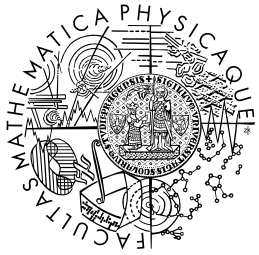

FACULTY
OF MATHEMATICS
AND PHYSICS
Charles University

MASTER THESIS

Pavel Marek

ALTREP Data Representation in FastR

Department of Distributed and Dependable Systems

Supervisor of the master thesis: Mgr. Štěpán Šindelář
Study programme: Computer Science
Study branch: Software Systems

Prague 2020

I declare that I carried out this master thesis independently, and only with the cited sources, literature and other professional sources. It has not been used to obtain another or the same degree.

I understand that my work relates to the rights and obligations under the Act No. 121/2000 Sb., the Copyright Act, as amended, in particular the fact that the Charles University has the right to conclude a license agreement on the use of this work as a school work pursuant to Section 60 subsection 1 of the Copyright Act.

In ..... date ....

Author's signature

I would like to thank my supervisor, Štěpán Šindelář, his colleague Tomáš Stupka, and my consultant, Petr Tůma, for their advices and patience. I would also like to thank my wife, Hana, for her support and unconditional love.

Title: ALTREP Data Representation in FastR

Author: Pavel Marek

Department: Department of Distributed and Dependable Systems

Supervisor: Mgr. Štěpán Šindelář, Oracle Labs

Abstract: R is a programming language and a tool used mostly in statistics and data analysis domains, with a rich package-based extension system. GNU-R, the standard interpreter of R, in version 3.5.0 introduced a new native API (ALTREP) for R extensions developers. The goal of the thesis is to implement this API for FastR, an interpreter of R based on GraalVM and Truffle, and explore options for optimization of FastR in context of this API. The motivation is to increase the number of extensions that can be installed and run on FastR.

Keywords: R interpreters programming languages compilers

Contents

Introduction

1. Introduction

This thesis is built on FastR [1], an interpreter of R programming language based on Graal [2] and Truffle [3]. This introduction should give the reader a general idea of how FastR works and how the ALTREP API relates to FastR. In the following text, we provide a high-level description of the components of FastR, and state the goal of this thesis.

R and ALTREP

R is an interpreted, dynamically-typed, programming language with a rich package-based extension system. Many packages use *native code* (typically C or C++), that interoperates with R objects via a native extensions API called *R Internals API*. In *GNU-R*, the standard interpreter of R, up until version 3.5.0, every object had the same memory layout - a header and a continuous array of data. With the introduction of ALTREP, this is no longer true - ALTREP (alternative data representation) is a part of the R Internals API that allows R package developers to provide an alternative data representation, i.e., the alternative memory layout of the objects. With ALTREP one can, for example, define an object with the data that are dynamically fetched from some database or shared with another process.

GraalVM and Truffle

GraalVM is a JDK that is an extension of a standard JDK and atop of "standard" components it provides many others including *GraalVM Compiler*, a Just in Time (JIT) compiler, and *Truffle*. Truffle is a platform for building high-performance language implementations. With Truffle, a language is implemented as a self-optimizing Abstract syntax tree (AST) interpreter where the nodes represent the semantics of the implemented language operations.

FastR

FastR is an implementation of the R language interpreter built on top of GraalVM and Truffle. It aims to be compatible with GNU-R. The ALTREP API is part of GNU-R since version 3.5.0. and is not yet implemented in FastR. This makes FastR incompatible with GNU-R, as there already exist some packages that use the ALTREP API.

FastR already supports many native packages (packages that use the native code) and implements the majority of functions in the R Internals API. This means that FastR supports calls from the R code to the native code, as this is what the native packages frequently do. However, as we will see in chapter 2, the calls from the R code to the native code and vice versa, in the context of ALTREP, are different than the "standard" already existing calls. Therefore, we have to explore the options of how to support ALTREP in the current infrastructure for native code calls in FastR.

Goals

The goal of this thesis is to analyze the semantics and implementation of mostly undocumented ALTREP API in GNU-R, and implement the ALTREP API in FastR, along with extensive tests, and also to explore the possibilities of optimizations in FastR in the context of the ALTREP API.

2. Technical background

In this chapter, we provide a high-level overview of the technologies used later in the thesis.

2.1 R

R is a dynamically-typed, interpreted programming language mostly used in statistics and data analysis domains with a rich package-based extension system. We provide a list of concepts and constructs of R that we will use throughout the rest of the thesis:

- *Vectors* ... "Vectors can be thought of as contiguous cells containing data" [4]. We will use integer, double, and raw vectors. Raw vectors are vectors containing bytes.
- Vectors are *immutable*. In fact, many other objects in R are immutable, but we will focus on vectors.
- R interpreter uses *copy-on-write* semantics. Let us demonstrate this principle on the following snippet:

```
x <- c(1,2,3)
y <- x  # x is not copied yet
y[[1]] <- 42  # x is copied before the assignment
x[[1]] != y[[1]]
```

- *Lists* ... A list is essentially a vector of vectors.
- *Attributes* ... Attributes are a set of `name=value` pairs that can be attached to any object (except `NULL`).
- *Pairlists* ... A pairlist is a list where every item has a *header* (`CAR`), a *tail* (`CDR`) that can be either `NULL` or a reference to another item, and optionally a `TAG` which is typically a string. "Pairlists are extensively used in the internals of R and are rarely visible in the interpreted code" [4].

For the complete R language specification please refer to [4].

There exist more implementations of R interpreter:

- *GNU-R* is the standard interpreter.
- *Renjin* is an R interpreter implemented in Java. It was initially designed to be embedded into JVM as a scripting engine - similarly to *Jython* [5].
- *terr* (TIBCO Enterprise Runtime for R) which is a closed-source implementation of R in C++.
- *pqR* (a "pretty quick" version of R) is a fork of GNU-R interpreter with some minor differences causing pqR to be faster for certain operations, e.g., pqR claims to have a faster garbage collector.

- *riposte* is another open-sourced implementation of R written mostly in C++, no longer maintained.

This thesis focuses on GNU-R and on FastR, because terr is commercial and other interpreters do not implement ALTREP.

As is the case for many other dynamically-typed languages, there is a native API that enables package authors to use C or C++ in their packages to *interoperate* between native code and R, where interoperation typically means allocating R objects from C, modification of the objects from C, or calling various other functionality from C. This native API, denoted as *R Internals API* [6] is widely used in packages, especially in packages where substantial performance is crucial.

2.1.1 Packages

A package in R is a collection of the source files (mostly R, C, and C++), data files, manuals, and various other optional files, that together may be built, published to the *CRAN* repository [7], and later downloaded and installed by other users. The entire structure of a package is described in [8]. The package has to conform to the standard directory layout - all native source files in the directory `src`, all R source files in the directory `R`, all data files in the directory `data`, and so on. In the `NAMESPACE` file, the developer provides a list of all the R functions accessible from outside the package, i.e., a list of public functions. This is called a namespace of a package and is described in "Search path" section in [4].

In the rest of the thesis, we will focus on the packages that contain native code, denoted as *native packages*. A native package has to provide at least an *initialize function*, which is called when the package is loaded by the R interpreter, and optionally a *deinitialize function*, which is called when the package is unloaded by the R interpreter. These functions must conform to certain naming conventions so the R interpreter can locate them once the shared library of the package is loaded. Initialize function typically registers native functions that should be treated as "public" and therefore used by the `.Call` and `.C` R functions (see "Interface functions .Call and .External" section in [8]). Such registration is done with `R_registerRoutines` function and will be used by the R interpreter later for lookups of the native functions.

Build

Building a package is as simple as invoking `R CMD build <pkgdir>` command from the shell, which does everything necessary to build a package - checks for syntax errors in R sources, compiles native sources if there are any, checks validity of manual pages, and so on. Optionally, a developer may provide a `Makevars` file, which adds new options for the compiler or overrides the existing ones. A build of a native package ultimately produces a shared library that exports initialization and deinitialization functions. Once the native package is loaded, the shared library is loaded into the memory and the initialization function is called, which in turn typically registers the native functions with `R_registerRoutines` and the R interpreter stores the names of the functions for later lookup. When the developer uses `.Call`, the R interpreter looks up the symbol in the registered native functions.

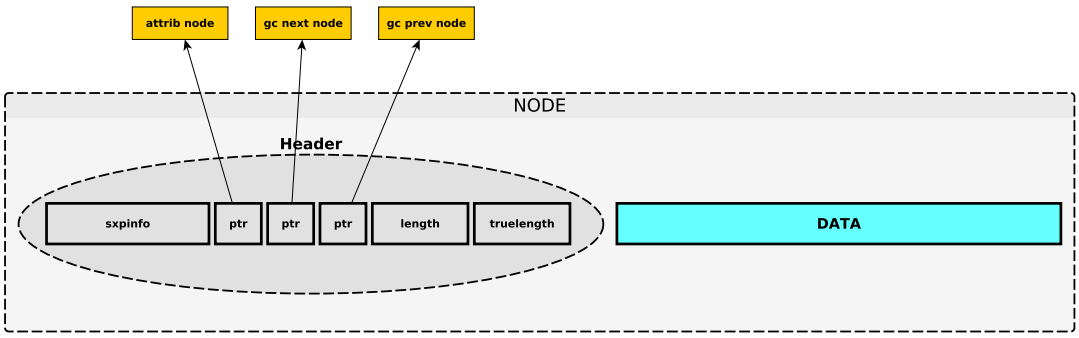

Figure 2.1: Integer vector memory layout

2.1.2 Object memory layout

Up until GNU-R version 3.5.0, every object in GNU-R was represented in memory by a *header* containing various metadata like length and type of the object, and by a continuous range of memory denoted as object's *data*. A header along with data represents one R object and is denoted as a *node* in GNU-R terminology.

In Figure 2.1 we can see the memory layout of an integer vector. Note that this layout is specific for GNU-R version 3.5.2 and is subject to change in the future versions. The header consists of:

- 64 bit long `sxpinfo` that contains 5 bit `SEXPTYPE`, which identifies 32 possible data types, in our case it is `INTSXP` - an integer type.
- `attrib` ... Pointer to the attributes node.
- `gen_gc_next` ... Pointer to the next node, used by the Garbage collector (GC).
- `gen_gc_prev` ... Pointer to the previous node, used by the GC.
- `length` ... Length of the data.
- `truelength` field, which is the same as length field for an integer vector.

After the header, possibly with some alignment, there is the data part. The data is represented as a continuous region of *length* `int` elements.

2.1.3 R Internals API

The *R Internals API* is a C API through which a developer can manipulate R objects. The majority of functions and macros that comprises this API is in the `Rinternals.h` and `R.h` header files.

In this section, we will briefly describe just those functions, macros and types that are used throughout the thesis inside various code snippets. Note that for many of these functions we will not provide a precise signature for the sake of clarity. Below is a list of functions, macros, and types with their description:

- `SEXP` ... An opaque pointer type used to point to almost every object in memory. In fact, `SEXP` is a pointer to `SEXPREC` structure, which is not

exposed to package developers. The vast majority of functions have `SEXP` as parameters or return values.

- `R_NilValue` ... A singleton value equivalent to `NULL`.
- `SEXP allocVector(int length, int type)` ... Allocate a vector object with some `type` and `length`
  - `type` is, for example, `INTSXP` (integer), `REALSXP` (double), `STRSXP` (string), etc.
- `int TYPEOF(SEXP x)` ... Returns the type of the object `x`. The type is inside the header of the object.
- `int LENGTH(SEXP x)` ... Returns the length of the data part of an object `x`. Length is also inside the header.
- `void * DATAPTR(SEXP x)` ... This is by far the most important function in the R internals API. This function returns a pointer to the data of an object `x`. Before ALTREP, it was the only way how to iterate over an object, which generally looked like this:

```
int summarize(SEXP vec) {
    assert(TYPEOF(vec) == INTSXP);
    int sum = 0;
    int *ptr = (int *) DATAPTR(vec);
    for (int i = 0; i < LENGTH(vec); i++) {
        sum += ptr[i];
    }
    return sum;
}
```

  Note that `DATAPTR` has many variants depending on the type of the pointer that it returns, i.e., `INTEGER` for returning an integer pointer. In GNU-R, these variants usually just call `DATAPTR` and cast the pointer.

