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

May 9, 2018

Acknowledgements

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

Declaration

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

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

In Prague on May 9, 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 the advanced bytecode disassembler tool for GNU-R language. Implemented tool is returning the information in human readable compact format for passed function. This second feature was reused in the second part of the thesis which is about native support for bytecode debugger. The current debugging options for the bytecode in GNU-R are very limited because of implementing through AST call. This has been improved in this thesis by implementing an native support for debugging bytecode. Presented implementation shows all internal information of the bytecode evaluator status. The solution written in this thesis is also focused on making any negative impact to overall performance of the language which has been successfully tested and proven with testing.

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

Contents

List of Figures

Introduction

Motivation and objectives

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

GNU-R language is dynamically typed interpreted language which usually means that it needs **Virtual Machine** to interpret. There are more ways to 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 into GNU-R in 1998.

Abstract syntax tree evaluator which is the slowest one has in the GNU-R all debugging features already implemented, but since now there was now no way how to easily analyze and debug the bytecode one. The only method provided is just very basic bytecode disassembler which was showing the data in human unfriendly. There is currently also no support for debugging evaluation of bytecode in the GNU-R. Instead of debugging the bytecode it is currently switching into the AST interpreter once user request the debugging features. 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.

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

the options for analysis and debugging any program code running on the bytecode engine of the language. This implementation was written with focus of having any negative in overall performance of the bytecode engine which was proven in the performance testing on the ending of the work. On top of this there was implemented feature for **breakpoint instruction** which can be used for the **simulation of the conditional breakpoints**.

All the implemented features were successfully tested. The disassembler package has written it's own automated tests with an

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.

Chapter 1

Analysis and design

1.1 GNU R from user perspective

1.1.1 Basis usage (main commands and REPL loop)

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

1.1.2 GNU R package system

The **GNU-R** has 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. The bytecode compiler is written in the GNU-R in it's own package (named `compiler`). However this comes already pre-installed so we don't need to manipulate with it in it's own regular way. The bytecode disassembler in this thesis is written as separate package named `bctools` because it is better to keep the language core minimal and the disassembly tool can be implemented as plugin into the language.

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

- `R CMD INSTALL <pkgs>` - install specified packages
- `R CMD build <pkgname>` - build package

- `R CMD check` <`pkgname`> - check package (check requirements, run tests etc.)

1.1.3 GNU R class system

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

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

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

1.2 GNU R internal structure

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

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. The `compiler` package also needs to be loaded due to internal usage. However it's loading mechanism is different because it is hardcoded in the code. It is hardcoded in `loadCompilerNamespace` (in the *src/main/eval.c* file) to be independent on the whole code.

1.2.2 Calling internal C functions from R

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

1.3 Computer Program Compiler Structure

1.3.1 Virtual Machine

Static languages compilers usually uses as the running environment directly the computer operating system. However the **GNU R** is the dynamically typed computer language with the internal Virtual Machine (VM) providing **running environment**. It 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 usually some performance slowdown but on the other hand there is big safety advantage of isolation. The code is much more safer.

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

The **register based** ones 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. However they can profit from the nature of the register-based hardware of the most common processor architectures (ARM, x86 etc.) so they should get better evaluation performance.

The **stack based** ones on the other hand are simpler ones because every instruction arguments 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. This type is one of the most common and also the GNU R Bytecode engine one of it's representative.

The examples of virtual machines are

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

1.3.1.1 Internal parts of the VM

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

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

1.3.2 GNU R memory types and memory management

Each memory node is represented as `SEXP` type. It contains internal representations such as code definition (`LANGSXP`, `BCOSESXP`, `WEAKREFSXP`, promises etc.) and also regular memory types (such as logical vectors, integer vectors, strings vectors etc.). GNU-R is 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

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

1.3.3 GNU R garbage collector

The memory management in dynamic languages is maintained by **Garbage Collector**. It releases allocated memory once it is no longer used. 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. There are is also protection stack managed by `PROTECT` (and `UNPROTECT`) functions which is used inside C code for internal purposes (widely used in 2.3). It is allows to push locally allocated variables into it so they are reachable by the garbage collector and would not get removed during an garbage collection run (memory cleanup).

