Insert here your thesis' task.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

GNU-R Debugger Bytecode Support

Bc. Aleš Saska

Department of System Programming
Supervisor: Ing. Petr Máj

June 2, 2018

Acknowledgements

Thanks to my adviser Petr Máj for help with reviewing thesis, Tomáš Kalibera for useful help with the GNU-R code and functionality, my father for assisting with the submission of this work, and big thanks to my girlfriend for psychological support and all the tea required to make this work happen.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended, in particular that the Czech Technical University in Prague has the right to conclude a license agreement on the utilization of this thesis as school work under the provisions of Article 60(1) of the Act.

In Prague on June 2, 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Saska, Aleš. *GNU-R Debugger Bytecode Support*. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Tato práce se zabÃ · vÃą analÃ · zou a implementacÃŋ pokroÄĎilÃľho bytecode debuggeru pro jazyk GNU-R. ImplementovanÃ · systÃľm vypisuje informace v ÄĎitelnÃľ a uÅ"ivatelem srozumitelnÃľ podobÄŻ pro kaÅ"dou instrukci zadanÃľ funkce. DruhÃą ÄĎÃąst prÃące se zabÃ · vÃą

V několika větách shrňte obsah a přínos této práce v českém jazyce.

Klíčová slova Replace with comma-separated list of keywords in Czech.

Abstract

This thesis is about analysis and implementation of the advanced bytecode disassembler tool for GNU-R language. The implemented tool is returning the information in a human readable, compact format for passed function. This second feature was reused in the second part of the thesis which is about native support for bytecode debugger. The current debugging options for the bytecode in GNU-R are insufficient because of implementing through AST call. The debugging capabilities of the languare are improved in this thesis by implementing native support for debugging bytecode. Presented implementation shows all internal information of the bytecode evaluator status. The solution written in this thesis is focusing on making any negative impact to

the overall performance of the language which has been successfully tested and proven with testing.

Keywords computer language,R,GNU-R,bytecode,disassembler,debugger

Contents

List of Figures

Introduction

Motivation and objectives

Almost everyone who has been trying to write computer program has made some logical mistakes in them. To help to solve them, we usually run program **step-by-step** with debugging tools with some debugger. The **GNU-R** which is one of the most widely used scientific languages across the whole world also has its implementation for debugging code.

GNU-R language is dynamically typed interpreted language which usually means that it needs **Virtual Machine** to interpret. There are more ways to represent and implement its evaluation. The first one **Abstract Syntax Tree** (see 1.3.5) evaluation is the simplest one. To make speedup of its internal evaluation, there has been introduced the **Bytecode** (see 1.3.6) compiler and interpreter into GNU-R in 1998.

While the abstract syntax evaluator is the slowest of the two analyzers it already contains the debugging features GNU-R implemented. However, it was previously difficult to analyze and debug the other analyzer, bytecode. The only method provided to analyze the bytecode is very basic disassembler showing the data in human unfriendly way. It means that there is currently no support for real-time debugging of GNU-R bytecode evaluation. Instead of it there is implemented switching into the AST interpreter once user requests the debugging features. The whole lack of the native bytecode debugger is causing potential issues because the code used for bytecode can be slightly different than the code used while not-debugging (AST one) even if it produces the same result. Eventually, this can also cause an issue when there is an error inside AST or BC interpreter core. This can result in confusing and very hard to solve issue.

The work done in this thesis is solving the insufficient **bytecode disassembler** by replacing it with a **new easy-to-use and human-friendly one**. The another implemented feature is **advanced and user-friendly native support for bytecode debugger** which is significantly improving the op-

tions for analysis and debugging any program code running on the bytecode engine of the language. This implementation was written with the focus on having any performance slowdown of the bytecode engine which was proven in the performance testing part on the ending of the work (see 3.2). On top of this, there was implemented the feature containing **breakpoint instruction** which is used for the **simulation of the conditional breakpoints**.

All the implemented features were successfully tested. The disassembler `bctools` package has written automated tests and there was concluded an performance testing of the GNU-R engine.

This thesis is organized as follows: Section is the introduction into problems of dynamic languages evaluation. Section 1.7 contains the analysis of possible solutions how to implement the bytecode disassembler and the debugger. Section 2 is describing design of current implementation. Section 3 describes what type of testing has been done on the work. Section 3 proposing work for the future improvements of the work followed by thesis summarization in conclusion.

Chapter 1

Analysis and design

1.1 GNU R from user perspective

1.1.1 Basis usage (main commands and REPL loop)

The main **GNU R** language is written as the console application evaluating the infinite **REPL** - Read Eval Print Loop. As the abbreviation says, it is evaluating the expressions right as is entered by the user (of the `R` program) to the program standard input (`stdin`). Alongside of this, there is also the `Rscript` command in the package which supports running the program from the input file. However, it is internally implemented just as wrapper piping the file content into the `R` command.

1.1.2 GNU R package system

The **GNU-R** has integrated package subsystem **CRAN** with plenty of inbuilt packages. These packages are intended to be the easy way for developers (R users) how to make a user-friendly extension for other people. The bytecode compiler is written in the GNU-R in the separate package named `compiler`. This package comes already pre-installed so users don't need to manipulate with it as with the other user-installed libraries. However, the bytecode disassembler in this thesis is written as the separate package named `bctools` because it is better to keep the language core minimal and the disassembly tool can be implemented as a plugin into the language.

Because we would be creating our package we need to be able to manipulate and work with it. These are few basic commands to work with packages:

- `R CMD INSTALL <pkgs>` - install specified packages
- `R CMD build <pkgname>` - build the package
- `R CMD check <pkgname>` - check package (check requirements, run tests, etc.)

1.1.3 GNU R class system

Classes are way how to in Object-oriented Programming (OOP) group functionality of one meaning. For example instead of the two functions `drawButton` and `pressButton` we can have the two methods `draw` and `press` belonging to the object `button`. The GNU-R is dynamic language it is variable can be any type. It means that once we'd call method of class (function on the variable), the language engine has to look up the specific code for the class which would be then evaluated. This functionality also is used in the printing of the values or objects via the `print` method which we can use for the disassembler. The user would then be able to print out the disassembly in the same way as he is used with other data types. The `print` method is also dispatched by default in the REPL loop in printing phase.

According to the Hadley Wickham's article about GNU-R has 4 possible class systems - `S3`, `S4`, **Reference classes** and **Base classes** (internally used). It means that the **R** class system is not strictly defined as in languages like Java, C++, **Python**, etc. The whole system provides the end user to more flexibility, but on the other hand, it can be a little bit more confusing for the programmers who are used to conventional programming languages.

The system used in the print method is `S3`. It implements a style of OO programming called generic-function OO. This is different from most programming languages, like Java, C++, and C#, which implement message-passing OO. With message-passing, messages (methods) are sent to objects and the object determines which function to call. Typically, this object has a special appearance in the method call, usually appearing before the name of the method/message: e.g., `canvas.drawRect("blue")`. S3 is different. While computations are still carried out via methods, a special type of function called a generic function decides which method to call, e.g., `drawRect(canvas, "blue")`. S3 is a very casual system. It has no formal definition of classes.

1.2 GNU R internal structure

The core GNU-R VM core is written in C language with the broad number of supported platforms (Windows/MAC/Linux...) and computer architectures (ARM/x86/x64...). The multi-compiler support also means that there could be the difference in supported features which would need conditional checks for supporting this feature in the compiler code. One of the examples is support for the Computed GOTO 1.3.4.

1.2.1 Implementation of the core features of the language in R itself

The GNU-R has internally written the loading mechanism in the way that the `base` package is loaded first and then all of the packages contained in

list `getOption("defaultPackages")` are loaded into the global environment. This mechanism allows that just the core features and language constructs are written in performance-optimized `C`, and the rest can be written inside R language itself. It means that inside these packages there is substantial amount of functionality for the whole language. The `compiler` package is little different than the other packages because it needs to be loaded due to internal bytecode compiler usage (see JIT chapter 1.3.7). However, it's loading mechanism is different because it is hardcoded in the code. It is hardcoded in `loadCompilerNamespace` (in the *src/main/eval.c* file) to be independent on the whole code.

1.2.2 Calling internal C functions from R

C code compiled into R at build time can be called directly in what are termed primitives or via the `.Internal` interface, which is very similar to the `.External` interface except in syntax. More precisely, R keeps a table of function names and corresponding C functions to call, which by convention all start with `do_` and return a `SEXP`. This table (`R_FunTab` sitting in file *src/main/names.c*) also specifies how many arguments to a function are required or allowed, whether or not the arguments are to be evaluated before calling, and whether the function is internal in the sense that it must be accessed via the **.Internal** interface, or directly accessible in which case it is printed in **R** as **.Primitive**.

1.3 Computer Program Compiler Structure

1.3.1 Virtual Machine

Static languages compilers usually use directly the computer operating system as the running environment. However, the GNU-R is the dynamically typed computer language with the internal Virtual Machine (VM) providing the **running environment**. It is simulating the environment by executing code, managing memory, and providing communication layer with the underlying computer and its external devices (accessing the filesystem, network communication, etc.). The nature of all being simulated usually results in performance slowdown, but on the other hand, there is s significant safety advantage of isolation which results of the code being much safer.

There are more types of virtual machines. The most common ones are stack machines followed by the register-based machines.

The **register-based** ones simulate register processors (such as **x86** architecture Intel or **ARM** ones) with instructions which are capable of working with more registers. The bigger number of registers implicates more sophisticated instructions because they needs to contain information about what registers should be accessed (for example Android **Dalvik VM** `add-int` in-

struction has 3 parameters - Destination parameter, the first source register, and second source register). The more information which the instruction needs to contain results in longer instruction sizes and more expensive instruction decoding which both negatively impacts execution speed. However, they can profit from the nature of the register-based hardware of the most common processor architectures (ARM, x86, etc.) so they should get better evaluation performance.

The **stack-based** ones, on the other hand, are simpler ones because every instruction arguments lie on the top of the stack in the specified order (for example ADD instruction removes two topmost arguments from the stack, make the addition, and push back the result to the stack). The simplicity of instruction coding implies easier instructions and also easier implementation of both compiler and evaluator code. This type is one of the most common and also the GNU R Bytecode engine one of its representative.

The examples of virtual machines are

- JVM - Java VM
- Python
- GNU R
- Dalvik VM - Android VM (the only register one based in this list)
- JavaScript V8
- Chakra (JavaScript inside MS Edge browser)

1.3.1.1 Internal parts of the VM

GNU R virtual machine is stack based one. Due to simplicity it consists of these parts:

- **Parser** (mainly in src/main/gram.y which generates src/main/gram.c)
- **Memory management** (mainly in src/main/memory.c)
- **AST evaluator** (eval function inside src/main/eval.c)
- **Bytecode Compiler** (inside R package Compiler)
- **Bytecode evaluator** (bcEval function inside src/main/eval.c)
- **Runtime environment**

1.3.2 GNU R memory types and memory management

Each memory node is represented as `SEXP` type. It contains internal representations such as code definition (`LANGSXP`, `BCOSESXP`, `WEAKREFSXP`, promises, etc.) and also regular memory types (such as logical vectors, integer vectors, strings vectors, etc.). GNU-R is a vector language, so every value is internally represented as a vector (e.g., integer 3 is represented and boxed as `INTSXP` vector of size 1 containing value 3).

