Assignment of master's thesis

Title:	Out of process byte-code copiler for the R programming language
Student:	Bc. Adam Plodek
Supervisor:	doc. Ing. Filip Křikava, Ph.D.
Study program:	Informatics
Branch / specialization:	System Programming
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2024/2025

Instructions

R is a popular dynamic programming language for statistics and data science. Since 2.13, R code can be compiled into a byte code, which is then interpreted by the R virtual machine. All of the standard functions, as well as all installed packages in R, are pre-compiled into byte-code. The compiler is written in R as an R package but is tightly integrated with the R virtual machine. This thesis should explore an alternative approach in which the compiler is entirely separate from the virtual machine.
Not running in R could open a range of benefits, such as faster compilation, parallel compilation, and better maintainability.

Analyze the current byte-code compiler and familiarize yourself with the byte-code instructions and byte-code interpreter. Design a compiler separated from the R virtual machine and only communicates with it over a clear interface. Prototype and experiment with the compiler.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 23 January 2024 in Prague.

Master's thesis

OUT OF PROCESS BYTE-CODE COPILER FOR THE R PROGRAMMING LANGUAGE

Bc. Adam Plodek

Faculty of Information Technology
Katedra teoretické informatiky
Supervisor: doc. Ing. Filip Křikava, Ph.D.
May 9, 2024

Citation of this thesis: Plodek Adam. *Out of process byte-code copiler for the R programming language.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2024.

Contents

List of Tables

List of code listings

I would like to thank friends, family, and people who have encouraged me throughout the last year. To my roommate for words of occasional encouragement and less rare banter. To all my schoolmates who shared this experience with me, and most importantly to my adviser Filip, who was always ready for very useful and enjoyable consultation.

Declaration

In Praze on May 9, 2024

Abstract

R is a dynamic programming language used mainly in statistics and data visualization. Its unique set of features and extensive ecosystem of packages enables statisticians to write software without the need to be software engineers. The GNU R implementation of an interpreter for R programming language is considered primary implementation. To speed up the execution of the R programs, the bytecode interpreter was implemented next to the standard AST interpreter. To compile the AST representation of the program into its bytecode representation, the compiler for GNU R bytecode was introduced.

This thesis explores one possibility of improvement for this compilation process, namely the out-of-process compilation. This approach allows the implementation of the compilers in different languages and could unlock more possibilities for sharing the compiled code between clients. Moreover, the compiler process can be located outside of the machine on which the R interpreter is running, which can be used to move the compilation overhead to a more powerful machine.

I describe the process of creating the experimental implementation of such a solution done in Rust programming language, which can serve as a baseline for future work. To achieve this, the custom representation of R values and serialization of those values was created. This was then used to implement the compiler and server, which communicates with the package that can be used by the interpreter.

Finally, I evaluate the current state of implementation of my compilation server. This is split into two parts: correctness and performance. Both of these criteria are compared against the current implementation embedded in the GNU R interpreter. The result of this evalutation showed that the compilation process could be sped up 20 times in best case scenario, when only compilation it self is counted. When the loading of the data is included the speed up ended up being 3 times compared to GNU R implementation.

Keywords R programming language, GNU R, compilation, JIT, server, RDS, bytecode, interpreter

Abstrakt

R je dynamický programovaní jazyk, který je převážně používaný ve statistice a pro vizualizaci dat. Jeho netypické vlastnosti a bohatý ekosystém balíčků umožňuje statistikům psát software bez pokročilých programátorských znalostí. Hlavní implementace toho programovancího jazyka je GNU R. Pro zrychlení běhu R programů, byla vytvořena GNU R implemetace bytekódu interpretru, který se používá souběžně s AST interpretrem. Součástí tohoto rozšíření byl kompilátor pro GNU R bytekód.

Tato práce se zabývá jednou z možností pro vylepšení tohoto procesu, a to kompilace mimo proces interpretru. Tento přístup umožňuje implemetaci v jiných jazycích a otevírá nové možnosti pro sdílení kompilovaného kódu. Dále by toto řešení umožnilo kompilátor přesunou ze zařízení, na kterém je spuštěn interpreter, což by umožnilo přesunout náročný výpočet na výkonější zařízení.

Dále popisuji průběh vývoje experimentální implemetace v programovacím jazyce Rust tohoto řešení, která může sloužit jakožto počáteční bod pro budoucí práci. Pro tyto účely byla vytvořena nová reprezentace hodnot v programovacím jazyce R a serializační formát pro tyto hodnoty. Toto bylo následně využito k implementaci samotného kompilátoru a serveru, který je schopný komunikovat s interpretem pomocí balíčku pro programovací jazyk R.

Na závěr se zabývám zhodnocením aktuálního stavu kompilačního serveru. Tato kapitola je rozdělena do dvou částí a to na část, která se zabývá korektností implementace, a na část která hodnotí její výkon. V těchto kritériích je moje impletace porovnána s implemetací, která je součástí GNU R intepretu. Výsledky těchto testů ukázaly, že v nejlepším případě je možné zrychlit proces kompilace až 20 krát, pokud je pouze čas na kompilaci počítán, a když bylo do měření přidáno i načítání data, tak zrychlení bylo trojnásobné.

Klíčová slova R programovací jazyk, GNU R, kompilace, JIT, server, RDS, bytekód, intepret

List of abbreviations

AST	Abstract syntax tree
CRAN	The Comprehensive R Archive Network
GC	Garbage collection
GNU	GNU’s Not Unix
TCP	Transmission Control Protocol
SEXP	S-expression

Introduction

The R is a programming language used in statistical computing and data visualization. The language is known for its flexibility, extensibility, and rich ecosystem of libraries written either in R itself or some lower-level language such as C or C++. The GNU R implementation of R programming language interprets R programs in two forms, either as an AST representation or as a bytecode compiled code. The bytecode representation was added to speed up the notoriously slow execution of R programs. The compiler for GNU R bytecode is currently implemented mainly in R itself, with parts written in C.

The compilation outside of the process of the interpreter can enable implementation of the compiler in different languages other than those used in the interpreter and the possibility to speed the compilation by using one faster machine for compilation to compile code from multiple clients, which would free up the resources of client interpreters. These techniques were used to implement the JIT compilation of programming languages such as Java.

The goal of this thesis is to implement the proof of concept of similar solution for a compilation of GNU R bytecode, on which it would be possible to build up the full solution and experiment with possible improvements.

At the start of the thesis I introduce the R programming language, following that I examine the implementation of all relevant parts of the GNU R implementation of interpreter, this include the compiler itself but also the parts of the interpreter that are relevant to implementation. Next I explore the similar solutions for different languages and what possible improvements the compilation done out of the process of the interpreter.

In the second chapter I discuss the design and implementation of compilation server. The implementation was done mainly in Rust programming language and to reach the goals of the thesis it had to reimplement multiple parts of GNU R interpreter. These include:

The representation of the internal values of R programming languages, also known as SEXP, is the serialization format in which the values and expressions could be sent between client and server. For this purpose, the RDS serialization format was implemented. This serialization format is a proprietary serialization format implemented for R values.

The compiler itself had to be reimplemented in the server language, and the parts of the data that are stored in the interpreter and necessary for compilation had to be identified and queried by the interpreter before compilation on the server could proceed. The original compiler implements a few optimizations, which would have to be implemented to achieve full feature parity with the current implementation. However, at the current point, only a subset of this optimization has been implemented.