1.3.4 Computed GOTO

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

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) 1.3.1. Enabling or disabling 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 (`INITIALIZE_MACHINE`, `BEGIN_MACHINE`, `OP`, `NEXT` and `LASTOP`). These macros source code are conditionally defined to either be compiled for supporting Computed GOTO or not (according to the `THREADED_CODE` flag).

1.3.4.1 Computed GOTO not used - Switch dispatch

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

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

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

1.3.4.2 GOTO not used

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.

In the GNU-R implementation of bytecode interpreter there is performance optimization called **direct threaded code** associated with dispatch table and the **computed GOTO**. In process of loading bytecode into the VM internal structure (done by `R_bcDecode` and `R_bcEncode` inside *src/main/eval.c*) there is translation between instruction codes (integer 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 VM 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 code array. It would cause saving one array lookup every step of

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

int pc = 0;
int val = initval;

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

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

BC interpreter compared to the classic interpreter (implemented for example inside **cpython** VM). The drawback of the

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. 1.3). 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). The current AST evaluator debugging implemen-

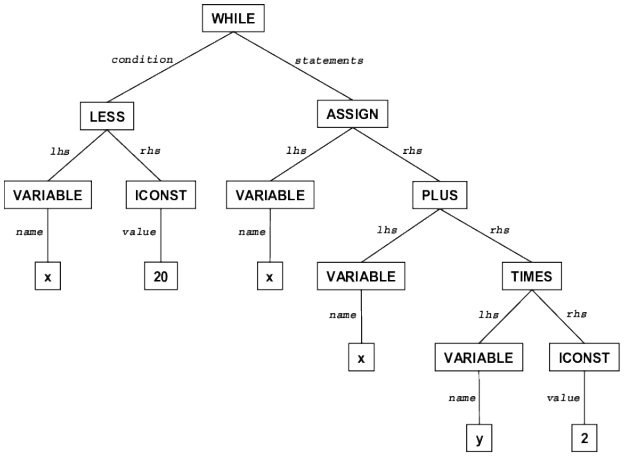

Figure 1.3: While loop example

tation was in this thesis used as an inspiration for the functionality of the newly implemented features. Its behavior is also simulated while bytecode debugger is set into the non-verbose mode 2.3.11.1.

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 byte operation code. Structure contains the instruction code which follows parameters (depending on how many / if parameter given instruction have). This thesis content is about an implementation of support for the bytecode evaluator of an GNU-R language.

1.3.7 Just in Time compilation

In order to speedup evaluation of inside **VM** there has been developed various techniques of the **performance optimization**. One of them is Just in Time (JIT) compilation of code. The underlying idea is to internally translate code into some more efficient representation (from **AST** to either **Bytecode** or to the native machine code). However this transformation (compilation) is usually pretty expensive so it is called once the execution of 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). This means that the bytecode engine of GNU-R

code can be evaluated even without user knowing it (explicitly calling compilation) which would make work for the bytecode debugger in our thesis very important. The second important thing is that the both interpreters (AST and BC one) can be run together so there has to be need to clearly decide which is presently active (implemented in 2.3.11).

1.3.8 Computer language promises

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

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

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

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

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

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

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

This shown approach is manual and pretty straightforward for user to understand. However the GNU-R has promise based lazy evaluation of arguments. Every argument in the function is the promise and instead of evaluating it before the function call (like in other old-fashioned languages like `C` or `Java`), the argument is evaluated inside the function code once it is accessed (see the figure **??**). It causes that the GNU-R is internally heavy dependent on the promises even it is not obvious for the normal user at the first sight.