Types of memory nodes are

- **NILSXP** nil = NULL
- **SYMSXP** symbols
- **LISTSXP** lists of dotted pairs
- **CLOSXP** closures
- **ENVSXP** environments
- **PROMSXP** promises: [un]evaluated closure arguments
- **LANGSXP** language constructs (special lists)
- **SPECIALSXP** special forms
- **BUILTINSXP** builtin non-special forms
- **CHARSXP** "scalar" string type (internal only)
- **LGLSXP** logical vectors
- **INTSXP** integer vectors
- **REALSXP** real variables
- **CPLXSXP** complex variables
- **STRSXP** string vectors
- **DOTSXP** dot-dot-dot object
- **ANYSXP** make "any" args work.
- **VECSXP** generic vectors
- **EXPRSXP** expressions vectors
- **BCODESXP** byte code
- **EXTPTRSXP** external pointer

- **WEAKREFSXP** weak reference
- **RAWSXP** raw bytes
- **S4SXP** S4, non-vector
- **NEWSXP** fresh node created in new page
- **FREESXP** node released by GC
- **FUNSXP** Closure or Builtin or Special

These types are holding values which can be printed in the disassembler function 2.1 and also inside the stack printer 2.2.2 in this thesis.

1.3.3 GNU R garbage collector

The memory management in dynamic languages is maintained by **Garbage Collector**. It releases allocated memory once it is no longer used by program. GNU R implementation of memory management lies inside *src/main/memory.c*. It implements a non-moving generational garbage collector with two or three generations. Memory is allocated by `R_alloc` and is maintained in a stack. There is also protection stack managed by `PROTECT` (and `UNPROTECT`) functions which is used inside C code for internal purposes (widely used in 2.3). It allows to push locally allocated variables into it so they are reachable by the garbage collector and would not get removed during a garbage collection run (memory cleanup).

1.3.4 Computed GOTO

The Computed GOTO technique is used in the VM for code **evaluation speedup**. Also, the GNU-R VM has its internal support for this feature (managed by macro `THREADED_CODE`). Because the DEBUG instruction is internally dispatching an old previous instruction after execution of debug features there was a need to understand the whole bytecode instruction jumping.

GNU R has the support of threaded code (implemented by the **computed GOTO** technique) although there is still support for non-GCC compilers (and compilers which does not support this feature) - see 1.3.1. Enabling or disabling of this feature is managed by (`THREADED_CODE` `define` preprocessor command.

The all of the while loop and switch cases are in the GNU-R code then defined with macros (`INITIALIZE_MACHINE`, `BEGIN_MACHINE`, `OP`, `NEXT` and `LASTOP`). They are conditionally defined to either be compiled for supporting Computed GOTO 1.3.4.2 or not 1.3.4.2 (according to the `THREADED_CODE` flag).

1.3.4.1 Computed GOTO not used - Switch dispatch

The internal representation of the traditional implementation of the BC evaluator acts like big loop going through all function instructions. It causes that in each loop step there has to be branching of program flow according to instruction (`if` command). In the traditional way, this is done as the switch-case where case values are the instruction codes. Example of this approach:

```
while(1){
  switch(*opcode++){
    case POP:    //POP=1
      ... do instruction POP ....
      break;
    case GETVAR:          //GETVAR=2
      ... do instruction GETVAR ....
      break;
    case ADD:    //GETVAR=3
      ... do instruction ADD ....
      break;
  }
}
```

Figure 1.1: Example of the VM with switch dispatch architecture.

1.3.4.2 Computed GOTO used - threaded code

The `switch` statement should be implemented very efficiently by C compilers - the condition serves as an offset into a lookup table that says where to jump next which means that it is evaluated for every bytecode instruction. However, it turns out that there's a popular GCC extension that allows the compiler to generate even **faster code**. The main idea behind this is to store the address of the label into the value of a variable which allows the dynamic lookup of the next value.

In the GNU-R implementation of bytecode interpreter, there is performance optimization called **direct threaded code** associated with dispatch table and the **computed GOTO**. In process of loading bytecode into the VM internal structure (done by `R_bcDecode` and `R_bcEncode` functions inside *src/main/eval.c*) there is translation between instruction codes (integer codes) and the current location of jump labels inside `bcEval` - see *computed goto* 1.3.5(`void*` type). The nature of the operating system loader would cause that this position can (and usually is) changed every time the program is started (R VM is loaded into memory by a operating system) - so value has to be computed every time again. This allows BC interpreter to jump

```
/* The indices of labels in the dispatch_table
 * are the relevant opcodes
 */
static void* dispatch_table[] = {
        &&do_halt, &&do_inc, &&do_dec, &&do_mul2,
        &&do_div2, &&do_add7, &&do_neg};
#define DISPATCH() goto *dispatch_table[code[pc++]]

int pc = 0;
int val = initval;

DISPATCH();
while (1) {
    do_halt:
        return val;
    do_inc:
        val++;
        DISPATCH();
    do_dec:
        val--;
        DISPATCH();
    do_mul2:
        val *= 2;
        DISPATCH();
}
```

Figure 1.2: Example of the VM with computed goto architecture through dispatch table.

directly at the position which is stored inside the code array. It would cause saving one array lookup every step of BC interpreter compared to the classic interpreter (implemented for example inside **CPython** VM).

1.3.5 Abstract Syntax Tree

To be able to internally interpret the syntax of every language the code is first parsed into abstract syntax tree (**AST**, see the example in fig. 1.3). Tree in the GNU-R contains nodes of LANGSXP type with references to the symbol table (pointers to the function). These references are represented as an pointers to functions which have implemented the evaluation of the code. This tree is then traversed and evaluated inside `eval` function (which lies in file *src/main/eval.c*). The evaluator also has built-in support for debugging which is done internally by the checking of the `RDEBUG` flag of the current

executed environment. The flag is checked every evaluator step which creates performance overhead even though the debug mode is not active. However this checking overhead is usually minimized by branch prediction feature of current CPUs.

The current AST evaluator debugging implementation was in this thesis used as an inspiration for the functionality of the newly implemented features 2.3. The other thing inspired by the this debugger is the user-output in case the bytecode debugger is in the non-verbose mode. In that case the bytecode debugger is simulating the command-line behavior of the AST - see section 2.3.11.1.

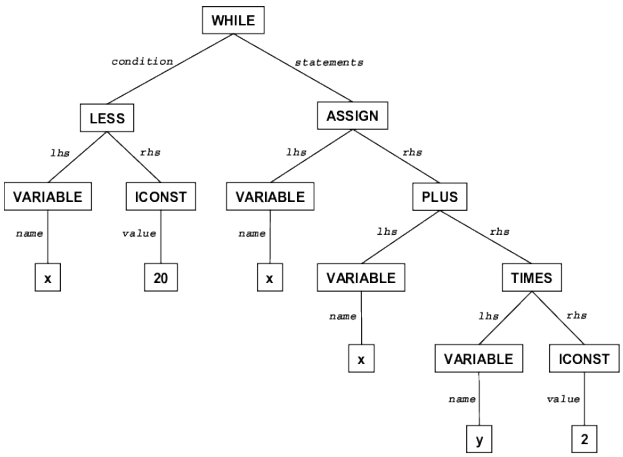

Figure 1.3: While loop example

1.3.6 Bytecode

Another option how to represent the source code to evaluate is the **bytecode**. It has an array of transferable instruction codes and constants designed for easy evaluation. The name bytecode stems from the the instruction set that have one byte operation code. Structure contains the instruction code which follows parameters (depending on how many / if parameter given instruction have). This thesis content is about an implementation of support for the bytecode evaluator of an GNU-R language.

1.3.7 Just in Time compilation

In order to speed-up evaluation of inside **VM**, there have been developed various techniques of the **performance optimization**. One of them is Just in Time (JIT) compilation of code. The underlying idea is to internally translate code into some more efficient representation (from **AST** to either **Bytecode** or to the native machine code). However, this transformation (compilation) is usually pretty expensive so it is called once the execution of a specified piece

of code reached some limit. GNU-R has the basic internal support of **JIT**. Implementation lies inside `src/main/eval.c` mainly in functions `R_CheckJIT` and `R_cmpfun`). It is executing the `Compiler::tryCmpfun` to compile function **AST** into **bytecode**. According to the posts from running code with ByteCode, the JIT enables speedup up to 10 times (theoretically up to 25 times but these cases are very rare). This means that the bytecode engine of GNU-R code can be used even without a user knowing it (explicitly calling compilation) which would make work for the bytecode debugger in our thesis very important. The second important thing is that both interpreters (AST and BC one) can be run together therefore there has to be a need to clearly decide which is currently in use (this feature is implemented in chapter 2.3.11).

1.3.8 Computer language promises

GNU-R computer language is heavily dependent on the promise pattern. As its name says this pattern represents a promise into the future that some code would be evaluated (instead of running it immediately). For example instead direct call (see figure 1.4) you can manually force GNU-R to wrap the function evaluation in the promise via the `future` function (see figure 1.5).

```
#fires immediatelly the read function
value <- read.csv(... some datafile ...)

#just prints the value
print(value)
```

Figure 1.4: GNU R - reading an data from csv table

```
#create just an promise containing the read function call
value <- future(read.csv(... some datafile ...))

#evaluates the promise (do the read.csv function)
#  on the background
#  and finally printing out the result
print(value)
```

Figure 1.5: GNU R - reading an data from csv table wrapped in promise

This shown approach is manual and pretty straightforward for the user to understand. However, the GNU-R has promise based lazy evaluation of arguments. Every argument in the function is the promise and instead of evaluating it before the function call (like in other old-fashioned languages

like `C` or `Java`), the argument is evaluated inside the function code once it is accessed (see the figure **??**). It causes that the GNU-R is internally heavy dependent on the promises even it is not obvious for the normal user at the first sight. The promises and printing of their content was done in this thesis in the stack printer 2.2.

```
getB <- function(){
        print("getB")
        5
}

calc <- function(a,b){
        print("calc enter")
        ret <- a*2
        print("accessB")
        ret <- b*10
        print("calc exit")
}
calc(2,getB())

#will produce output:
# [1] "calc enter"
# [1] "accessB"
# [1] "getB"
# [1] "calc exit"
```

Figure 1.6: GNU R - example of promise based arguments evaluation

1.4 GNU R Bytecode

GNU-R has the internal support of the BC which consists of `compiler` package for compiling to the bytecode. The language bytecode is interpreter by function `bcEval` (inside *src/main/eval.c*). The BC compiler can be used explicitly by calling certain functions to carry out compilations or implicitly by enabling compilation to occur automatically at certain points.

- **Explicit compilation** - primary functions are: `compile`, `cmpfun`, `cmpfile`
- **Implicit compilation** - can be used to compile packages as they are installed or for JIT compilation of functions or expressions.

For now, the compilation of packages is enabled by calling `compilePKGS` with argument `TRUE` or by starting R with the environment variable `R_COMPILE_PKGS` set to the positive integer value.

1.4.1 GNU R internal representation of bytecode

The internal representation of bytecode is `SEXP` node of `BCODESXP` type. It is internally represented as a linked list (CONS of cells) of two variables:

- **Bytecode code** (body) array which contains set bytecode instructions following its' parameters

  internally represented as first element (*CAR*) of the list

  accessed in the code through the `BCODE_CODE` macro

  The array contains the representation of version number followed by the bytecode instructions

- **Constant pool** array

  internally represented as second element (*CDR*) of the linked list

  accessed in the code through the `BCODE_CONSTS` macro

  contains the of the constant expressions (which are referenced in the bytecode array)

1.4.2 Expression and source references

At the end of the constant pool array, there can be (are optional) some additional information about the bytecode. This information is not used for the evaluation but are provided for specifying the original location of the compiled code. They are used in the disassembler tool 2.1 and are used in the implemented feature which is doing jumping granularity restriction 2.3.12. They can be of 2 types:

- **Expression reference**

  describing the expression representation of bytecode (for example *b+a+4*)

- **Source reference**

  describing the location in the source file (for example *main.R#4*)

The data structures in the end of the constant array can contain these class types:

- **srcref**

  Source reference representing the whole function (it's beginning)

- **srcrefsIndex**

  Array corresponding source references to code for each instruction (length of the array is length of BC code array - see 1.4.1)

- **expressionsIndex**

  Array corresponding expression references (expressions) to code for each instruction (length of the array is the length of BC code array - see 1.4.1)

1.5 Current implementation of AST debugger

AST evaluator of GNU-R is implemented as a recursive descent of the AST tree. The current implementation of debugger inside GNU-R language is made on top of the AST interpreter and it uses the `RDEBUG` flag of current evaluated to check if enable the debugging features. For user, there are written functions (user interface) managing this functionality. They are:

- `debug(fun, text = "", condition = NULL, signature = NULL)`

  enables debug features on the function `fun`

- `debugonce(fun, text = "", condition = NULL, signature = NULL)`

  run debug features on the function `fun` next time it is called

- `undebug(fun, signature = NULL)`

  disable debug features on the function `fun`

- `isdebugged(fun, signature = NULL)`

  check whether the debugging features on the function `fun` are enabled

- `debuggingState(on = NULL)`

  manages the debugging features by turning them off / on by managing R internal state

  returns boolean representing whether debugging is globally turned on. In the case that the `on` parameter is not NULL, the internal state is modified according to that parameter.