In the end, communication between the client and server had to be implemented. To achieve this, the basic TCP server was implemented, and to enable a client to communicate with the server, the package for R programming language was implemented that handles the querying, serialization, and deserialization of the data that are requested and needed for compilation.

The last chapter explains the process that was used to assess the properties of the implemented compiler. The implementation was measured in terms of correctness and performance. To achieve this, I used the GNU R as baseline implementation. The Rust solution contains a subset of the logic provided by the reference compiler since the goal of the thesis was to create a proof of concept that would create a framework for future work. However, when compared to the GNU R compiler, my implementation can compiler 45% of functions from the base environment identically and is able to do so more efficiently.

Chapter 1

Background

In this chapter, I discuss prerequisite knowledge for the implementation of the out-of-process compilation. The first section contains a basic introduction to R programming language, its usage, and properties, which become important in the thesis going forward.

The second section discusses the current implementation of GNU R interpreter, which is considered the main implementation of the R programming language, and the implementation of the interpreter of bytecode, which is the target of the implemented compilation server. The relevant parts of the interpreter for this thesis are the internal representation of the values known as SEXP, the RDS serialization format, and the compiler. However, the compiler is discussed in a separate section this section contains description of the representation of the bytecode, basic structure of the compilator and optimizations that are implemented in compiler.

The last section explores the implementation of compilation servers, most importantly what are the challenges compared to implemention within the interpreter and what are the possible gains from the implementing the compiler out of the process.

1.1 The R programming language

R is a dynamic programming language that is used mainly in statistics, which is part of the GNU project. Its origins could be traced to the S programming language, which was created by Bell Labs in the 1980s. Another big part of the R is its development environment and its package ecosystem, which are either built-in or hosted on CRAN. The package could be written using the R programming language itself, or if the task it is performing is performance-sensitive, C, C++, or Fortran could be used.[1] [2]

The R programming language has an unusual set of features that have been implemented mostly with statisticians' needs in mind. The main data structure that R operates on is a vector, which represents ordered homogeneous data. You can operate on these values via vector arithmetic that allows users to use normal arithmetic, such as:

```r
c(1, 2) + 1
```

The function is a first-class object in R, and in fact, many of the constructs that would be in different languages implemented as core parts of the language are in R implemented as normal function calls. To this set of functions belongs, for example, the `if`, which is implemented as a primitive in the interpreter. However, since the R interpreter handles it as a normal function, it allows its reassignment to a different definition. This would be achieved by just simple reassignment such as:

```r
`if` <- function(...) print("if")
```

This complicates the implementation of the compiler since the AST interpreter handles all of these situations dynamically, but if the compiler would like to create a more optimized version of the if statement, the implementation must consider all of these possibilities, and I will discuss what decisions were made in GNU R implementation to overcome these problems.

Other than the environments, the objects in R follow the value semantics with the copy-on-write mechanism. Most of the function arguments are lazily evaluated. Exceptions are built-in functions that are written in C or other lower-level language.

1.2 GNUR implementation

The GNU R is not only the implementation of interpreter of R programming language, however it is considers as default implementation. Moreover for purposes of this thesis this is the implementation that uses GNU R bytecode which is the target bytecode of the implemented compiler.

The implementation of the interpreter is done in C and R. The C is used for the interpreter itself and build-ins, and the R is used to implement libraries, most importantly for our purposes, the compiler library. [3]

Other then compiler itself the relevant parts of GNU R, to implement goals of this thesis, are representation of the R values in the interpreter and RDS serialization format. The representation is called SEXP and must have been reimplemented within my implementation, reasoning for this is discussed at start of the chapter 2. The RDS serialization was chosen as a format that is used for communication between client and server, again reasoning for this is discussed in further chapters.

1.2.1 SEXP

SEXP is the way that the GNUR represents the R data and expressions. SEXP is a pointer to a structure SEXPREC. The SEXPREC contains a header and union of possible SEXP types. The header contains metadata such as the type of the SEXP and has attributes or tag flag and flag that are set for all objects that contain class as one of their attributes. Most of this information is stored in `sxpinfo_struct` structure, which you can see in 1.2. Additional metadata, not stored in `sxpinfo_struct`, are attributes and previous and next node in same generation of GC, all of these values are represented as pointers to SEXPREC. You can see the whole structure in code snippet 1.1. [4]

There are 27 types of SEXP, of which 25 refer to actual value type, and the rest, such as FUNSXP, are used as wildcards for matching some class of the SEXPs. The type of the SEXP can be found via the typeof function in R. [4]

The data in SEXP are, in most cases, represented as a triplet of pointers to SEXPREC, as you can see in example 1.3, which is used to represent list expression and language expressions. The exception to this rule is `primsxp_struct` and `vecsxp_struct`. Structure `primsxp_struct` is used to represent specials and build-ins to achieve this only offset to table internally stored in R interpreter. The vector expression is a bit out of the ordinary compared to other SEXP values since their structure only shares a header with SEXPREC and is otherwise different. The rest of the vector expression contains length and true length, which are usually the same value but may differ in some circumstances, and aligned data itself. You can see all the structures that are used to achieve this in code 1.4. [4]

To create and manipulate SEXPs, the interpreter implements many macros and functions that let you use these structures without needing to delve into representation itself. These constructs include CAR and CDR for manipulating lists or CHAR and PRINTNAME, which are used to get the value of the symbol from symbol expression. To inspect the structure of the SEXP you can use internal function `inspect` by calling:

```
.Internal(inspect(sexp))
```

The most important SEXP types for compilers are those representing language, closure, list, symbol, environment, and bytecode. The list is represented by three SEXPREC pointers to the head (`car`) of the list, tag, and tail (`cdr`) of the list. The language does not have a special structure and is represented as a list in whose head is the function that should be called, and the rest of the list are arguments to this function. The tag can be used to represent named arguments by storing the name of the argument, which is stored in the car of the list. This representation creates a tree structure for expressions. For example, you can see the representation in code example 1.5, which corresponds to expression [4]

```
f(1 + a)
```

The closure is represented as a triplet of pointers to SEXPREC, and you can see this structure in code reference 1.6. The formals represent the names of the arguments of the function, with an optional default value. This value is stored as a list. The body field stores the body of the function. This can represented in multiple ways, most importantly by language expression or bytecode. The env field stores the environment in which the closure should be called. [4]

The environments form a tree-like structure of parents and children. This structure has its roots in an empty environment. Moreover, there are multiple special environments, such as the base environment, that are maintained as singleton in runtime. There are two types of environments: list environments and hash environments. The environment can be only one type, and the other must be set to the null value. The list environment is represented as a list in which the tags are names of the variables that exist in this environment, and the car stores the value of the variable. The hash environment is represented as a generic vector with a fixed size, which contains the lists with names and values of the variables, similar to the list environment. The index into the vector position is calculated with the hash of the name of the variable. [4]

The bytecode is stored similarly to the language expression without any dedicated structure and reuses the listsxp structure. The car in the list stores the bytecode instructions, which are represented as integer vectors, and the cdr stores the constant pool of bytecode. The constants contain not only the values that are used by bytecode instructions but also the expression and source location for each bytecode instruction. [4]

1.2.2 RDS serialization

R has builtin way to serialize any SEXP into ASCII or binary format, which could even be compressed. For our purposes, we will cover only uncompressed binary format since that is a format that will be used in the implementation.

