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

April 23, 2018

Acknowledgements

Thanks to my adviser Petr Máj for help with reviewing thesis, my code adviser Tomáš Kalibera for useful help with the VM code structure, my father for assistation with the submiting of this work, and big thanks to my girlfriend for psychological support.

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 April 23, 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

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 current bytecode disassembler tool. The second part is describing the current debugger implementation and there is implementation of it's support for bytecode.

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

Contents

Citation of this thesis vi

Introduction **1**
Motivation and objectives 1
Problem statements 2

1 Analysis and design **3**
1.1 GNU R from user perspective 3
1.1.1 Basis usage (main commands and REPL loop) 3
1.1.2 GNU R package system 3
1.1.3 GNU R class system 3
1.2 GNU R internal structure 4
1.2.1 Implementation of the core features of language in R itself 5
1.2.2 Calling internal C functions from R 5
1.3 Computer Program Compiler Structure 5
1.3.1 Internal structure 5
1.3.2 Virtual Machine 6
1.3.3 Garbage Collector 7
1.3.4 Computed GOTO 8
1.3.5 Abstract Syntax Tree 10
1.3.6 Bytecode 10
1.3.7 Just in Time compilation 11
1.4 GNU R Bytecode 12
1.4.1 GNU R internal representation of bytecode 12
1.4.2 Expression and source references 13
1.5 Current implementation of AST debugger 13
1.6 Current implementation of Bytecode disassembler 15
1.7 Analysis of disassembler improvements 16
1.7.1 Java bytecode disassembler 16
1.7.2 Python bytecode disassembler 18

1.7.3 Summary 19
1.8 Analysis of Bytecode debugger implementation 19
1.8.1 Inspiration with current AST implementation 19
1.8.2 Implementation inside Python VM 20
1.8.3 Implementation inside V8 VM 20
1.8.4 User interace and state of the BC evaluator 22
1.8.5 Summary 23

2 Realisation **25**
2.1 Implementation of the disassembler 25
2.1.1 User interface 25
2.1.2 Instruction arguments 27
2.1.3 Annotation of instructions 28
2.1.4 Instruction arguments and labels 29
2.1.5 Computing of labels 30
2.1.6 Verbosity and formatting 31
2.1.7 Function types in the constant pool 32
2.1.8 Printing functions 33
2.1.9 Printing of different types 33
2.1.10 Documenting of code 34
2.2 Implementation of the bytecode stack printer 35
2.2.1 Stack definition 35
2.2.2 Printing of the stack values 36
2.2.3 RAWMEM stack type tag 36
2.2.4 Persisting stack pointers 37
2.3 Conditional breakpoints 38
2.4 Implementation of the debugger 39
2.4.1 Main idea 39
2.4.2 Global design 39
2.4.3 Instruction for debugging 40
2.4.4 Storing of the original instruction when the breakpoint is setted 41
2.4.5 Setting and unsetting debug instruction 41
2.4.6 Listing breakpoints 43
2.4.7 Setting the next breakpoint 44
2.4.8 Support in the disassembly tool 46
2.4.9 Temporary and regular breakpoints 47
2.4.10 Implementation of the breakpoint instructions 47
2.4.11 Handling of the recursive character of the bytecode . . . 48
2.4.12 Threaded and non-threaded design of the application . . 48
2.4.13 Handling of the debugger user input 50
2.4.14 Default behavior of the bytecode debugger 50
2.4.15 Verbose mode of bytecode debugger 51
2.4.16 Initial entry points to the bcEval functions 51

3 Testing and future work **55**
3.1 Bytecode disassembler 55
3.2 Bytecode debugger 56
3.2.1 Effectiveness of testing 57
3.2.2 Side effects of the scripts 57
3.3 Conditional breakpoints 58
3.4 Future work 58
3.4.1 Push into working repository 58
3.4.2 Merging the bctools package into the compilers 58
3.4.3 GNU R Memory optimization 58

Conclusion **59**

Bibliography **61**

A Acronyms **63**

B Contents of enclosed DVD **65**

List of Figures

1.1 While loop example 11
1.2 V8 internal architecture 22

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 sort of debugger. The **GNU R** is one of the most widespread used scientific languages across the whole world.

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

Abstract syntax tree which is usually the slowest one has all debugging features already implemented, but since now there was now no way how to analyze and eventually debug the bytecode one. There has been just very basic bytecode disassembler which was showing the data in human unfriendly way with actual no support for real debugging evaluation of bytecode. Also while debugging function instead of supporting debugging inside bytecode mode there is just fallback that switch to **AST** in case the function is debugged. This is causing potential issues because the code used for debugging can be slightly different than the code used while not-debugging even if it produces the same result. Eventually this can also cause issue when there is an error inside core AST or BC interpreter, and because the debug and non-debug mode is internally using the different code it can cause to confusing and very hard to solve issue.

This work is solving this issue by analyzing problem, suggesting solutions, and implementing them. The solutions include the implementation of more advanced and user-friendly bytecode disassembler tool and implementing of support for the bytecode debugger. This would enable and significantly improve the support for analysis and debugging any program.

At the beginning of the thesis there is introduction into problems of evaluation of dynamic languages followed by the analysis of possible solutions how to implement the bytecode disassembler and the debugger. In the middle of work there is design of current implementation. After tgat there is described what type of testing has been done on the work. At the end there is proposed work for the future improvements of the work followed by thesis summarization in conclusion.

Problem statements

There are two main types of programming languages. Dynamic typed and static typed ones. Static such as **C** / **C++** / **FORTRAN** languages has strictly defined types and are usually more machine inspired/focused. Dynamic ones usually do not have strict type inference so they allow the user (programmer) more freedom. As trade-off they usually needs Virtual Machine (**VM**, see 1.3.2) to runs efficiently which cause possible performance slowdown. **GNU R** language was primarily developed for the scientists so user-friendliness and no need to necessary care about data types was one of the decision to make the language dynamic typed.

TODO: add more of some good shit

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** loop. As the abbreviation says, it is evaluating the expressions right as is entered by user (the **R** program). Alongside of this there is also the **Rscript** command in the package which supports running the program from the input file, but 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 its own integrated package subsystem **CRAN** with plenty of inbuilt packages. These packages are intended to be easy way for developers (**R** users) how to make user friendly extension for other people.

Therese are few basic commands to work with packages:

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

1.1.3 GNU R class system

GNU R Citing the Hadley Wickham's article OO field guide [**?**]:

R has s three OO systems differ in how classes and methods are defined:

- **S3** 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.

- **S4** works similarly to S3, but is more formal. There are two major differences to S3. S4 has formal class definitions, which describe the representation and inheritance for each class, and has special helper functions for defining generics and methods. S4 also has multiple dispatch, which means that generic functions can pick methods based on the class of any number of arguments, not just one.

- **Reference classes**, called RC for short, are quite different from S3 and S4. RC implements message-passing OO, so methods belong to classes, not functions. $ is used to separate objects and methods, so method calls look like *canvas$drawRect("blue")*. RC objects are also mutable: they don't use R's usual copy-on-modify semantics, but are modified in place. This makes them harder to reason about, but allows them to solve problems that are difficult to solve with S3 or S4.

  There's also one other system that's not quite OO, but it's important to mention here:

- **base types**, the internal C-level types that underlie the other OO systems. Base types are mostly manipulated using C code, but they're important to know about because they provide the building blocks for the other OO systems.

Basically 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 little bit more confusing for the programmers which are used to regular programming languages.

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...).

1.2.1 Implementation of the core features of 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. These 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 a lot of functionality for the whole language environment.

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 Internal structure

To make compilers structure internally more organized most of the compilers or **VM**s are divided into separate passes.

The common basic ones are:

- Frontend - parsing source code into **IR** or **AST** (see chapter 1.3.5)
- Optimizer - optimizing **IR** or **AST** (can be left out)
- Backend - generating machine code or running **VM** (see chapter 1.3.2)

Each step can be also divided into more steps (for example frontend consists usually of lexer and parser).