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

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

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

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

1.4 GNU R Bytecode

GNU-R has internal support of the BC which consists of `compiler` package for compiling to the bytecode. Its 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 `SEXP` node of `BCODESXP` type. It is internally represented as an linked list (CONS of cells) of two variables:

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

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

  accessed in the code through the `BCODE_CODE` macro

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

- **Constant pool** array

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

  accessed in the code through the `BCODE_CONSTS` macro

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

1.4.2 Expression and source references

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

- **Expression reference**

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

- **Source reference**

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

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

- **srcref**

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

- **srcrefsIndex**

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

- **expressionsIndex**

  Array corresponding expression references (expressions) to code for each instruction (length of the array is 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 current GNU-R 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)`

  enables debug features on the function `fun`

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

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

- `undebug(fun, signature = NULL)`

  disable debug features on the function `fun`

- `isdebugged(fun, signature = NULL)`

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

- `debuggingState(on = NULL)`

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

  returns boolean representing whether debugging is globally turned on. The `on` parameter is not NULL this internal state is modified according to that parameter.

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 (stack content and showing the current evaluating instruction in the code) 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) two arrays. This provided functionality is very user unfriendly. The source code of current disassembler consists of 2 short functions `disassemble` and `bcDecode`:

1.7 Analysis of disassembler improvements

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

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

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

Figure 1.7: Current implementation of disassembler in GNU-R

1.7.1 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.

1.7.2 Python bytecode disassembler

Python has in-built support of disassembler for it's internal BC 1.9. 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).

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**.

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. Its implementation of the debug code 1.10 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** 1.1.1 functionality of whole language (implemented by the function `Rf_ReplIteration` or `ParseBuffer` inside *src/main/main.c*).

1.8.2 Implementation inside Python VM

One of the good examples of the similar language and VM is the Python one.

In the following paragraph there is description of *cpython* VM (Python itself is language and not a VM) - there are different VMs 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 implementation of the debugger inside this VM is pretty straightforward. There is implemented runtime checking of the debug flag 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 design of the implementation implicates that debugger implementation is causing performance overhead even while function not being debugged (for every evaluated BC instruction there is at least one value comparison and conditional jump needed for processor compute). However this implementation is easy to implement, does not require any specific debug instruction and does not cause any changes to the memory subsystem (potential GC issues). Also the overhead would in real world usage not be huge due to branch prediction feature in the modern CPUs.

1.8.3 Implementation inside V8 VM

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

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

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

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

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). In the current AST implementation there is used `"debug"` as prefix printed while showing the environment browser in the debugger. This could be modified to the `"debugBC"` to signalize the user that the BC debugger is active.

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

- Current position inside code
- Stack content

Alongside the showing the current position there would be also need to show the function BC source code. For this feature we can use the disassembler feature 2.1 proposed in the first part of the analysis 1.7. Showing current position inside code can be implemented as an feature inside the 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. There was decided that in order to keep better maintainability and more flexibility of the tool it would be implemented as separate `print.disassembly` function inside the new `bctools` package.

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

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

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

.... other contant pool elements ...