There are multiple functions implemented in the interpreter that handle serialization and deserialization. Most important for the rest of the theses are saveRDS, readRDS, and serialize. Function saveRDS and readRDS are used to write and load R objects into the file. Another argument of saveRDS that is important for us is compress, which is a boolean argument that must be set to FALSE to force uncompressed serialization. [5]

As an example of the serialized data, you can see in code 1.7 the serialization of function:

```
function(x) x + 1
```

The data starts with the header, which contains the type of the format (0x58 in this case), format version, writer version, and minimal reader version. If the format version has value 3, then the format that is used for representation of strings is outputted. In our case, since the format version is 3, the end of the header contains:

```
05 55 54 46 2d 38
```

Code listing 1.1 Representation of list

```
#define SEXPREC_HEADER \
    struct sxpinfo_struct sxpinfo; \
    struct SEXPREC *attrib; \
    struct SEXPREC *gengc_next_node, *gengc_prev_node


typedef struct SEXPREC {
    SEXPREC_HEADER;
    union {
        struct primsxp_struct primsxp;
        struct symsxp_struct symsxp;
        struct listsxp_struct listsxp;
        struct envsxp_struct envsxp;
        struct closxp_struct closxp;
        struct promsxp_struct promsxp;
    } u;
} SEXPREC;
```

Code listing 1.2 Structure of sxpinfo_struct

```
struct sxpinfo_struct {
    SEXPTYPE type      :  TYPE_BITS;
                            /* ==> (FUNSXP == 99) %% 2^5 == 3 == CLOSXP
                             * -> warning: `type' is narrower than values
                             *              of its type
                             * when SEXPTYPE was an enum */
    unsigned int scalar:  1;
    unsigned int obj   :  1;
    unsigned int alt   :  1;
    unsigned int gp    : 16;
    unsigned int mark  :  1;
    unsigned int debug :  1;
    unsigned int trace :  1;  /* functions and memory tracing */
    unsigned int spare :  1;  /* used on closures and when REFCNT is defined */
    unsigned int gcgen :  1;  /* old generation number */
    unsigned int gccls :  3;  /* node class */
    unsigned int named : NAMED_BITS;
    unsigned int extra : 32 - NAMED_BITS; /* used for immediate bindings */
}; /*                    Tot: 64 */
```

Code listing 1.3 Representation of list

```
struct listsxp_struct {
    struct SEXPREC *carval;
    struct SEXPREC *cdrval;
    struct SEXPREC *tagval;
};
```

Code listing 1.4 Representation of vector

```
struct vecsxp_struct {
    R_xlen_t        length;
    R_xlen_t        truelength;
};

...

typedef struct VECTOR_SEXPREC {
    SEXPREC_HEADER;
    struct vecsxp_struct vecsxp;
} VECTOR_SEXPREC, *VECSEXP;
```

Code listing 1.5 Representation of the language expression

```
@587566950a68 06 LANGSXP g0c0 [REF(1)]
  @587563975db8 01 SYMSXP g0c0 [MARK,REF(177)] "f"
  @587566950a30 06 LANGSXP g0c0 [REF(1)]
    @58756380ec10 01 SYMSXP g0c0 [MARK,REF(76),LCK,gp=0x5000] "+" (has value)
    @587566947cc8 14 REALSXP g0c1 [REF(2)] (len=1, tl=0) 1
    @587563c14230 01 SYMSXP g0c0 [MARK,REF(23)] "a"
```

Code listing 1.6 Representation of closure

```
struct closxp_struct {
    struct SEXPREC *formals;
    struct SEXPREC *body;
    struct SEXPREC *env;
};
```

The 05 is the length of the string end, and the rest is the hexadecimal values of the string ”UTF-8”.

Following the header, the data itself continues. Every value starts with a flag, which is 4 bytes in size and contains the type of the value and metadata, such as if the value has the attribute. You can see the implementation of the serialization flag in code snippet 1.8 after the flag follows data that are part of the value if needed.

As an example, we can look at what follows after header in 1.7. The first value is closure, which type has value 3. The other metadata that its flag contains is the bit that signals that this closure has a tag, which is always set in serialized closure value. The data continues with environment in which this closure was created, formals which represent the arguments for the function and are represented as a list. As you can see in 1.3, the list SEXP is represented as a head (car) and tail (cdr) with the possibility of a tag, the same is true in RDS. In the case of formals in our example, the head is the possible initial value of the variable, and the tag is the name of the variable.

At the end, the body of the function itself is serialized, which contains language expression. This is represented similarly to the list expression, so much so that it even shares the `listsxp_struct` with the list expression. The value of language expression in our example starts with symbol expression, which represents the name of the function, namely ”+”, followed by a list of arguments, which is closed with the null value.

To shrink the final size of the serialized data, the RDS format implements interning for symbols, environments, namespaces, and external and weak pointers into the references. In some cases, they are even necessary to be able to serialize the cyclic data structures. These references are used implicitly in the resulting data, meaning that the reference is created when the value that has to be treated as a reference is encountered for the first time, is added to the reference pool, and is outputted in its normal form. The reference is represented with its own type identifier, with the index in the reference pool, which is either stored within the flag field of value or as an int following the flag, depending on the value of the index.

Moreover, the bytecode serialization implements a separate version of the references used in the constant pool serialization. These kinds of references are done only for repeating values and must be explicitly stated in the serialized data with a flag that represents the definition value followed by the serialized value itself and a tag that represents the reference of the value. The set of the values the are represented with usage of these references must be selected before the start serialization of the bytecode value.

1.3 Compiler implementation

The implementation of the current bytecode compiler is part of the standard library of the R programming language and is written in the R language itself, with parts of the implementation done by C functions which provide performance sensitive operations. By default, the interpretation of the R program is done in an AST interpreter, but for performance purposes, the function, library, or file can be compiled into bytecode either ahead-of-time (AOT) or just-in-time (JIT). The JIT compilation is enabled by default and is done on the function and package level. [6]

1.3.1 GNU R bytecode

The runtime for bytecode interpretation is a stack-based virtual machine. Virtual machine (VM) and bytecode is designed to be able to reuse as much of the AST interpreter as possible. The bytecode is represented via SEXP with type BCODESXP, which contains the instruction buffer and constant pool. The instruction buffer is represented by an integer vector, which starts with the bytecode version (currently version 12) and continues with instruction, indexes into a constant pool, and jumps values for the branch instruction. The constant pool is represented

Code listing 1.7 RDS serialization of the function(x) x + 1

```
// header
58 0a 00 00 00 03 00 04 03 03 00 03 05 00 00 00 00 05 55 54 46 2d 38

// closure tag
00 00 04 03
// environment (fd represents global environment)
00 00 00 fd
// formals
00 00 04 02 00 00 00 01 00 04 00 09 00 00 00 01 78 00 00 00 fb 00 00 00 fe

// body (x + 1)
00 00 00 06 00 00 00 01 00 04 00 09 00 00 00 01 2b // 2b - ascii code for +
00 00 00 02 00 00 01 ff // reference to symbol "x"
00 00 00 02 00 00 00 0e 00 00 00 01 3f f0 00 00 // real vector - c(1)
00 00 00 00 00 00 00 fe // null
```

Code listing 1.8 Code for packing flags into flag integer

```c
static int PackFlags(int type, int levs, int isobj, int hasattr, int hastag)
{
    /* We don't write out bit 5 as from R 2.8.0.
       It is used to indicate if an object is in CHARSXP cache
       - not that it matters to this version of R, but it saves
       checking all previous versions.

       Also make sure the HASHASH bit is not written out.
    */
    int val;
    if (type == CHARSXP) levs &= (~(CACHED_MASK | HASHASH_MASK));
    val = type | ENCODE_LEVELS(levs);
    if (isobj) val |= IS_OBJECT_BIT_MASK;
    if (hasattr) val |= HAS_ATTR_BIT_MASK;
    if (hastag) val |= HAS_TAG_BIT_MASK;
    return val;
}
```