As long as the **GNU R** is dynamic language it is build over the internal Virtual Machine. It does an on-demand parsing of the source code which directly evaluates.

The optimization problem is very difficult inside the **GNU R**. One of its' example why this is not possible is extreme dynamism of the language. For example you can:

- Change the body, arguments, and environment of functions.
- Change the S4 methods for a generic.
- Add new fields to an S3 object, or even change its class.
- Modify objects outside of the local environment with «-.

Basically the **GNU R** is monolithic structure built in over the Virtual Machine. This design is not ideal but it came up from to historic reasons.

1.3.2 Virtual Machine

As long as the **GNU R** is the dynamically typed computer language it needs the **Virtual Machine** to runs on. Static compiled languages use as the environment directly the computer operating system. On the other hand the **Virtual Machine** is simulating the environment by executing code, managing memory, and providing communication layer with the underlying computer and it's external devices (accessing filesystem, network communication etc.). This means there is performance slowdown but on the other hand there is big safety advantage of isolation which means that you cannot write the harmful code that interacts directly with the underlying system.

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

The **register based** ones simulates register processors (such as **x86** architecture Intel or **ARM** ones) with instructions having more registers to get internally values from. This implicates more sophisticated instruction coding because each one has to also have encoded what registers should be used as input for the instruction (for example Android **Dalvik VM** add-int instruction has 3 parameters - Destination parameter, first source register, and second source register). This results in longer instruction sizes and more expensive instruction decoding which both negatively impacts execution speed.

The **stack based** ones on the other hand are simpler ones because every instruction arguments lies on the top of the stack in the specified order (for example ADD instruction removes two topmost arguments from the top, make addition, and push result back to the top of the stack). This implies much easier instructions and also easier implementation of whole compiler and evaluator.

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)

GNU R virtual machine is register based one with one byte size bytecode instructions. 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)
- etc.

1.3.3 Garbage Collector

Memory management in dynamic languages is maintained by **Garbage Collector**. It releases allocated memory once it is no longer used. **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 allocated by *R_alloc* is maintained in a stack. Code that *R_alloc*ates memory must use *vmaxget* and *vmaxset* to obtain and reset the stack pointer.

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 internally an vector language so every value is internally represented as 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

1.3.4 Computed GOTO

The internal representation of **BC** evaluator inside **VM** acts like big loop going through all **BC** instructions of function. In the each loop step there has to be branching of flow according to instruction. 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;
  }
}
```

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. 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 value of variable which allows the dynamic lookup of the next value.

Example of computed-goto code:

```
/* 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();
}
```

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). Enabing of this feature is managed by (*THREADED_CODE* define macro flag.

The all of the while loop and switch cases are in the **GNU R** code then defined with macros (e.g. *OP*, *BEGIN_MACHINE* etc.). These macros code are conditionally defines to either be compiled to enable or disable support for threaded code. Basically according to the *THREADED_CODE* flag there is conditional two implementations of these macros which enables or disables this functionality.

In the **GNU R** implementation of bytecode interpreter there is performance optimization 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* inside *src/main/eval.c*) there is translation between instruction codes (1 byte/char codes) and 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 is loaded into memory by operating system) - so value has to be computed every time again. This allows **BC** interpreter to jump directly at the position which is stored inside the **BC** definition which means saving one array lookup every step of **BC** interpreter (**BC** instruction) 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**, example in fig. 1.1). Tree in the **GNU R** contains nodes of *LANGSXP* type with references to the symbol table (pointers to the function) which contains function to evaluate 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 current executed environment in its every step (creates performance overhead even though the debug mode is not active).

1.3.6 Bytecode

Another option how to represent the source code to evaluate is the **bytecode**. It is array of transferable instruction codes and constants designed for easy evaluation. The name bytecode stems from instruction set that have one

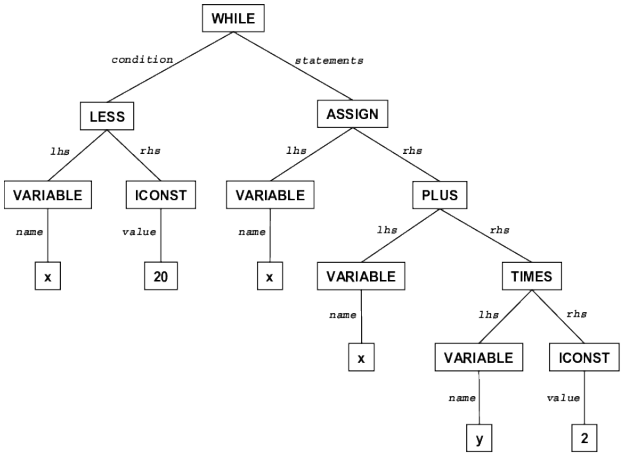

Figure 1.1: While loop example

byte operation code. Structure contains the instruction code which follows parameters (depending on how many / if parameter given instruction have).

1.3.7 Just in Time compilation

In order to speedup evaluation of inside **VM** there has been developed various techniques of the performance optimization. One of them is Just in Time 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 specified piece of code reached some limit. **GNU R** has basic internal support of **JIT**. Implementation lies inside src/main/eval.c mainly in functions *R_CheckJIT* and *R_cmpfun*) and doing running of the *Compiler::tryCmpfun* to compile function **AST** into **bytecode**. According to the posts from running code with ByteCode JIT enables speedup up to 10 times (theoretically up to 25 times but these cases are very rare).

As mentioned before, the compilation can take significant amount time, so it makes sense for the virtual machine to compile just the most used parts of code (if the every function would be compiled, it would have even cause the slowdown). The **GNU R** has for this more strategies:

- *NO_CACHE*

  functions are compiled 1st time seen

  code is never cached

- *NO_SCORE*

  functions are compiled 1st time seen,

code is cached,

in case of conflict function may be marked *NOJIT*

- *ALL_SMALL_MAYBE*

  functions with small score are compiled 2nd time seen,

  function with high score are compiled,

  1st time seen if top-level, 2nd time seen otherwise

- *TOP_SMALL_MAYBE*

  functions with small score compiled,

  2nd time seen if top-level, never otherwise

  functions with high score compiled

  time seen if top-level, 2nd time seen otherwise

1.4 GNU R Bytecode

GNU R has internal support of the **BC** which consists of Compiler package for compiling to the bytecode and interpreting function *bcEval* (inside *src/main/eval.c*) which evaluates 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 positive integer value.

1.4.1 GNU R internal representation of bytecode

Internal representation of bytecode is wrapped inside *SEXP* node of *BCODESXP* type. It's represented as an linked list with these parameters (arrays of integers)

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

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

  The array contains char (1 Byte size) representation of version number followed by the bytecode instructions

- **Constant pool** array

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

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. These information are not used for the evaluation, but are provided for specifying the original location of compiled code. 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 length of BC code array - see 1.4.1)

1.5 Current implementation of AST debugger

AST evaluator of **GNU R** is implemented as recursive descent of the **AST** tree. The debugger 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)
- debugonce(fun, text = "", condition = NULL, signature = NULL)
- undebug(fun, signature = NULL)
- isdebugged(fun, signature = NULL)

- debuggingState(on = NULL)

To keep the same debug functionality for functions running on top of the bytecode there is 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 an issues in case of the internal problems (there is error in either **AST** or **BC** evaluator). In that case the code while debugging would be using the different code while not and potentially could have been very confusing. There is also no way to debug **BC** internals (**BC** stack and showing the current evaluating instruction in the **BC**) while running.

The **GNU R** debugger internal implementation of interacting with user is made internally by calling the *browse()* function which is running the environment browser. Its purpose is to wait for user console input and evaluate it. 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 - 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 is very basic and just works the way that converts the code instructions and constant buffer to array and dputs it into the console. It means that the user would see (the function would return) just two arrays with which is definitely not user friendly. The source code consists of 2 short functions *disassemble* and *bcDecode*:

```
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
}
```

1.7 Analysis of disassembler improvements

The current implementation of bytecode disassembler is unsufficient so it needs to be improved. In the following paragraphs there is detailed analysis of another implementations of diassemblers and possibility of implementation advanced one inside **GNU R**.

1.7.1 Java bytecode disassembler

The nice example of disassembler is in Java language (*javap* command of Java package). The java bytecode is although very specific and each file contains the one class. The whole file contains representation of the file just as the bytecode. 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.

Example output of the *javap* command is the:

```
#>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
   #6 = Class                #22
                  // java/lang/Object
   #7 = Utf8                 <init>
   #8 = Utf8                 ()V
   #9 = Utf8                 Code
  #10 = Utf8                 LineNumberTable
  #11 = Utf8                 main
  #12 = Utf8                 ([Ljava/lang/String;)V
  #13 = Utf8                 SourceFile
  #14 = Utf8                 HelloWorld.java
  #15 = NameAndType          #7:#8
                  // "<init>":()V
  #16 = Class                #23
                  // java/lang/System
  #17 = NameAndType          #24:#25
                  // out:Ljava/io/PrintStream;
  #18 = Utf8                 Hello, World
  #19 = Class                #26
                  // java/io/PrintStream
  #20 = NameAndType          #27:#28
                  // println:(Ljava/lang/String;)V
  #21 = Utf8                 HelloWorld
  #22 = Utf8                 java/lang/Object
  #23 = Utf8                 java/lang/System
  #24 = Utf8                 out
  #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:
          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
   // Field java/lang/System.out:Ljava/io/PrintStream;
         3: ldc           #3
                  // String Hello, World
         5: invokevirtual #4
 // Method java/io/PrintStream.println:(Ljava/lang/String;)V
         8: return
      LineNumberTable:
        line 4: 0
        line 5: 8
}
SourceFile: "HelloWorld.java"
```

1.7.2 Python bytecode disassembler

Python has in-built support of disassembler for it's internal **BC**. It is provided inside package *dis* which is part of python (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 any separate array for constant array).

The usage example:

```
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
```

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 dumps 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.

As long as the R can internally combine in memory **AST** and **BC** representation of the code, the way how to disassemble code is just for the one specific function. In case there are a lot of information inside **GNU R** bytecode it would be also nice to have ability to show these information just as parameters through verbosity level. The inline showing values from the constant pool would be also useful because it can result in compact and shorter output with easily enabling the feature for printing just specified are of function.

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. Instead of it we can inspire ourselves with the current AST implementation which is done in the first part of this section. The following parts are analyzing the implementation of the **BC** debugger in other VMs (*Python* VM and *V8* javascript VM).

1.8.1 Inspiration with current AST implementation

The general idea how to run the is taken from the current implementation of the **AST** debugger. The example of **AST** implementation of the debug:

```
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
}
```

Basically this code check if there is *RDEBUG* flag on the current executed function and if there is it would print information about the current evaluated code (source reference if available + evaluated expression). Following command is the *do_browser()* which is internally calling the environment browser

with support for evaluating expression (seeing what is value of which variable + evaluating functions) and inbuilt handling of 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** 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 language and **VM** is the **Python**.

In the following paragraph there is describtion of *cpython* **VM** (**Python** itself is language and not a **VM**) - there are different **VM**s supporting evaluation of the language but the *cpython* is the most common used one. It is supporting just the **BC** interpreter with the quite similar instruction set to the **GNU R**.

The **BC** evaluator in **Python VM** is inside *Python/ceval.c* file. Interpretation of the debugger inside this **VM** is pretty straightforward. There is implemented runtime checking of the in the label *fast_next_opcode*. To speedup this there is 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 particular interpretation means that debugger implementation is still causing performance overhead even while function is not being debugged because for every evaluated **BC** instruction there is at least one value comparison and conditional jump needed for processor compute.

1.8.3 Implementation inside V8 VM

V8 is **Javascript VM** developed by the **Google** initially to be used for **Chrome** browser. By the time it has been used for desktop / server applications through **Node.js** and also **Electron** project.

It's internal implementation is consisting of the **BC** interpreter (*Ignition*) and **JIT** compiler (*TurboFan*).

Bytecode interpreter main file is located inside *src/interpreter/interpreter-generator.cc* file. It is single stack registed based **VM** (similarly to our **GNU R VM** and **cpython VM**). The way how the core functionality of the debugger works on the **BC** level is that **VM** defines separate Debug instruction for every number of the arguments (e.g. *DebugBreak0*, *DebugBreak1*, *DebugBreak2* etc.)

The bytecode definition contains the separate bytecode instructions. If the breakpoint is set on the instruction (for example *ShiftRight* instruction with the 2 parameters) it means it get replaced by the corresponding breakpoint instruction according to number of the parameters (*DebugBreak2* for

ShiftRight). These bytecode instructions works like special instructions which calls the handler for debugger and also dispatch the original instruction to maintain the same behavior of the code.

Bytecode definition for the breakpoint instructions (see file **src/interpreter/bytecodes.h**):

```
  /* 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)
```

Current implementation of breakpoint inside the **V8** *Ignition* **BC** interpreter (see file *src/interpreter/interpreter-generator.cc*):

```
// 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 =
```

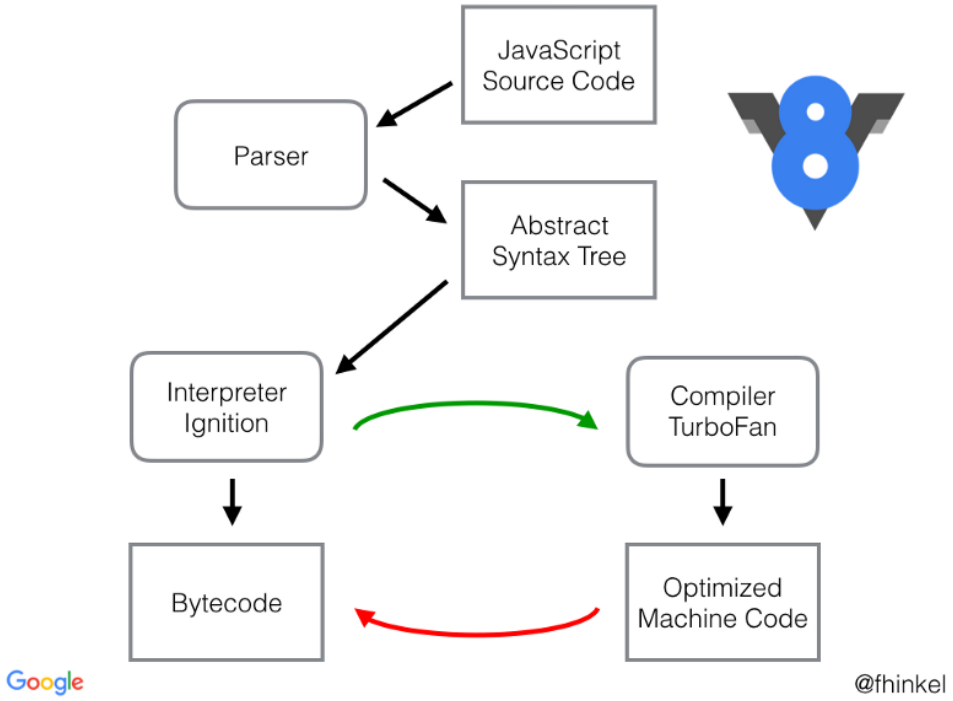

Figure 1.2: V8 internal architecture

