

GraalVM FastR

Mixed interactive debugging of R and native code with FastR and Visual Studio Code

Zbyněk Šlajchrt
Oracle Labs

zbynek.slajchrt@oracle.com

GraalVM™

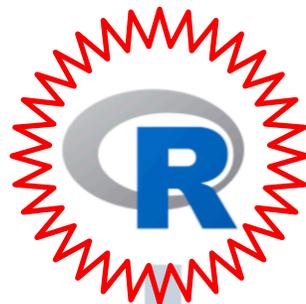


Safe Harbor Statement

The following is intended to provide some insight into a line of research in Oracle Labs. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. Oracle reserves the right to alter its development plans and practices at any time, and the development, release, and timing of any features or functionality described in connection with any Oracle product or service remains at the sole discretion of Oracle. Any views expressed in this presentation are my own and do not necessarily reflect the views of Oracle.

Agenda

- Quick Intro to GraalVM
- FastR Overview
- Debugging Native Code Examples
- Conclusion
- Q&A



Automatic transformation of interpreters to compiler

GraalVM™

Embeddable in native or managed applications



standalone



Top 5 Things To Do With GraalVM

1. High-performance modern Java (just-in-time mode)
 2. Low-footprint, fast-startup Java (ahead-of-time mode)
 3. Combine JavaScript, Java, Ruby, Python and R
 4. Run native languages (e.g. C, Fortran) via LLVM
 5. Tools that work across all languages (**debugger**, profiler ...)
- *To learn the Top 10 Things, visit a comprehensive article by Chris Seaton on <https://medium.com/graalvm/graalvm-ten-things-12d9111f307d>*

What's FastR?

- An R implementation running on **GraalVM**
- Mature, but still in the **experimental** stage
- Goals
 - Efficient
 - Polyglot
 - Compatible
 - <https://www.graalvm.org/docs/reference-manual/compatibility/>
 - Embeddable
- Licence: GPLv3

GraalVM Debugger

- GraalVM supports debugging of guest language applications
- Implements ChromeDev Tools protocol
- GraalVM applications can be debugged using, e.g.:
 - Chrome Developer Tools
 - **Visual Studio Code**
- Visit <https://www.graalvm.org/docs/reference-manual/tools/>

Native Code Debugging Examples - Prerequisites

- GraalVM installed
- FastR installed in GraalVM
- Visual Studio Code installed
- VSC R Plugin installed
- Debugging examples cloned from GitHub
- The examples folder added to VSC workspace
- Visit **fastr-mixed-debug** in <https://github.com/graalvm/examples> repository for detailed information

Example 1: Debugging Simple Native Code

- Agenda
 - How to debug a simple R and C code using FastR and GraalVM debugger
 - How to use Visual Studio Code and the R plugin to debug the code
 - How to use FastR's LLVM backend to debug native code
 - How FastR objects are displayed when debugging native code
 - <https://youtu.be/xc9mS09B7Fk>

Debugging Simple Native Code

lapplyNative.c

```
#include <R.h>
#include <Rdefines.h>

SEXP lapplyNative(SEXP list, SEXP fn, SEXP rho) {
    int n = length(list);
    SEXP R_fcall, ans;

    R_fcall = PROTECT(lang2(fn, R_NilValue));
    ans = PROTECT(allocVector(VECSXP, n));
    for(int i = 0; i < n; i++) {
        SETCADR(R_fcall, VECTOR_ELT(list, i));
        SET_VECTOR_ELT(ans, i, eval(R_fcall, rho));
    }
    setAttrib(ans, R_NamesSymbol,
              getAttrib(list, R_NamesSymbol));
    UNPROTECT(2);
    return ans;
}
```

lapplyNative.R

```
lapplyNative <- function (x, fun, env = new.env()) {
  .Call("lapplyNative", x, fun, env)
}
```

```
R CMD SHLIB -o lapplyNative.so lapplyNative.c
```

```
> dyn.load("lapplyNative.so")
> source("lapplyNative.R")
> x <- list(a = 1:5, b = rnorm(10))
> lapplyNative(x, sum)
$a
[1] 15

$b
[1] -1.45445
```

Source: <https://cran.r-project.org/doc/manuals/r-release/R-exts.html#Named-objects-and-copying>

Debugger Activation in FastR

- Launch FastR with this additional argument
 - `--inspect` – activates the GraalVM debugger