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

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

```
import dis

def myfunc(alist):
    return len(alist)

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

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

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

Figure 1.10: GNU-R AST implementation of the debugger

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

Figure 1.11: V8 BC definition for the breakpoint instructions

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

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

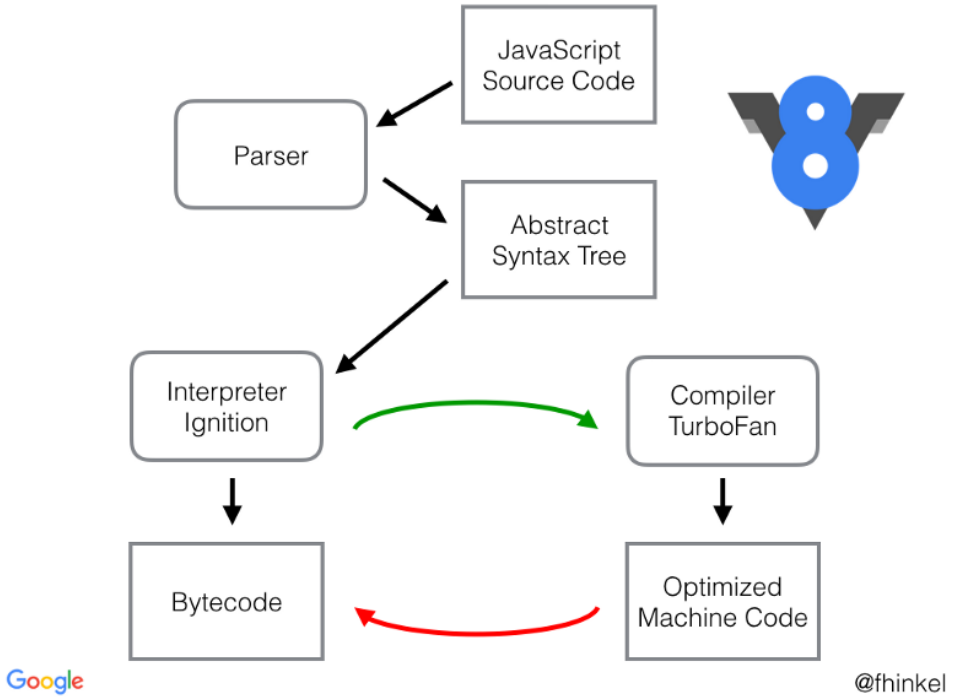

Figure 1.13: V8 internal architecture

Chapter 2

Realization

There are three different things which has to been done - disassembler, stack printer and finally the BC debugger itself.

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. **As much as possible code in the R** language (most of it in separate package `bctools`, but some also to the `compiler` one) and just necessary **minimum in C** for keeping the changes into the GNU-R core code simple.

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

2.1.1 User interface

For better user friendliness of the BC print function there has been made an decision to use advantage of the S3 class system (see 1.1.3). The old `disassemble` function inside the `compiler` package was kept intact except of the **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 `bctools` package. That function would be then 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 simple list (see figure 2.1) to the user friendly disassembly code (see figure 2.2):

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 5 basic types --see2.3 (`BOOL`, `INT`, `LABEL`, `CONSTANT_LABEL` and `CONSTANT`). The printing of the output was

The `CONSTANT` parameter can have two different meanings in the code. 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 some 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). This means that some arguments are stored because the internal implementation of the bytecode evaluator and they contains the duplicate information. The optional parameter kept due to internal purposes of evaluator was named an `CONSTANT_DBG`.

These arguments are printed by the different functions.

2.1.3 Annotation of instructions

These 6 types should be printed in different way according to the annotation. There has been created an definition for each instruction argument (annotation of the instructions). The pretty printing disasembly function (`print.disassembly` method in the `bctools` package) would be then taking instruction definition and printing the arguments according to the definition.

In the compiler package there is already some sort of annotation annotation specifying the number of arguments for each instruction (`Opcodes.argc` list). This was list was replaced by the `Opcodes.descr` list containing the fully annotated instruction was named `Opcodes.argdescr` (see the figure 2.4). To keep the old behavior the old previous `Opcodes.argc` list was computed from `Opcodes.argdescr` by applying the `length` function to each element (see figure 2.5). This solution is not duplicating any information in the `compiler` package.

The `compiler` package is made with the `noweb` tool which is the tool to write documentation alongside with the code. Its source is written in the *src/library/compiler/noweb/compiler.nw* file. 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, we have to rerun the `make from-noweb`

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

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

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

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

Figure 2.1: Old disassembly user interface

command inside the `compiler` to regenerate the R sources for the `compiler` package.

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.

2.1.5 Computing of labels

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

There is no direct list containing locations of the labels, but it can be computed from the code. Generating of this list was done in the two-pass linear lookup through code (see figure 2.6). The result is 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 (`-2`) or the label (number `>=0`).

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)

  see figure 2.7

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

  see figure 2.8

- `2` - the same as 1 + display every operand's argument (including ones used just for debugging)

  see figure 2.9

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 the location (function pointer) to call. These functions are printed as an *<INTERNAL_FUNCTION>*.