Code listing 1.9 Disassembly of compiled function

```
str(compiler::disassemble(compiler::cmpfun(function(x) x + 1)))

list(.Code, list(12L, GETVAR.OP, 1L, LDCONST.OP, 3L, ADD.OP,
    0L, RETURN.OP), list(x + 1, x, structure(c(1L, 44L, 1L, 60L,
44L, 60L, 1L, 1L), srcfile = <environment>, class = "srcref"),
    1, structure(c(NA, 1L, 1L, 3L, 3L, 0L, 0L, 0L), class = "expressionsIndex"),
    structure(c(NA, 2L, 2L, 2L, 2L, 2L, 2L, 2L), class = "srcrefsIndex")))
List of 3
 $ : symbol .Code
 $ :List of 8
  ..$ : int 12
  ..$ : symbol GETVAR.OP
  ..$ : int 1
  ..$ : symbol LDCONST.OP
  ..$ : int 3
  ..$ : symbol ADD.OP
  ..$ : int 0
  ..$ : symbol RETURN.OP
 $ :List of 6
  ..$ : language x + 1
  ..$ : symbol x
  ..$ : 'srcref' int [1:8] 1 44 1 60 44 60 1 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x60a5317aa9b0>
  ..$ : num 1
  ..$ : 'expressionsIndex' int [1:8] NA 1 1 3 3 0 0 0
  ..$ : 'srcrefsIndex' int [1:8] NA 2 2 2 2 2 2 2
  ..$
```

via a list of the SEXPs and contains the constant, symbols, expression locations, and source locations. You can see an example of a compiled function in code snippet 1.9, which is displayed with the disassemble function provided by the compiler package.[6]

1.3.2 Compiler structure

The compiler implements the interface for compiling functions, expressions, and files, namely cmpfun, compile, and cmpfile. The implementation of the compiler is done via recursive code generation that traverses the expression that is being compiled. [6]

The compiled instruction, indexes into the constant pool, and constants are stored in the code buffer, which is a structure that contains functions such as putcode of putconst that are used to manipulate the code buffer. Additionally, the code buffer contains the current expression and source location, if available. Moreover, the context contains functions that are used to signal warnings and errors to the user and the current compiler environment. [6]

Next to the code buffer, the compiler uses the current context in which the compilation is done. The context contains information about current compiler settings, most importantly whether the expression is in the tail position and if the expression must be compiled in loop context. For creating a different context, the compiler package provides a number of functions that either create a whole new context or create a new context from the current context. These function are: [6]

- `make.toplevelContext`
- `make.callContext`
- `make.nonTailCallContext`
- `make.nonTailCallContext`
- `make.noValueContext`
- `make.functionContext`
- `make.loopContext`
- `make.argContext`
- `make.promiseContext`

1.3.3 Basic compilation

Any R expression can be compiled with a small subset of bytecode instructions. When using this subset, the bytecode interpreter mimics the flow of the AST interpreter very closely. The set of these instructions contains instructions for loading constant, pushing argument for function calls, creating promise, querying function from the environment, calling the function, and returning the function. This kind of compilation is done when the inlining level is set to 0.

As explained above, the constructs that would be generally seen as a part of the language, such as the `if` statement, are implemented in R as a function. This allows for the compilation of without any special bytecode instructions for logic or branching. This is true for all other kinds of language constructs. This creates significant overhead, and the next subsection explores optimizations that are used to improve these cases.

1.3.4 Optimizations

The compiler implements the main way to optimize the bytecode, namely inlining and constant folding. The constant folding is done only for the selected set of functions, and inlining is done depending on the inlining level for all of the functions in the base environment. However, there are some functions that are considered syntactically special or part of the language. You can see these selected functions in code example 1.11 in which the foldFuns variable is for constant folding, and the languageFuns variable is for functions that are considered syntactically special or part of the language. There are four levels of inlining, which increasingly broaden the number of functions that could be inlined and loosen the check of correctness. The default level of inlining is currently set to level 2. These levels are:[6]

- Level 0 : No inlining
- Level 1 : Function found from base packages found through namespace that are not shadowed and add base guard
- Level 2 : Level 1 + base package function found via a global environment that is not shadowed during compilation and the base guard is omitted for syntactically special or considered part of the core language.
- Level 3 : Any function from base packages found via the global environment can be inlined

We can see the comparison of the compiled function with inlining in code example 1.9 and without inlining in 1.10. We can see that without inlining, the compiled code must find a function for addition and then call this function. On the other hand, the inlined version can use the addition instruction. These optimizations are implemented in the tryInline function in the compiler package, which tries to use one of the inlining handlers and return a boolean informing the compiler about the result.

There is many handlers for different kinds of function that occur in R programs. For function like `if` there are special handlers that are capable of compiling these function without any additional call. However for generic function from base environment that are builtins or specials the inlining only uses different kind of call function and slightly different way of argument compilation. You can see example of custom inline handler in code example 1.12

You can notice that if the inlined function does not need the base guard the behaviour that was expected during the compilation persist even when the function is override during the execution. This is different compared to AST interpreter which would change the behaviour to new function, this is solved by addition of the base guard.

The base guard instruction is emitted if needed, and its arguments is the expression that runtime needs to check if the function that should be called is the same as in the base package and the position where to resume bytecode interpretation if the interpretation must be handed over to AST interpreter.

The constant folding is done very similarly. There are a small amount of functions that can be constant fold. When the compiler detects that the call can be constant folded and the function is coming from the base package, then the expression is computed at compile time by calling the function. This process can be done multiple times on the result of the constant folded expression. For example, expression

```
1 + 2 * 3
```

is optimized to only load constant instruction with the value of 7. The constant folding is done on all levels of the optimizations. [6]

1.4 Compilation server

To achieve the speedup of JIT compilation the time that is spend compiling the original intepreted bytecode or AST as in case of the GNU R must be smaller that the time that would be gained in execution of the resulting compiled code. This is tradeof the is constant problem that is at heart of the all intepreters that use this technique. The compilation done outside of the compiler is one of the ways to mitigate the cost of the compilation by moving the computation into the server that could have more resources than the client machine.

This however adds cost of the comunication, this include not only the code that would be requested for compilation but also the data necessary for compilation such as environment in which the compilation is done and the information about the context gather during the execution that is used during the compilation to implement optimizations.

Despite these challenges the there are alot of posibilities to optimize compilation process. For example to run a compilation for multiple clients on one server that has more computational power or caching results of the compilation and reusing it in different clients.