Enable LLVM Debugging in FastR

- To debug **native** code, FastR must be instructed to use the **LLVM** version of **shared** libraries
 - The LLVM bitcode is **bundled** with a shared library during **compilation**
 - Native code is compiled by the **GraalVM LLVM toolchain** (clang)
 - The **LLVM** bitcode is **interpreted** just as another GraalVM language
- Use these **LLVM-related** additional arguments
 - `--R.BackEndLLVM` – to instruct FastR to use the LLVM version of **libs**
 - `--R.DebugLLVMLibs` – to enable debugging of the LLVM bitcode

Attaching Visual Studio Code to GraalVM Debugger

```
bash-3.2$ R --inspect --inspect.Suspend=false --R.BackEndLLVM --R.DebugLLVMLibs  
Debugger listening on port 9229.  
To start debugging, open the following URL in Chrome:  
chrome-devtools://devtools/bundled/js\_app.html?ws=127.0.0.1:9229/1fbc7afb-2f50d632a8171
```

Note: The URL can be copied and pasted to Chrome to start debugging in DevTools

Debugging in Visual Studio Code

- Locate `lapplyNative.R` in VSC Explorer and toggle a breakpoint in the `lapplyNative` function
- Execute `lapplyNative` again

```
▲ VARIABLES
  ▶ env: promise <unevaluated>
  ▶ fun: promise .Primitive("sum")
  ▲ x: promise list(1:5, c(0.03381604384...
    expression: "x"
    isEvaluated: 0
  ▶ value: NULL

toulouse ▶ fastr_llvm_debug_demo ▶ simple ▶ lapplyNative.R
4
5   lapplyNative <- function (x, fun, env = new.env()) {
6     .Call("lapplyNative", x, fun, env)
7   }
8
9
10
11
```

```
▲ CALL STACK          PAUSED ON BREAKPOINT
lapplyNative         lapplyNative.R  6:5
<repl wrapper>      <REPL>         1:1
(anonymous function) <REPL>         1:1
```

Debugging in Visual Studio Code

- Locate `lapplyNative.c`, toggle a breakpoint and press F5

VARIABLES

- ans: SEXP <foreign>
 - <foreign>: list \$a\n[1] 15\n\n\$b\n[...]
 - a: integer [1] 15
 - b: double [1] -2.144418
 - 0: integer [1] 15
 - 1: double [1] -2.144418
 - <offset>: 0
 - fn: SEXP <foreign>
 - list: SEXP <foreign>**
 - <foreign>: list \$a\n[1] 1 2 3 4 5\n...

WATCH + [icon] [icon]

toulouse ▸ fastr_llvm_debug_demo ▸ simple ▸ C lapplyNative.c

```
1 #include <R.h>
2 #include <Rdefines.h>
3
4 SEXP lapplyNative(SEXP list, SEXP fn, SEXP rho) {
5     int n = length(list);
6     SEXP R_fcall, ans;
7
8     R_fcall = PROTECT(lang2(fn, R_NilValue));
9     ans = PROTECT(allocVector(VECSXP, n));
10    for(int i = 0; i < n; i++) {
11        SETCADR(R_fcall, VECTOR_ELT(list, i));
12        SET_VECTOR_ELT(ans, i, eval(R_fcall, rho));
13    }
14    setattr(ans, R_NamesSymbol,
15            getattr(list, R_NamesSymbol));
16    UNPROTECT(2);
17    return ans;
18 }
```

CALL STACK PAUSED ON BREAKPOINT

lapplyNative	lapplyNative.c	16:5
lapplyNative	lapplyNative.R	6:5
<repl wrapper>	<REPL>	1:1
(anonymous function)	<REPL>	1:1

Example 2: A Package With Rcpp Code

- Agenda
 - Debugging a package containing **Rcpp** code
 - Stepping into and debugging **Rcpp** functions
- Prerequisites
 - Rcpp 1.0.0 installed from the unpacked **source tarball**
 - R CMD INSTALL package-sources/Rcpp
 - The **gibbs** sampler example installed
 - R CMD INSTALL ./gibbs
 - <http://adv-r.had.co.nz/Rcpp.html#rcpp-package> by Hadley Wickham