There are also stored the **BC** compiled functions which should be also printed. One way is to print out flag like *<BYTECODE_FUNCTION>*. The second approach is to be able to print it out nested with some indentation. The second described approach was chosen. In order to implement this feature there were introduced these 3 parameters in the 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 the `depth` reaches this level, the <`FUNCTION`> instead of calling the disassembly print would be shown in the output)

If the `maxdepth` would be set to the 0, the disassembly tool would print out just the *<BYTECODE_FUNCTION>* for every nested bytecode (the nested BC printing would be disabled).

2.1.8 Printing functions

One type of the constant expressions in the constant array are the functions written as an expression references (non-byte-compiled and being able to potentially evaluate with `eval` function). They have to be printed in user-friendly way which would not break the consistency of disassembly output. One way how to achieves that is to call the `dput` function printing the but it would format them line by line which is not desired output. For making the arguments look as dense as possible (and not break consistency and compactness of the disassemble function) there has been put decision to write the function in 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 because there has been put the decision to make just minimal amount of the changes into the C core of the language (for being able to release the disassembler in separate package) the second way was chosen.

2.1.9 Printing of different types

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

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

The `bctools` package user-documentation was created with the `roxygen` tool (the GNU-R inbuilt documenting system). There are several ways for developer to rebuild the documentation (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 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 (see figure 2.10). Once it is defined, the option for saved unboxed values enabled. It causes that instead of the `SEXP` stored in the stack every time (which is more expensive to handle), there could be also stored raw values (`int` / `double`). There is also `RAWMEM` memory which is used in the BC evaluator to store for example evaluation context frame (but generally speaking this data type can be out of any other data type). Once the stack is enables, the stack values could be then out of these types:

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

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

To be able to not have to write specific code for each stack type there is implemented an macro `GETSTACK_PTR` returning an boxed `SEXP` type equivalent of stack position (no matter if the value in stack is boxed or not).

2.2.2 Printing of the stack values

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

2.2.3 RAWMEM stack type tag

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

array. This pass is in the forward order (from bottom to top of the stack) so we are able to tell whether the memory is the raw or not.

The whole algorithm to dump the stack is then having 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 2.2.2).

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

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

all of these satisfying a equation:
$R\_BCNodeStackBase <= R\_BCNodeStackTop <= R\_BCNodeStackEnd$

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

2.3 Implementation of the debugger

The 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:

- Bytecode
- Position inside bytecode
- Bytecode stack content

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

2.3.1 Main idea

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

2.3.2 Global design

The idea used behind the debugger implementation is **inspired by the JS V8 VM** (see 1.8.3). It is to replace the original instruction with 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 by default disabled. Managing the state for enabling it is done by *enableBCDebug* function (written in *src/main/debug.c*) which is internally handling an `R_is_bc_debug_enabled` variable. See the figure 2.11 for example.

2.3.3 Instruction for debugging

To be able to dispatch breakpoints there has been created 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) 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.).

2.3.4 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.

2.3.5 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 2.3.4). 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.

2.3.6 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))
```