To keep the same debug functionality for functions running on top of the bytecode there is a fallback for switching back to the AST implementation. It implicates that for users the code behaves in the same way (both BC and AST representation are producing equivalent output), but can cause issues when there are bugs in the internal engine (either AST or BC evaluator). In that case, the code while debugging would be using the different code than while not-debugging. This can potentially cause confusing and hard to solve issues.

There is currently also no way to debug BC internals (stack content and showing the current evaluating instruction in the code) while running. This would be changed in this work by implementing an stack printer 2.2 and disassembler 2.1.

The GNU-R debugger internal implementation of interacting with the user is made by calling the `browse()` function which is running the environment browser. Its purpose is to wait for user input. Once the user types expression, it evaluates the typed expression. Its internal representation is reusing the function shared with main REPL loop (mainly functions `Rf_ReplIteration` and `ParseBrowser` inside *src/main/main.c*) for support user input (parsing and evaluating).

The `browse()` function also has support for the commands managing the debug mode. They are:

- `c` - exit the browser and continue execution at the next statement.
- `cont` - a synonym for c.
- `f` - finish execution of the current loop or function
- `help` - print this list of commands
- `n` - evaluate the next statement, stepping over function calls. For byte-compiled functions interrupted by browser calls, n is equivalent to c.
- `s` - evaluate the next statement, stepping into function calls. Again, byte-compiled functions make s equivalent to c.
- `where` - print a stack trace of all active function calls.
- `r` - invoke a ”resume” restart if one is available; interpreted as an R expression otherwise. Typically ”resume” restarts are established for continuing from user interrupts.
- `Q` - exit the browser and the current evaluation and return to the top-level prompt.

1.6 Current implementation of Bytecode disassembler

There is already implemented the way how to see bytecode representation - the bytecode disassembler function `disassemble` in `compile` package. Even though its current functionality is very minimal and insufficient. It works the way that it converts the code instructions and constant buffer to array which can be after then printed to the console by user (by default in the REPL loop or manually with `print` function). It means that the user would

see (the function would return) two arrays which is user unfriendly. The source code of current disassembler consists of 2 short functions `disassemble` and `bcDecode` (see figure 1.7).

```
disassemble <- function(code) {
    .CodeSym <- as.name(".Code")
    disasm.const<-function(x)
        if (typeof(x)=="list" && length(x) > 0
                        && identical(x[[1]], .CodeSym))
            disasm(x) else x
    disasm <-function(code) {
        code[[2]]<-bcDecode(code[[2]])
        code[[3]]<-lapply(code[[3]], disasm.const)
        code
    }
    if (typeof(code)=="closure") {
        code <- .Internal(bodyCode(code))
        if (typeof(code) != "bytecode")
            stop("function is not compiled")
    }
    dput(disasm(.Internal(disassemble(code))))
}

bcDecode <- function(code) {
    n <- length(code)
    ncode <- vector("list", n)
    ncode[[1]] <- code[1] # version number
    i <- 2
    while (i <= n) {
        name<-Opcodes.names[code[i]+1]
        argc<-Opcodes.argc[[code[i]+1]]
        ncode[[i]] <- as.name(name)
        i<-i+1
        if (argc > 0)
            for (j in 1:argc) {
                ncode[[i]]<-code[i]
                i<-i+1
            }
    }
    ncode
}
```

Figure 1.7: Current implementation of disassembler in GNU-R

1.7 Analysis of disassembler improvements

In the following paragraphs, there is a detailed analysis of other implementations of disassemblers and the possibility of implementation advanced one inside GNU-R. The user-interface and output of the disassembler tool 2.1 implemented in this thesis was inspired by these implementations.

1.7.1 Java bytecode disassembler

The nice example of the disassembler is in Java language (`javap` command of Java package). However, it works with the Java bytecode which is very specific because each file contains the one class (the file is named classfile). Although the GNU-R implementation is different - there can be mixed up the non-compiled (AST) and compiled (BC) code. It means that the bytecode printer is showing just the one function at once.

1.7.2 Python bytecode disassembler

Python has inbuilt support of disassembler for it's internal BC 1.9. It is provided inside package `dis` which is part of the package (no need to manually installing). Source code location is in the *Lib/dis.py*. As you can see the code is showing just one function at once. It is also showing the combined output of constant array at one line (not printing separately code and constant array). See the figure 1.9 for example.

1.7.3 Summary

The difference between the Java `javap` and the Python `dis` command is that `javap` works on the whole file instead of the Python `dis` which is printing just one function. They both dump the BC in the human-readable form with **instructions line-by-line**. The Python one is showing the parameters from the constant pool altogether with the instruction. The `javap` tool, on the other hand, supports **more levels of verbosity**.

The R can internally combine AST and BC representation of the code. It means that the architecture of the disassembler output cannot be the same as in the `javap` command which shows the whole file. Instead of it we can print out to the user the information provided by the old disassemble function which works over the whole functions (instead of files as the `javap command`). The printed out information also have lot of information which are additional information for the user (not necessary needed to interpret the code). To show these it would be nice if the GNU-R bytecode would have the ability to show these information conditionally according to the verbosity level. The inline showing values from the constant pool (inspiration by the Python `dis` function) would be also useful because it would enable the result to be printed in compact and shorter way.

1.8 Analysis of Bytecode debugger implementation

Currently, there is no support for debugging the bytecode evaluation in real time (just the fallback to the AST one is present) so there is no current implementation of the BC debugger to go through. To improve this it should be done the full native implementation of bytecode debugger. User interface of that native debugger implementation can be inspired with the current AST implementation - which is described part 1.8.1. The following parts (1.8.2 and 1.8.3) are analyzing the implementation of the **BC** debugger in other VMs (Python VM 1.8.2 and V8 javascript VM refbcdebug-implementation-in-v8).

1.8.1 Inspiration with current AST implementation

The general idea how to implement the debugger features is taken from the current implementation of the AST debugger 1.10. Its implementation of the debug code check if there is `RDEBUG` flag on the current executed function. It yes, then it prints information about the current evaluated code (source reference if available + evaluated expression). After that the `do_browser()` command is called which is internally showing the environment browser. The browser has in-build support for evaluating user-entered expressions altogether with support for handling the debugger commands (*next step*, *step into*, *continue* etc.). It also has support for showing backtrace (`where` command). The environment browser is internally reusing the REPL 1.1.1 functionality of whole language (implemented by the function `Rf_ReplIteration` or `ParseBuffer` inside *src/main/main.c*).

1.8.2 Implementation inside Python VM

One of the good examples of the similar VM is the CPython one which is the most popular Python VM. It is internally supporting just the BC interpreter (not even having AST evaluator) with the instruction set similar to the GNU-R.

The BC implementation of the its debugger is straightforward. There is implemented runtime checking of the debug flag in the label `fast_next_opcode`. To speed up the language evaluation there is a shortcut for dispatching computed goto (see 1.3.4) through dispatch table inside `FAST_DISPATCH` macro. Inside this macro is check for the `_Py_TracingPossible` &&`PyDTrace_LINE_ENABLED()` (eventually also combined with the `!lltrace` flag). This design of the implementation implicates that debugger implementation is causing performance overhead even while function not being debugged (for every evaluated BC instruction there is at least one value comparison and conditional jump needed for a processor to compute). However, this implementation is easy to implement, does

not require any specific debug instruction and does not cause any changes to the memory subsystem (potential GC issues). Also, the overhead would in the real-world usage not be huge due to branch prediction feature in the modern CPUs.

1.8.3 Implementation inside V8 VM

V8 is Javascript VM developed by the Google. It was initially used by the Chrome browser. By the time it has been also used for desktop (for example Electron framework) / server applications using the Node.js which is the V8 engine with written filesystem access, networking etc.

The V8 internal implementation is consisting of the BC interpreter (*Ignition*) and JIT x86 compiler (*TurboFan*) -see the figure 1.13.

The bytecode interpreter is single stack registed based VM (similarly to the GNU-R and *CPython* VM). Its core functionality of the debugger works on the BC instruction level. The VM defines separate **Debug instruction for every number of the arguments** (e.g. `DebugBreak0`, `DebugBreak1`, `DebugBreak2` etc.) 1.11.

If the breakpoint is set on some specific instruction (for example *ShiftRight* instruction with the 2 parameters) it causes its the replacement of the original instruction by the corresponding breakpoint one according to the number of the parameters of original one (*DebugBreak2* for *ShiftRight* because it has 2 parameters). These BC instructions work like special instructions 1.12 which internally calls the handler for the debugger and also dispatch the original instruction to maintain the same behavior (consistency) of the original code.

1.8.4 User interface and state of the BC evaluator

To keep the implementation consistent from the user perspective BC debugger should use the same user-interface as the AST. There can also be visible distinguishing between the internal state of the language (if the language is currently inside the AST or BC evaluation mode). In the current AST implementation, there is used `"debug"` as prefix printed while showing the environment browser in the debugger. This could be modified to the `"debugBC"` to signalize the user that the BC debugger is active.

The internal state of the GNU-R BC stack machine should be printed out to the output while the debugging. This state consists of:

- Current position inside the code
- Stack content

Alongside the showing the current position, there would also be need to show the function bytecode. For this feature we can use the disassembler feature 2.1 proposed in the first part of the analysis 1.7. Showing current

position inside code can be implemented as a feature inside the disassembler but for the stack content, there has to be implemented a separate tool.

1.9 Summary

In order to improve bytecode debugging, there is a need for improving (implementing the human-readable) disassembler. There was decided that the disassembler would be implemented as the separate `print.disassembly` function inside the new `bctools` package.

Another part of the thesis is about implementing the native bytecode debugger support into the GNU-R. The solution proposed in this thesis was inspired by the JS V8 VM 1.8.3 and Python VM 1.8.2. It consists of the implementing set of bytecode instructions (`BREAKPOINT0` through `BREAKPOINT4`) alongside with storing of the original instructions in the separate array. This proposed solution is performance oriented because the bytecode debugger architecture should not have any negative effect on overall language performance.