- `SEXP duplicate(SEXP object, Rboolean deep)` ... Duplicates the given `object`. The `deep` argument is only considered when `object` is a list (vector of vectors) - in such a case, with `deep = true`, all the inner atomic vectors are duplicated right away, whereas with `deep = false`, the reference count of the elements of the inner atomic vectors is set to the maximal value. Note that there is no R counterpart of this function - the R interpreter duplicates objects transparently, based on the copy-on-write semantics (see 2.1).

2.1.4 Garbage collector

GNU-R has a generational, single-threaded, stop-the-world, non-compacting GC [6]. Garbage collection may be triggered only from functions that allocate some memory or from some other functions like `allocVector`, `eval`, or `forceAndCall`.

Protection of values

The allocated objects might be protected against garbage collection via `PROTECT` function call. The protection mechanism is stack based - there is a *pointer protection stack* internal data structure - `PROTECT` pushes a new value and `UNPROTECT` pops a value. Every native code application should protect every value it allocates either directly or indirectly, and unprotect it once it is no longer needed. However, the native package developers sometimes misuse the fact that most of the functions in R Internals API do not call the GC, and they do not protect the values. Note that the documentation suggests protecting all values, there is no mention about some functions being safe from the garbage collection - it is just an artifact of the implementation [6].

With ALTREP, the lack of the protection of the values returned from certain functions causes problems. With ALTREP, the list of functions that might trigger garbage collection grew - almost all of the functions within the ALTREP API might trigger a garbage collection. A particularly good example is the `INTEGER` function call, which is used in many packages. Before ALTREP, `INTEGER` never triggered a garbage collection, therefore the native package developers did not protect the values they had created before the `INTEGER` function call. With ALTREP, this behavior changes, and it might break some packages. The current solution chosen by the R core developers to address this issue is to suspend the garbage collection within these ALTREP API functions [9]. This, unfortunately, means that if `INTEGER` tries to allocate a new object and there is not enough memory, then GNU-R fails immediately, even if there would be enough memory after the garbage collection.

2.2 ALTREP

In this section, we will define ALTREP and describe its basic constructs and how it is implemented in GNU-R. We first define ALTREP and then provide some examples in 2.2.4.

ALTREP (alternative data representation) was first introduced in GNU-R version 3.5.0. As the name suggests, ALTREP enables package developers to provide an alternative data representation of R objects, as opposed to standard memory layout described at 2.1.2.

Some goals of ALTREP are:

- Data of vectors may be:
  - In a memory-mapped file.
  - Distributed, e.g., within Apache Spark or Hadoop.
  - Shared with different applications, e.g., with Apache Arrow.
- Allow compact representation of arithmetic sequences.
- Allow adding meta-data to objects.
- Allow computations or allocations to be deferred.
- To existing C code, ALTREP objects should look like ordinary R objects.

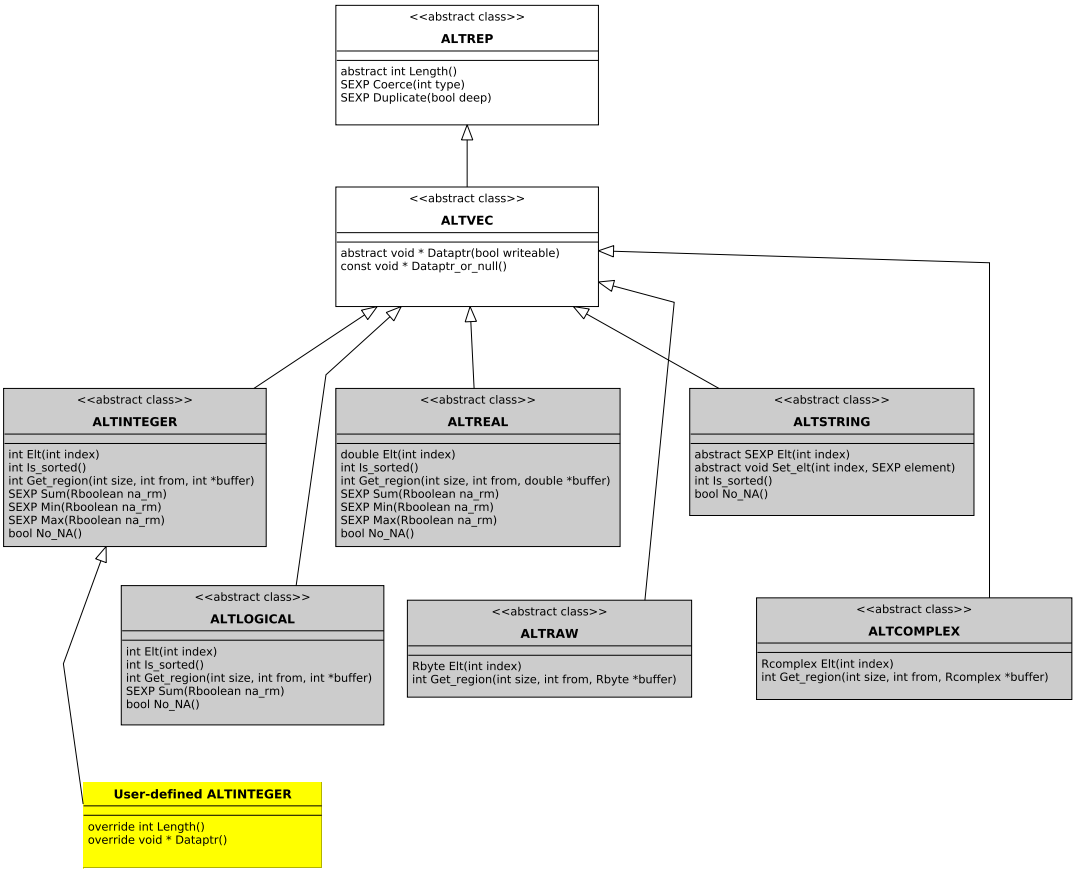

Figure 2.2: ALTREP abstract class hierarchy

In a high-level overview, ALTREP provides a hierarchy of abstract *classes* with the `ALTREP` base class, `ALTVEC` extending this base class, and `ALTINTEGER`, `ALTREAL`, `ALTSTRING`, and others extending `ALTVEC` and representing basic R data types as depicted in Figure 2.2. Note that a more detailed description of the figure is further in the text. Unless specified otherwise, a *class* will denote an ALTREP class, and a *method* will denote an ALTREP method, in the rest of the thesis. The list of methods is not complete and the signatures are simplified. Also note that the authors of ALTREP claim that the class hierarchy may not be final, i.e., some classes may be added or some may be merged [10].

Most of the methods have clear semantics based on the name. The documentation of all the ALTREP methods is not in the codebase of GNU-R, as is the case for most other functions, neither it is in [6]. Let us, therefore, describe the semantics of some of the methods:

- `void * Dataptr()` ... The semantics for this method is the same as for `DATAPTR` function (see section 2.1.3) - a valid pointer to a continuous array with the size given by the `Length` method and with the type of the elements given by the type of this ALTREP instance, e.g., `int` for altinteger, `double` for altreal, `Rbyte` for altraw, etc. Note that the data of an ALTREP instance does not have to be represented as a continuous array in memory, but once the `Dataptr` method is called, the ALTREP instance has to *materialize* its data. We say that an ALTREP instance is *materialized* when its data is converted to the continuous array format. In the rest of the thesis, the term materialization is used for other objects with the same meaning.

  We generally want to avoid materialization of objects, therefore we avoid calling the `Dataptr` method unless it is necessary.

- `const void * Dataptr_or_null()` ... This method should return `NULL` in case the underlying ALTREP object is not materialized, otherwise it returns a pointer to the data. Note that unlike `Dataptr`, it returns a constant pointer that should be used only for reading the data.

- `Elt(int n)` ... Returns the `n`-th element of the ALTREP object. The return type of this method differs based on what class it is defined in.

- `void Set_elt(int index, SEXP element)` ... This method is declared only for `ALTSTRING` because string vectors are not atomic (they are lists), and the garbage collector has to ensure that the elements of such a non-atomic vector are at least as old as the vector itself. This is described in "Write barrier" section in [6].

- `int Is_sorted()` ... Lazily returns an enum value of sortedness - which may be ascending, ascending with NAs as first elements, decreasing, decreasing with NAs as first elements, and unknown sortedness. *Lazily* means that `Is_sorted` method should not do any computation and should only return the information about sortedness of the ALTREP object that is known at the time.

- `bool No_NA()` ... Lazily returns true if there are no NA values in the vector, or false if there are some NA values or if the information is unknown.

- `SEXP Sum(bool na_rm)` ... Computes the sum of the elements of the instance, treating `NA`s as defined by `na_rm` argument. Note that the return value is not a primitive type but a `SEXP`, which gives the method flexibility to return a vector.
- `Min` and `Max` have the same signature as `Sum` and return the minimal and the maximal element, respectively.

Let us describe Figure 2.2 in greater detail:

- The greyed-out classes are the ones that correspond to some basic R data type. These are the only classes that a developer can extend.
- The method is either abstract, or it provides some default implementation.
- The yellow class is a user-defined altinteger. It overrides all the abstract methods, therefore it is a valid class.

2.2.1 Implementation in GNU-R

This section provides the implementation details of ALTREP objects in GNU-R. All the information provided in this section can be deduced from the GNU-R sources `src/main/altrep.c` and `src/include/R_ext/Altrep.h`.

Each ALTREP object has a class, denoted as class descriptor, and *instance data*. The class descriptor can be thought of as a virtual method table with pointers to the implementation of the methods. An ALTREP method invocation corresponds to the virtual method dispatch. Every ALTREP instance is limited to two `SEXP` objects as instance data.

Class descriptors

Class descriptors are raw vectors with:

- Attributes representing the class name, the package name, and the type of the ALTREP instance.
- Data containing pointers to ALTREP methods.

As of GNU-R version 3.5.2, there is little error handling implemented - for example, not overriding an abstract method results in an error during the invocation of such method, rather than returning some error result from the `R_alt*_set_method` functions.

Instances

ALTREP instances (objects) are pairlists with:

- `alt` bit set in their `sxpinfo` in header (see section 2.1.2).
- A pointer to the class descriptor as their `TAG`.
- A pointer to `data1`, i.e., first instance data as their `CAR`.

- A pointer to `data2`, i.e., second instance data as their `CDR`.

Pairlist is an already-existing data structure in R used mostly internally in the interpreter (see 2.1). The advantage of representing an ALTREP object with a pairlist is that it does not require additional work spend on programming and debugging new data structures. The disadvantage is that the package developers may use standard R internals API for accessing pairlists to access various meta-data about an ALTREP object. And obviously, ALTREP object should not be treated as a pairlist, because they have different semantics. Unfortunately, as we can see in the package `vroom`, this knowledge of internal representation of an ALTREP object is already used by the package developers.

Note that the instance may use an arbitrary object as its data, including `R_NilValue`.