Code listing 1.10 Compiled bytecode of closure function(x) x + 1 without inlining

```
str(compiler::disassemble(compiler::cmpfun(function(x) x + 1, options=list(optimize=0))))

list(.Code, list(12L, GETFUN.OP, 1L, MAKEPROM.OP, 3L, PUSHCONSTARG.OP,
    4L, CALL.OP, 0L, RETURN.OP), list(x + 1, `+`, structure(c(1L,
44L, 1L, 60L, 44L, 60L, 1L, 1L), srcfile = <environment>, class = "srcref"),
    list(.Code, list(12L, GETVAR.OP, 0L, RETURN.OP), list(x,
        x + 1, structure(c(1L, 44L, 1L, 60L, 44L, 60L, 1L, 1L
        ), srcfile = <environment>, class = "srcref"), structure(c(NA,
        1L, 1L, 1L), class = "expressionsIndex"), structure(c(NA,
        2L, 2L, 2L), class = "srcrefsIndex"))), 1, structure(c(NA,
    0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L, 0L), class = "expressionsIndex"),
    structure(c(NA, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L, 2L), class = "srcrefsIndex")))
List of 3
 $ : symbol .Code
 $ :List of 10
  ..$ : int 12
  ..$ : symbol GETFUN.OP
  ..$ : int 1
  ..$ : symbol MAKEPROM.OP
  ..$ : int 3
  ..$ : symbol PUSHCONSTARG.OP
  ..$ : int 4
  ..$ : symbol CALL.OP
  ..$ : int 0
  ..$ : symbol RETURN.OP
 $ :List of 7
  ..$ : language x + 1
  ..$ : symbol +
  ..$ : 'srcref' int [1:8] 1 44 1 60 44 60 1 1
  .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x5c2a091d92b0>
  ..$ :List of 3
  .. ..$ : symbol .Code
  .. ..$ :List of 4
  .. .. ..$ : int 12
  .. .. ..$ : symbol GETVAR.OP
  .. .. ..$ : int 0
  .. .. ..$ : symbol RETURN.OP
  .. ..$ :List of 5
  .. .. ..$ : symbol x
  .. .. ..$ : language x + 1
  .. .. ..$ : 'srcref' int [1:8] 1 44 1 60 44 60 1 1
  .. .. .. ..- attr(*, "srcfile")=Classes 'srcfilecopy', 'srcfile' <environment: 0x5c2a091d92b0>
  .. .. ..$ : 'expressionsIndex' int [1:4] NA 1 1 1
  .. .. ..$ : 'srcrefsIndex' int [1:4] NA 2 2 2
  ..$ : num 1
  ..$ : 'expressionsIndex' int [1:10] NA 0 0 0 0 0 0 0 0 0
  ..$ : 'srcrefsIndex' int [1:10] NA 2 2 2 2 2 2 2 2 2
```

Code listing 1.11 Syntactically special function or consider part of the language

```
foldFuns <- c("+", "-", "*", "/", "^", "(",
              ">", ">=", "==", "!=", "<", "<=", "||", "&&", "!",
              "|", "&", "%%",
              "c", "rep", ":",
              "abs", "acos", "acosh", "asin", "asinh", "atan", "atan2",
              "atanh", "ceiling", "choose", "cos", "cosh", "exp", "expm1",
              "floor", "gamma", "lbeta", "lchoose", "lgamma", "log", "log10",
              "log1p", "log2", "max", "min", "prod", "range", "round",
              "seq_along", "seq.int", "seq_len", "sign", "signif",
              "sin", "sinh", "sqrt", "sum", "tan", "tanh", "trunc",
              "baseenv", "emptyenv", "globalenv",
              "Arg", "Conj", "Im", "Mod", "Re",
              "is.R")

languageFuns <- c("^", "~", "<", "<<-", "<=", "<-", "=", "==", ">", ">=",
                  "|", "||", "-", ":", "!", "!=", "/", "(", "[", "[<-", "[[",
                  "[[<-", "{", "@", "$", "$<-", "*", "&", "&&", "%/%", "%*%",
                  "%%", "+",
                  "::", ":::", "@<-",
                  "break", "for", "function", "if", "next", "repeat", "while",
                  "local", "return", "switch")
```

Code listing 1.12 Example of custom inline handler

```
setInlineHandler("function", function(e, cb, cntxt) {
    forms <- e[[2]]
    body <- e[[3]]
    sref <- if (length(e) > 3) e[[4]] else NULL
    ncntxt <- make.functionContext(cntxt, forms, body)
    if (mayCallBrowser(body, cntxt))
        return(FALSE)
    cbody <- genCode(body, ncntxt, loc = cb$savecurloc())
    ci <- cb$putconst(list(forms, cbody, sref))
    cb$putcode(MAKECLOSURE.OP, ci)
    if (cntxt$tailcall) cb$putcode(RETURN.OP)
    TRUE
})
```

Chapter 2

Design and implementation

The aim of the implementation was to create an experimental implementation of a compilation server for GNU R byte code. The original implementation of the compiler has all of the necessary information for compilation that is reachable from the interpreter itself. However, the server, by definition, must run separately from the interpreter. Because of this constraint, the way of communication between the interpreter (client) and compiler (server) must have been implemented.

For the implementation of the server and compiler, I decided to use the Rust programming language. This language was chosen because of my prior familiarity with the language, good support for algebraic data types, and low runtime overhead. To implement the client, I created a package for R programming language. This was done with a combination of R and C programming languages.

There have been multiple options as to how the basic structure of the implementation could be done. Firstly, the Rust library libR_sys implements the Rust binding for the GNU R interpreter. However, for this to work, the Rust compiler would be dependent on R binary. Moreover, the representation of the R values that is provided by this library strictly follows the representation used in GNU R implementation since the library provides only bindings. For this reason, I decided against the use of this library.

The option I chose was to implement all the necessary parts without the need to rely on the implementation of the GNU R. To achieve this, the custom representation of the SEXP, RDS serialization, and deserialization, compiler with an implemented subset of inlining, the server which is able to communicate with the client.

This approach requires more work at the beginning of the implementation since there is more prerequisite functionality before the implementation of the compiler and communication can be started. However, the advantage of this approach is the isolation of the compiler from the GNU R interpreter and better representation of the R values, which helps when with the implementation of the logic in the compiler itself. I discuss advantages and disadvantages of each part of the implementation in the following chapters.

2.1 Value representation

The original SEXP implementation representation is very permissive when it comes to what kind of data can be set to many fields. The example of this is could be seen in hash environments, the implementation is done via vector of lists but in the data structure for representing environments the hash environment field is represented with pointer to SEXPREC. This enables to have any data into position where only either vector or nil value should be.

■ **Code listing 2.1** Enum representing SEXP types

```rust
// SXP
#[derive(Debug, PartialEq, Clone)]
pub enum SexpKind {
    Sym(lang::Sym),
    List(data::List),
    Nil,

    // language contructs
    Closure(lang::Closure),
    Environment(lang::Environment),
    Promise {
        environment: lang::Environment,
        expr: Box<Sexp>,
        value: Box<Sexp>,
    },
    Lang(lang::Lang),
    Bc(Bc),
    Buildin(lang::Sym),

    // vecs
    Char(Vec<char>),
    NAString,
    Logic(Vec<data::Logic>),
    Real(Vec<f64>),
    Int(Vec<i32>),
    Complex(Vec<data::Complex>),
    Str(Vec<String>),
    Vec(Vec<Sexp>),

    MissingArg,

    BaseNamespace, // as in GnuR fake namespace
}
```

I wanted to reduce these possibilities to in my implementation to be able to take advantage of Rusts compile time checks. You can see the part of the implementation that represents all the possible types of SEXPs in code example 2.1. As you can see the data are more structured. When we look on the environment and more specifically the hash environment implementation in code example 2.2, we can see that compared to GNU R implementation in the hash environment there has to be vector of the SEXP which is more restrictive.