Finally in the debugger, there has to be done some showing of the evaluator internal status to the user. The status consists of the bytecode of the function, current evaluated instruction, and the stack content. For showing the function bytecode and printing the current evaluator position in the code there can be reused the disassembler tool (see 1.7) but for the dumping of the stack content, there has to be implemented completely new and separate tool.

```
#>javap -c -verbose ./HelloWorld.class
Classfile
 /C:/Users/aless/skola/thesis/java_bc/HelloWorld.class
  Last modified Mar 8, 2018; size 426 bytes
  MD5 checksum 2855c0c8a8386e26943e1bce67c9fc96
  Compiled from "HelloWorld.java"
public class HelloWorld
  minor version: 0
  major version: 53
  flags: (0x0021)
                          ACC_PUBLIC, ACC_SUPER
  this_class: #5
                // HelloWorld
  super_class: #6
                // java/lang/Object
  interfaces: 0, fields: 0, methods: 2, attributes: 1
Constant pool:
   #1 = Methodref          #6.#15
                // java/lang/Object."<init>":()V
   #2 = Fieldref           #16.#17
 // java/lang/System.out:Ljava/io/PrintStream;
   #3 = String             #18
                // Hello, World
   #4 = Methodref          #19.#20
 // java/io/PrintStream.println:(Ljava/lang/String;)V
   #5 = Class              #21
                // HelloWorld

.... other contant pool elements ...

  #25 = Utf8               Ljava/io/PrintStream;
  #26 = Utf8               java/io/PrintStream
  #27 = Utf8               println
  #28 = Utf8               (Ljava/lang/String;)V
{
  public HelloWorld();
    descriptor: ()V
    flags: (0x0001) ACC_PUBLIC
    Code:
      stack=1, locals=1, args_size=1
         0: aload_0
         1: invokespecial #1
                // Method java/lang/Object."<init>":()V
         4: return
      LineNumberTable:
22      line 1: 0

  public static void main(java.lang.String[]);
    descriptor: ([Ljava/lang/String;)V
    flags: (0x0009) ACC_PUBLIC, ACC_STATIC
    Code:
      stack=2, locals=1, args_size=1
         0: getstatic     #2
```

```
import dis

def myfunc(alist):
    return len(alist)

dis.dis(myfunc)
# Generating output
#  2           0 LOAD_GLOBAL              0 (len)
#              2 LOAD_FAST                0 (alist)
#              4 CALL_FUNCTION            1
#              6 RETURN_VALUE
```

Figure 1.9: Example output of the python `dis` command

```
if (RDEBUG(rho) && !R_GlobalContext->browserfinish) {
  SrcrefPrompt("debug", R_Srcref);
    //Print "debug" followed by
    //  source reference of the current evaluated code
  PrintValue(CAR(args));
        //print current evaluated expression
  do_browser(call, op, R_NilValue, rho);
        //run the environment browser
}
```

Figure 1.10: GNU-R AST implementation of the debugger

```
/* Debug Breakpoints - one for each possible
                size of unscaled bytecodes */
/* and one for each operand widening prefix
                bytecode                    */
V(DebugBreak0, AccumulatorUse::kReadWrite)
V(DebugBreak1, AccumulatorUse::kReadWrite,
                OperandType::kReg)
V(DebugBreak2, AccumulatorUse::kReadWrite,
                OperandType::kReg,
  OperandType::kReg)
V(DebugBreak3, AccumulatorUse::kReadWrite,
                OperandType::kReg,
  OperandType::kReg, OperandType::kReg)
V(DebugBreak4, AccumulatorUse::kReadWrite,
                OperandType::kReg, OperandType::kReg,
                OperandType::kReg, OperandType::kReg)
V(DebugBreak5, AccumulatorUse::kReadWrite,
                OperandType::kRuntimeId, OperandType::kReg,
                OperandType::kReg)
V(DebugBreak6, AccumulatorUse::kReadWrite,
                OperandType::kRuntimeId, OperandType::kReg,
                OperandType::kReg, OperandType::kReg)
V(DebugBreakWide, AccumulatorUse::kReadWrite)
V(DebugBreakExtraWide, AccumulatorUse::kReadWrite)
```

Figure 1.11: V8 BC definition for the breakpoint instructions

```
// DebugBreak
//
// Call runtime to handle a debug break.
#define DEBUG_BREAK(Name, ...)
  IGNITION_HANDLER(Name, InterpreterAssembler) {
    Node* context = GetContext();
    Node* accumulator = GetAccumulator();
    Node* result_pair =
        CallRuntime(Runtime::kDebugBreakOnBytecode,
                    context, accumulator);
    Node* return_value = Projection(0, result_pair);
    Node* original_bytecode =
        SmiUntag(Projection(1, result_pair));
    MaybeDropFrames(context);
    SetAccumulator(return_value);
    DispatchToBytecode(original_bytecode, BytecodeOffset());
  }
DEBUG_BREAK_BYTECODE_LIST(DEBUG_BREAK);
#undef DEBUG_BREAK
```

Figure 1.12: V8 current source code implementation of the breakpoint instruction

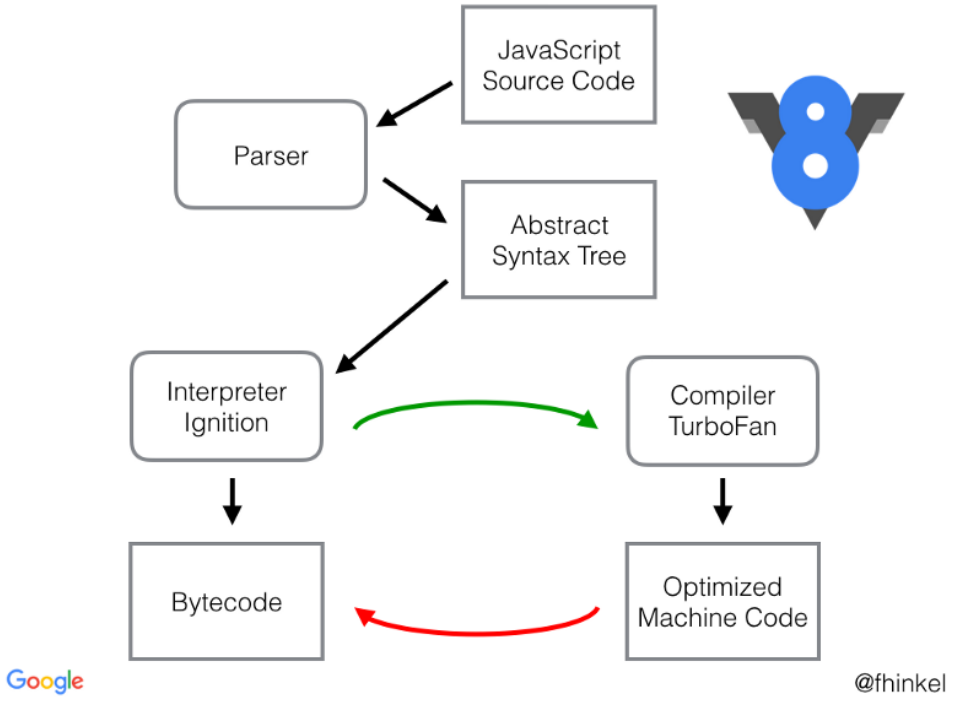

Figure 1.13: V8 internal architecture

Chapter 2

Realization

There are three different things which have to be implemented - disassembler, stack printer and finally the BC debugger itself.

2.1 Implementation of the disassembler

The whole project was structured the way that **disassembler can be easily released to the CRAN repository** without the debugger. **As much as possible code is written in the R** language (most of it in separate package `bctools`, but some also to the `compiler` one) and just necessary **minimum in C** for keeping the changes into the GNU-R core code simple.

In the GNU-R language, there is the `compiler` package written in R. Inside it there is lying the current BC compiler and BC disassembler (very minimal - see 1.6). The changes made in order to realize the implementation of the advanced disassembler involved also modifying the `compiler` package. This package lies inside the GNU-R core code so the general idea about implementation was to put the just the bare minimum inside it (the annotation of instructions 2.1.3 and putting a class into the disassembly code 2.1.1). The most of the functionality was implemented then in the `bctools` package.

2.1.1 User interface

For better user-friendliness of the BC disassembler there has been made a decision to use the advantage of the S3 class system (see 1.1.3). The old `disassemble` function inside the `compiler` package was kept intact except the **putting class into the disassembly code**. The name of the class has been decided to be the `"disassembly"`. Having this class then allows us to have *print.disassembly* function (*print* method of *disassembly* class) inside our `bctools` package. That function would be then automatically dispatched once the user calls the `print` function on the object (if the `bctools` package would be loaded inside user library).

The usage then changed from simple list (see figure 2.1) to the user-friendly disassembly code (see figure 2.2):

```
#initialization
library(compiler)
f<-function(x) {
    y <- x*2
    while(x < y)
        x <- x+1

    if(x %% 2 == 0)
        x
    else
        -x
}
compiled <- cmpfun(f)

#disassembly
#  same output due internal to behavior of REPL as the
#    print(disassemble(compiled))
disassemble(compiled)

#generated output:
# list(.Code, list(8L, GETVAR.OP, 1L, LDCONST.OP,
#         2L, MUL.OP, 3L, SETVAR.OP, 4L, POP.OP, GETVAR.OP,
#         1L, GETVAR.OP, 4L, LT.OP, 5L, BRIFNOT.OP, 6L, 30L,
#         GETVAR.OP, 1L, LDCONST.OP, 7L, ADD.OP, 8L,
#         SETVAR.OP, 1L, POP.OP, GOTO.OP, 10L, LDNULL.OP,
#         POP.OP, GETBUILTIN.OP, 9L, GETVAR.OP, 1L, PUSHARG.OP,
#         PUSHCONSTARG.OP, 2L, CALLBUILTIN.OP, 10L, LDCONST.OP,
#         11L, EQ.OP, 12L, BRIFNOT.OP, 13L, 51L, GETVAR.OP,
#         1L, RETURN.OP, GETVAR.OP, 1L, UMINUS.OP, 14L,
#         RETURN.OP), list({
#    y <- x * 2
#    while (x < y) x <- x + 1
#    if (x%%2 == 0)
#        x
#    else -x
#}, x, 2, x * 2, y, x < y, while (x < y) x <- x + 1, 1, x + 1,
#    `%%`, x%%2, 0, x%%2 == 0, if (x%%2 == 0) x else -x, -x))
```

Figure 2.1: Old disassembly user interface

2.1.2 Instruction arguments

The **BC** instruction contains an integer code identifying it followed by a variable number of arguments. These arguments can be of different 5 basic types - see2.3 (`BOOL`, `INT`, `LABEL`, `CONSTANT_LABEL` and `CONSTANT`).

The `CONSTANT` parameter can have two different meanings in the code. Some of the arguments could be a whole expression (e.g. a+b+c) kept due to usage in some corner cases during evaluation (e.g., `ADD` instruction is in the most cases taking the two topmost variables from the stack. Just in some corner cases, it is calling the internal functions which were initially designed to work on the AST evaluator, so they expect the expression as an input). These additional arguments are then stored only because the internal implementation of the bytecode evaluator and they contain the duplicate information. The optional parameter kept due to internal purposes of evaluator was named a `CONSTANT_DBG`.

These arguments are printed by the different functions in the disassembler (see 2.1.9).

- `BOOL` boolean value
- `INT` integer value
- `LABEL` - jump target / reference (integer index) to the code array itself
- `CONSTANT_LABEL` variation (extension) of the *LABEL* which allow more than one referenced index

  represented as reference (integer index) to the constant pool where is located an array containing the references (integer indexes) to the code array itself
- `CONSTANT` representing reference (integer index) to the constant pool where is located constant expression (can be either number or function)

  used for most of common cases
- `CONSTANT_DBG` - constant expression inside the argument used internally just for the corner cases (technically containing duplicitous information)

Figure 2.3: Bytecode instruction argument types

2.1.3 Annotation of instructions

These 6 types (see figure 2.3) should be printed in different way according to the annotation. There has been created a definition for each instruction argument (annotation of the instructions - see figure 2.4). The pretty printing

disassembly function (`print.disassembly` method in the `bctools` package) can then take instruction definition and print out the arguments accordingly.