```
        SmiUntag(Projection(1, result_pair));
    MaybeDropFrames(context);
    SetAccumulator(return_value);
    DispatchToBytecode(original_bytecode, BytecodeOffset());
  }
DEBUG_BREAK_BYTECODE_LIST(DEBUG_BREAK);
#undef DEBUG_BREAK
```

1.8.4 User interace and state of the BC evaluator

To keep the implementation consistent from the user perspective **BC** debugger should use the 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).

There is also need for the print current state of the **BC** stack machine which consists of the:

- Current position inside code
- Stack content

Alongside the showing the current position there would be also need to show the function **BC** content. For this feature we can use the disassembler feature proposed in the first part of the analysis. Showing current position inside code can be implemented as an feature inside the **BC** disassembler but for the stack content there has to be implemented an separate tool (function).

1.8.5 Summary

In order to improve bytecode debugging there is need for improving (implementing the human-readable) disassembler.

After analyzing the **GNU R** internal implementation of *bcEval* function and the implementation of the **JS V8 VM** and Python VM there is definitely need to mark and put some condition for the marking of the bytecode instruction that the bytecode has been setted on that one. As long as there is focus on performance the bytecode debugger should in the best way put no overhead while it is not debugging. Looking into the internals of the V8 we can reuse the idea of making the separate instructions for bytecode debug flag which would call the neccesary debug routines (as shown in the AST debugger 1.5) and after that call the original instruction to keep functionality of BC (needed to not break program integrity).

Finally there has to be done some showing of the internal status to end user. This consists of the printing the current position in the code and printing the stack content. For the current position in the code there can be reused the disassembler (see 1.7) but for the dumping of the stack content there has to be implemented an separate tool.

Chapter 2

Realisation

There are three different things which has to been done - disasembler, stack printer and finally the **BC** debugger. The BC debugger is using use the stack printer and disassembler so it make sense to implement the disassembler first.

2.1 Implementation of the disassembler

The whole project was structured that disassembler can be easily released (to the **CRAN** repository) without the debugger. This is affecting the whole structuring of the project. The disassembler feature is made to be ready-to-deploy to the **GNU R** beta-testing codebase.

The main idea behind the disassembly rool was to write as much as possible code in the R language and just neccesary minimum in C for keeping the code simple. In the **GNU R** language there is the *compiler* package written in R which contains the current **BC** compiler and **BC** disassembler (very minimal - see 1.6). As long as the contribution to the **GNU R** repository has very strict code styling rules which are much more relaxed for the 3rd party libraries inside **CRAN** repository decision was to make as much as possible inside the separate library which would be then published with some minimal changes to the main repository. Name of the package was decided to be the *bctools*.

2.1.1 User interface

For more user friendliness of the **BC** print function there has been make an decision to use advantage of the **S3** class system (see 1.1.3). The old *disassemble* function inside the *compiler* package was kept almost intact. The only change that has been made there was putting class into the disasembly code. The name of the class has been decided to be the "disassembly". This allows us to have *print.disassembly* function (*print* method of *disassembly* class) inside our package which would be automatically dispatched once user

call the *print* function on the object if the *bctools* package would be loaded inside user library.

The usage then changed from (in the previous version / before the changes made in this thesis):

```
#initialization
library(compiler)
f<-function(a) a+1
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, ADD.OP, 0L,
#   RETURN.OP), list(a + 1, a, 1))
```

To the:

```
#initialization
library(compiler)
library(bctools) #new package
f<-function(a) a+1
compiled <- cmpfun(f)

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


#generated output:
#
#  @ a
#  GETVAR                 a
#  @ 1
#  LDCONST                1
#  @ a + 1
#  ADD
#  RETURN
#
```

2.1.2 Instruction arguments

The **BC** instruction contains the integer code identifying it followed by variable number of arguments. These arguments can be of different types (described in more detail bellow).

The implementation of these operators is expecting arguments with different types (definition is hardcoded inside compiler and evaluator function) but there is currently no list containing list of these 4 types for each instruction:

- **CONSTANT** representing reference (integer index) to the constant pool where is located constant expression (can be either number or function)
- **BOOL** boolean value
- **INT** integer value
- **LABEL** 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

The **CONSTANT** parameter although can have two different meanings in the code. In the **GNU R** bytecode some of the arguments could be just 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 arguments, just in these corner cases it is calling the other internal functions which are originally designed to work on the **AST** evaluator so they expect the expression as an input). Basically it means that these arguments are stored just because the internal implementation and contains the duplicate information. These two meaning should be distinguished inside the disassembler print so there has been put decision to split the **CONSTANT** parameter into the:

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

So the final list of the argument types is:

- **CONSTANT** representing reference (integer index) to the constant pool where is located constant expression (can be either number or function)
- **CONSTANT_DBG** - constant expression inside the argument used internally just for the corner cases (technically containing duplicitous information)
- **BOOL** boolean value
- **INT** integer value
- **LABEL** 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

2.1.3 Annotation of instructions

These 6 types should be printed in different way. There are 2 ways how to implement these behaviors in the bytecode disassembler. The first is to hardcode the behavior for the each instruction. This make sense for the compiler (*compiler* package) and evaluator (mostly *bcEval()* function) because they are basically generating or evaluating the instructions which does not need any anotation at all. On the other hand in our disassembly script only difference between printing these argument types would formatting of output. For this reason it makes a sense to write the code as much generic as possible. The best solution is to just take the definition and prints out the instruction according to its' definition.

In the compiler package there is already some annotation of the **BC** instructions in the form specifying the number of arguments for each instruction. After fully annotating all instruction by specifying each argument type, these values can be computed from the instruction annotation. This definition of the instruction parameters annotation also is more logically connected with the **BF** which generation and support functions are inside *compiler* package than the disassembler (would be just using this feature). Despite the fact that in the purpose of the work there has been idea that as much as possible code should be put inside the 3rd party library (*bctools*) it was decided to put the annotation inside the compiler package. This annotation of the instructions inside the *compiler* package would also allow easier maintainance for the *bctools* package because every time there would be added new instruction its

definition has to be added in the *compiler*. The argument annotation would now be part of the instruction annotation so even the disassembler would support then this instruction (it would be reading the instruction definition from the *compiler* package).

The *compiler* package is made with the *noweb* tool which is the tool to write documentation alongside with the code. Because of the fact that build command (*make*) is not written to re-generate the source code from *noweb* each time the compiling is provided, each time this noweb source is changed there has to be run the *make from-noweb* command inside the compiler package directory (*src/library/compiler*) to regenerate the C sources.

Example of the Instruction argument annotation inside *compiler* package:

```
<<opcode argument description>>=

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

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 ....
)
```

Calculating of the argument count array is done then by:

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

2.1.4 Instruction arguments and labels

Label is form of representation of the reference to the code. It is used in the jumps through the code. Main **BC** array can store just one value itself so if there is an need for few jump locations (e.g. like *GOTO* operator) it is just value of the argument itself (*LABEL* type of argument). The switch operator

could have also multiple locations where to jump so it is internally utilizing an *CONSTANT_LABEL* arguments type.

2.1.5 Computing of labels

To visualize the jumping locations in the diassembler there are usually showed labels in the code to specify target location (usually incrementally numbered from the 1) and arguments are then in forms of references to these locations. These references are printed in the way that is has to be obvious that it is aiming to the specific location. This can be achieved using some specific symbol - e.g. $1) (reference to label with number 1). However there is no real direct information about these locations which should contain labels directly in the bytecode definition (even the x86 code does not have these). Instead of it there are just references in the instructions to the code array, but we can compute the list of labels from these references. For generating this list there was done the two-pass linear lookup through code to generate auxiliary array containing the information direct information of number of the label (the list is expressed as this array).

Steps to generate labels are then:

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

2. Go through all instruction in forward direction. For each argument if it contains any label (see labels inside arguments 2.1.4) label then mark the according instruction on its location inside the auxiliary array.

3. Go through the auxiliary array and set on each position marked as target for some instruction number of label. This number of label is calculated incrementally (in the beginning set counter to 1, and on each marked position set the value of counter to the array and increment the counter)

This array then would contain the information whether there is no label at the instruction or the label unique number.

Code used for computing labels:

```
#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
}
```

2.1.6 Verbosity and formatting

Bytecode compiled function has optionally the information about location of the source. These informations are not neccesary 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 previous chapter 2.1.2). All of these information are not necessary to be displayed for the user for the

basic information but it is nice for them to provide ability to display even these information. To provide this conditional ability to show more basic or more advanced information there has been put an decision to implement more levels of the verbosity in the disassembly tool.

The levels are:

- **0** - display only source references (if they are available, if they aren't print expression references instead)
- **1** - the same as 0 + display bytecode version and display expression references (if they are available)
- **2** - the same as 1 + display every operand's argument (including ones used just for debugging)

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 of more 3 types:

- regular (ordinary) constant expressions (e.g. numbers, array of numbers etc.)
- native functions (calls to 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 and compiled inside the runtime core in C language and the R interpreter knows just it's location (function pointer) to call. These functions are printed as an *<INTERNAL_FUNCTION>*. Alongside of these internal functions there are also stored the **BC** compiled functions which should be also printed out. The way how to do it is just say that this is the function and print out flag like *<BYTECODE_FUNCTION>*. The second approach is to be able to print it out nested with some indentation. The second described approach was chosen. To enable this feature there are 3 parameters in out disassembly tool:

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

With these arguments it is possible to provide functionality disassembly recursiveness. Even though if the maxdepth would be set to the 0, the disassembly tool would print out just the *<BYTECODE_FUNCTION>* for every bytecode information (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 potentially evaluate with *eval* function inside *src/main/eval.c*). These ones has to be printed in some user-friendly way which would not break the consistency of disassembly output. One way of printing the is to use the *dput* function but they would be printed line by line which is not desired output. In order to make 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 single line. There are 2 ways how to solve this:

- Write an specific call into the *print.c* (*src/main/print.c*) - *deparse1* function
- Call an *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 more clean in order of the code but as long as there has been put the decision to make just minimal amount of the changes into the C core of the language to be able to release the disassemble package the second way was chosen.

TODO: maybe add some good shit (code example)

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 dump*NAME* (e.g. *dumpConstant*) convention. The complete list of types to print:

- **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 *compiler* package with the *.OP* (e.g. *ADD.OP*). The knowledge that it is operator is obvious so we do not want to print it. Because of it function for printing operands is extracting the *.OP* suffix and printing just the actual name (e.g. *ADD*).

- **Value**

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

  printing the expression by calling directly the *cat* function

  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 special type of the constant expressions. For printing them we can reuse then the print function for constant ones (the *dput* and eventually even *print* functions has internal support for printing out the expression references). The only difference we need to print the @ as prefix. So final implementation of the print function is to dump @ to the output and then call the function for printing out the constant expression.

2.1.10 Documenting of code

Code itself was written to be self-explainable and easily readable with any developer for good maintainability. On top of that there was also written a lot of comments thorough the code.

From the user-perspectiwe there was created an user-documentation for R the package by the *roxygen* tool (the **GNU R** inbuilt documenting system). There are several ways for developer to rebuild the documentation:

There are three main ways to run *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

In order to implement the **BC** debugger we need to be able to print in user-friendly way the **BC** stack content (see 1.8.4). The way how the implementation was designed was that it was written in the **C** language (**GNU R** language core) a printing function capable of stack of current pointer.

2.2.1 Stack definition

Stack definition depends on the *TYPED_STACK* typedef conditional. If it is definied then the stack optimalization (type of **unboxing**) is enabled so there can be stored even raw values on the stack alongside an boxing **SEXP** type. So then the types which could be stored on the stack are of type:

- **int**
- **double**
- **RAWMEM** - additional piece of raw memory piece of length SEXP-¿ival (in sizeof(SEXP))
- **SEXP** - internal representation of the boxed object storing any value

In case this *TYPED_STACK* is not defined it is just the boxed *SEXP* value.

The definition of the stack element is here:

```
#ifdef TYPED_STACK
typedef struct {
    int tag;
    union {
        int ival;
        double dval;
        SEXP sxpval;
    } u;
} R_bcstack_t;

... other definitions ...
#else
typedef SEXP R_bcstack_t;

... other definitions ...
#endif
```

Luckily there is implemented an macro *GETSTACK_PTR* which would return us an boxed *SEXP* type equivalent no matter if the value in stack is boxed or unboxed. This would allow us to write the stack print function much easier without any need of separate handling of unboxed values.

2.2.2 Printing of the stack values

This functionality is completely written in *C* and it was designed to reuse as much code as possible.

In analyzing printing of the stack values there has been realized that it is not possible to use the *print* (*PrintValue* call in **C**) function because it is dumping output in the very spread out manner (array and object items are printed each at separate line). The better output formatting has the *dput* function, but it is written inside *base* package (written in **R** instead of **C**). After the research what is *dput* doing under hood it was observed that is calling the internal **C** implementation of the *do_dput* (inside *src/main/deparse.c*) which unfortunately cannot be reused without any changes. The reason is that it is internally caling the *deparse1* function with the parameter to internally evaluate the promises. However this printing should not evaluate any promises because they could potentially change the output of the bytecode (they can have side effects). To disable this feature (promise evaluation) we can call the *deparse1* function with first argument setted to *SHOWATTRIBUTES — DELAYPROMISES*. The function would instead of evaluating them just print the <promise> text signalizing that there is promise in that particular element.

2.2.3 RAWMEM stack type tag

Items on the stack are formatted in C-style array but in case the *TYPED_STACK* is defined the one of the types can be *RAWMEM*. In that case the following memory contains the some raw data so we cannot call to these items regular function for printing elements and instead of we have to skip (jump over) that chunk of memory.

The data on the stack would be in our function printed with the first line pointing to the top of the stack. Because of this this reason we want to print the stack from the top to the bottom. The *RAWMEM* type however is pointing to the followed data instructions and not the previous. This means that if we would be printing the stack from the end (top) we would not be able to decide whether the next position contains raw memory or regular stack value.

To be able to detect this we created the auxiliary *LOGICAL* (internal representation of binary inside **GNU R**) array which would contain boolean flag if the value is printable. Tis means we also needed to add one linear pass through the stack to fill in this array. This pass is in the forward order (from bottom to top of the stack) so we are able to tell whether the memory is the raw or not.

This caused that whole algorithm to dump the stack needs to have two passes:

- **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

- **Second pass** from top to bottom to actually print out the values on the stack

  works in the way that skip values for every place where the are auxiliary value is set to FALSE. If TRUE then look to the tag (if *TYPED_STACK* available) if is *RAWMEM*, then print <rawmem of size %d> otherwise call the print function for the value (see 2.2.4).

If there is *TYPED_STACK* not defined it means that the there is no support for *RAWMEM* which would induce that all of the values are printable. In that case the auxiliary array is useless but we still decided to keep this auxiliary array to be able to use as much same code for every case as possible. This would induce one linear lookup of the array which is not neccesary but in term of performance it is just constant slowdown. This function would be used mainly in debugging we are not concerned about the performance that much (the **VM** would be sitting and waiting for user input at some point of the time anyways).

For style formatting we chose to print just simply <rawmem of size %d> text to specify that there is raw memory chunk on the stack.

2.2.4 Persisting stack pointers

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

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

Which satisfies 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 beginning of the whole stack and not to the beginning of current evaluated function). For knowing this information we don't have enough information in these variables so we decided **R_BCNodeStackFnBase** which would point to the beginning (bottom) of the current evaluated function so after that we would have an memory range in within is the current evaluated stack frame. As long as this variable is internally represented as C global

variable but it is used in function context (mimicking of the CPU function local stack frame behavior) we need to add this variable also to the handling the context (in the *src/main/context.c*).

2.3 Conditional breakpoints

Due to lack of the the ability of the conditional breakpoins which are simple and user-friendly there has been put decision to add simple user-friendly *breakpoint()* function which fires the debugger functionality on the spot. It is inspired by the **JS** *debugger;* command, but instead of being an separate command which would be big change in the language parser we decided it to be just an callable function.

This function is simply modifying the *RDEBUG* feature of the current evaluating environment and resetting the temporary bytecode variables (*R_BrowserLastCommand* and *browserfinish* member of function context). This command works like every other function, so it is called with new environment. On the other side we want to change variables of not this environment but the evaluated function, so there has to be done the environment lookup for the parent function context.