2.2 Data serialization

To communicate between server and client, the data must be serialized. For this purpose, you can use multiple possibilities. First, let's look at the data that we need to send between endpoints. The first and most important part of the request is the code that is to be compiled. Second, the environment in which the code is currently running, in the end, the three sets of the string that inform the compiler what function can be considered part of some distinct set, such as builtins.

■ **Code listing 2.2** Hash environment implementation

```
#[derive(Debug, PartialEq, Clone)]
pub struct NormalEnv {
    pub parent: Box<Environment>,
    pub locked: bool,
    pub frame: ListFrame,
    pub hash_frame: HashFrame,
}

...

#[derive(PartialEq, Clone, Default)]
pub struct HashFrame {
    pub data: Option<Vec<super::Sexp>>,
    pub env: HashMap<String, (usize, usize)>,
}
```

I discuss why these data are needed for compilation both in sections 1.3 and 2.3.

Any format that would be used in communication has to be able to serialize all of these data types. The code could be easily represented in its textual form. This would require parsing the R code on the server, which could prove more complicated than necessary. The same approach would be feasible for sets of functions since the only information that is needed is the names of the functions. However, this simple format fails when it comes to the serializing environment since it contains general data. Another possibility was to create a specialized format that would match all of my criteria; I ultimately decided against it because this would require the implementation of serialization and deserialization on both client and server.

In the end, I settled on the RDS serialization format since the GNU R interpreter that plays the role of the client already implements this serialization format. This let me save some time that would have to be spent implementing different ways of serialization. Moreover, any property of the RDS format that could be seen as nonoptimal is the server interacts with this serialization format only at the start and end of the request, and in between, the custom representation of the R values is used.

The reader is implemented as a trait that extends the Reader trait. This way, any Reader can implement RDSReader without any modifications. For example, you can implement RDSReader on File by writing.

```
impl RDSReader for File {}
```

The writer is implemented the same way, except that the extended trait is Writer. The error handling in both of these traits is done with the Result type with a special error type, which represents either RDSReader or RDSWriter errors. You can see these types in code example 2.3.

The implementation of the RDS reader and writer follows the C implementation, which is part of the GNU R interpreter. However, there are parts of serializations that could not be implemented the same way. These are caused by the more rigid SEXP type in Rust implementation. In the original implementation, there are two main functions that handle the parsing of the RDS format ReadItem_Recursive and ReadItem_Iterative, the first of which can be followed quite closely. However, the iterative version implements reading for all items whose type follows the shape of the list. So this means that one function implements reading LISTSXP, LANGSXP, CLOSXP, PROMSXP, and DOTSXP. Since these types are represented differently from each other in my implementation, their reading is implemented differently from each other.

Code listing 2.3 RDSReaderError and RDSWriterError types

```
#[derive(Debug)]
pub enum RDSReaderError {
    DataError(String),
    WrongFlag(i32),
    IO(std::io::Error),
}

...

#[derive(Debug)]
pub enum RDSReaderError {
    DataError(String),
    WrongFlag(i32),
    IO(std::io::Error),
}
```

Moreover, when reading RDS data, my implementation of SEXP requires transformation and checks of a few data types, such as environments or formals in closure value. These values must be read in a way that is not only good for handling in Rust but also able to be written out in the same way as it came in, which is essential for testing. The most interesting example is in environments that are represented as hash maps. These, however, do not maintain the same structures or order as in the original data that were read from RDS. For this reason, we must not only store the hash map but also the original value, as can be seen in code example 2.2.

2.3 Compiler

The compiler implements basic compilation and inlining that follows the rules of inlining on the second level in the GNU R implementation. So far, only a subset of inline handlers has been implemented. The implementation follows the original implementation as closely as possible. The main structure of the compiler remains the same. You can see the structure that is used for implementation in code example 2.4. You can see that this structure contains additional information that the original compiler used runtime itself to get, and since my implementation is outside of the interpreter process, this information must be passed to the compiler with the code that is requested for compilation. These values include base environment, specials, builtins, and builtin internal values.

As in the original implementation, the code buffer handles the addition of the instruction, constants, and labels. You can see this data structure of the code buffer in code example 2.5. The instructions buffer itself represents the instructions as a vector of i32. This allows for straightforward serialization and deserialization since this is the way that the original implementation represents bytecode instructions.

To add data into a code buffer, it implements an interface, which can be seen in code reference 2.6. All of the interaction with outputted code is done via this interface. Other than handling instructions and constants, the code buffer handles labels. In the resulting bytecode, the labels are represented as an integer that has the value of the absolute position of the instruction to which the label points. In the original implementation of the compiler, the labels are represented during the compilation as strings, which contain the name of the label and store the map from the name of the label to the position in the bytecode. These labels are then patched to match the correct values at the end of the compilation. Since the bytecode instructions are represented even during the compilations as a vector of i32, the labels cannot be represented as a string. To overcome this, the information about the labels is stored entirely out of the bytecode instruction buffer, and dummy labels are inserted throughout the compilations. These dummy labels are patched at the end of the compilation using stored data.

The context is handled in the same way as in the original implementation. At the current point, the context contains less information than in the GNU R implementation. You can see structures that represent the context in the code reference 2.7. Similarly, the helper functions that act as the constructors for different kinds of contexts are implemented. During the compilation, the context is used to guide compilation and store information that is necessary for compilation in the future.

The entry point for compilation is the cmpfun method, which can be seen in code reference 2.8. The function sets up the environment for compilation collects local variables with find_local method and call gen_code method which creates the code buffer itself stars the compilation. After the compilation is finished the body of closure, that has been passed into the method, by resulting bytecode and this closure is returned.

Same as the original implementation root part of the compilation is implemented in cmp method which contains the match expression that call other methods that implements compilation of individual expressions. As you can see the structure of the compiler follows as closely as possible the original source code written in R.

Without inlining, the compilation uses only the instructions that load the constant from the constant pool, push arguments into the stack, query the environment for the function by its name, set tags, and create the promise. The inlining is triggered when compiling the language expression and only when the inlining is allowed. The disallowing of the inlining is triggered if the inlining fails and must be called from inline handling to prevent the cyclic calling of inlining. Rules for inlining follow level 2 of inlining from the original implementation. First, the information about the function to be inlined is figured out in the method get_inlineinfo. You can see this method in code reference 2.9. This function, depending on the inline level and if the function can be found via base environment, either returns None or inline information containing if the function call requires base guard instruction. The base guard is omitted for a set of chosen defined in global constant LANG_FUNCS. Right now, the base guard is omitted even when the inline level is set to 3 or more, however, this option has not been tested so far. Depending on the result of the get_inline info the inlining either ends unsuccessfully and basic function call compilation is done, or the compilation continues with or without the base guard instruction inserted.