2.2.2 Class definition API

In this section, we will give an example of how the ALTREP API is used to define an ALTREP class. We will define a simple ALTREP class that is a wrapper for an ordinary integer vector. The creation of ALTREP classes is usually done in the package initialization function. Bellow is the commented code snippet demonstrating the usage of this API.

```
R_altrep_class_t class_descriptor;

R_xlen_t my_Length_method(SEXP instance) {
  // Get the instance data - in this case we use only the first instance data.
  SEXP wrapped_data = R_altrep_data1(instance);
  // Return the length of the wrapped vector.
  return LENGTH(wrapped_data);
}

void * my_Dataptr_method(SEXP instance, Rboolean writeabble) {
  SEXP wrapped_data = R_altrep_data1(instance);
  return DATAPTR(wrapped_data);
}

/**
 * A package initialization function.
 */
void R_init_mypackage(DllInfo *dll) {
  // Create a new altrep class via R_make_*_class functions
  // Other alternatives are: R_make_altreal_class, R_make_altstring_class, etc.
  class_descriptor = R_make_altinteger_class("ClassName", "PackageName", dll);

  // We override all the "abstract" methods.
  R_set_altrep_Length_method(class_descriptor, &my_Length_method);
  R_set_altvec_Dataptr_method(class_descriptor, &my_Dataptr_method);

  // We could override more methods, e.g., Elt, but let us keep the exaple
  // simple and clear.
}
```

```
/**
 * Function exported in the package's namespace, i.e., this function is called
 * from .Call R function.
 */
SEXP create_wrapper(SEXP data) {
  // Create new instance of class_descriptor, with data as first instance data,
  // and R_NilValue as the second instance data.
  return R_new_altrep(class_descriptor, data, R_NilValue);
}
```

Let us summarize the class definition API:

- Functions `R_make_alt*_class` are used for creating a class descriptor. These class descriptors are objects used to describe which methods the class overrides.
- Functions `R_set_*_*_method` *register* a specific method for the class descriptor. Note the terminology: we say that we *register* a method on a class descriptor rather than override some method from class descriptor. Note that some methods need to be registered because there are no defaults for them.
- Finally, with the function `R_new_altrep` we create a new instance of the given ALTREP class. Furthermore, we specify two instance data - both of them might be `R_NilValue`.
- With functions `R_altrep_data1`, `R_altrep_set_data1` and their versions for `data2`, we can get or set the data associated with a particular instance. Note that we may access the instance data both from ALTREP methods and outside ALTREP methods, so there is no encapsulation as in typical object-oriented programming language.

2.2.3 Rest of ALTREP API

In this section, we will show the rest of the ALTREP API that is an extension of R Internals API. Note that most of the functions presented take any kind of object as an argument, not necessarily an ALTREP object, but if the argument is an ALTREP object, usually some specific ALTREP method on the object is invoked. All these functions are present in GNU-R from version 3.5.0. Some of the presented functions are specific for the integer data type - those with the `INTEGER` prefix in the name. Note there are also functions specific for all other basic R data types - real, raw, logical, complex and string - that are not included in the list below as they differ only in the signature and not in the semantics. Also, note that the list is not complete, we present only the functions that will be used in the rest of the thesis.

- `int INTEGER_ELT(SEXP object, int i)` ... This function returns the `i`-th element of the `object`. Before GNU-R version 3.5.0, the only possible way to get the i-th element of an object was with `INTEGER(object)[i]` statement (see section 2.1.3). That is: first obtain a pointer to a continuous

array in memory via the INTEGER function call, then do pointer arithmetics on the result. From a certain perspective, the INTEGER function represents a flaw in the design of the API, because it leaks an implementation detail - there has to be a continuous array of elements as the data of the object. INTEGER_ELT, on the other hand, does not expose the underlying object's memory layout.

- `const void * DATAPTR_OR_NULL(SEXP object)` ... This function is an addition to the DATAPTR function. It is the DATAPTR's lazy counterpart - DATAPTR_OR_NULL may return NULL if the underlying object is not materialized. On the other hand, DATAPTR must return a valid pointer, therefore materialize the object if necessary.

- ```
  int INTEGER_GET_REGION(SEXP object, int from_idx,
                         int size, int *buffer)
  ```

  This function fills the given buffer with contents of an object's data region defined by from_id and size arguments. When this function is called with an ALTREP object as the argument, its Get_region method is invoked. The default implementation just iterates over the vector via INTEGER_ELT function described above. In the current version of GNU-R, this is the preferred way of iteration over vectors, as we will later see on ITERATE_BY_REGION macro.

- `Rboolean INTEGER_IS_SORTED(SEXP object)`

  This function returns true if the given object is sorted. This function is lazy - it does not compute anything, it just returns false when the result is unknown at the time of the invocation. If the object is an ALTREP object, its Is_sorted method is invoked.

- `ITERATE_BY_REGION(vec, size, expr)` ... A macro that iterates over the given vector vec. For simplicity, we omit some parameters (the real macro has 6). This macro is not a part of the ALTREP API but uses some functions from it, so we decided to describe it in this section rather than in section 2.1.3. Using the macro in the following way:

  ```
  int acc = 0;
  ITERATE_BY_REGION(vec, nbatch, {
      for (int i = 0; i < nbatch; i++) {
          acc += dataptr[i];
      }
  })
  ```

  has the following simplified expansion:

  ```
  int acc = 0;
  int *dataptr = DATAPTR_OR_NULL(vec);
  if (dataptr != NULL) {
      for (int i = 0; i < LENGTH(vec); i++) {
          acc += dataptr[i];
  ```
  ```

```
        }
    }
    else {
        int buff[512];
        for (int k = 0; k < LENGTH(vec); k += 512) {
            INTEGER_GET_REGION(vec, k, 512, buff);
            dataptr = buff;
            for (int i = 0; i < 512; i++) {
                acc += dataptr[i];
            }
        }
    }
```

In other words, if `DATAPTR_OR_NULL` returns a valid pointer, then we iterate over this valid pointer. If `DATAPTR_OR_NULL` returns `NULL`, then we iterate over batches retrieved with `INTEGER_GET_REGION`.

2.2.4 ALTREPs in GNU-R

In this section, we list some use cases of ALTREP already implemented in GNU-R in version 3.5.2. Note that besides the ALTREP classes we mention in this section, there are also "deferred string conversions" and "memory-mapped vectors" [9].

Compact sequences

In the following code snippet:

```
for (item in 1:1e10) {
    do.some.sideeffect()
}
```

Pre-3.5.0 GNU-R (GNU-R without ALTREP) would allocate $10^{10} * 4$ bytes of memory and fill it with the values of the sequence, although this allocation is unnecessary - we do not even read the values of the sequence.

Post-3.5.0 GNU-R (GNU-R with ALTREP) has an alternative representation for this kind of sequences with only three integers - the start of the sequence, the end of the sequence, and the step. As long as we do not write into this sequence, the compact representation is preserved.

Wrapped meta-data

Because all vectors in R are immutable, it makes sense to cache some computed values on these vectors. For example, whether the vector is sorted or whether it contains some NA values.

In the following code:

```
x <- rnorm(1e8)
y <- sort(x)
sort(y)
```

non-ALTREP GNU-R would first sort the unsorted vector (the `sort(x)` statement), and then it would sort the already sorted vector again (the `sort(y)` statement). The ALTREP GNU-R would only sort the vector once. In ALTREP GNU-R, this is possible because `y` is an ALTREP object containing not only the data, but also the information about sortedness. Therefore, the statement `sort(y)` will be a no-op in ALTREP GNU-R.

2.2.5 Packages using ALTREP

These three packages are the most popular real-world packages on CRAN using ALTREP:

- `vroom` [11] ... A package for reading structured files. In this package, ALTREP is used to lazily read just a portion of a file once it is needed. It is a native package that uses a thread pool for the actual reading. In [12] the package authors claim that `vroom` is faster than `readr` on larger datasets. According to the blogpost at `https://www.tidyverse.org/blog/2019/05/vroom-1-0-0/`, `vroom` will be merged with `readr` eventually.
- `stringfish` ... This package contains an altstring class definition that is a wrapper for C++'s STL `std::string`.
- `qs` ... Package for fast serialization and deserialization of objects. This package depends on the `stringfish` package for fast serialization of character vectors.

There are also the following packages that are not published on CRAN. They mostly serve as demo packages showing the usage of the ALTREP API.

- `AltWrapper` [13] ... A package that contains R wrapper functions for the ALTREP native API.

2.3 GraalVM

GraalVM [14] is a Java Development Kit (JDK) that is an extension of a standard JDK (OpenJDK or Oracle JDK). On top of the components and features that the standard JDK provides, it also provides:

- The *GraalVM Compiler* - A JIT compiler that uses the JVM Compiler Interface and is itself implemented in Java. This is in contrast to previous JIT compilers, like the HotSpot JIT compiler, which is implemented in C++.
- *Truffle* - A platform for building high-performance language implementations.
- *Interoperability* between languages that are implemented in Truffle.
- Native Image - A tool for Ahead of Time (AOT) compilation of Java into a standalone self-contained executable which also contains the runtime.

2.3.1 GraalVM Compiler

The JVM Compiler Interface started as an experimental feature in JDK 9. It provides access to the Java Virtual Machine (JVM) data structures like classes, methods, fields, various profiling information, and a way to install compiled code into the memory of the JVM. More details can be found in [15].

The GraalVM Compiler is a successful demonstration of the usage of this API - the peak performance is on par with C2 JIT compiler across a wide range of benchmarks [15].

The GraalVM Compiler does some additional dynamic optimizations in contrast to the "older" ones found in HotSpot - for example *partial escape analysis* [16]. The GraalVM Compiler aims to produce a highly optimized code through extensive use of *speculative optimizations* [17].

Dynamic compilation Let us provide a brief description of what *dynamic compilation* is. Unlike static compilation, dynamic compilation happens at runtime. One of the initial phases of a dynamic compilation is the *profiling phase*, during which the bytecode interpreter profiles various information. The compilation starts after the profiling phase and the profiles collected previously are used for the compilation. This resembles the standard Profile-guided optimizations (PGO) of the static compilers. The *speculative optimizations* assume that the types of the objects frequently seen at the runtime will not change in the future and the compiler compiles only a subset of code handling these types, therefore leaving out a lot of unnecessary cases. Because the speculations might turn out to be false in the future, Truffle (or Graal) inserts *guard* statements and *deoptimization points* in the code. The guard statements check whether the assumption still holds and if not, the native code is discarded and the control is transferred back to the interpreter. Note that unlike PGO, the speculative optimizations cannot be done by the traditional static compilers.

2.4 Truffle

Motivation

To implement an interpreted language, we have to implement at least the parser and the interpreter. Such an implementation would have poor performance compared to languages with industry-scaled interpreters and dynamic optimizers - like the Javascript V8 engine [18]. A dynamic optimizer (compiler) is a different component than the interpreter and it usually cannot share much of its code, therefore we need to develop the compiler separately from the interpreter. Moreover, we usually need to maintain a specification of some IR that is emitted by the parser. To make the implementation even more difficult, compilers and interpreters are usually written in a low-level system programming language like C or C++. All in all, producing a completely new interpreted (dynamic) language or developing a new interpreter along with the compiler for an existing language, is a lot of work.

With Truffle, we can implement the interpreter in a high-level *host* language - in Java. The IR of the Truffle language implementation (TLI) is an AST. This

means that for every operation in a guest language, there is a separate node implementing the operation. More specifically, the guest language operations are represented as classes extending the `Node` abstract base class and overriding its `execute` method. At runtime every node of this AST is traversed by calling the `execute` methods on its children and then on this node. During the traversal, various profiling information is collected, e.g., types of values returned by nodes.

TLI is a *self-optimizing* AST interpreter. Truffle does a lot of optimizations in context of the AST, as described in [19], [2], and [3]. The following text is a reworded excerpt from these papers.

Node specialization

A particular guest language operation may have different semantics based on the type of its operands, e.g., `+` operator in R either adds two integer vectors, two real vectors, or a combination of them where the R interpreter has to first coerce one of the arguments. In Truffle, the nodes have to implement the whole guest language operation semantics. However, having the code for the whole semantics active all the time would not lead to any performance improvement, therefore the node has different *specializations* for different subsets of semantics, often linked to different types of operands. For example, the `+` operator in R would have one specialization for adding two integer vectors, one specialization for adding two real vectors, and other specializations that handle coercion from integer to real or vice versa, recycling, attributes, etc. After a new node is created, it is in an *uninitialized* state. Later, it specializes itself (activates one specialization) for particular types of operands. At one moment only one specialization can be active, and only active specializations are eventually compiled into the machine code, whereas inactive specializations are replaced with deoptimization points.

AST rewriting

The specializations have *guard* conditions that check the types of operands for that specialization, e.g., in a case when a specialization for integer vector operands is active and the node gets a real vector as an operand, the guard condition fails. In such a case, the specialization can no longer be active, and the node has to transition from one specialization to another. This process is called *node rewriting*. Node rewriting has these limitations that are not enforced by Truffle:

- Finiteness: After a finite number of node rewrites, the last specialization has to be able to cover the whole semantics of the operation. This is important because we want the AST to stabilize eventually.
- Completeness: Every specialization can handle every possible input by rewriting to another specialization

Method inlining

"Whenever there is a hot call site, the AST of the called method is copied and put back into uninitialized state. This way, the AST of the called method is specialized based on its usage patterns from the call site. Figure 2.3 illustrates this

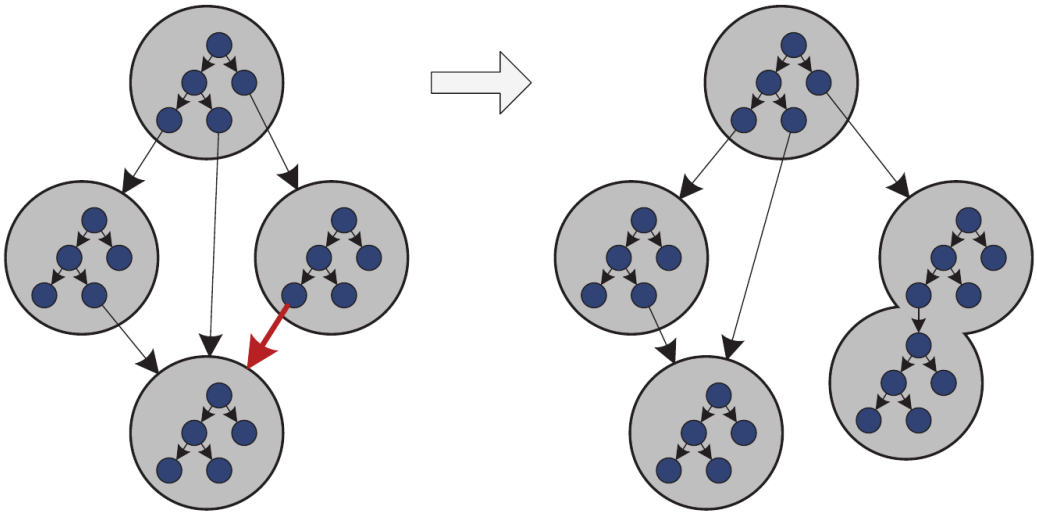

Figure 2.3: Method inlining in the dynamic call graph, taken from [19]

method inlining process, where the big circles represent methods, and the small circles represent AST nodes." [19]. This is an important optimization because it allows the compiler to optimize across the method boundaries, but also to specialize for call sites.

Compilation

Once the AST becomes stable, Truffle *partially evaluates* [2] the AST with the Graal Compiler, resulting in a highly optimized speculative machine code with inserted deoptimization points. If one of the guard statements of this machine code fails, the machine code is discarded and control is transferred back to the interpreter.

2.4.1 Truffle DSL

In this subsection, we will focus on Truffle from the perspective of the TLI developer. Truffle provides a Domain specific language (DSL) for specification of nodes, node specializations, guard statements and various other constructs required for a TLI [3]. The majority of constructs are implemented as Java annotations. We will describe some of these constructs very briefly. For a full reference, please refer to the javadoc located in [20] and tutorials contributed by third parties located in [21] and in [22].

Specialization

As mentioned before, a node typically corresponds to a Java class that extends `Node` abstract class. The different node specializations are achieved with `@Specialization` annotation on methods of this node. For example, the `+` operator in R can have this simplified implementation:

```
class AddOperatorNode extends Node {
    public abstract Object execute(Object vec1, Object vec2);
```

```
    @Specialization
    Object addIntegerVectors(RIntVector vec1, RIntVector vec2) { }

    @Specialization
    Object addRealVectors(RRealVector vec1, RRealVector vec2) { }
}
```

Truffle generates code for a class that extends `AddOperatorNode` and implements `execute` method. In the `execute` method, this generated class dispatches either to `addIntegerVectors` or to `addRealVectors` based on the types of the arguments, and it also keeps track of which particular specialization is active. We denote a specialization activation as *specialization instantiation*. The `@Specialization` annotation have various parameters to control the behavior of the specialization, e.g., with `guards` parameter we can specify further conditions that have to be satisfied in order to execute the corresponding specialization, and the `replaces` parameter specifies which specialization will be replaced by this specialization.

Cached

We can also *cache* various values inside specializations. In the example of `+` operator, we can have two specializations for integer vectors - one that has the length of the first vector cached, and the other that is uncached:

```
@Specialization(guards = "vec1.getLength() == cachedLen")
Object addIntVecCachedLen(RIntVector vec1, RInteVector vec2,
            @Cached("vec1.getLength()") int cachedLen) { ... }

@Specialization(replaces = "addIntVecCachedLen")
Object addIntVecUncached(RIntVector vec1, RIntVector vec2) { ... }
```

When a specialization is instantiated, all the `@Cached` statements are executed and the return values are cached in the generated class. Note that the `@Cached` statements are executed just once. We should insert a guard condition that checks whether the dynamic parameters (*dynamic parameters* are parameters without an annotation) have not changed since the specialization instantiation and, therefore, the cached value is the correct one. The example above is artificial and useless because the guard conditions are evaluated every time before a specialization is executed. Let us provide a more useful example, although still artificial:

```
@Specialization(guards = "vec == cachedVec")
Object someMethod(RIntVector vec,
            @Cached("vec") RIntVector cachedVec,
            @Cached("vec.getLength()") int cachedLen) {...}
```

This is useful if `getLength()` is an expensive operation - `getLength()` will be called just once in the specialization instantiation, and in the subsequent executions only references will be compared (the `vec == cachedVec` statement). Note that in this example we assume that the length of the `RIntVector` instance does not change.

Truffle library

A *Truffle library* is similar to a Java interface, where the implementations of that interface can use the Truffle DSL inside their methods. Note that we denote a method from the library as a *message* and we say that a class *exports* a library if it implements the messages of the library. There is a guide about Truffle libraries in [23].

Let us demonstrate this concept on an example of `com.oracle.truffle.api.interop.InteropLibrary` - a Truffle library that specifies the interoperability message protocol between Truffle languages. The simplified definition of `InteropLibrary` is as follows:

```
@GenerateLibrary
public class InteropLibrary extends Library {
    public abstract boolean isExecutable(Object receiver);
    public abstract Object execute(Object receiver, Object... args);
    public abstract boolean hasArrayElements(Object receiver);
    public abstract Object readArrayElement(Object receiver, long index);
    // More messages
    // ...
}
```

Let us have a Java class `RVector` that represents some R vector and exports `InteropLibrary`.

```
@ExportLibrary(InteropLibrary.class)
class RVector {
    @ExportMessage
    boolean hasArrayElements() {
        return true;
    }

    @ExportMessage
    Object readArrayElement(long i, @Cached(...) someCachedValue) {
        // return i-th element
    }
}
```

Example usage of this library would be:

```
@Specialization
Object useLibrary(Object vec,
        @CachedLibrary("vec") InteropLibrary vecInterop) {
    if (vecInterop.hasArrayElements(vec)) {
        return vecInterop.readArrayElement(vec, 0);
    }
}
```

If `vec` is `RVector`, then corresponding messages in `RVector` are called, and their cached values (if any) are initialized in the same way as for any other specialization.

We will use `InteropLibrary` in the implementation for invoking ALTREP methods.

2.5 FastR

FastR is a TLI of R. The fact that FastR is implemented with Truffle implicitly provides these features:

- Interoperability with other TLIs.
- High peak performance.
- Various tools for TLIs.

FastR aims to be compatible with GNU-R. There are some tests with expected output in R source repository that can be thought of as a Test compatibility kit (TCK) for R. FastR passes most of these tests. However, support for base R language is just one portion of the compatibility with GNU-R, another portion is support for packages, mostly native packages.

In previous sections, we noted that native packages use the R Internals API, or at least they should use this API. Unfortunately, it is common to see a native package that bypasses this API and uses the knowledge of internal structures layout for some reason [1].

In FastR, the semantics of the R language by defined as Truffle nodes in Java. However, FastR also has to handle native packages and calls to C functions from the R Internals API.

2.5.1 Native code evaluation

In this section, we will provide a high-level overview of native code evaluation in FastR. Remember that a native package has to contain a shared library with the compiled code as described in 2.1.1. In FastR, there are two possible implementations for an evaluation of the native code - the *NFI backend*, and the *LLVM backend*. NFI uses Java Native Interface (JNI) under the hood, which means that it calls the native code directly. NFI loads the shared library and looks up the corresponding function.

The LLVM backend is actually *Sulong* [24] - a Truffle interpreter of the LLVM bytecode. Being able to install and run all the native packages in Sulong is the ultimate goal of FastR, because it has many advantages over NFI:

- The performance is better because Sulong is yet another TLI and as such generates a Truffle AST that can be inlined into other ASTs and GraalVM Compiler can perform optimizations across the whole AST.
- The safety is potentially better because with Sulong we can intercept some memory accesses.
- The debugging of a native package is much easier because we can use just one tool to debug both R and C code and the calls between them.

FastR version 20.0.0 is not able to install and run all the packages in Sulong, there are still some fundamental native packages that are installed and run in NFI.

[1] `data.table` is an example

For Sulong to function properly, the shared library has to be compiled into LLVM bytecode rather than into machine code. FastR package installation process takes care of the compilation transparently.

Downcalls

A call from R code into native code via `.Call` or `.C` is denoted as a *downcall* in a sense that from R we *downcall* into the native code. A downcall in GNU-R is fairly straightforward as R is interpreted in C and has no moving GC, there is even no need to marshall arguments or to copy them into a different section of memory. In FastR this process is more complicated, as we need at least to convert the arguments to their native representation and keep them alive (protect them from the JVM GC) as long as the downcall does not return.

Further in the text, we will use the term *downcall return* for a return from the downcall, i.e., when the down-called native code stops execution and returns to the JVM (it might return some value but that is not important for this definition).

FastR has to simulate the behavior of the garbage collector (see section 2.1.4), which means that we have to make sure that those R Internals functions that are expected to not cause GC in GNU-R, should also not trigger GC in FastR. collector.

Upcalls

An *upcall* denotes a call from the native code (C or C++) to the JVM. The upcall typically happens when we call some function from the R Internals API. In FastR, the implementation of most of the functions from the R Internals API has this pattern:

```
int LENGTH(SEXP x) {
    return ((int (*) (SEXP)) callbacks[LENGTH_x])(x);
}
```

where `callbacks` is a table of pointers to functions that *callback* into JVM and `LENGTH_x` is an index into this table. This callback takes us to the implementation of `LENGTH` in FastR - which is implemented as a Truffle node. In fact, the majority of functions from the R Internals API are implemented as Truffle nodes.

Further in the text, we will use the term *upcall return* for a return from the upcall, i.e., when we return from the JVM to the native code.

From a downcall, we can do arbitrary many upcalls, and from an upcall we can do a downcall. The scheme of such interactions is depicted in Figure 2.4.

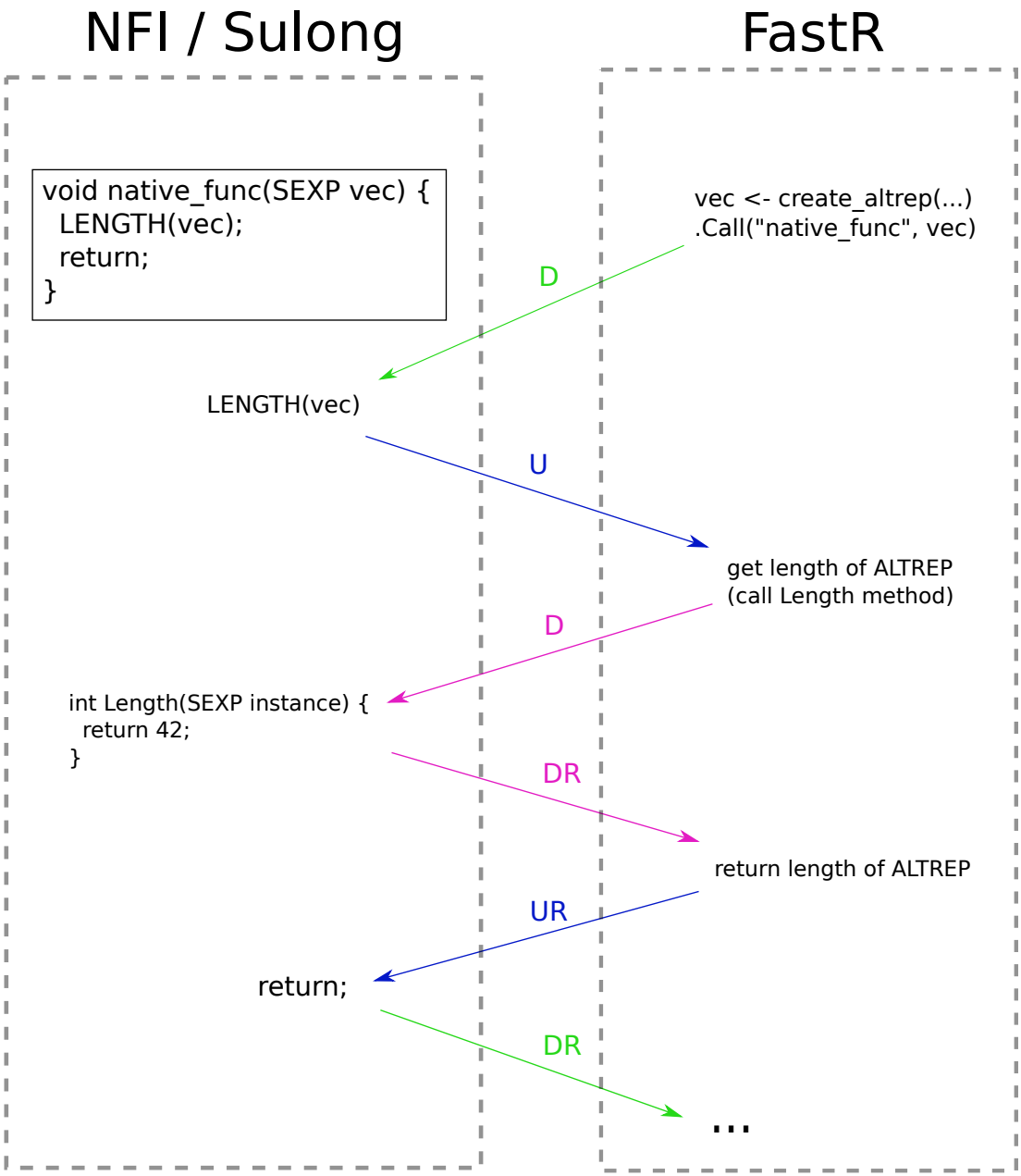

Figure 2.4: Scheme of downcalls/upcalls interaction
Legend: D (downcall), U (upcall), DR (downcall return), UR (upcall return)

3. Design and implementation

In this chapter, we will analyze the design and the possible implementation of ALTREP. As we know from section 2.2, ALTREP consists of the class definition API, and of some other functions (see section 2.2.3). We will start the analysis with the observations about the ALTREP methods, and then we will analyze possible support for the whole ALTREP object lifecycle, which is as follows:

- Create an ALTREP class descriptor.
- Register some methods on this descriptor.
- Create a new ALTREP instance, optionally with some instance data.

Later on, we will analyze the rest of the ALTREP API (see section 2.2.3).

3.1 ALTREP method contracts

ALTREP is a relatively new API and as is the case for the whole R Internals API, it is rather undocumented. GNU-R base developers themselves claim that the best - and many times the only - documentation is inferred by looking at the R codebase. This is not only time consuming, but in many cases also ambiguous. This section describes the *contracts* of various ALTREP methods (see section 2.2) infered from the GNU-R codebase. We will build on these contracts for the rest of the thesis.

Dataptr

Signature: `void * Dataptr(SEXP instance, Rboolean writeable)`

`Dataptr` forces the ALTREP instance to materialize - to return a pointer to valid memory with the data of the instance with the correct length. Most objects in R are immutable and once the data pointer for the object is returned to the caller, it should not change in the future. In other words: once a vector is materialized, the data for this vector stays at the same memory location. Consequently, the following code snippet should run without an error:

```
void *dataptr = DATAPTR(object);
// do something
// ...
assert(dataptr == DATAPTR(object));
```

For a specific ALTREP instance, it means that the `Dataptr` method should always return the same value. Therefore, we can cache the return value after the `Dataptr` method is invoked.

Sum

Signature: `SEXP Sum(SEXP instance, Rboolean narm)`

From the behavior of the `sum` R builtin function, implemented in `do_summary` C function in the `summary.c` source file, we infer that the `Sum` method is expected

to return the summary of the elements in the ALTREP instance, although it could theoretically return an arbitrary object. The analysis follows.

The `sum` R builtin function works as follows:

- If there is just one argument and it is an ALTREP instance, its `Sum` method is invoked.
- If there are more arguments they are all iterated one-by-one via the `ITERATE_BY_REGION` macro and summed together. This means that even if there are two ALTREP vectors as arguments, their `Sum` methods are not invoked.

By considering the behavior for more arguments mentioned in the second point, we infer that the `sum` builtin function expects the `Sum` ALTREP method to return the actual sum of all its elements.

This contract is not forced anywhere and if broken, some operations might return nonsensical results, as demonstrated in the following code snippet:

```
x <- altrep.with.bad.sum(...)
print(x) # [1, 2, 3]
y <- 1:3
sum(x) # 100 - altrep method invoked
sum(y) # 6
sum(x, y) # 12
sum(x, y) != sum(x) + sum(y)
```

Duplicate

Signature: `SEXP Duplicate(SEXP instance, Rboolean deep)`

Under normal circumstances, `Duplicate` just duplicates the object's data and attributes. There is no R function that can be called in order to duplicate an object - the interpreter does that transparently with copy-on-write semantics (see section 2.1). While there is no R function that would duplicate an object, there is a C function in the R Internals API `SEXP duplicate(SEXP object, Rboolean deep)` (see section 2.1.3).

When the `object` argument is an ALTREP, its `Duplicate` method is invoked and the `object`'s attributes are duplicated - the `Duplicate` method itself should not copy the attributes. When the `object` argument is not an ALTREP instance, or the ALTREP instance does not have the `Duplicate` method registered, the instance's data is copied into a new non-ALTREP vector.

In conclusion: `Duplicate` may return both ALTREP and non-ALTREP vectors with the same data as the original object.

3.2 Class descriptors

In this section, we will analyze the requirements for a data structure that will represent a class descriptor.

In GNU-R, the class descriptor is a raw vector with the class name and the package name as attributes (see section 2.2.1). This is an internal information and different access to class descriptors than via the ALTREP API is discouraged.

However, there is already at least one violation in the `vroom` package - in order to get the name of the class descriptor, the authors of `vroom` access directly the attributes of a class descriptor, with this code: `CAR(ATTRIB(ALTREP_CLASS(x)))`, where `x` is an ALTREP instance. In this particular case, it might be a temporary solution as there is currently no other way to get the name of a class descriptor via the ALTREP API. But as long as the ALTREP API does not provide a function for getting the name of the class descriptor, we should expect even more packages to access the attributes of class descriptor directly. Therefore, we have to make sure that class descriptors in FastR also have the appropriate attributes. This is not too difficult as it should not require any modifications of the existing codebase.

In GNU-R, the class descriptors form a hierarchy (as explained in section 2.2). This hierarchy, however, is not complete and new types might be added, or some might be removed. In [10], Luke Tierney (one of the authors of ALTREP) mentions that `ALTENV` might be added or `ALTVEC` might be removed. Moreover, new methods might be added to the existing types, which is the case for the `Match` method that was added after GNU-R version 3.5.0. Because of these ongoing changes, it seems like a good idea to implement the same hierarchy in FastR. In the future, it might be easier to stick to the modifications introduced in GNU-R if the hierarchy is the same.

Storage for class descriptors

We have to store the class descriptors inside some data structure that represents a global state of an R session.

In FastR, there is a class `RContext` that encapsulates the runtime state of an R session, e.g., R profile, paths to libraries, command-line options, etc. Class descriptors are also specific for a session, therefore `RContext` seems as a natural destination for some data structure containing the ALTREP class descriptors. More precisely, class descriptors are specific for a particular package, therefore some Java class that represents an R package might more suitable location for class descriptors. In FastR, it is `DLLInfo` class - a static inner class of `DLL` class. Since the implementation of serialization of ALTREP vectors and classes does not have a high priority, let us keep the implementation simple and put all the class descriptors inside `RContext`, rather than in `DLLInfo`.

Methods in class descriptors

In GNU-R, a class descriptor is a raw vector containing pointers to the methods (see section 2.2.1). It makes no sense for a native package developer to extract a particular method from some class descriptor and call it manually. Consequently, we do not have to care about the data of these raw vectors and leave them empty. In FastR, an ALTREP method might have two types, depending on which backend was used to load the corresponding shared library - either LLVM or NFI. References to methods of both of these types conform to the interop API (see Truffle libraries in section 2.4.1), which means that we can invoke such a method as easily as in this snippet:

```
Object method = ...;
InteropLibrary interop = InteropLibrary.getUncached(method);
Object retValueFromMethod = interop.execute(method, argsToMethod);
```

In conclusion, the references to the ALTREP methods can be stored as `Objects` in class descriptors.

Redefining methods In GNU-R, it is possible, although not very useful, to *redefine* an already registered method in a particular class descriptor. Caching methods is a good optimization in FastR, but we have to make sure that the method is not redefined while being in a cache. For this purpose, a Truffle assumption is appropriate. Note that in the future, the GNU-R core developers might disallow method redefinition.

3.3 Downcalls and upcalls

In this section, we will first describe the already existing downcall infrastructure of FastR (introduced in section 2.5.1) in greater detail, and then discuss how ALTREP-specific downcalls could fit into this infrastructure. For an ALTREP instance, we expect that the downcalls will happen frequently. Therefore, the design of the downcall infrastructure is essential.

In FastR, there exists a complex downcall infrastructure that handles all possible downcalls specified by R: `.Call`, `.Internal`, and `.Primitive` interfaces [6]. We will describe the downcall mechanism on an example.

3.3.1 Simple example

In Diagram 3.1 we can see an overview of the steps described below. The entry point for this example is located in the source file `CallAndExternalFunctions.java`. For the sake of simplicity, we will omit some details like package initialization code, package namespace declaration, etc. Consider this simple native function defined in `my_package`:

```
SEXP print_length(SEXP arg) {
    Rprintf("Length is: %d\n", LENGTH(args));
    return R_NilValue;
}
```

that just prints the length of the argument it receives, and this snippet of R code that calls the aforementioned native function:

```
arg <- 1:5 #Create an integer vector of length 5
.Call("print_length", arg, PACKAGE="my_package")
```

We start in R and allocate `arg`. It will be represented as `RIntVector`, which is a class in FastR representing an integer vector.

Now we evaluate the `.Call` downcall:

1. The symbol `print_length` is looked up in a loaded shared library from `my_package`.

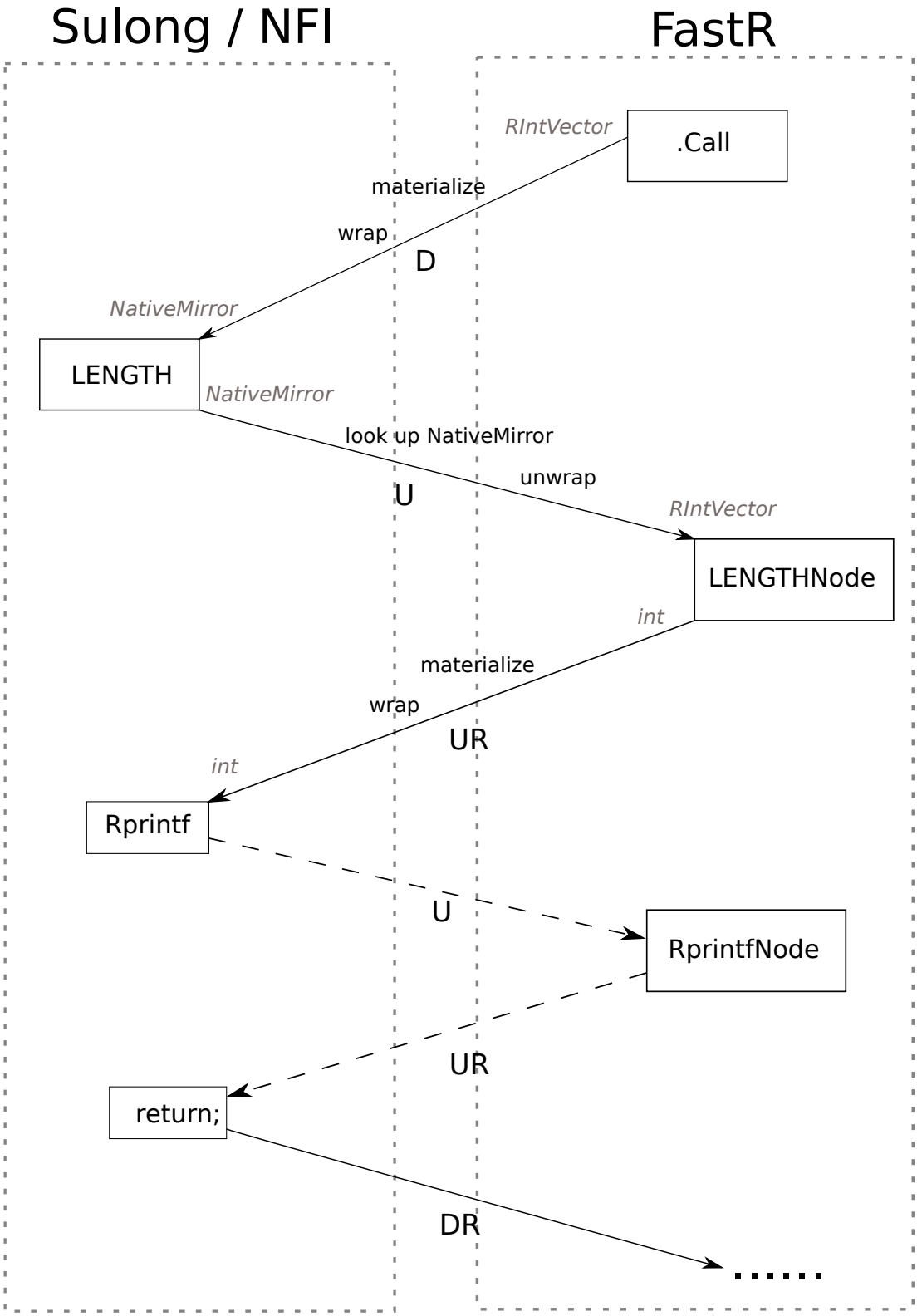

Figure 3.1: Diagram of downcalls and upcalls
Legend: D (downcall), U (upcall), DR (downcall return), UR (upcall return)

2. We have to keep track of the downcalls because, e.g., we have to keep the arguments passed to the downcall alive until the downcall returns. To do that, we call `RFFIContext.beforeDowncall`. Note that this method has a different implementation for NFI and LLVM.

3. We *materialize* `arg` with `FFIMaterializeNode`. Note that the materialization in context of `FFIMaterializeNode` has a different meaning than the materialization that we defined previously in this thesis. The materialization in context of `FFIMaterializeNode` means that all the primitive-type arguments are converted to vectors of length one. This is because the target of the `.Call` function receives all the arguments as `SEXP`. In the rest of the example, materialization will mean materialization with `FFIMaterializeNode`.

   In our case the materialization does not actually do anything - `arg` is already `RIntVector`.

4. We wrap `arg` in a `NativeMirror`. `NativeMirror` is a native representation of an `RBaseObject`, which is a Java object representing some R object. `NativeMirror` roughly corresponds to `SEXP` in the native code. `NativeMirror` is stored in a hash map and if it will get into some upcall later, we will retrieve it from the hash map. More specifically, the key in the hash map is used as the actual value of the pointer to the native code (the key is a `long` value). In this step, no native memory is allocated.

5. We execute the function `print_length` via `InteropLibrary` - `InteropLibrary.execute(nativeFunc)`. This step is different for NFI and LLVM backends. In NFI, the native function is invoked via JNI. In LLVM, the Sulong starts interpreting the root node of the AST corresponding to the native function.

Now we switch to the native code and start evaluating `print_length`. The `print_length` function has two upcalls - `Rprintf` and `LENGTH`. Every upcall is implemented as a native function that looks like this:

```
int LENGTH(SEXP x) {
    return ((int (*) (SEXP)) callbacks[LENGTH_x])(x);
}
```

where `callbacks` is a table of pointers to functions that callback into the JVM and `LENGTH_x` is an index into this table. Once we callback into the JVM, the JVM knows the mapping between callbacks and upcall nodes. Therefore, JVM invokes the corresponding upcall node. Upcall node receives a pointer as the argument and it looks up this pointer in the hash map of `NativeMirror`s, and unwraps the underlying object. Now we are in an upcall node (in our case `LENGTHNode`) with `RIntVector` as the argument.

We return to the native code and do another upcall from there - `Rprintf`. We will not explain the steps of this upcall as it is similar to the `LENGTH` upcall.

After the downcall returns to R, we discard the references to the objects that were allocated in the native code, and thus JVM's GC will be able to eventually collect the unused objects. Implemented in `RFFIContext.afterDowncall`.

3.3.2 Nativization

The aforementioned example used upcalls with primitive return values - `LENGTH` returns an integer and `Rprintf` does not return anything. The upcalls might return `SEXP` or a different kind of pointer. The upcalls that return a `SEXP` comply with the principles mentioned in the example - some Java object is wrapped into `NativeMirror` and the returned `SEXP` is represented as `NativeMirror`. However, for `DATAPTR`, `INTEGER`, and similar upcalls we have to return a pointer to valid native memory.

Consider this C code snippet:

```
SEXP vec = ...; // vec is an integer vector
int sum = 0;
int *dataptr = INTEGER(vec);
for (int i = 0; i < LENGTH(vec); i++) {
    sum += dataptr[i];
}
```

We will focus on the `INTEGER` upcall. In GNU-R, the `INTEGER` function returns a valid pointer to the vector's data. In FastR with NFI backend, the `INTEGER` upcall has to *nativize* the vector (a Java object), where *nativization* means allocating a native memory and copying the vector's data into that memory, i.e., it is a materialization of the vector into the native memory. However, with LLVM (Sulong) this does not have to be the case - the `INTEGER` upcall does not have to nativize the vector immediately, it can rather *delay* the nativization up to the point where a write into that memory occurs.

Here is how the delay of the nativization works. The `INTEGER(vec)` statement returns `RObjectDataPtr`, that is just a wrapper for the data of the actual vector. `RObjectDataPtr` is a type that exports `InteropLibrary` (see Truffle libraries in section 2.4.1). The `dataptr[i]` statement calls `readArrayElement` message on `RObjectDataPtr`, which retrieves the `i`-th element from the underlying vector. No native memory has been allocated.

For ALTREP vectors there are two options - we can either eagerly nativize or delay the nativization as mentioned above. In GNU-R, the materialization of an ALTREP instance happens once its `Dataptr` method is called, and `INTEGER` is an R Internals API function that dispatches to this method. Simply put: `INTEGER` calls `Dataptr` in GNU-R. Although discouraged, the `Dataptr` method might have some side effects and some part of the system might depend on the `Dataptr` method being called once `INTEGER` is called. To achieve the full compatibility with GNU-R, the eager nativization would be necessary.

The delayed nativization for ALTREP vectors is also possible, but more complicated and not as beneficial as delayed nativization for standard vectors. Let us use the code snippet at the beginning of section to explain. The main issue is what to do in the `dataptr[i]` statement. If `Elt` method is registered, we could dispatch to this method and avoid nativization. If `Elt` method is not registered, we need to nativize and obtain the element via `Dataptr` method. So for the delayed nativization to work for an ALTREP vector we need the `Elt` method registered for that vector. But is this kind of delay worth it? Invoking `Elt` method has certainly less overhead than invoking `Dataptr` for the first time,

but in the long run, invoking `Dataptr` once and then obtaining elements straight from the native memory is faster than invoking `Elt` method all the time.

In conclusion, FastR can delay the nativization of all types of vectors with LLVM. The delayed nativization for the standard vectors is already implemented, but we will not implement it for the ALTREP vectors.

3.3.3 Specifics for ALTREP

An ALTREP-specific downcall is a call from the JVM to some ALTREP method, that is, native function. The aforementioned downcall infrastructure supports `.Call` and `.C` R functions. All the parameters of the native functions called through `.Call` and `.C` R functions are expected to have the `SEXP` type, including the return type, if any. This is in contrast to ALTREP, where some methods have primitive type parameters, e.g., the `Elt` method has `R_xlen_t idx` parameter which is long or integer. This means that we cannot use the existing downcall infrastructure without some modifications, specifically, we need to ensure that not all the arguments are wrapped. We still need to call the methods in FastR that record the downcalls (`RFFIContext.beforeDowncall` and `RFFIContext.afterDowncall`) though, but they are, luckily, independent of the existing downcall infrastructure.

So the wrapping of the arguments and return types is incompatible for ALTREP downcalls and standard downcalls. Another incompatibility that might cause issues in the future is in performance. For ALTREP downcalls, we want to be able to cache the ALTREP method. This seems tricky to implement in the existing downcall infrastructure but is not complicated to implement from scratch - it requires just a few `@Cached` annotations and guard conditions.

Apart from the downcall infrastructure that would do just the low-level downcall into a specific ALTREP method, we also need some Truffle nodes that will dispatch to a particular ALTREP method based on the provided ALTREP instance. For example, the `Elt` method needs different handling for altstring and altinteger. This is implemented as nodes in `AltrepRFFI`. See javadoc for more information.

Finally, let us discuss the behavior of some upcalls for ALTREP instances. Consider the `INTEGER` upcall as an example, and suppose that we run FastR with LLVM. For the standard vectors, the `INTEGER` upcall returns `RObjectDataPtr` which is a representation that allows delayed nativization of these vectors. We have already decided that ALTREP instances will not implement delayed nativization. So the best return value for the `INTEGER` upcall for an ALTREP instance is just a wrapper for its data pointer (returned by its `Dataptr` method). In this way, Sulong will interpret the value returned from `INTEGER` as just a pointer, and interpret the statement `dataptr[i]` from the following snippet as access to native memory.