```
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';

        ...other support for bytecode
                debugger ( described below )...

    return R_NilValue;
}
```

Usage example:

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

#evaluated function
f<-function(){
```

```
  for(x in 1:5){
    if(x == 3){
          breakpoint();
        }
  }
}
compiled <- cmpfun(f)
compiled()
```

2.4 Implementation of the debugger

The main purpose of this work is enable to do the bytecode debugging. In order to do debugging we need to visualize the current bytecode internal state of the evaluating function (debugger work in the each function separately) which consists of:

- Position inside bytecode
- Bytecode stack content

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

2.4.1 Main idea

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

2.4.2 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 special breakpoint instruction together with backuping (saving) the original instruction alongside the bytecode. This 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.

This feature is still experimental so it is by default disabled. Managing this state is done in *src/main/debug.c* by *R_is_bc_debug_enabled* and *do_enableBCDebug* (exposed into R as *enableBCDebug*).

Example (notice the *enableBCDebug* call):

```
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)
```

2.4.3 Instruction for debugging

To be able to dispatch breakpoints there has been created an specialized set of debug instructions. For dispatching breakpoint on the instruction the original instruction is replaced with it's equivalent (according to the number of arguments, see below) breakpoint one. The debugging instructions are:

- **BREAKPOINT0**
- **BREAKPOINT1**
- **BREAKPOINT2**
- **BREAKPOINT3**
- **BREAKPOINT4**

There has been put the decision to make an instruction for each number of the arguments instead of one generic instruction. All the functions which are working with the debug instructions are then setting the debug on the specific instruction and code for checking the breakpoint (*bcInstrIsBreakpoint* function in the *compiler* package) which would be affected by this decision of creating more separate instruction. On the other hand there are already other parts of the R code (translating bytecode instructions to function pointers and vice versa through *bcEncode*/*bcDecode*, bytecode disassembler etc.) which works with the instruction argument counts inside the current C code. These ones would remain intact and working with any change. This decision would also allow better forward compatibility because this is allowing less code duplication (in the case adding instruction with more than 4 arguments it would mean changing just that two places. Even though if these changes would not be applied in future which should never happen (but we eventually can think even this case) the interpreter **BC** functionality would be intact and would not crash. The only affected feature would be non-working disassembler and debugger (setting the next breakpoint) features. It is very bad and critical but not the worst case scenario of the whole interpreter crashing.

The other possible option (which was not chosen) would have been to have just one debug instruction for whole bytecode set of instructions. This solution would save some possible BC instruction space (for now at least 4 instructions) and allow more simple implementation of the setting/unsetting breakpoint (no need to look up original instruction argument count) but would add more complexity to the all functions working with the **BC** instructions argument count (*bcEncode*/*bcDecode*/disassembler etc.).

2.4.4 Storing of the original instruction when the breakpoint is setted

There is need to somehow remember the original instruction in case the breakpoint is set on the instruction (the original instruction is overwritten by the corresponding breakpoint one but we still need to execute the original one alongside with the debugger features). The function can have more than one instruction replaced with the debug so there cannot be stored just one original instruction for each function but there has to be option ton store one for each position in the BC code array.

The idea behind implemented solution is on the first change (first setting of the breakpoint) make an deep copy of the whole **BC** code array and attach it into the linked list of the bytecode (see internal representation of the bytecode refR-internal-bc-representation). Once this is done we can set or unset the breakpoint on any instruction without worrying of losing any information. The representation of the garbage collector is treating the bytecode as a generic linked list (it's code is inside src/main/memory.c) so the there are any changes needed for keeping the memory consistent.

2.4.5 Setting and unsetting debug instruction

To be able to set (and unset) the debug instruction on the bytecode there has been created an *bcSetBreakpoint* function inside compiler package (but can be very easily moved into another one if neccesary because it is just dependent on the C core function and the instruction annotation which is already exported through *bcinfo* function). It has support for both setting and unsetting the instruction (parameter *is*). This function is internally used in the **BC** debugger which internally needs the informations where has been placed the new breakpoint instruction. For this case it returns the array containing the positions of newly set instructions.

Function implementation:

```
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
}
```

The function is exposed to the end user so it needs to be fail-proof. The **BC** code array does not contain just the instruction operands but also its arguments which cannot be rewritten in any case scenario. The only way how to check whether the value is operand or its instruction is going from the beginning of the code array instruction by instruction. The function would then find the first avaThe way how the function was designed was then it would set or unset (depending on the parameter *is*)

it would firstly find the first instruction which position (index) ¿= parameter *pos*. This feature would prevent user from the corrupting of bytecode array (marking the position of the argument instead of operand). The nice side-effect is that it would allow us to to elegantly solve **next step** feature of debugger. The debugger could have had just take the position of the current evaluating instruction and just add the 1 no matter how many arguments has the current instruction because it would all the time set the following instruction.

As we can see in the code the function is calling internally the *modifybcbreakpoint*. It taking the bytecode function representation (first argument), position (second argument) and flag if the breakpoint should be set or removed (third argument). The function is at first check if there is backup of the original bytecode (to save the potential old representation of the code (see storing of original instructions 2.4.4). If not, it would create an shallow copy of the bytecode representation assigned to the function call (so basically just this specific function call would have an modified breakpoint code). The bytecode array copy would then be appended to this shallow function definition

copy. This way would keep the memory overhead the minimal (shallow copy is small and the only bytecode array is once duplicated and assigned to it's copy). After this is done there could have been provided the setting / unsetting the breakpoint on the array. Setting is done by looking to the definition of the instruction to it's number of arguments and unsetting is done just by copying the value from the second bytecode array (with saved instructions) which contains every time unmodified instructions. As you can notice there is no removing of the auxiliary array in case the last debug instruction was removed. This decision was made because the memory overhead of keeping the auxiliary array is minimal and the removing would add additional complexity to this function.

2.4.6 Listing breakpoints

In order to manage the breakpoint status there has been implemented a *bcListBreakpoints* function for listing the setted ones.

```
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
print(bcListBreakpoints(compiled))
```

2.4.7 Setting the next breakpoint

Consequential instruction does not necessary mean the following instruction right next in the bytecode array due to labels. The breakpoint for the next instruction can be either one of these:

- following in the bytecode array
- at the position which were instruction labels pointing at (see types of labels 2.1.4)

To support this feature there has been developed the*bcSetNextBreakpoint* function inside the *compiler* package. There has been also written internal bindings wrapper inside the **C** package (call to the **R** code) in the function *Rf_breakOnNextBCInst*.

The code is checking all possible locations for the jump locations also with the check if there is already one breakpoint (in that case it is not adding). If there is no breakpoint on the location it is then putting there breakpoint. For placing the breakpoint it is using the internal call to the C function *modify-bcbreakpoint* (see 2.4.5).

It is also internally generating an array of added breakpoint locations which is then returning. This feature is then used in the internal implementation of the breakpoint instruction in the C core (this returned array is used to keep tracking of the added breakpoints - see **??**).

The function code:

```
#simulate next behavior with breakpoints
# returns the vector of added breakpoints
bcSetNextBreakpoint <- function(code, pos){
    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]]
    origbcode <- bc[[3]]
    consts    <- bc[[4]]
```

```
tryAddBreakpoint<-function(pos, ret){
    #if there is no breakpoint at position pos,
    # set breakpoint to that position and add
    # append this position into list ret
    # finally return the list ret
    #  of breakpoint positions
    if(!(pos %in% ret) && pos <= length(bcode)){
        opcode <- bcode[pos]
        if(!bcInstrIsBreakpoint(opcode)){
            .Internal(modifybcbreakpoint(code,
                                         pos-1,
                                         TRUE));
            ret <- append(ret, as.integer(pos-1))
        }
    }
    ret
}

#handle adding breakpoint to the first instruction
#  we need this after entering
#  bcEval in case of RDEBUG == 1
ret <- vector(mode="integer", length=0);
if( pos < 1 ) {
    return(tryAddBreakpoint(2, ret))
}

opcode    <- origbcode[pos]
descr     <- Opcodes.argdescr[[opcode+1]]

#iterate over all parameters of instruction
i <- 1
while( i <= length(descr) ){
    if(descr[[i]] == LABEL.ARGTYPE){
        #if it is label, add breakpoint to target
        # instruction and append its position to
        # return array
        ret <- tryAddBreakpoint(
                            origbcode[i + pos] + 1,
                            ret)
    }else if(descr[[i]] == CONSTANTS_LABEL.ARGTYPE){
        #the label arguments are through
        # constants array
        labelvals <- consts[[ origbcode[i + pos] + 1 ]]
        if(!is.null(labelvals)){
```

```
                for( v in 1:length(labelvals) ){
                    ret <- tryAddBreakpoint(
                                labelvals[[v]]+1,
                                ret)
                }
            }
        }
        i <- i+1
    }
    #if instruction has no labels, add breakpoint to
    # its following instruction
    if(opcode != RETURN.OP &&
        opcode != RETURNJMP.OP &&
        opcode != GOTO.OP) {
        ret <- tryAddBreakpoint(
                pos + 1 + length(descr), ret)
    }

    ret
}
```