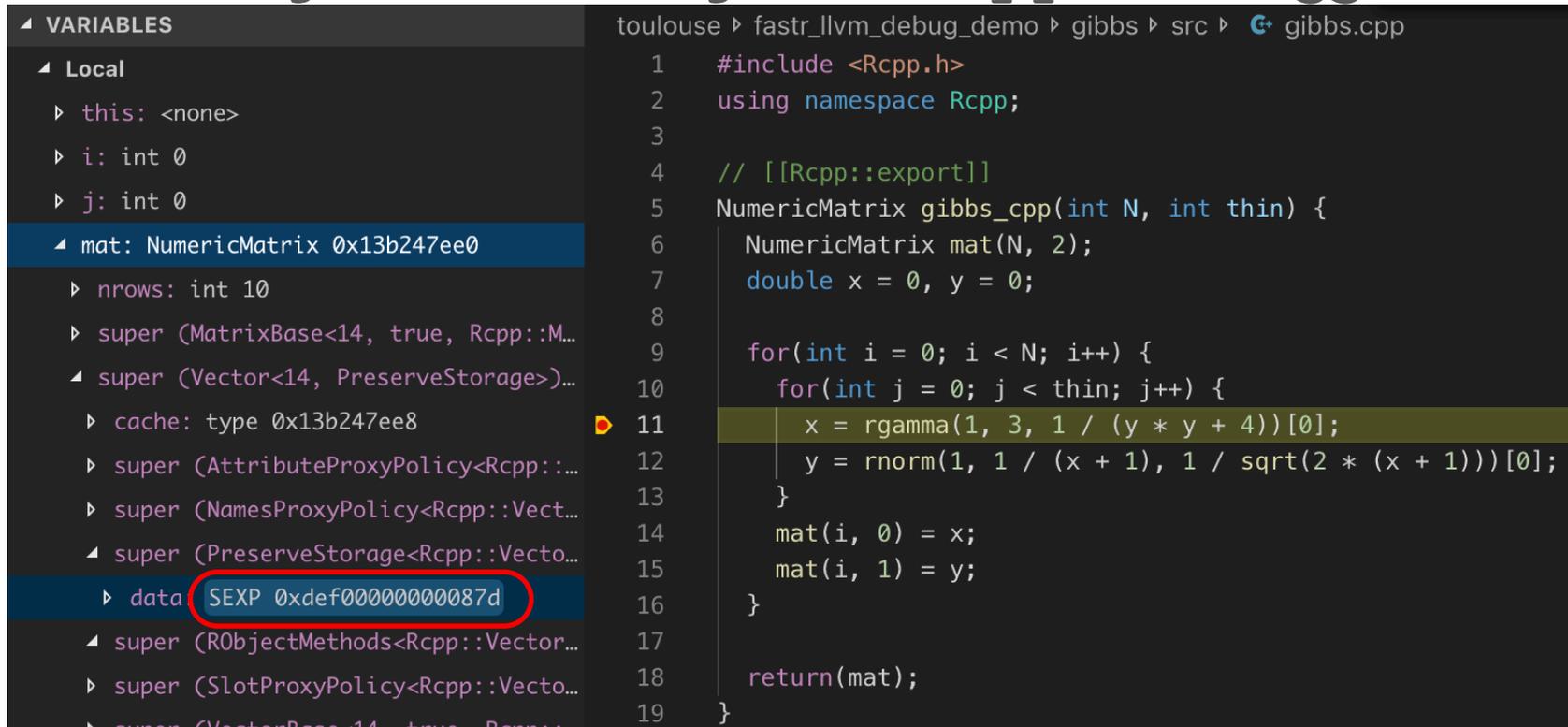
Debugging Rcpp Code

- Launch FastR in debug mode
- Load the gibbs package and execute `gibbs_cpp(100, 10)`

```
> library(gibbs)
> gibbs_cpp(10, 10)
      [,1]      [,2]
[1,] 0.4922806 0.9848679
[2,] 0.3332298 0.7986238
[3,] 0.1811279 2.3413093
[4,] 0.2363802 2.0305358
[5,] 0.6196257 0.8939006
[6,] 0.4226072 0.8424776
[7,] 0.4207100 0.2881861
[8,] 0.5958476 0.6877602
[9,] 0.3973855 1.0024997
[10,] 0.9245635 0.8350409
```

Debugging Rcpp Code (cont.)