In the compiler package, there is already basic annotation specifying the number of arguments for each instruction (`Opcodes.argc` list). This list was replaced by the `Opcodes.descr` list containing the fully annotated instruction (see the figure **??**). To keep the old behavior the old `Opcodes.argc` list was computed from `Opcodes.argdescr` by applying the `length` function to each element (see figure 2.5). This solution was chosen due to the fact that the definition is made just at one place and there is no need to maintain the two lists.

The `compiler` package is made with the `noweb` tool which is the tool used for writing documentation alongside with the code. Its source is written in the *src/library/compiler/noweb/compiler.nw* file. Because the build command (`make`) is not written to re-generate the source code from `noweb` each time the compiling is provided, we have to rerun the `make from-noweb` command inside the `compiler` to regenerate the R sources for the `compiler` package.

2.1.4 Instruction arguments and labels

A label is a form of representation of the reference to the code which is used in the jumps through the code.

2.1.5 Computing of labels

Labels are shown in the code in specific styling (usually incrementally numbered from the 1, e.g., `2:`). Arguments are then referencing to these locations (e.g., `$2`).

There is no direct list containing locations of the labels, but this list can be computed from the code. Its generating was done in the two-pass linear lookup through code (see figure 2.6). The result is an auxiliary array containing the number of the label.

Steps to generate labels are then:

1. **Initialize the auxiliary array** with the size of code buffer. Each element has the default value representing the fact, that there is no instruction which argument is pointing to that position.

2. **Go through all instruction** in a forward direction. For each argument, if it contains any label (see labels inside arguments 2.1.4) mark the position (which it is pointing to) with information that there is label pointing to that position.

3. **Go through the auxiliary array**. If the instruction is marked then replace the mark with the unique number of the label. This number

is calculated incrementally (at the beginning set a counter to 1, and on each marked position set the value of the counter to the array and increment the counter).

The result of this algorithm is an array containing the information whether there is no label at the instruction (`-2`) or the label (number `>=0`).

2.1.6 Verbosity and formatting

Bytecode compiled function can contain the information about a location of the source code (they are optional). These data are not necessary for the evaluation of the code but are good for human-readability (see 1.4.2). Alongside with this, there are also some of the instruction arguments which are used just for the reason of the internal implementation and have duplicate value (see the previous chapter 2.1.2). All of this information is not necessary to be displayed to the user by default, but it would be nice for them to provide an ability to display even these information. To provide this optional ability to show more basic or more advanced information there has been put a decision to implement more levels of the verbosity in the disassembly tool.

The levels are:

- `0` - display only source references (In case they are available. When not they aren't print expression references instead)

  see figure 2.7

- `1` - the same as 0 + display bytecode version and display expression references (if they are available)

  see figure 2.8

- `2` - the same as 1 + display every operand's argument (including ones used only for internal uses - see **??**)

  see figure 2.9

The default value can be pre-set by `bcverbose` function (provided in the `bctools` package).

2.1.7 Function types in the constant pool

The constant expressions in the constant pool can be of more 3 types:

- regular (ordinary) constant expressions (e.g. numbers, an array of numbers etc.)
- native functions (calls to the inside of the GNU R C implementation)
- BC compiler function code

There is no way how to print out the native functions because its structure is written inside the GNU-R runtime core in C language (the R interpreter knows just the location/function pointer/ to call). These native functions are printed in the bytecode disassembly as an <INTERNAL_FUNCTION>.

There are also stored the BC compiled functions in the constant bool. This means that there is recursion of the bytecode functions which we need to solve in the disassembler. The first approach is to print out flag like <BYTECODE_FUNCTION>. The second approach is to be able to print it out nested with some indentation. The second one was chosen due to the ability to print out more information. In order to implement this feature there were introduced these 3 parameters in the disassembly tool:

- `prefix` - the string prefix which is put before each line printed in the whole function
- `depth` - current depth of the recursion.
- `maxdepth` - maximal depth for the recursion (once the `depth` reaches this level, the <FUNCTION> instead of calling the disassembly print would be shown in the output)

This condiguration allows us to have one feature. If the `maxdepth` would be set to the 0, the disassembly tool would print out just the <BYTECODE_FUNCTION> for every nested bytecode (the nested BC printing would be disabled).

2.1.8 Printing functions

One type of the constant expressions in the constant array are the functions written as an expression references (non-byte-compiled and being able to evaluate with `eval` function). They have to be printed in a user-friendly way which however would not break the consistency of disassembly output. One way how to achieves that behavior is to call the `dput` function. However, it would format the functions line by line which is not the desired output. For making the arguments look as dense as possible (and not break consistency and compactness of the disassemble function), there has been put decision to write the function in a single line. There are 2 ways how to solve this:

- Write a specific call into the *print.c* (*src/main/print.c*) - *deparse1* function
- Call a `dput` to print out line-by-line into the buffer by `capture.output` and after that make string modifications over this output.

The first approach is cleaner that the other in sense of the code. However, it was previously decided to make only a minimal amount of the changes into the C core of the language (for being able to release the disassembler in the separate package `bctools`) so the second way was chosen.

2.1.9 Printing of different types

In the application, there are different types of the actions to print (e.g. Constants, Operators etc.). The corresponding implementation of printing functions in the disassembler is named by the `dumpNAME` convention (e.g. `dumpConstant`). The complete list of types to print is:

- **Constant**

  used for printing any constant value

  description of the functionality of the whole operator is described in the following chapter subsection (see **??**)

- **Operator**

  used for printing the operator name

  The operator names are received by the `bcinfo` function from the *compiler* package with the `.OP` suffix (e.g., `ADD.OP`). It means that after extracting the suffix these names can be printed.

- **Value**

  used for printing the `INT` and the `BOOL` argument types

  printing the expression by calling the *cat* function directly

  formatting notation - directly the `NUMBER` (e.g., 1)

- **Label**

  formatting notation - $`LABEL_NO` (e.g., $1)

- `SrcRef` a.k.a. **source reference**

  formatting notation - `SRCREF` (e.g., simple_bc_verbosity1.R#4)

- `ExprRef` a.k.a. **expression reference**

  formatting notation - @`EXPRESSION` (e.g., @$a + 1$)

  The expressions are stored in the constant pool, so technically they are a special type of the constant expressions. For printing them, we can reuse the print function for the constant expressions-`dumpConstants`. The only difference we need to print the @ as prefix. So final implementation of the dump function is printing the @ to the output and after that calling the `dumpConstants`.

2.1.10 Documenting of code

The `compiler` package has documentation written altogether with the code. It is managed through `noweb` tool (see 2.1.3).

The `bctools` package user-documentation was created with the `roxygen` tool (the GNU-R inbuilt documenting system). There are several ways for a developer to rebuild the documentation (run the `roxygen`):

- `roxygen2::roxygenise()`, or
- `devtools::document()`, if the `devtools` are used, or
- `rCtrl + Shift + D`, if the `RStudio` is used

The second listed (calling `devtools::document()`) was used during an development of this package.

2.2 Implementation of the bytecode stack printer

To implement the BC debugger we need to be able to print the `BC` stack content (see 1.8.4). The way how the implementation was designed is that it was written in the C language (GNU-R language core).

2.2.1 Stack definition

Stack definition depends on the `TYPED_STACK` typedef conditional (see figure 2.10). Once it is defined, the ability to save the unboxed values on the stack is enabled. It causes that instead of the `SEXP` stored in the stack every time (which is more expensive to handle), there could also be stored raw values (`int` or `double`). There is also `RAWMEM` memory which is used in the BC evaluator to store any data chunk - for example evaluation context frame. Once the stack is enabled, the stack values could be then out of these types:

- **`int`**
- **`double`**
- **`RAWMEM`** - a piece of raw memory

  its size is defined in a number of `sizeof(SEXP)` sized chunks
- **`SEXP`** - internal representation of the boxed object storing any value

To be able not to have to write a specific code for each stack type there is used a macro `GETSTACK_PTR` returning a boxed `SEXP` type equivalent of stack position (no matter if the value in stack is boxed or not). Ability to get the boxed value of the stack means that we can handle every place on the stack the same (except the `RAWMEM`).

2.2.2 Printing of the stack values

The printing of stack values is done through a direct call of `deparse1` function in the C core. It is inspired by the `dput` function which is used for writing an `ASCII` representation of the R object to the text output or file. The `dput` function cannot be reused without any changes because it is internally evaluating the promises while dumping the output (see 1.3.8). However the evaluating of the promises can potentially introduce some unwanted side-effects. To disable the evaluating the promises, the `deparse1` function was then called with the `DELAYPROMISES` argument (instead of evaluating the promises it is showing <`promise`> text).

2.2.3 RAWMEM stack type tag

In case the `TYPED_STACK` is defined (see 2.2.1) then the stack values can contain raw memory chunks which cannot be printed (see figure 2.10). Information about the size of these chunks is directed to the top of the stack (from bottom to the top). However we want to print the values in the direction from the top to the down. It results in the question whether the cell is printable or not. To be able to answer this question there has been created an auxiliary array containing the values if the cell is printable (see figure 2.10). It is causing some additional complexity by running one more linear pass through the stack (to fill out this array). This pass is in the forward order (from bottom to top of the stack), so we can tell whether the memory is the raw or not.

The whole algorithm to dump the stack then has two passes:

- The **first pass** from bottom to top to fill out the auxiliary array.

  Sets the `TRUE` value on the visited value. If the visited value is the `RAWMEM` type, then mark the n following (size parameter of the `RAWMEM` cell) cells `FALSE`.

- The **second pass** from top to bottom to print out the values on the stack.

  Works in the way that skip values for every place where there are set auxiliary value to FALSE. If it is set to TRUE then look at the tag (if `TYPED_STACK` available) whether is `RAWMEM`. In case it is `RAWMEM` then print the <`rawmem of size %d`>, otherwise call the `print` function for the `SEXP` value (see 2.2.2) to print out the value laying on the position.

The `TYPED_STACK`, however, could also be disabled. It means that there cannot be `RAWMEM` stored on the stack. Even though we decided to keep the whole algorithm intact which would result in the auxiliary array having just the `TRUE` values (every item is printable). This decision would cause better maintainability of the code because there are less `IFDEF` preprocessor conditions.

2.2.4 Persisting stack pointers

In regards to the **BC** stack information, there are currently these (global) variables representing the current state.

- **R_BCNodeStackBase** - the bottom of the stack
- **R_BCNodeStackTop** - current top of the stack
- **R_BCNodeStackEnd** - the end of allocated space for the stack (stack is represented internally as an array)

all of these satisfying an equation:
$R\_BCNodeStackBase <= R\_BCNodeStackTop <= R\_BCNodeStackEnd$
To be able to print values on the current evaluating bytecode stack we need to know when function stack frame starts and ends (*R_BCNodeStackBase* points to the bottom of the whole stack and not function). There is currently not any enough information from which easily we can get a start of the function stack frame. To achieve this, there has been added the `R_BCNodeStackFnBase` variable representing a begin of function stack frame. It is global variable but kept and managed through context handling (in the *src/main/context.c*) to simulate the CPU function register stack frame.

2.3 Implementation of the debugger

The primary purpose of this work is to enable the bytecode debugging in a user-friendly way. To do debugging we need to visualize the current bytecode internal state of the evaluating function (debugger work in each function separately) which consists of:

- Bytecode
- Position inside bytecode
- Bytecode stack content

The position inside bytecode can be printed alongside with the bytecode (we can re-use already implemented bytecode disassembler 2.1). For the second part, we already implemented the stack printer function.

2.3.1 Main idea

The main idea behind the debugger implementation is to maintain the same functionality and user interface as the current *AST* implementation.

2.3.2 Global design

The idea used behind the debugger implementation is **inspired by the JS V8 VM** (see 1.8.3). It is to replace the original instruction with the special breakpoint instruction together with saving the original instruction alongside the bytecode. The whole idea to make separate instruction and not the runtime check for debug flag would enable the dispatching of bytecode debug features with evaluating the same code while not causing any performance overhead in case the code is not debugged.