bcInstrIsBreakpoint function is used for checking if the instruction code is breakpoint. Internally it is just wrapper over the condition to limit the code duplication (this check is used on more places in the whole code).

```
bcInstrIsBreakpoint <- function(x) {
  x == BREAKPOINT0.OP ||
    x == BREAKPOINT1.OP ||
    x == BREAKPOINT2.OP ||
    x == BREAKPOINT3.OP ||
    x == BREAKPOINT4.OP
}
```

2.4.8 Support in the disassembly tool

To visualize the set breakpoints there has been added an support into the disassembly tool. The R disassembly script was modified to getting an an 3 arrays as an bytecode input (added an field with backup of all bytecode array). So the arrays now contain the constant array, code array possibly containing breakpoints and original code array which never contains any breakpoint instruction. The original array is returned every time no matter if the code array is modified or not (in case not modified there is returned the same 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.

Loading an arrays with breakpoints:

```
#can contain BREAKPOINT[0-9] instructions
code_breakpoint    <- x[[2]]
#never contains BREAKPOINT[0-9] instruction
code               <- x[[3]]
#constant buffer
constants          <- x[[4]]
```

Checking for breakpoint:

```
if(grepl("^BREAKPOINT[0-9]+\\.OP$", code_breakpoint[[i]])){
    instrname <- paste0("(BR) ", instrname)
}
```

2.4.9 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 (by calling *bcSetBreakpoint* function from compiler package). Once bytecode interpreter reaches them it breaks and show the interface for debugger.

The temporary ones are used on the other hand for handling debugger commands. They are implemented with the same instruction except they are also held their locations on the bytecode local stack (variable in which points global *R_BCtmpBreakpoints* also kept through context in *src/main/context.c*). This value contains an array in which each elements is pointing to the location of the breakpoint. These breakpoints always point to the instruction succeeding current the evaluated one (see 2.4.7.

In order to keep the garbage collector satisfied and because it is not possible to store the in the local protection stack (*PROTECT/UNPROTECT* function), there has been decided to dedicate the one field on the bytecode stack for this array. The *R_BCtmpBreakpoints* is then pointer (*SEXP** type) to this location which allows changing of this variable while modifying the bytecode body. Changing this array and not keeping an new one also reflects the fact that the changes inside the bytecode code array are also made in-place in destructive manner.

2.4.10 Implementation of the breakpoint instructions

The reason why there are implemented separate breakpoint instructions for each number of instruction arguments is just because they would have different annotation which would contain the number of arguments. Basically the breakpoint instruction contains the information of the number of its arguments in order to not break the structure of bytecode. The code itself inside all of them would be the same. This means that it can be simply generalized by writing an macro for all breakpoint instruction and calling this macro for each instruction.

The code for instructions ended up like:

```
OP(BREAKPOINT0,  0): DO_BREAKPOINT();
OP(BREAKPOINT1,  1): DO_BREAKPOINT();
OP(BREAKPOINT2,  2): DO_BREAKPOINT();
OP(BREAKPOINT3,  3): DO_BREAKPOINT();
OP(BREAKPOINT4,  4): DO_BREAKPOINT();
```

The breakpoint instructions algorithm (*DO_BREAKPOINT* macro) is:

- Remove all temporary breakpoints from the bytecode
- Call debug browser
- Set the debug flags (RDEBUG or mark next bytecode instruction for debgging)
- Call the original instruction

As you can see in the time of calling the browser there are all temporary breakpoints removed from the bytecode. It means that just the regular user-defined breakpoints would the printed to the output (see 2.4.8) which is desired behavior because we do not want to print the user breakpoints which are used just for implementation of debugger.

2.4.11 Handling of the recursive character of the bytecode

As you can see the debugger implementation while going through the code is modifying the breakpoint code with adding an temporary breakpoins to support breaking on the next instruction (see 2.4.7). In the time of the evaluating instruction there can be set temporary breakpoints somewhere in the code and if this instruction is CALL to the current instruction we have to delete them. In that case the algorithm is to

TODO this shit is fucked up, it is erasing also regular breakpoints

2.4.12 Threaded and non-threaded design of the application

Because of the speedup there is implemented support for the THREADED code which would remove the main loop from the bytecode instruction evaluation loop (see 1.3.4). So basically there has to been two different 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).

2.4.12.0.1 THREADED_CODE defined In this case the changes were minimal. It required just adding an macro **BREAKPOINT_GOTO_ORIGIN_OP** for an switching to the instruction defined in the backup array (*inst* argument). Because of the implementation design of the bytecode instruction in case of the instruction operand it is holding internally position of it's label to jump to (see 1.3.4). All it is doing is then jumping into the location defined in passed instruction value.

```
#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)
```

2.4.12.0.2 THREADED_CODE not defined On the other hand in case of not defined support for *THREADED_CODE* situation is more complicated. In that case the whole evaluator design is one big loop. The loop originally had big switch (beginning defined in *BEGIN_MACHINE* macro) which was deciding which instruction according to the value of operand (*op). We changed this behavior that we loaded operand into variable, placed another jump label (*jmp_opcode*) and after that decided which instruction has to be evaluated according to the value of that variable. In the case of executing BREAKPOINT and evaluating original instruction (through **BREAKPOINT_GOTO_ORIGIN_OP ??**) we set the auxiliary variable (*jmp_opcode*) to the desired instruction code and jumped into jump label which we added (*jmp_opcode*) which would dispatch the another instruction code (without increasing program count pointer etc.).

```
#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)
```

2.4.13 Handling of the debugger user input

The original environment browser has inbuilt support for handling and modifying the debugging (step into, next step, continue etc.) via the user input (defined in *src/main/main.c*) implemented by setting up the *RDEBUG* flag altogether with *R_BrowserLastCommand* variable on the evaluated environment. The intend of implementing bytecode debugger interface was to reuse as much of this feature so the bytecode debugger is managed also through these variables. Bytecode interpreter provides the more status information, so the user interface has been extended by these commands:

- **bc** - print current bytecode with the position (internally calling *R_printCurrentBCbody*)
- **bcstack** - print current bytecode stack (internally calling *R_printCurrentBCstack*)

Implementation in **ParseBrowser**:

```
...
    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
        ...
    }
```

2.4.14 Default behavior of the bytecode debugger

The whole bytecode debugger engine was designed in it's default way for the user to behave in the most similar way as the current AST one. The only difference is printing *"debugBC"* instead of printing *debug* in the beginning of the each line used for user command input (environment browser). Making the debugger output behaving as much similar as the current AST implementation is beneficial for other dependent software which is relying on the specific output format (such as IDEs etc.).

2.4.15 Verbose mode of bytecode debugger

The bytecode debugger on the other hand can jump in much more granular way (not according to the changes of expression references but step for each bytecode instruction) and also have additional state information (bytecode stack). Support for this feature was implemented in this thesis through function *bcVerbose()* called with parameter TRUE or FALSE.

Once the verbose mode is enabled the debugger is stepping through each bytecode instruction. For printing current state there is reused the calling *R_printCurrentBCbody* in *src/main/eval.c* internally dispatching *print* method of bytecode object from the *bctools* package (called with *select* parameter to show current instruction and *peephole* turned to show just surrounding area across the current instruction).

This flag is internally represented through the boolean *R_DebugVerbose* global variable.

