org.rjava.compiler.semantics.SootEngine`. In this step, Soot parse all the related classes (including the imported classes). Note that we do not actually 'run' Soot and its optimizations, we simply use Soot to parse sources and use its Jimple AST. *All* the analyses and compilation in RJava Compiler are based on Jimple AST.
- **Instantiating RJC class representation**
  After last step, we have Soot representation of classes (`soot.SootClass`) /methods (`soot.SootMethod`)/etc. To make analyses convenient, we wrap those into RJC representations (see package `org.rjava.compiler.semantics.representation.*`). We only make our own representations for classes, methods, statements and invoke expressions; for the rest, we use Soot representations. Classes that are not denoted as source files but are referenced will also have their RJC representations and RJC will generate code for them (as long as they are application classes, see `SemanticMap.isApplicationClass()`).
- **Removing generic bridging methods**
  Soot may generate bridge methods for generics, which are two methods that have same signature but different return types. One has the actual method body, and the other simply calls that method. In RJC code base, these methods are referred as 'twin methods' (see `org.rjava.compiler.semantics.representation.RMethod` for more details) . We do not need bridging methods, and methods with the same signature complicates demangling. Thus we remove the bridging methods.
- **Running multiple analyses**

All static analyses that RJC needs are done here. The implementation of analyses inherits from CompilationPass (`org.rjava.compiler.pass.Com` Note that one analysis may do several passes internally. The analyses include class hierarchy analysis (`ClassHierarchyPass`), call graph (`CallGraphPass`), class initialization dependency (`DependencyGraphPass`), points-to analysis (`PointsToAnalysisPass`), constant propagation (`ConstantPropagationPass`). In addition, a restriction pass is executed here to apply RJava's restriction model (e.g. propagating restrictions to nested classes).

3.2 Codebase Overview

3.3 Unit Tests

4 RJava Compiler Details

4.1 Magic/Unboxed Types

4.2 java.lang.* Package

4.3 RJava Compiler AST

4.4 Analysis and Optimization passes

4.5 RJava Compiler Targets

5 MMTk/RJava Manual

5.1 Unofficial Changes

5.2 MMTk-VM Interface