The bytecode debugger feature is by default disabled. Managing the state for enabling it is done by *enableBCDebug* function (written in *src/main/debug.c*). This function is is internally handling a `R_is_bc_debug_enabled` variable. See the figure 2.11 for example.

2.3.3 Instruction for debugging

To be able to dispatch breakpoints there has been created a specific set of debug instructions. For dispatching breakpoint on the instruction, the original instruction is replaced with its equivalent (according to the number of arguments) breakpoint one. The debugging instructions are:

- `BREAKPOINT0`
- `BREAKPOINT1`
- `BREAKPOINT2`
- `BREAKPOINT3`
- `BREAKPOINT4`

The decision to make a separate instruction for each argument count was made because of simplicity and forward compatibility. Because the normal instruction is replaced with the debug one but as long as the debug instructions are for each number of the arguments we know the number of the arguments of the previous instruction. This knowledge would allow us to keep all of the functions iterating over the argument count intact.

2.3.4 Storing of the original instruction when the breakpoint is set

In case the breakpoint is set (the breakpoint instruction overwrites the corresponding original one) we need to store the original instruction (to keep the functionality of the evaluated function intact).

The idea begin implemented solution is to make a deep copy of unchanged code array to preserve the original instruction list. This copy is made the first

change of the breakpoint (first replacing with the breakpoint instruction) and attached to the `BCODESXP` by making new `CONS` cell. Once this is done, we can set or unset the breakpoint on any instruction without worrying about losing any information. The placing / removing of the breakpoint instruction is done in the `modifybcbreakpoint` function (written in the *src/main/main.c* file).

The internal representation of the `BCODESXP` memory object holding the GNU-R bytecode is the same as the `LISTSXP` (the `BCODESXP` is internally wrapper over `LISTSXP`). The changes made by making an cons cell means that there is added one more nested layer. All of these functionality (going through `BCODESXP` and `LISTSXP`) is already implemented in the Garbage Collector (see 1.3.3), so there is no need to update it to support this change.

2.3.5 Setting and removing debug instruction

To be able to set (and unset) the debug instruction on the bytecode there has been created a `bcSetBreakpoint` function inside `compiler` package (see fig. 2.12 for source code). It has support for both setting and removing the breakpoint on instruction (parameter `is`). It is returning the position of newly set instruction which is used in the setting next breakpoint 2.3.7. This returned position can be different than the position passed through the parameter. The reason behind this is that this function needs to be fail-proof. It means that it cannot break the code by modifying the argument instead of the instruction (the function is exposed to the end-user). The way to solve is to implement finding of the first instruction which position (index) is the first after the position given through `code` argument (satisfies the $>=$ *code* condition). Then the `bcSetBreakpoint` function can place the breakpoint to the founded position and return that position.

2.3.6 Listing breakpoints

To manage the breakpoint status, there has been implemented a `bcListBreakpoints` function (implemented inside the `compiler` package). It is returning the array of the instruction positions which contains breakpoint (see figure 2.13 for usage example).

2.3.7 Setting the next breakpoint

The consequential instruction does not necessarily mean the following instruction right next to each other in the bytecode array (due to labels - see 2.1.4). The breakpoint for the next instruction can be either one of these:

- following in the bytecode array
- at the position where are the instruction labels pointing to (see types of labels 2.1.4)

The handling of this fact was implemented inside the`bcSetNextBreakpoint` function (inside the `compiler` package). This function is also used inside the `C` code in the debugger. To support easier dispatching it has been also written a `Rf_breakOnNextBCInst` function which is internally dispatching a `bcSetNextBreakpoint` function by a call to the `R` code through `eval`.

The implementation of the `bcSetNextBreakpoint` function is checking all possible locations for the jump locations. It is also checking if there is already set a breakpoint in that position. If no, then it is setting there breakpoint instruction. For placing the breakpoint instruction it is using the internal call to the `C` function `modifybcbreakpoint` (see 2.3.5).

It is also returning an array of newly added breakpoint locations. This feature is used in the implementation of the `BREAKPOINT` instruction (see 2.3.10) in the `C` core - this returned array is used to keep tracking of the added breakpoints.

2.3.8 Support in the disassembly tool

There has been added support for visualizing the breakpoints set on the instruction into the disassembly tool. The R disassembly script was modified to get 3 arrays as a input (added field with the original code array - see 2.3.4). The original array is returned every time no matter if the `BCODESXP` code array is modified or not (in case not modified there is returned the same code array twice).

This means that we can just simply modify the disassembler to go always through the original array. Then for every instruction we would be checking in the code array (which possibly contains breakpoints) if there is breakpoint or not. In case there is we would just simply print an mark `(BR)` as prefix for the instruction name to signalize that this instruction contains breakpoint (see the figure 2.14 for example).

2.3.9 Temporary and regular breakpoints

Currently there are two types of the breakpoint inside the bytecode:

- temporary breakpoint
- regular breakpoint

The regular ones are used for user-defined breakpoint, these ones are set by calling `bcSetBreakpoint` function (from `compiler` package). Once the bytecode interpreter reaches them the breakpoint functionality is called and the R shows the debugger interface.

The temporary ones are used on the other hand for handling debugger commands. They are implemented with the same instruction except there are also held their locations on the bytecode local stack (variable in which points

global `R_BCtmpBreakpoints`). This value contains an array in which each elements is representing the location of the currently set temporary breakpoint. These temporary breakpoints always point to the instruction succeeding current the evaluated one (see 2.3.7).

In order to keep the garbage collector satisfied and because it is not possible to store the variable in the local protection stack (through `PROTECT/UNPROTECT` function - see 1.3.3) during `bcEval`, there has been dedicated one field on the bytecode stack for storing this array. The `R_BCtmpBreakpoints` variable is then pointer (`SEXP*` type) to this location. This design allows changing of this variable while modifying the bytecode body from the different evaluated context. Changing this array and not keeping an new one also reflects the fact that the modifications inside the bytecode code array are also made in-place by modifying this array.

2.3.10 Implementation of the breakpoint instructions

The reason why there are implemented separate breakpoint instructions for each number of instruction arguments is its annotation. It allows us that breakpoint instruction would have the information of number of its arguments with itself. It would prevent from the breaking the bytecode code array structure. However the evaluated code inside all of the breakpoint instructions would be the same. This means that it can be simply generalized by writing single macro for all breakpoint instructions. This macro was decided to be named `DO_BREAKPOINT` and it is containing this algorithm with the functionality for showing the debugger feature:

- Remove all temporary breakpoints from the bytecode
- Print bytecode interpreter internal status (see 2.3.11)
- Call debug browser
- Set `RDEBUG` debug flag (see 2.3.15)
- Call the original instruction (see 2.3.14)

As you can see in the time of calling the browser there are all temporary breakpoints removed from the bytecode code array. It means that just the regular user-defined breakpoints (see 2.3.9) would the printed to the output (see 2.3.8). It is desired behavior because we do not want to print the user breakpoints which are used just for internal purposes of a step-by-step feature of the debugger (see 2.3.15).

2.3.11 Bytecode interpreter internal status

Users need to know whether the interpreter is in the BC or in the AST mode. In order to achieve this, the `"debugBC"` string was put as a prefix in the each debugger step (instead of `"debug"` in the AST evaluator).

Thing of importance while debugging the bytecode is the ability to locate the currently evaluated code (position inside the code). For achieving this there has to be printed the internal status of the BC interpreter. This was implemented in two possible ways:

- **Short compact way** inspired by AST status printing
- **Long verbose way** showing the all information available in the **BC** interpreter - used by default

To control whether to print out the short compact way or the long way the `R_DebugVerbose` boolean variable is used. This flag is accessible for the user through the `debugVerbose()` function which acts as getter and setter altogether. It is returning the value of the variable as return value while having an optional parameter used for the modifying of the flag.

2.3.11.1 The short compact way of status printing

It is used to simulate the AST printing behavior. It is printing the data in the same way as AST to remain the backward compatibility support (for example for the debuggers in IDEs). This way is used by default.

2.3.11.2 The long verbose way of showing all information

It is used for the printing of the whole internal state bytecode interpreter. To support this feature there has been implemented function `printBCStatus()` (see the figure 2.15).

```
void printBCStatus(){
    Rprintf("      --- Evaluating bytecode --- \n");
    R_printCurrentBCbody(R_BCbody, R_BCpc, TRUE, 1);
    Rprintf("      ---------- Stack dump ---------- \n");
    R_printCurrentBCstack(
        R_BCNodeStackFnBase,
        R_BCNodeStackTop);
}
```

Figure 2.15: printBCStatus function for printing the whole BC status information

As you can see the code is reusing the bytecode disassembler (see 2.1) and stack printer (see 2.2).

The bytecode disassembler function is printing the current bytecode instruction with the surrounding instructions (`R_printCurrentBCbody`. It is calling the bytecode with the `select` parameter set to the selected instruction position (to print out the `>>>` string) and the `peephole` argument set to `TRUE` to show just the surrounding instructions instead of the whole function (current instruction and the 5 following ones).

2.3.12 Debugger jumping granularity

The AST debugger is making one jump for every AST expression. The bytecode debugger, on the other hand, can jump in the much more granular way (not according to the changes of expression references but one step for each bytecode instruction). Because the short compact way should be a simulation of the AST debugger it means that the debugger should also jump in similar way as the AST (according to changes of expressions and not just by the bytecode instructions). This can be implemented in two ways:

- calculate the next breakpoint location in the `bcSetNextBreakpoint`
- runtime checking by skipping the debug functionality in case the expression reference has not changed

The first way has better performance - the breakpoint would be placed on the right instruction where should be debugger functionality dispatched and there would be no runtime checking. The second one would work by placing the breakpoint instruction to the very next one while skipping the debugger functionality unless the change in expression reference occur (runtime checking).

The runtime-checking was chosen due to implementation simplicity. The performance overhead would be only when the debugger is enabled which we are not worried in the first place.

2.3.13 Handling of the recursive character of the bytecode

The debugger implementation is modifying the breakpoint code by adding temporary breakpoint instructions (see 2.3.7). This means that while calling recursive call there can be already set breakpoint instruction in the evaluated code. However, we do not want to have any of them executed in the recursive call of the function. To solve this there was done checking (at the beginning of the `bcEval` function) whether the code is modified. If yes then we are creating a shallow copy of the current code without a modified bytecode code array by making new `BCODESXP` object created from an original code array and constant array (see figure 2.16).

Because of this implementation is allocating a new element we have to satisfy the language GC by putting its reference to the bytecode stack. It was decided to push the current evaluated body even when this change is not done. It is creating tiny memory overhead by having one unnecessary element on the stack but it is resulting in better code readability because of less conditions in the code.

2.3.14 Threaded and non-threaded design of the application

Because of the speedup of the GNU-R evaluating there is support for the `THREADED` code (see 1.3.4).

This means there were two different dispatch systems which had to be analyzed and modified to support the jumping to the different direction (jump for calling the original instruction - inside the *DO_BREAKPOINT* macro).

2.3.14.1 THREADED_CODE defined

In this case, the needed changes were minimal (see figure 2.17). It required adding a macro `BREAKPOINT_GOTO_ORIGIN_OP` for a switching to the different instruction. In this case, the original bytecode instruction is represented with the location inside the code (see 1.3.4) so the jump is being executed directly to that location.