To support this feature there has been implemented function *printBCStatus()* for printing the bytecode interpreter status.

Its code is:

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

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

2.4.16 Initial entry points to the bcEval functions

Because the in the bytecode implementation there are no checks for the RDEBUG flag due to performance (avoiding condition instruciton every each evaluated bytecode instruction) there is need for setting the bytecode instruction 2.4.7 in order to dispatch bytecode functionality manually.

Currently there are two cases in which this scenario has to be handled.

- At the beginning - the bytecode function was just called
- At the return from the function call

The first case is handled in the start of the bytecode evaluation *bcEval* function. In this case we just call the *Rf_breakOnNextBCInst* which call internally *bcSetNextBreakpoint* function from the *compiler* package. In the start point of the function evaluation *R_execClosure* (R function implementation is wrapped in closure) is condition if the function has *RDEBUG* flag enabled and there are met criteria to start debugging, the *RDEBUG* flag is set in the current evaluated environment. In the *AST* evaluator there is then checked the environment *RDEBUG* flag. The user interface *debug*, *debugOnce* etc. are changing the function flag (not the environment one) which implies that there cannot be changed the *RDEBUG* by calling debug on the function while executing the function (this implies any instruction so also the any function all from this code). It means that for support the current behavior the only place where is need to add BREAKPOINT instruction is in the beginning of the *bcEval*.

Code in the beginning of the *bcEval* function:

```
if(RDEBUG(rho)){
  ptrdiff_t diff = pc - codebase;
  Rf_breakOnNextBCInst();
  codebase = BCCODE(body);
  constants = BCCONSTS(body);
  pc = codebase + diff;
}
```

The other entry point would be used just in case of the implementation of the conditional breakpoints 2.3. In this case there has to be done the two changes for fully support this feature.

- If the breakpoint 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 2.3).
- The function described above can possibly change the internal structure of bytecode object structure so there has to be done the update of the changed internal variables after call to the function

Marking following instruction as breakpoint:

```
SEXP attribute_hidden do_breakpoint
        (SEXP call, SEXP op, SEXP args, SEXP rho)
{
        ... setting debugging flags ( described above ) ...

    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;
}
```

The **CHECK_ADD_BREAKPOINT** function is used for resetting the internal variables (after the) state in the bcEval function. This reset it done in case of the *DEBUG* flag on the function is enabled (the *do_breakpoint* is turning this flag on). This added function is adding some overhead in case of debugging but in this case we are not worried about the performance. The bigger concern is adding one variable check even if the function is not in *DEBUG* mode.

```
#define CHECK_ADD_BREAKPOINT(pcdiff) do { \
    if(RDEBUG(rho)){ \
      ptrdiff_t diff = *((BCODE**)R_BCpc) - codebase - 1; \
      *((BCODE**)R_BCpc) = BCCODE(R_BCbody) + diff + 1; \
      codebase = BCCODE(R_BCbody); \
      constants = BCCONSTS(R_BCbody); \
      pc = *((BCODE**)R_BCpc) + pcdiff + 1; \
   } \
} while(0)
```

Chapter 3

Testing and future work

The each 3 implemented parts of this thesis were tested separately. The best case scenario was to make set of automated tests which would test out the all of the implemented features.

3.1 Bytecode disassembler

The main part of the work was done as the separate **GNU R** package (named *bctools*). The R package system has it's testing system which runs every file in the *tests* directory and checks its the return code.

The main test parts 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 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 modify the tests to use **test_that** framework which supports naming the tests and printing the name in case of it's fail.

3.2 Bytecode debugger

The **BC** debugger tests was done to make sure that there are no bugs in the code. The testing was done manually altogether with the writing of the some automated tests. Automated testing was done by step by step simulation of the step-into (**"s" command** in the browser) by modifying beginning of the **Rf_ReplIteration** function (found in **src/main/main.c**).

The modified function:

```
int Rf_ReplIteration(
        SEXP rho,
        int savestack,
        int browselevel,
        R_ReplState *state
  )
{

  ... variables initialization ...

  if(!*state->bufp) {
    R_Busy(0);
    if(browselevel){
        /* environment browser is on */
      strcpy((char*)state->buf, "s\n");
    }else{
      if (R_ReadConsole(
          R_PromptString(browselevel, state->prompt_type),
            state->buf,
            CONSOLE_BUFFER_SIZE,
            1)
             == 0)
        return(-1);
    }
    state->bufp = state->buf;
  }

  ... end of the function ...

}
```

Altogether with these changes there was made an script (modification of the test feature from the *R CMD test* command) to utilize the R package test files as testing examples. The tool was comparing the output with and without an bytecode debugger support enabled and trying to simulate the same output as normal. Although this tool was not 100% successful because

internally the tests (tested on the test directory from the *bctools* package) was using catching output with it's internal feature. Running the debugger in this function meant that there was infinite recursion (catching output which causes another catching of the output). This testing can be considered just partially successful, because it was not able to run across any of the input test cases.

Next to this automated testing this feature was tested manually, which results were considered successful.

3.2.1 Effectiveness of testing

During the tests there has been observed numerous issues. Some of them were caused by the oversight during analyzes of the **BC** compiler and/or evaluator. Example of these are:

- handling of the **SWITCH** command arguments - lack of support for the **CONSTANT_LABEL** type - this feature support has to be also done into the disassembler code
- memory issues caused by **Garbage collector** due to forgotten support for the added global variables (such as **R_BCNodeStackFnBase**)
- memory issue caused by modification of the body variable while duplication of the current evaluated one (see 2.4.5) - solved by pushing this variable onto the BC stack

3.2.2 Side effects of the scripts

There are limited options to debug the memory errors inside the language **VM** (and generally any **C** code) and finding these errors are even more difficult when the R code is evaluated - in that case the code is internally represented as *SEXP* type parameter in the *eval* or eventually as an bytecode in the *bcEval*.

To be able to better understand whether some of the memory issues which occurred were caused by the injected R code and not other changes there has been made decision to rewrite some of the parts to not use **R** call to the **compiler** package but use just native **C** implementation. These parts are not having the full functionality as their **GNU R** counterparts and were not fully tested. Although of this they can be useful for future performance improvements so they are included in the attached DVD.

These parts of code are:

- **Rf_breakOnNextBCInst**
- **do_BCdisasm** used in **R_printCurrentBCbody** function

3.3 Conditional breakpoints

Technically conditional breakpoints are in term of the code part of the debugger support so even the testing was done altogether other testing of the bytecode debugger support.

3.4 Future work

3.4.1 Push into working repository

The big part of the planned future part of the future work is to collaborate with the *GNU R* core team to pushing this work into it's 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 the 2013 was approx 2 millions), the work done in this thesis would have big worldwide effect to lot of the people.

3.4.2 Merging the bctools package into the compilers

The **bctools** package was developed independently as an 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** while the source of the bctools one is relatively short (approx. 600 lines). The reason why this was not already done but just proposed here as 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.

3.4.3 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 investigation of these errors there has been noticed one thing. The core of the **GNU R** language is heavy dependent on the linked list memory data structure. During the modern computer implementation of the memory system this can cause a lot of memory cache misses which affect the performance. The most straightforward way to do this is to use an smallvector technique (inspired by an **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 using worldwide with not just scientific community but also non-scientiscs. For better evaluation performance it's VM is internally supporting the bytecode alongside with old-fashioned and slow AST interpreter. Since nowadays options for debugging the bytecode were very limited. The disassembler shows just the plain array of instructions and constant buffer and the debugger is internally dispatching the AST debugger so it is not going through the same code as while not debugging.

In this thesis there has been proposed and implemented changes which contains the user friendly easy human readable bytecode disassembler. The second part is about the implementation of the native debugger support for bytecode evaluator. As the additional fun feature there has been added support for conditional breakpoints in the code in very user-friendly manner inspired by the JavaScript breakpoint command.

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

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.

Bibliography

Appendix A

Acronyms

VM Virtual Machine

BC Bytecode

GC Garbage collector

AST Abstract syntax tree

IR Intermediate representation

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` ..... 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