2.3.7 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 setting there breakpoint instructions. For placing the them there it is using the internal call to the C function *modifybcbreakpoint* (see 2.3.5).

It is also internally generating an array of added breakpoint locations which is then returning. This feature is 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).

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).

Its source code is pretty simple:

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

2.3.8 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 3 arrays as an bytecode input (added 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)
}
```

2.3.9 Temporary and regular breakpoints

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

- temporary breakpoint

- regular breakpoint

The regular ones are used for user-defined breakpoint (by calling *bcSetBreakpoint* function from compiler package). Once bytecode interpreter reaches them the breakpoint functionality is called and shows 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 where each elements is pointing to the location of the breakpoint. These breakpoints always point to the instruction succeeding current the evaluated one (see 2.3.7).

In order to keep the garbage collector satisfied and because it is not possible to store the variable in the local protection stack (through *PROTECT/UNPROTECT* function) during *bcEval*, there has been dedicated one field on the bytecode stack for storing this array. The *R_BCtmpBreakpoints* is then pointer (*SEXP**) 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 modifications inside the **BC** code array are also made in-place in destructive manner.

2.3.10 Implementation of the breakpoint instructions

The reason why there are implemented separate breakpoint instructions for each number of instruction arguments is its annotation. It allows us that instructions would have different annotation containing the number of arguments. Basically the breakpoint instruction would then contain the information of the number of its arguments in order to not break the structure of bytecode. However the evaluated code inside all of them would be the same. This means that it can be simply generalized by writing single macro for all breakpoint instructions.

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
- Print bytecode interpreter internal status (described below)

- Call debug browser
- Set debug flags (RDEBUG or mark next bytecode instruction for debugging)
- 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 2.3.8). It is desired behavior because we do not want to print the user breakpoints which are used just for internal purposes of step-by-step feature of debugger.

2.3.11 Bytecode interpreter internal status

For user there is useful to know whether the interpreter is in the BC or in the AST mode. In order to achieve this there was put as `debugBC` prefix instead of `debug` in the AST evaluator.

The second thing to think about is the ability to locate the currently evaluated code (position in code). For achieving this there has to be printed the internal status of interpreter. Which was implemented in two possible ways:

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

2.3.11.1 The short compact way of status printing

It is used to simulate the AST printing behavior. It printing the data in the same manner to remain the support for the debuggers in other dependent IDEs to keep the backward compatibility. This way is used by default.

2.3.11.2 The long verbose way showing all information

It is used for the printing the whole internal state bytecode interpreter. 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 );
}
```

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).

To decide control whether to print out the short compact way or the long way is used the *R_DebugVerbose* boolean variable. This flag is accessible for user through the *debugVerbose()* function which acts like getter and setter altogether. It is returning the value of the variable as return value. It has also optional parameter which is used for the setting the variable.

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

2.3.12 Debugger jumping granularity

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

- calculate these in bcSetNextBreakpoint
- runtime checking

The first way is more proper in order of performance because there would be placed breakpoint instruction right to the place where should be debugger functionality dispatched. The second one would works by placing the breakpoint instruction to the suceeding one while skipping the debugger functionality unless the change in expression reference occur (runtime checking). The second one was chosen due to simplicity of this solution because it means just handling and storing one runtime variable of expression reference and checking if it changes. The performance overhead due to runtime checking would be just in case debugger is enabled which is not think we are worried in the first place.

On the other hand when the long verbose mode is set up. The debugger would jump directly to the following instruction which would be then visualized for user. It means that there is no need for putting any condition for skipping debug functionality.

2.3.13 Handling of the recursive character of the bytecode

The debugger implementation is modifying the breakpoint code while going through the code with adding an temporary breakpoints. This is done in order to support breaking on the next instruction (see 2.3.7). However it means that while calling recursive call there can be already set breakpoint instruction in the evaluated code. However we want to have any executed in the recursive call of the function. To solve this there was put an decision to check in the beginning of *bcEval* function whether the code is modified. In that case it is creating an shallow copy of the current code without an modified code array (taking just an code array and constant array). This would mean that we are sure that after this there are no breakpoints set in the function.

Because of this is allocating an new element it has to have reference to some object for satisfying the **GC**. This was done by pushing to the bytecode stack. It was decided to push the current evaluated body even when this change is done due to less if-then clauses and better code readability (the value has to be also popped from the stack at the ending of evaluating).

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

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

2.3.14 Threaded and non-threaded design of the application

Because of the speedup of the **GNU R** evaluating there is support for the THREADED code. When it is enabled it causes that there is no the main loop in the bytecode instruction evaluation (see 1.3.4).

So basically this means there were 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).