2.3.14.2 THREADED_CODE not defined

In this situation, things are more complicated because the whole evaluator design is one big loop. The changes in this code were described in the figure 2.18. The loop originally had one big switch (its beginning is defined in `BEGIN_MACHINE` macro) which was deciding which instruction to execute according to the value of operand (`*op`). We changed this behavior so that it is loading operand (`*op`) into a variable (`jmp_opcode`), then placed another jump label (`do_instruction`) and after that finally decided which instruction has to be evaluated according to the value of that variable (by `switch` command). In the case of executing breakpoint instruction and evaluating original instruction (jumping to another instruction through `BREAKPOINT_GOTO_ORIGIN_OP`) we are setting the auxiliary variable `jmp_opcode` to the desired instruction code and evaluating the switch statement for instruction (jumping to the `jmp_opcode` label).

2.3.15 Handling of the debugger user input

The original environment browser has inbuilt support for handling the debugging (*step into*, *next step*, *continue* etc.) via the user input (see 2.3.15). It is managed by setting up the `RDEBUG` flag and the `R_BrowserLastCommand` variable on the evaluated environment. The intent of implementing bytecode

debugger interface was to reuse as much of this feature. It resulted in the fact that the bytecode debugger control is also managed through these variables.

Bytecode interpreter also provides the more status information (see 2.3.11.2). It has been decided to extend the user interface to support showing these pieces of information by those commands (see implementation in the figure 2.19):

- `bc` - print the current bytecode with marked the current evaluated position (see 2.1)
- `bcstack` - print current bytecode stack (see 2.2)

2.3.16 Entry points to the bcEval function

While debugging bytecode, there are no runtime checks inside the `bcEval` function for the `RDEBUG` flag (due to performance reasons - see 2.3.3). Because of this and the nature of dispatching breakpoint functionality through the special `BREAKPOINT` instructions (see 2.3.3) there is need for setting the bytecode instruction 2.3.7 to dispatch bytecode functionality. The modifying of the flag of the debugger functionality (`RDEBUG`) means that we have to check and potentially apply the breakpoint instruction to the code every time the `RDEBUG` flag can be modified.

Currently, there are two places where this change can happen (entry places into the `bcEval` function):

- At the beginning of `bcEval` - the bytecode function was just called
- Inside `bcEval` at the positions of the return from the function calls

The first case is handled at the beginning of the bytecode evaluation function (`bcEval` function). In this case, there is a call to the `Rf_breakOnNextBCInst` which is setting the breakpoint to the first instruction.

The other entry point (after the return from the function calls) would be used just because of the conditional breakpoints implementation 2.4. If the debugger modified the state of the evaluated bytecode function (the function was not in debug mode and now is due to calling breakpoint), add the debugging instruction to the next following instruction (see 2.4).

2.4 Simulated conditional breakpoints

Due to lack of the ability of the simple and user-friendly conditional breakpoints, there has been put a decision to add support for simulating them. This feature was implemented via `breakpoint()` function which fires the debugger functionality on the spot. It allows the user to simulate conditional breakpoint behavior by calling the environment browser. This `breakpoint()` (see fig. 2.21) function is inspired by the JS `debugger;` command, but instead

of being a separate command (which would be big change in the language parser) it is a callable function.

As you can see in the figure 2.20, this function is modifying the `RDEBUG` flag of the currently evaluating environment and resetting the temporary bytecode variables (`R_BrowserLastCommand` and `browserfinish` member of function context).

```
#initialization
library(compiler)
library(bctools) #new package
f<-function(x) {
    while(x < y) x <- x+1

        x
}
compiled <- cmpfun(f)

#disassembly
#  same output due internal to behavior of REPL as the
#    print(disassemble(compiled))
disassemble(compiled)

#generated output:
#
1:
  @ x
  GETVAR              x
  @ 10
  LDCONST             10
  @ x < 10
  LT
  @ while (x < 10) x <- x + 2
  BRIFNOT             while (x < 10) x <- x + 2  | $2
  @ x
  GETVAR              x
  @ 2
  LDCONST             2
  @ x + 2
  ADD
  @ x <- x + 2
  SETVAR              x
  @ while (x < 10) x <- x + 2
  POP
  GOTO                $1
2:
  LDNULL
  POP
  @ x%%2
  GETBUILTIN          `%%`
  GETVAR              x
  PUSHARG
  PUSHCONSTARG        2
46CALLBUILTIN         x%%2
  @ 0
  LDCONST             0
  @ x%%2 == 0
  EQ
  @ if (x%%2 == 0) x else -x
  BRIFNOT             if (x%%2 == 0) x else -x   | $3
```

```
<<opcode argument description>>=

SKIP.ARGTYPE<--1L
LABEL.ARGTYPE<-0L
CONSTANTS.ARGTYPE<-3L
CONSTANTS_DBG.ARGTYPE<-4L
CONSTANTS_LABEL.ARGTYPE<-5L
BOOL.ARGTYPE<-6L
INT.ARGTYPE<-7L

Opcodes.argdescr <- list(

BCMISMATCH.OP = c(),
RETURN.OP = c(),
GOTO.OP = c(LABEL.ARGTYPE),
BRIFNOT.OP = c(CONSTANTS.ARGTYPE,LABEL.ARGTYPE),
POP.OP = c(),
DUP.OP = c(),
PRINTVALUE.OP = c(),
STARTLOOPCNTXT.OP = c(BOOL.ARGTYPE, LABEL.ARGTYPE),
    #  bool is_for_loop, pc for break
.... all remaining instructions ....
)
```

Figure 2.4: Example of instruction annotation

```
Opcodes.argc <- lapply(Opcodes.argdescr, length)
```

Figure 2.5: Computation of argument count in the compiler package

```
#first pass to mark instruction with labels
#labels is array that describes if each Aq
#    instruction has label
n <- length(code)
#labels now contains -2=not used, -1=used
labels <- rep(-2, n)
i <- 2
instrCnt<-0 # count number of instructions
while( i <= n ) {
    v <- code[[i]]
    argdescr <- Opcodes.argdescr[[paste0(v)]]
    j <- 1
    while(j <= length(argdescr)){
        i<-i+1
        if(argdescr[[j]] == argtypes$LABEL){
            labels[[ code[[i]] + 1 ]] <- -1
        }else if(argdescr[[j]] == argtypes$CONSTANT_LABEL){
            v <- constants[[ code[[i]] + 1 ]]
            if(!is.null(v)){
                for(k in 1:length(v)){
                    labels[[v[[k]] + 1]] <- -1
                }
            }
        }
        j<-j+1
    }
    instrCnt<-instrCnt+1
    i<-i+1
}

#second pass to count labels
#loop through labels array and if
#   that instruction has label marked on it
#labels array now contains values:
#   -2=not used, -1=used, >0=index of label
i <- 2
lastlabelno <- 0;
while( i <= n ) {
    if(labels[[i]] == -1){
        lastlabelno <- lastlabelno+1
        labels[[i]] <- lastlabelno
    }
    i<-i+1
}
```


Figure 2.6: Code for generating labels

```
1:
 - #1: function(a) while(a) a <- a-1
   GETVAR                a
   BRIFNOT               while (a) a <- a - 1        | $2
   GETVAR                a
   LDCONST               1
   SUB
   SETVAR                a
   POP
   GOTO                  $1
2:
   LDNULL
   INVISIBLE
   RETURN
```

Figure 2.7: Disassembly output with verbose lvl 0

```
Bytecode ver. 10

1:
 - simple_bc_verbosity1.R#4: function(a) while(a) a <- a-1
   @ a
     1: GETVAR               a
   @ while (a) a <- a - 1
     3: BRIFNOT              while (a) a <- a - 1    | $2
   @ a
     6: GETVAR               a
   @ 1
     8: LDCONST              1
   @ a - 1
    10: SUB
   @ a <- a - 1
    12: SETVAR               a
   @ while (a) a <- a - 1
    14: POP
    15: GOTO                 $1
2:
   17: LDNULL
   18: INVISIBLE
   19: RETURN
```

Figure 2.8: Disassembly output with verbose lvl 1

```
Bytecode ver. 10

1:
 - simple_bc_verbosity2.R#3: function(a) while(a) a<-a+1
  @ a
    1: GETVAR                 a
  @ while (a) a <- a + 1
    3: BRIFNOT                while (a) a <- a + 1    | $2
  @ a
    6: GETVAR                 a
  @ 1
    8: LDCONST                1
  @ a + 1
   10: ADD                    a + 1
  @ a <- a + 1
   12: SETVAR                 a
  @ while (a) a <- a + 1
   14: POP
   15: GOTO                   $1
2:
   17: LDNULL
   18: INVISIBLE
   19: RETURN
```

Figure 2.9: Disassembly output with verbose lvl 2

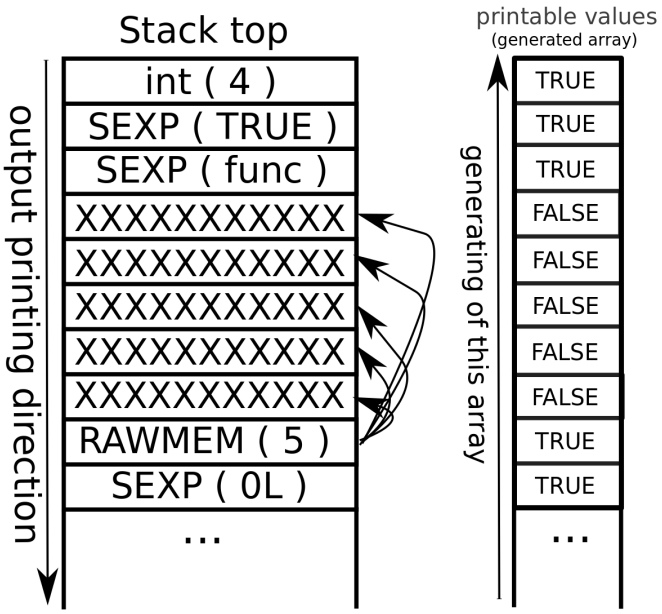

Figure 2.10: Definition of stack elements and generated auxiliary array showing the pritable elements

```
options(keep.source=TRUE)
library(compiler)
enableBCDebug(TRUE)

f<-function(a){
      c<-a+1
      d<-c+a
      c-d
}
compiled <- cmpfun(f)
debug(compiled)
compiled(2)
```

Figure 2.11: Example of debugged function with bytecode debugger enabled

```
bcSetBreakpoint <- function(code, pos, is=TRUE) {
  if (typeof(code)=="closure")
    bc <- .Internal(bodyCode(code))
  else
    bc <- code
  if (typeof(bc)!="bytecode")
    stop("Internal error - code is not bytecode")

  bc <- .Internal(disassemble(bc))
  bcode <- bc[[2]]
  newbcode <- rep(bcode) #replicate original bytecode

  #loop through bytecode over instructions and find
  #   matching instruction
  setpos <- 2
  repeat{
    if(!(setpos < length(bcode) && setpos <= pos)) break
    setpos <- setpos + 1 + Opcodes.argc[[bcode[setpos]+1]]
  }

  .Internal(modifybcbreakpoint(code, setpos-1, is));

  setpos-1
}
```

Figure 2.12: Source code of bcSetBreakpoint function

```
options(keep.source=TRUE)
library(compiler)
library(bctools)

f<-function(a){
    c<-a+1
    d<-c+ac
    c-d
}

compiled <- cmpfun(f)

#set breakpoints

 #this breakpoint would be set into position 12,
 # because at 11 is argument and
 # the implemented functionality is setting
 # the breakpoint in that cases
 # to the first following instruction
bcSetBreakpoint(compiled, 11);
 #14 is regular instruction
bcSetBreakpoint(compiled, 14);

#print the current function
# - notice the (BR) in the instruction
print(disassemble(compiled),verbose=2)

#print the bytecode instructions
# - see the 12 and 14
# -   returning an c(12,14) equivalent
print(bcListBreakpoints(compiled))
```