The inlining itself is handled in method handle_inline, this function returns boolean that signals if the inlining succeeded. The handlers that implement inlining for different kind of function are selected with match expression. Most of the handlers can be selected my simple string match however there are few inline handlers that require more complex check. Basic example of this is that there are two different handlers for the addition (function with the name `+`) for unary plus operator and binary plus operator. There are currently 35 special inline handlers out of which 9 are handlers for `is` functions, these functions are:
`if, {, <-, +, -, *, /, ^, exp, sqrt, while, break, function, [[, .Internal, ==, !=,`
`<, <=, >=, >, &, |, !, &&, ||`
`is.character, is.complex, is.double, is.integer, is.logical, is.name, is.null, is.object,`
`is.symbol`

Moreover, over the inline handlers for builtins, specials, and math functions are implemented. The builtins and specials are dependent on the environment. To determine if the function is builtin or special, the compiler stores a hash set of the string of names for both of these sets of functions. Math functions are hardcoded into the compiler the same as in the original compiler in global constant MATH1_FUNCS, and these functions are:
`floor, ceiling, sign, expm1, log1p, cos, sin, tan, acos, asin, atan, cosh, sinh,`
`tanh, acosh, asinh, atanh, lgamma, gamma, digamma, trigamma, cospi, sinpi, tanpi`

Code listing 2.4 Compiler structure

```
pub struct Compiler {
    options: CompilerOptions,
    context: CompilerContext,
    code_buffer: CodeBuffer,

    pub warnings: Vec<Warning>,

    env: lang::Environment,
    localenv: HashSet<String>,
    baseenv: Option<lang::NormalEnv>,
    namespacebase: Option<lang::NormalEnv>,

    pub specials: HashSet<String>,
    pub builtins: HashSet<String>,
    pub internals: HashSet<String>,
}
```

Code listing 2.5 Code buffer structure

```
pub struct CodeBuffer {
    pub bc: Bc,
    pub current_expr: Option<Sexp>,
    pub expression_buffer: Vec<i32>,
    labels: Vec<Label>,
}
```

Code listing 2.6 Code buffer interface

```
impl CodeBuffer {
    ...

    pub fn insert_currexpr(&mut self, bc_count: usize) { ... }

    pub fn add_instr(&mut self, op: BcOp) { ... }

    pub fn add_instr2(&mut self, op: BcOp, idx: i32) { ... }

    pub fn add_instr_n(&mut self, op: BcOp, idxs: &[i32]) { ... }

    pub fn add_const(&mut self, val: Sexp) -> i32 { ... }

    pub fn set_current_expr(&mut self, sexp: Sexp) -> Option<Sexp> { ... }

    pub fn restore_current_expr(&mut self, orig: Option<Sexp>) { ... }

    pub fn make_label(&mut self) -> LabelIdx { ... }

    pub fn set_label(&mut self, label: LabelIdx) { ... }

    pub fn put_label(&mut self, label: LabelIdx) { ... }

    pub fn patch_labels(&mut self) { ... }
}
```

Code listing 2.7 Code buffer interface

```
#[derive(Default, Clone)]
pub struct LoopContext {
    pub loop_label: LabelIdx,
    pub end_label: LabelIdx,
    pub goto_ok: bool,
}

#[derive(Default, Clone)]
pub struct CompilerContext {
    pub top_level: bool,
    pub need_returnjmp: bool,
    pub tailcall: bool,
    pub loop_ctx: Option<LoopContext>,
    pub call: Option<Sexp>,
}
```

Code listing 2.8 Code buffer interface

```
pub fn cmpfun(&mut self, closure: lang::Closure) -> lang::Closure {
    let mut closure = closure;
    self.env = lang::NormalEnv::new(
        Box::new(closure.environment.clone()),
        false,
        lang::ListFrame::new(
            closure
                .formals
                .iter()
                .map(|x| {
                    data::TaggedSexp::new_with_tag(
                        x.value.as_ref().clone(),
                        x.name.data.clone(),
                    )
                })
                .collect(),
        ),
        lang::HashFrame::new(vec![]),
    )
    .into();
    if self.options.inline_level > 0 {
        self.localenv = HashSet::new();
        self.find_locals(&closure.body);
    }
    let body =
        SexpKind::Bc(self.gen_code(closure.body.as_ref(), Some(closure.body.as_ref()))).into();
    closure.body = Box::new(body);
    closure
}
```

Code listing 2.9 get_inlineinfo method

```
fn get_inlineinfo(&self, function: &str) -> Option<InlineInfo> {
    let base_var = self.is_base_var(function);
    if self.options.inline_level > 0 && base_var && self.has_handler(function) {
        let info = InlineInfo {
            guard: !(self.options.inline_level >= 3
                || (self.options.inline_level >= 2 && LANG_FUNCS.contains(&function))),
            base_var,
        };
        Some(info)
    } else {
        None
    }
}
```

Code listing 2.10 Main server loop

```
pub fn run() {
    let listener = TcpListener::bind("127.0.0.1:1337").unwrap();

    for conn in listener.incoming() {
        match conn {
            Ok(stream) => handle_conn(stream),
            Err(x) => println!("{}", x),
        }
    }
}
```

2.4 Server

Server is implemented as via TCP server from standard library. The server is able to handle multiple clients at one. Every request creates new thread, you can see this creation in main loop of the server in code example 2.10. All data must be newly loaded and no such as data, such as environments, are maintained throughout the request. The data are send exclusively through out the RDS format. Since the TcpStream implements the Reader trait in Rust, the implementation of RDS reading and writing can be done trivially by just declaring:

```
impl RDSReader for TcpStream {}
```

```
impl RDSWriter for TcpStream {}
```

After reading through the full request, the necessary data are extracted from the loaded SEXP. The SEXP must be a generic vector that has as a first argument closure to be compiled, followed by options and optionally other data that could be used for compilation.

Following the extraction of the data, the compiler is set up. The inlining level is set to 2 if not set otherwise throughout the options from request. If the information about what contains the base environment, what the builtins, specials, and the builtins internal are, the empty set is assumed. After all the settings are set, the closure is handed over to the compiler. The compiled closure is then serialized with TcpStream. At the end, the stream is flushed and closed.

In its current form, the server is a basic implementation without any advanced features. Most importantly, as mentioned above, there is no maintenance of data between requests, even from the same client. This would allow to slim down the request by not needing to send the whole environment and other additional information with every request for compilation and reuse of some compiled functions.

2.5 Client

The client is implemented as an R package written mainly in C programming language with some parts written in R programming language that handle the correct collection of the needed data for the compilation. The package exposes as its public interface one function named server_cmpfun. The header of this function is defined as:

```
server_cmpfun <- function(sexp, options=NULL, bundle_env=FALSE)
```

The first and second arguments are directly into the C part of the implementation, and depending on the last argument, the base environment is queried for necessary data. All of the data are bundled into a list, and this list is passed as a single argument into the C function with the same name.

The C function implements serialization, deserialization, and communication with the server. For serialization into RDS format, the functions implemented in the interpreter are used. The communication is implemented via TCP connections in the standard library. The inputted data are first serialized into the buffer, and this buffer is written into the TCP socket. After the server handles the request, the result is read from the same socket, deserialized, and returned.

Chapter 3 Assessment

This chapter contains the practices and results of the evaluation of the state of the compilation server and its parts at the current point. There are two considerations: correctness and performance. The correctness is assessed throughout the testing parts of the compiler. The testing is done using the Rust standard testing tools. The performance was measured against the GNU R implementation of the compiler.

3.1 RDS serialization testing

The RDS serialization is tested with a set of R expressions, which are evaluated and serialized with an R script that stores this evaluated data in the file. After the data are stored, my implementation of the RDS reader reads data from the file and compares the output with a snapshot of the data. Insta library is used to create and handle these snapshots. In order to test the implementation of the RDS writer, the test function loads the data and serializes it into the buffer. The result in the buffer is compared to the original input.

Since I compare the result to the results of the writer to the original values, there are some serializations that would represent the same values but would not be considered correct by the testing that is set up. The example would be the serialization of the hash environment, which could be represented with different order of insertion of values, resulting in different data serialization, but the value itself has the same properties. This is done because the alternative would require being able to compare the values that would be represented by the resulting data.