```
SEXP alt_vec = ...; // alt_vec is an altinteger
int sum = 0;
int *dataptr = INTEGER(alt_vec);
for (int i = 0; i < LENGTH(alt_vec); i++) {
    sum += dataptr[i];
}
```

To summarize, we have decided to design a new downcall infrastructure that is specific for ALTREP. The most important concepts we want to implement are:

- The ability to choose which arguments to wrap and whether to wrap the return value.
- Cache the target of the ALTREP method.
- Profile LLVM or NFI backend, i.e., whether to choose LLVM-specific or NFI-specific `RRFFIContext.beforeDowncall` and `RFFIContext.afterDowncall` methods.

Moreover, we have decided that we will not implement delayed nativization of the ALTREP instances.

3.4 ALTREP instances

In GNU-R, an ALTREP instance is a pairlist with instance data and a pointer to a class descriptor in the attributes (see section 2.2.1). From the ALTREP API perspective everything a developer can do with an ALTREP instance is:

- Create an instance with `R_new_altrep(descr, data1, data2)`.
- Read or modify the instance data (with `R_altrep_data1`, `R_altrep_data2`, `R_set_altrep_data1` and `R_set_altrep_data2`).
- Check whether an instance is of a particular class with `R_altrep_inherits(instance, class)`.

In FastR, there is no reason to make an ALTREP instance look like a pairlist, as there are no packages that access any ALTREP instance via the pairlist API, at least currently (GNU-R version 3.5.2). This gives us the freedom to represent an ALTREP instance similar to how standard vectors are represented - with `VectorDataLibrary`, which is a Truffle library (see section 2.4.1).

In FastR, a vector is just a wrapper for the data and the attributes. The data is represented with `VectorDataLibrary`, which contains messages for reading and writing the elements of the vector. This access is type-aware, i.e., there is a `getInt` message as well as a `getDouble` message. For the complete reference of `VectorDataLibrary`, please refer to the javadoc in the attachments. This design allows every class that represents some vector data to export `VectorDataLibrary`, and thus access all the possible vector's data in a unified way.

We can implement a class exporting `VectorDataLibrary` for each ALTREP type. In that way, an ALTREP vector would be represented as a standard vector, like `RIntVector`, with some ALTREP vector data. Consequently, the ALTREP vectors would be transparent to the rest of FastR and minimal modifications to the existing codebase would be required.

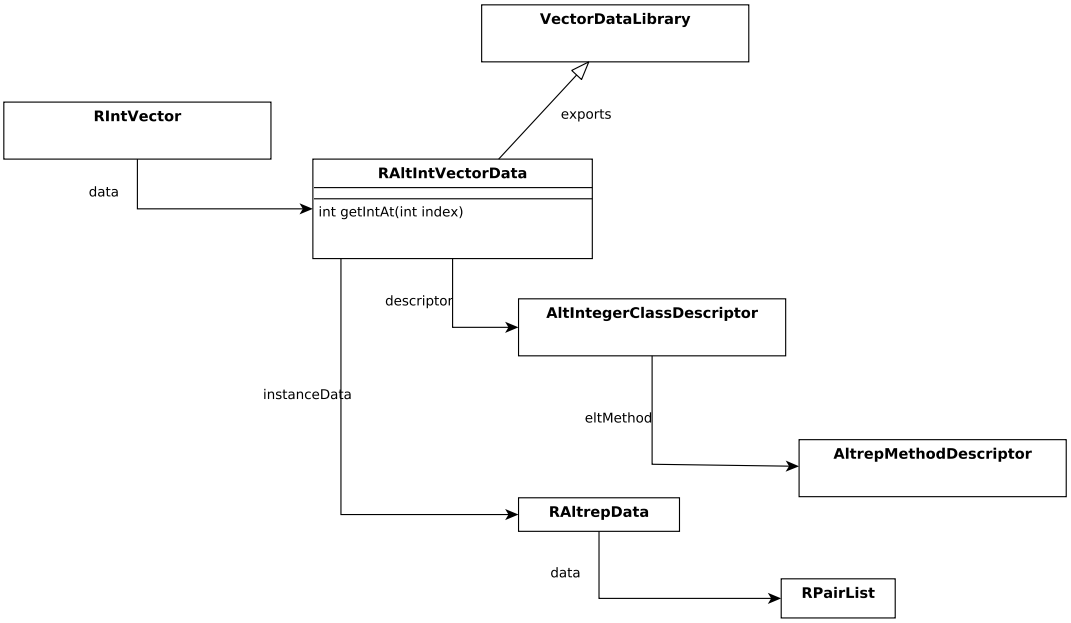

Figure 3.2: Diagram of ALTREP instance

3.4.1 ALTREP vector data

Let us focus on implementing the vector data for altintegers, other ALTREP types stick to similar principles and, therefore, have similar implementation. Most of the exported messages from `VectorDataLibrary` have to call various ALTREP methods, which means that we have to keep a reference to the class descriptor. Note that there might be more ALTREP instances with the same class descriptor. In Diagram 3.2 we can see the layout of an ALTREP instance in FastR:

- `RIntVector` has `RAltIntVectorData` as its field.
- `RAltIntVectorData` has a reference to the class descriptor.
- `AltIntegerClassDescriptor` has all the possible ALTREP methods as field. In Diagram, we depict only the `Elt` method.
- The `Elt` method is represented by `AltrepMethodDescriptor` - a Java class that encapsulates an ALTREP method along with information how to invoke it.
- `RAltIntVectorData` has an `instanceData` field that includes pairlist, because in GNU-R, the instance data is represented as pairlist (see section 2.2.1).
- See javadoc in the attachments for more information.

Below we enumerate the messages that we have to implement from `VectorDataLibrary` and provide a discussion about the implementation. Note that the enumeration is not complete - we omit the trivial messages.

getInt

The implementation of this message depends on whether the `Elt` ALTREP method is registered. If it is registered, we call this method, otherwise, we return an offset from the address obtained with the `Dataptr` method.

We also want to cache the return value from the `Dataptr` method. But this is the responsibility of `DataptrNode` in `AltrepRFFI`.

copy

First, we have to introduce `deep` parameter to the current implementation of `VectorDataLibrary.copy` method to better reflect the signature of `duplicate` from the R Internals API (see section 2.1.3). Note that `deep` parameter can be ignored in FastR because in GNU-R it is used only internally by the interpreter with no visible effects on the outside. However, the `deep` argument must still be forwarded to the ALTREP method.

Next, the implementation of this message will dispatch to the `Duplicate` ALTREP method if it is registered, otherwise, it will do the standard duplication. The return value from the `Duplicate` method needs some additional handling because this method might return `NULL`. So an ALTREP instance without registered `Duplicate` method, and an ALTREP instance with a `Duplicate` method that returns `NULL` will be duplicated in the same way.

We do not need to handle the attributes as they are handled by other nodes.

materialize

The materialization happens once `Dataptr` method is called. More specifically, nativization happens. As we cache the return value from the `Dataptr` method, we know whether it was called or not. Therefore, in the `materialize` message we call the `Dataptr` method only if it was not called yet.

asPointer

The `asPointer` message returns the address as `long`, therefore, it is sufficient to call the `Dataptr` method (if it was not called yet).

isComplete

The `isComplete` message returns true if the vector does not contain any `NA`s. This means that this message has the same semantics as the `No_NA` method, including lazy evaluation. So to implement the `isComplete` message, we dispatch to the `No_NA` method if it is registered, otherwise, we return false as the default.

isSorted

The `isSorted` message and the `Is_sorted` method have similar semantics but different signatures, and they are both lazily evaluated. It is, therefore, sufficient to dispatch to the `Is_sorted` method if it is registered. If it is not registered, the `isSorted` message returns false.

3.5 Rest of ALTREP API

Apart from the API that handles the creation of class descriptors, registration of the methods and instantiation of the ALTREP instances, there is an ALTREP API that can be used for any kind of vectors. Below we provide a list of these functions with their possible design. For the full signatures and the semantics of these functions, please refer to 2.2.3.

Note that all these upcalls, similarly to other upcalls, are implemented as Truffle nodes and, therefore, may take advantage of cached Truffle libraries, cached parameters, etc. We want to use Truffle libraries as much as possible - more specifically, we want to use `VectorDataLibrary` as much as possible. Therefore, in the implementation of the upcalls, we will try to primarily use some messages from `VectorDataLibrary` or add new ones.

`INTEGER_ELT`, `INTEGER_NO_NA`, and `INTEGER_IS_SORTED` are straightforward to implement, as they can dispatch to `getIntAt`, `isComplete`, and `isSorted` messages of `VectorDataLibrary` respectively.

INTEGER_GET_REGION

The most useful solution is to implement `getIntegerRegion` as a message to `VectorDataLibrary`. That way, we can provide a default implementation with reasonable performance. Moreover, ALTREP instances, i.e., ALTREP vector data can have an implementation of this message that considers whether the `Get_region` method is registered.

DATAPTR_OR_NULL

At the beginning of this section, we stated that we want to implement new messages in `VectorDataLibrary` if necessary. But only if such a message would have some use inside other parts of FastR as well. For example, `isSorted` is a very useful message inside `VectorDataLibrary` that is used inside many different parts of FastR. A message for `DATAPTR_OR_NULL` would be used only in this upcall, and nowhere else. Thus, we will not implement a new message for the `DATAPTR_OR_NULL` upcall. We will rather implement it as an isolated upcall node.

The documentation says that `DATAPTR_OR_NULL` function should return `NULL` rather than allocate any memory. In FastR, we represent R objects as Java object, but only some of them are nativized (see section 3.3.2). More specifically, only the objects for which the `DATAPTR` function was called are nativized. In other words, not all the R objects are materialized in the native memory in FastR. We either break the contract of the `DATAPTR_OR_NULL` function and nativize the vector, or we return `NULL` for many objects. We decided to stick to the contract of the `DATAPTR_OR_NULL` function and return `NULL` for not-yet-nativized objects. Note that this is in contrast to GNU-R which returns a valid pointer for the vast majority of the objects except for some ALTREP objects. This design decision will have non-trivial performance impact, as `DATAPTR_OR_NULL` function is used in `ITERATE_BY_REGION` macro (prefered way of iteration over a vector in the native code - see section 2.2.3). We will discuss the performance in section 4.1.

In conclusion, we decided that we will implement `DATAPTR_OR_NULL` as an isolated upcall node that, by default, returns `NULL` for many objects.

3.6 Tests

In this section, we will describe how the ALTREP implementation is tested.

3.6.1 Unit tests

Let us first discuss unit tests. Unit tests, as defined by [25], are used sparsely in FastR. The reason is that most of the implementation consists of Truffle nodes and it might be difficult to isolate and test one particular node, as many nodes require some parts of FastR to be initialized. In the end, it turns out that in order to allow FastR to be entirely covered by this kind of unit tests, it would require significant changes in the design.

assertEval

So taking one class or few classes, initialize them with some mock dependencies, and unit testing them conventionally is not a great option for FastR. FastR actually uses jUnit testing framework and inside the testing methods it uses `void assertEval(String input)` method. This method evaluates the `input` argument, which is some R code snippet, in FastR and in GNU-R, and then compares their outputs. When the outputs are more or less equal the test passes, otherwise, it fails. `assertEval` uses a lot of components - it constructs an AST from the given R code snippet, initializes the interpreter, and interprets the AST. From a software engineering perspective, these tests are rather integration tests than unit tests. And because `assertEval` is easy to use and has decent coverage, it is used frequently in FastR.

For ALTREP, `assertEval` is not a feasible option, because it can take only R expressions as the input, whereas for ALTREP we also need to evaluate some C statements, or even install a native package.

3.6.2 Package tests

FastR has to be able to install and run as many packages as possible in order to be compatible with GNU-R. In *package tests*, FastR tries to install certain package, run tests for this package, if there are any, or run at least examples of that package. In most of the packages, FastR uses the examples, demos, or tests published by the package authors to test the package. There are some packages, particularly those very popular on CRAN, for which we have some internal tests in FastR.

In an R package, tests are a collection of R scripts that uses, e.g., `stopifnot` R function to test some condition or even something more sophisticated like `RUnit`. Executing these tests in FastR without any errors is a good indicator of whether the package is supported by FastR. Packages that are not published with tests usually contain at least demos or examples, which are again a collection of R scripts. Executing the demo or an example in FastR is not enough - we also need to execute it in GNU-R and compare their outputs. The outputs do not necessarily have to be exactly the same - there might be some differences in producing various warning or informational messages. Only after the outputs match, we can safely claim that the package is supported by FastR.

In summary, a package test consists of executing the tests, examples or demos of the package by both FastR and GNU-R, and comparing their outputs.