Figure 2.13: Example usage of bcListBreakpoints function

```
Bytecode ver. 10

- #2: c<-a+1
 @ a
   1: GETVAR                 a
 @ 1
   3: LDCONST                1
 @ a + 1
   5: ADD                    a + 1
 @ c <- a + 1
   7: SETVAR                 c
   9: POP
- #3: d<-c+a
 @ c
  10: GETVAR                 c
 @ a
  12: (BR) GETVAR            a
 @ c + a
  14: ADD                 c + a
 @ d <- c + a
  16: SETVAR                 d
  18: POP
- #4: c-d
 @ c
  19: GETVAR                 c
 @ d
  21: GETVAR                 d
 @ c - d
  23: SUB                    c - d
  25: RETURN
```

Figure 2.14: Example of showing an instruction with breakpoint in the disassembly tool - (notice `GETVAR` instruction on position 12)

```
/* duplicate body in case this
        function has modified */
if(BCODE_HAS_TMPBREAKPOINTS(body)){
  SEXP expr = TAG(body);
  body = CONS(
            BCODE_CODE_UNBREAKPOINT(body),
            BCODE_CONSTS(body));
  SET_TAG(body, expr);
  SET_TYPEOF(body, BCODESXP);
}

/* satisfy GC */
BCNPUSH(body); /* pushing body is neccesary
        just in case of duplicated, but pushing
        even unchanged one is easier for code
    readability */
```

Figure 2.16: Modifying the bytecode array in the beginning of bcEval to erase breakpoints from the code

```
#define NEXT() (__extension__ ({ \
    currentpc = pc; goto *(*pc++).v; \
    }))

#define BEGIN_MACHINE  NEXT();
    init: { loop: switch(which++)


#define BREAKPOINT_GOTO_ORIGIN_OP(inst) do{ \
    __extension__ ({goto *(*(inst)).v;}); \
  } while(0)
```

Figure 2.17: Changes made for instruction handling macros in case `THREADED_CODE` defined

```
#define NEXT() goto loop

#define BEGIN_MACHINE  loop: \
    currentpc = pc; \
    jmp_opcode = *pc++; \
    do_instruction: switch(jmp_opcode)

#define BREAKPOINT_GOTO_ORIGIN_OP(inst) do{ \
    jmp_opcode = *(inst); \
    goto do_instruction; \
  } while(0)
```

Figure 2.18: Changes made for instruction handling macros in case `THREADED_CODE` not defined

```
...
other commands
...

}else if (!strcmp(expr, "bcstack")){
    rval = 2;
    RCNTXT* cntxt = GetBCDebugContext();
    if(R_BCIntActive)
        R_printCurrentBCstack(
            R_BCNodeStackFnBase,
            R_BCNodeStackTop);
    else
        Rprintf("Debugged context is not bytecode\n");
} else if (!strcmp(expr, "bc")) {
    rval = 2;
    RCNTXT* cntxt = GetBCDebugContext();
    if(R_BCIntActive)
        R_printCurrentBCbody(R_BCbody, R_BCpc, FALSE, 1);
    else
        Rprintf("Debugged context is not bytecode\n");
} else if () {
...
other commands
...
```

Figure 2.19: Implemenation of `bcstack` and `bc` commands in the debugger interface

```
SEXP attribute_hidden do_breakpoint
        (SEXP call, SEXP op, SEXP args, SEXP rho)
{
  Rboolean oldrdebug;
  oldrdebug = RDEBUG(rho);
  SET_RDEBUG(rho, 1);
  R_GlobalContext->browserfinish = 0;
  R_BrowserLastCommand = 'n';


  if(!R_is_bc_debug_enabled() && R_BCIntActive)
    warning(_("Calling breakpoint in the BC

  while bytecode debugger disabled"));

  /* Support for bytecode debugger */
  if(R_BCIntActive && !oldrdebug){
    R_RemoveBCtmpBreakpoints(
      R_BCbody, *R_BCtmpBreakpoints);
    Rf_breakOnNextBCInst(
      R_BCbody, R_BCpc, R_BCtmpBreakpoints);
  }

  return R_NilValue;
}
```

Figure 2.20: Implementation of simulated conditional breakpoints through breakpoint() function

```
#setting up the bytecode
options(keep.source=TRUE)
debugVerbose(TRUE)
enableBCDebug(TRUE)
library(compiler)

#evaluated function with breakpoint fired
# once the x variable equals to 3
f<-function(){
  for(x in 1:5){
    if(x == 3){
      breakpoint();
    }
  }
}
compiled <- cmpfun(f)
compiled()
```

Figure 2.21: Example usage of simulated conditional breakpoints through breakpoint() function

Chapter 3

Testing and future work

The each 3 implemented parts of this thesis were tested separately. The goal was to make as much code as possible to be able to run the automated tests on.

3.1 Bytecode disassembler

The main part of the work was done as the separate GNU-R package `bctools`. The R package system has its test subsystem which runs every file in the `tests` directory and checks if it returns an error or not.

The test files of the `bctools` package include:

- **basics.R** - basic functionality (printing the different cases - for loop, switch command, while loop etc.)
- **advanced.R** - advanced functionality
- **closure.R** - printing function closure
- **switch.R** - switch command
- **peephole.R** - peephole parameter
- **bcverbose.R** - verbose parameter
- **bcversion.R** - check if the current GNU R BC version is the supported one in this package (see 1.4.1)

Because of the simplicity, the tests were not written without any testing framework. Nowadays if the test fails, there is not any more detailed information where the error occurs inside the file. To support this functionality, it is possible to modify the tests to use `test_that` framework which supports naming the tests and printing the name in the test fails.

3.2 Performance testing

There was done performance a performance testing to prove that the changes done in the GNU-R language bytecode evaluator did not caused any slowdown. The testing was done through the set of the micro-benchmarks executing the computationally expensive bytecode functions.

Data were taken on the computer with these specifications (the system was running just the test scripts one at the time):

- Windows 10 Home - Ubuntu 16.04.4 Xenial subsystem
- Intel Core i7 8550u - 1.8GHz, 4 cores, 8 threads
- 16GB RAM

The result (see the fig. 3.1) showed just minimal fluctuations which are caused by inacurracy during the measuring. This means that there is no performance difference caused by implementation of the BC debugger.

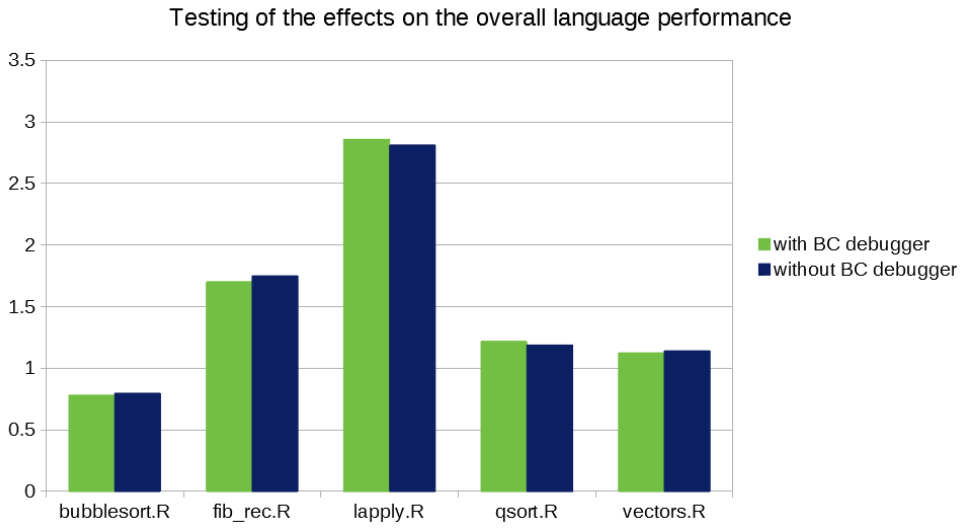

Figure 3.1: Performance testing

3.3 Future work

3.3.1 Push into working repository

The big part of the planned future part of the work is to collaborate with the GNU-R core team to push this thesis code into their development and potentially even the production code. After this would be done, and considering the

number of people using GNU-R language (the estimation in 2013 was approx 2 million), the work done in this thesis would have a significant worldwide effect to a lot of the people.

3.3.2 Merging the bctools package into the compilers

The `bctools` package was developed independently as add separate disassemble feature with the minimal changes to the GNU-R VM core. This tool was also used in the debugger which is part of the VM core. When the debugger would be deployed it is suggested to merge the `bctools` package into the `compiler` because the source code of the `bctools` package is short (approx. 600 lines). The reason why this was not already done, but just proposed as a future work, is that it would allow to deploy the disassembler into the working branch faster and eventually give a user-feedback for improvements before merging inside the `compiler` package and deploying the debugger.

3.3.3 GNU R Memory optimization

Some of the parts of the errors which were done and fixed during the development of this thesis were memory-based. During an investigation of these errors, there has been noticed one thing. The core of the GNU-R language is heavily dependent on the linked list memory data structure. Because of the modern computer memory system cache implementation), this can cause a lot of memory cache misses which affect the performance. The most straightforward way to do this is to use a `smallvector` technique (inspired by a llvm compiler), but the more appropriate way would be to do analysis of whole GNU-R memory subsystem optimization with focus on cache locality.

Conclusion

The GNU-R language is one of the most popular scientific language worldwide used (by not just scientific community). For better evaluation performance its VM internally supports the bytecode alongside with old-fashioned and slow AST interpreter. Since nowadays options for debugging the bytecode were very limited. The old disassembler showed only the raw array of instructions and constant buffer and there was no native implementation of bytecode debugging. Instead of it the evaluator was internally dispatching the AST debugger which technically resulted in code going over different code while debugging. This has changed by features proposed in this thesis.

There has been proposed and implemented changes which contains the **user friendly and easy-human readable bytecode disassembler** 2.1. The second part is about the implementation of the **native debugger support for bytecode evaluator** 2.3. As the additional feature there has been added `breakpoint()` function which enables to add support for simulating conditional breakpoints 2.20.

In the beginning of the work there is brief introduction to the problem with the motivation part . Following is the analysis of the current implementation of the GNU-R bytecode disassembler 1.7 and debugger 1.8 alongside with the comparison to similar tools and features in VMs of other languages. After that there are described implementation details of the work. Firstly there is described implementation of the disassembler 2.1 followed by the stack printer 2.2. Finally there is described debugger implementation 2.3 where is shown reuse of two previously described features. As an addition it is described the implementation of support for simulated conditional breakpoints 2.4. The end of the work contains chapter about the testing of all features 3.2 altogether with possible future work and improvements 3.

All the features were developed and tested and are included on the enclosed DVD. Personally it was a very good experience to work on the real language VM while implementing feature which would use millions of people. The work done in this thesis can be considered as successful.

Appendix A

Acronyms

VM Virtual Machine

BC Bytecode

GC Garbage collector

AST Abstract syntax tree

IR Intermediate representation

JS JavaScript

OO Object oriented

OOP Object oriented programming

JIT Just in time

REPL Read-Eval-Print-Loop

UI User interface

ARM Advanced RISC Machine, originally Acorn RISC Machine

CRAN The Comprehensive R Archive Network

GCC GNU Compiler Collection

TDD Test driven development

Appendix B

Contents of enclosed DVD

- `readme.txt` ..... instruction for the file
- `analysis` ..... analysis of other implementations
- `examples` ..... implementation examples directory
  - `cond_breakpoint` simulated conditional breakpoint examples directory
  - `r_breakpoints` ..... bytecode debugger examples directory
  - `r_disassembly` ..... bytecode disassembler examples directory
- `noncomplete_functions` ..... analysis of other implementations
- `bctools` ..... the bctools package directory
- `r_src` ..... modified GNU R source
- `thesis` ..... the thesis source code directory
  - `DP_Saska_Ales_2018.pdf` ..... the thesis text in PDF format
  - `DP_Saska_Ales_.tex` ..... the thesis text in LaTeX format