To streamline this testing, I implemented a macro, which creates test functions in a standard Rust testing environment for both reader and writer. You can see an excerpt from this macro in code reference 3.1 with example usage of this macro.

3.2 Compiler testing

The compiler is tested on two levels of inlining, either 0 or 2. The testing is done in two ways: the first is similar to RDS serialization testing, and the second is a compilation of the function queried from the base environment. The first method uses two macros for both inlining levels, similar to the RDS testing, and an example of usage can be seen in code reference 3.2. These macros use the R script that saves inputted closure, compiled code, and any additional data that are needed for compilation, depending on the inline level, into RDS format in files. Afterward, these files are used to load testing data into the SEXP. The closure is compiled, and the result from the Rust compiler is compared to the result of the original compiler.

The second type of compiler testing is used to benchmark how a large part of the compiler is implemented. Since, at the current time, only a subset of the inlining handler is implemented, there should be no possibility of compiling correctly all functions when compiling with inlining level 2. The R script is used to create test data. This script queries the base environment and then stores all the closures that can be compiled in both the compiled version and the AST version. Moreover, the script stores all additional information needed for compilation, such as a list of builtins. These data are then loaded into the test, and as in the first type of test, the result of my implementation of the compiler and the original compiler is compared. This testing is dependent on the base environment of the installed R binary. The current implementation can correctly compile 540 out of 1146 queried functions at inline level 2.

Similar to the RDS testing, these approaches disqualify some results that have the same properties. In the case of the compiler, there are more possibilities for creating outputs that represent the correct solution. However, to show that the compiled bytecode from my implementation is correct would require proof that it is equivalent to the output of the original compiler. In some cases, this could be possible, but unfortunately, in general cases, this is not feasible. More importantly, since the definition of the semantics of the bytecode is dependent only on the implementation of the GNU R bytecode interpreter, that would mean that if two bytecode representations are equal at the current time, they will not necessarily be equal in the future.

3.3 Performance testing

To assess the performance, I compared my compiler implementation to the original solution. This was done in the same way as the second type of compiler testing. All closures from the base environment are stored with all necessary data for compilation with inline level 2, and this data is then loaded and compiled with the Rust implementation of the compiler. The two times that have been examined are compilation itself, without the time that is necessary to read all the data and compilation with reading all data. These two times have been chosen since these actions are necessary for the whole process, but the loading of the whole environment and other additional data, other than the code itself, is strictly necessary only once per client initialization, although this is not part of the implementation as of yet.

To evaluate the results of my implementation, I measured the time that it took to compile all of the same functions. However, the time to load data is not counted since the interpreter already has all the data that are needed to compile the function. You can see the script that was used for this purpose in code reference 3.3.

These tests were then run multiple times, more precisely 100 times, to account for random time fluctuations within the execution times. These timings were then used to calculate the mean values of executions and speed up compared to original implementation. These tests were run on the machine with AMD Ryzen 5 7600X 6-Core with 16GiB of RAM. You can see results in table 3.1.

Table 3.1 Result of measurements

	Times	Speedup
Original implentation	4.065 s	1
My implementation with RDS data loading	1.130 s	3.598
My implementation without RDS data loading	0.193 s	21.009

■ **Code listing 3.1** RDS test macro

```
macro_rules! testR {
    ( $name:ident, $code:expr) => {
        mod $name {
            use super::*;
            #[test]
            fn reader() {

                ...

                let mut file = std::fs::File::open(path).unwrap();
                let RDSResult { header: _, data } = file.read_rds().unwrap();

                insta::assert_debug_snapshot!(data);
                std::fs::remove_file(path).unwrap();
            }

            #[test]
            fn writer() {

                ...

                let mut file = std::fs::File::open(path).unwrap();
                let RDSResult { header, data : input } = file.read_rds().unwrap();

                ...

                writer.write_rds(header, input).unwrap();
                writer.flush().unwrap();

                assert_eq!(writer.get_ref(), &input_vec);
                std::fs::remove_file(path).unwrap();
            }
        }
    }
}

...

testR![intsxp_02, "as.integer(c(1, 2))"];
```

Code listing 3.2 Compiler test macro

```
// compiler test for inlining level 0
test_fun_noopt![
    block,
    "
    function(a, b = 0) {
        x <- c(a, 1);
        x[[b]];
    }"
];

// compiler test for inlining level 2
test_fun_default![
    block02_opt,
    "
    function(x) {
        if (x) 1 else 2;
        if (x) 3 else 4
    }"
];
```

Code listing 3.3 RDS test macro

```
#!/usr/bin/Rscript

# load all the names of variables from base package
basevars <- ls("package:base", all.names = TRUE)

# get types of the values in basevars variable
types <- sapply(basevars, \(x) typeof(get(x)))

# get only closures in their AST form
orig <- sapply(basevars[types == "closure"], \(x) {
    tryCatch(eval(parse(text=deparse(get(x)))[[1]]), error = function(e) {
        NULL
    })
})

start_time = Sys.time();
# compiler these functions with default compiler
x <- sapply(orig, \(x) tryCatch(compiler::cmpfun(x), error=function(e) NULL));
end_time = Sys.time();
end_time - start_time
```

Chapter 4
Conclusion

The goals of this thesis was to create the proof of concept implementation which could be used to experiment in this space. To achieve this I created the custom representation of R values, RDS serialization, compiler for GNU R bytecode, basic TCP server and client package that communicates with the server. The server part was implemented with Rust programming language and client code was implemented with combination of R and C programming languages.

This implementation was then tested for its correctness and performance. These properties were tested against the GNU R implementation of compiler. This showed that the part of compiler that is already implemented can be more then 20 times faster then the baseline implementation and if the reading of the data that are necessary for compilation my solution can be around 3 times faster. Although these results are promising there must be more measurements to truly find the performance characteristics of Rust implementation.

The compiler implementation is at current point still incomplete, however the main structure of compiler is done. The main part that is necessary to implement, to achieve feature parity with the GNU R implementation, are not yet implemented inline handlers and constant folding. The support for inline handler is already created within the compiler, however the constant folding would need additional design.

Moreover the server and client implementation is done by basic TCP communication which does not implement more sophisticated scheme for communication. This could create possibilities for lowering overhead of communication and sharing the resulting compiled code.

Bibliography

1. PROJECT, R. *R About* [online]. [N.d.]. Available also from: `https://www.r-project.org/about.html`.
2. PROJECT, R. *R Language Definition* [online]. [N.d.]. Available also from: `https://cran.r-project.org/doc/manuals/R-lang.html`.
3. PROJECT, R. *GNUR source* [online]. [N.d.]. Available also from: `https://svn.r-project.org/R/`.
4. SIEK, Konrad. *Everything You Always Wanted to Know About SEXPs But Were Afraid to Ask* [online]. [N.d.]. Available also from: `https://gitlab.com/kondziu/everything-you-always-wanted-to-know-about-SEXPs/-/blob/master/2017-03-12-everything-you-always-wanted-to-know-about-sexp-but-were-afraid-to-ask.md`.
5. PROJECT, R. *readRDS: Serialization Interface for Single Objects* [online]. [N.d.]. Available also from: `https://www.rdocumentation.org/packages/base/versions/3.6.2/topics/readRDS`.
6. TIERNEY, Luke. *A Byte Code Compiler for R* [online]. [N.d.]. Available also from: `https://homepage.cs.uiowa.edu/~luke/R/compiler/compiler.pdf`.

Content of the attachment

```
dip_implementation ........................ Server implementation
└─ compile_client_package ........... Client package implementation
```