2.3.14.1 THREADED_CODE defined

It means that the needed changes were minimal. It required adding a 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)
```

2.3.14.2 THREADED_CODE not defined

It means on the other hand that 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 instruction 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)
```

2.3.15 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 inside **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
...
```

2.3.16 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.).

2.3.17 Initial entry points to the bcEval functions

Because in the during bytecode debugging inside **bcEval** function there are no runtime checks for the RDEBUG flag due to performance (avoiding condition instruciton every each evaluated bytecode instruction) there is need for setting the bytecode instruction 2.3.7 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 2.4. 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 2.4).
- 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)
```

2.4 Simulated conditional breakpoints

Due to lack of the the ability of the simple and user-friendly conditional breakpoints there has been put decision to add support for simulating them. It is done through `breakpoint()` function which fires the debugger functionality on the spot. It allows the user to simulate conditional breakpoint behavior by calling the environment browser as in the breakpoint instruction (if you'd call the environment browser manually through `browse()` the handling of debugger through *next step*, *continue* etc. is not provided). The `breakpoint()` (see fig. 2.13) function is inspired by the JS `debugger;` command, but instead of being an separate command which would be big change in the language parser it was decided to make it a callable function.

As you can see in the figure 2.12 this function is modifying the `RDEBUG` flag of the currently evaluating environment and resetting the temporary bytecode variables (`R_BrowserLastCommand` and `browserfinish` member of function context).

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

        x
}
compiled <- cmpfun(f)

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

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

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

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

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

Figure 2.3: Bytecode instruction argument types

```
<<opcode argument description>>=

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

Opcodes.argdescr <- list(

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

Figure 2.4: Example of instruction annotation

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

Figure 2.5: Computation of argument count in the compiler package

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

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


Figure 2.6: Code for generating labels

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

Figure 2.7: Disassembly output with verbose lvl 0

```
Bytecode ver. 10

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

Figure 2.8: Disassembly output with verbose lvl 1

```
Bytecode ver. 10

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

Figure 2.9: Disassembly output with verbose lvl 2

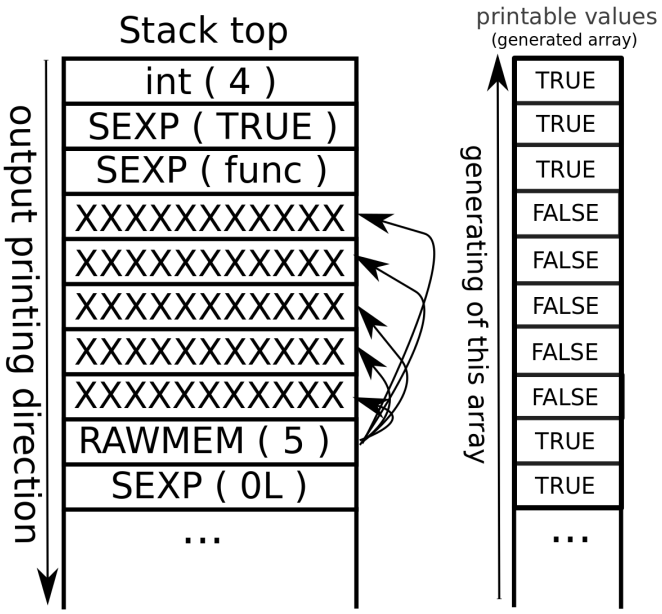

Figure 2.10: Definition of stack elements and generated auxiliary array showing the pritable elements

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

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

Figure 2.11: Example of debugged function with bytecode debugger enabled

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


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

  while bytecode debugger disabled"));

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

  return R_NilValue;
}
```

Figure 2.12: Implementation of simulated conditional breakpoints through breakpoint() function

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

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

Figure 2.13: Example usage of simulated conditional breakpoints through breakpoint() function

Chapter 3

Testing and future work

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

3.1 Bytecode disassembler

The main part of the work was done as the separate **GNU R** package (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 2.3.5) - 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 **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

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

3.4 Performance testing

TODO: performance testing

3.5 Future work

3.5.1 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.

3.5.2 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.

3.5.3 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

JS JavaScript

OO Object oriented

OOP Object oriented programming

JIT Just in time

REPL Read-Eval-Print-Loop

UI User interface

ARM Advanced RISC Machine, originally Acorn RISC Machine

CRAN The Comprehensive R Archive Network

GCC GNU Compiler Collection

TDD Test driven development

Appendix B

Contents of enclosed DVD

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