- Switch to VSC and attach to the GraalVM debugger
- Locate `gibbs/src/gibbs.cpp` and toggle a breakpoint

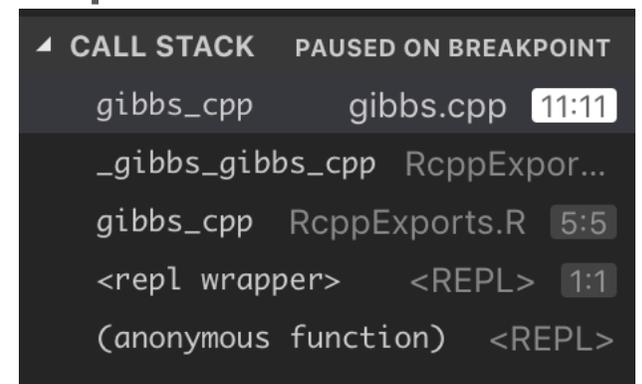


The screenshot shows the Visual Studio Code editor with the file `gibbs.cpp` open. The code is as follows:

```
1 #include <Rcpp.h>
2 using namespace Rcpp;
3
4 // [[Rcpp::export]]
5 NumericMatrix gibbs_cpp(int N, int thin) {
6     NumericMatrix mat(N, 2);
7     double x = 0, y = 0;
8
9     for(int i = 0; i < N; i++) {
10         for(int j = 0; j < thin; j++) {
11             x = rgamma(1, 3, 1 / (y * y + 4))[0];
12             y = rnorm(1, 1 / (x + 1), 1 / sqrt(2 * (x + 1)))[0];
13         }
14         mat(i, 0) = x;
15         mat(i, 1) = y;
16     }
17
18     return(mat);
19 }
```

The left sidebar shows the 'VARIABLES' panel with the following content:

```
▲ VARIABLES
  ▲ Local
    ▶ this: <none>
    ▶ i: int 0
    ▶ j: int 0
    ▲ mat: NumericMatrix 0x13b247ee0
      ▶ nrow: int 10
      ▶ super (MatrixBase<14, true, Rcpp::M...
      ▲ super (Vector<14, PreserveStorage>)...
        ▶ cache: type 0x13b247ee8
        ▶ super (AttributeProxyPolicy<Rcpp:::...
        ▶ super (NamesProxyPolicy<Rcpp::Vect...
        ▲ super (PreserveStorage<Rcpp::Vecto...
          ▶ data: SEXP 0xdef00000000087d
          ▲ super (RObjectMethods<Rcpp::Vector...
            ▶ super (SlotProxyPolicy<Rcpp::Vecto...
            ▶ super (VectorBase<14, true, Rcpp::...
```



The screenshot shows the 'CALL STACK' panel with the following content:

```
▲ CALL STACK   PAUSED ON BREAKPOINT
  gibbs_cpp    gibbs.cpp  11:11
  _gibbs_gibbs_cpp RcppExpor...
  gibbs_cpp    RcppExports.R  5:5
  <repl wrapper> <REPL>  1:1
  (anonymous function) <REPL>
```



Source: <http://adv-r.had.co.nz/Rcpp.html#rcpp-package>

Debugging Rcpp Code (cont.)

- Step into the `rgamma` Rcpp function (F11)

```
Local      165 inline NumericVector rgamma( int n, double a, double scale ){
  this: <none>      166   if (!R_FINITE(a) || !R_FINITE(scale) || a < 0.0 || scale <= 0.0) {
  a: double 3.0     167     if(scale == 0.) return NumericVector( n, 0. ) ;
  n: int 1          168     return NumericVector( n, R_NaN ) ;
  scale: double 0.25 169   }
  Closure         170   if( a == 0. ) return NumericVector(n, 0. ) ;
  Global          171   return NumericVector( n, stats::GammaGenerator(a, scale) ) ;
                  172 }

```

```
CALL STACK      PAUSED ON STEP
rgamma         random.h 166:10
gibbs_cpp     gibbs.cpp 11:11
_gibbs_gibbs_cpp RcppExpor...
gibbs_cpp    RcppExports.R 5:5
<repl wrapper> <REPL> 1:1
(anonymous function) <REPL>

```

Conclusion

- FastR as part of GraalVM provides an advanced support for **mixed debugging of native and R code**
- **Visual Studio Code** provides a comfortable debugger UI that can be used in tandem with **FastR/GraalVM**
- **TODO**
 - Completeness of LLVM implementation
 - Displaying “nativized” R objects (esp. Rcpp ones)
 - A better source-paths management for packages installed from CRAN

GraalVM™

Run Programs Faster Anywhere

Stay Tuned

Website

<http://www.graalvm.org/>

Github Repository

<https://github.com/oracle/graal>

<https://github.com/oracle/fastr>

<https://github.com/graalvm/examples>

Stay Tuned

graalvm-announce@oss.oracle.com



@GraalVM



/graalvm

Other Links

FastR overview:

<https://medium.com/graalvm/faster-r-with-fastr-4b8db0e0dceb>

GraalVM compatibility (can be used to check the status of a package):

<http://www.graalvm.org/docs/reference-manual/compatibility/>