Package tests for ALTREP

There are two native packages used to test ALTREP - `classtests`, and `altreprffitests`. Both of these packages aim to have as little dependencies as possible. They are listed as attachments in the thesis, as they represent a substantial contribution to the thesis.

classtests `classtests` is the native package used as a TCK for the ALTREP API. It gets an arbitrary object (not only ALTREP) as the input and it checks whether the object conforms to various ALTREP contracts, like those mentioned in section 3.1. More specifically, rather than just an object, it gets a factory function as the input and this function is executed before every test to create the object. The reason for a factory function is that in some tests we modify the values of the object return by the factory function. `classtests` package is implemented in C++ and contains a simple implementation of a unit testing framework with C++ Standard Template Library (STL) as the only dependency. The code and the comments of the tests inside `classtests` package serve also as a documentation of the ALTREP API that is absent in the core R. In the future, the `classtests` might be published on CRAN.

altreprffitests `altreprffitests` is a native package that tests whether ALTREP instances can be used transparently in R. It creates some ALTREP instances and use them inside various R statements that dispatch to the ALTREP methods, e.g., the `vec[[i]]` statement, where `vec` is an ALTREP instance, dispatches to the `Elt` or the `Dataptr` method . This package contains some ALTREP class definitions in C++ but all the tests are written in R.

A key difference from `classtests` is that `altreprffitests` tests more ALTREP instances with different methods at the same time, while `classtests` tests just one object. Moreover, `classtests` package may test non-ALTREP objects.

4. Results

In this chapter, we will describe some obstacles found in the implementation of altrep in FastR and how we managed to get across them. We will also describe how the performance of the implementation was measured and what kind of optimizations were implemented.

4.1 Benchmarks

Performance is one of the key features of FastR. This thesis aims to have the performance of various ALTREP usages comparable to the GNU-R counterpart.

In this chapter, we will describe benchmarks and their properties used to measure the performance of ALTREP implementation. In the whole chapter, we will target the LLVM backend of FastR.

The purpose of the benchmarks is to see what are the limitations of FastR compared to GNU-R and vice versa, and what are some possible future optimizations of FastR.

Note that all the benchmarks presented in this section are only *micro benchmarks*. The focus of the performance measurement is solely the ALTREP API, not the whole FastR. While we cannot conclude that the performance of the ALTREP API is generally better in FastR than in GNU-R, the microbenchmarks are still very valuable for various optimizations. Moreover, implementing standard benchmarks is currently impossible since `vroom` - the only real-world package that uses ALTREP - is not supported by FastR [1].

The benchmarks are designed to resemble a portion of the most typical workflow in an R application - iteration over a standard vector or ALTREP vector. If we iterate over an ALTREP vector, we invoke these ALTREP methods (for the list of all ALTREP methods refer to 2.2):

- `Elt`
- `Get_region`
- `Length`
- `Dataptr` or `Dataptr_or_null`

The list is roughly sorted by the frequency of the invocations - the methods mentioned in the beginning of the list have higher probability of being called more frequently than the methods mentioned in the end of the list. Note that the list of the methods is inferred from the `Itermacros.h` source file, i.e., from `ITERATE_BY_REGION` macro that is the most commonly used and preferred way of iteration over any vector (see 2.2.3). In a typical R application, these methods will be called most frequently.

The benchmarks are implemented with the goal of invoking the aforementioned ALTREP methods frequently. In GNU-R, the ALTREP method invocation does not present a large barrier - it is very similar to virtual method dispatch,

[1] The reason why `vroom` package and other features are not implemented is described in the following sections

which basically means double pointer dereference. But in FastR, the ALTREP method invocation presents many difficulties. Some of these difficulties were already mentioned in section 3.3. The reason for such a design of the benchmarks is that we want to know the impact of frequently calling a native function from a managed context. Theoretically, with Sulong this impact should be minimal.

There are 19 micro benchmarks in total for the ALTREP. Most of the benchmarks contain one loop or two nested loops that iterate over an ALTREP vector and does some trivial arithmetic operation on them. Some benchmarks are implemented in R, and some are implemented in C. Every benchmark has a *baseline* benchmark that it is compared to. Baseline benchmarks just iterate over a normal integer vector and do the same arithmetic operation as normal benchmarks.

Another goal of the implementation of benchmarks is *simplicity*. FastR is a huge and complex system and as such, various of its functionality may interfere with the measurements. Therefore, we stick to some best practices in the performance evaluation:

- Try to allocate as little memory as possible during the measurement.
- Keep the code that does the actual measurement as simple as possible, without unnecessary if branches and functions that do some side effects.
- Measure every benchmark in a separate process.

Note that the benchmarks only use altinteger and integer vectors. This should be enough for our purposes - altreal, altlogical, altraw and altcomplex vectors all have the same API and similar behavior with the difference just in the primitive values as their data. Altstring vectors do not have benchmarks because they are not used so frequently as numerical vectors and in cases where they are used, we do not expect high performance.

4.1.1 Description of the benchmarking framework

There exists a benchmarking framework for FastR that is integrated with `mx` [26] , along with many benchmarks from various sources. But these benchmarks along with the framework are part of the proprietary repository and as such cannot be published in this thesis. Therefore, the author implemented the framework along with the benchmarks just for ALTREP, from scratch.

The benchmarking framework is also integrated with `mx` - it is basically a Python script. This framework takes care of various setups and cleanups necessary for the benchmarks - like installation of some packages containing some ALTREP class definitions and native benchmarks.

One of the advantages of the integration with `mx` is that the `mx` fetches some configuration data for us - like ids of specific commits or CPU architecture of the host machine, and writes this information inside the output file together with some other measurement-specific information.

In the end, we can use this framework from other scripts to automatically obtain some measurement data and analyze them.

4.1.2 Parameters

This is a list of the possible parameters for benchmarks:

- The length of the vector that is iterated.
- The count of nested loop iterations for the benchmarks that have a nested loop - these are the native benchmarks.
- The durations of the warmup and measure phases.
- The ALTREP class used as a vector that is iterated over.
- Whether to run the benchmark in GNU-R or in FastR.

ALTREP classes

NativeMemVec `NativeMemVec` that is a simple wrapper around a region of native memory. During the class initialization, certain size (given as the parameter) is `malloc`ed and the pointer to this memory is saved and later returned as `Dataptr`. The `Elt` method simply returns an element from this native memory, and the return value of the`Length` method is obvious.

VecWrapper `VecWrapper` is the ALTREP class that wraps some standard vector and in its methods dispatches to the standard functions, e.g., the `Elt` method calls the `INTEGER_ELT` function on the standard vector. The standard vector is kept as an instance data and retrieved in each method with the `R_altrep_data1` function. `VecWrapper`, while still being simple and artificial class, resembles a real-world class more than `NativeMemVec` as it is very likely that real classes access their instance data in every method. Note that we do not use `data2` instance data, because accessing them has the same performance overhead as accessing `data1`.

4.1.3 Measured data

In this section, we will provide an analysis of the measurements of the aforementioned benchmarks. This analysis is an excerpt from the Jupyter notebook inserted in the attachment.

The data were measured with 300 seconds warmup, 25 seconds of the measurement phase, and 10^7 data length. It is our experience that for relatively small code sizes involved in the benchmarks, most compilation and optimization activities of Truffle and Graal have already finished by the 300 seconds warmup time. We chose 10^7 data length experimentally - with smaller data, the benchmarks tend to invoke too many operations per second which may negatively impact the measurements - the overhead for the loop in which the benchmark is executed would have non-trivial overhead. On the other hand, larger data causes some benchmarks to run too long, not even finishing one operation per one second.

In the rest of the text, the *score* represents throughput - the number of operations per second (higher is better).

In Table 4.1 we can see the measured data. In the upper part, we can see the benchmarks that have a better score in FastR and in the lower part we can see

Name	FastR score	GNU-R score
iterate-native-mem-vec	16.797427	1.034098
iterate-native-mem-vec-baseline	17.879114	1.135255
iterate-dataptr	1.409227	0.823220
iterate-dataptr-baseline	17.719459	1.134758
iterate-elt-baseline	18.197590	1.146264
native-dataptr-before-native-mem-vec	17.432065	3.939710
native-dataptr-before-baseline	736.569754	7.092524
native-dataptr-inside-baseline	8.388018	2.948475
native-iter-by-region-native-mem-vec	8.451850	7.772932
native-iter-by-region-vec-wrapper	4.218315	2.207535
iterate-elt	0.792010	0.918453
native-dataptr-before-vec-wrapper	0.345072	2.272194
native-dataptr-before-vec-wrapper-elt	0.361011	2.283996
native-dataptr-inside-native-mem-vec	0.188310	0.687725
native-dataptr-inside-vec-wrapper	0.182996	0.684873
native-dataptr-inside-vec-wrapper-elt	0.094913	0.886068
native-iter-by-region-vec-wrapper-elt	0.442845	4.134619
native-iter-by-region-vec-wrapper-get-region	3.270665	17.983512
native-iter-by-region-baseline	6.149700	24.723014

Table 4.1: Measured data

	FastR	GNU-R
Mean	45.204	4.411
Median	4.218	2.207
Variance	28081.729	41.135
Standard deviation	167.576	6.413

Table 4.2: Statistical characteristics of FastR and GNU-R scores

the benchmarks that have a better score in GNU-R. In the table 4.2 we can see some statistical characteristics of both GNU-R and FastR scores.

From the names of the benchmarks we can infere some parametrization of the benchmark:

- Benchmarks ending with `-baseline` are baseline benchmarks.
- The prefix `iterate` represents benchmarks that are implemented in R.
- The prefix `native` represents a benchmark implemented in C in an attached native package.
- The suffix `-native-mem-vec` represents benchmarks that get `NativeMemVec` class as parameter (defined later).
- The suffix `vec-wrapper` represents benchmarks that get `VecWrapper` class as parameter (defined later).
  - `vec-wrapper-elt` is a `VecWrapper` with `Elt` and `Dataptr` methods registered.
  - `vec-wrapper-get-region` is a `VecWrapper` with `Dataptr` and `Get_region` methods registered.

The code of the benchmarks is located in the attachment.

We can see that most of the baseline benchmarks are faster in FastR. This is an expected result because the baseline benchmarks iterate over standard vectors and FastR is able to optimize such an iteration heavily.

4.1.4 Outliers

In this section, we will discuss various outliers, both in FastR and GNU-R. As subsections, we will provide names of benchmarks that we consider outliers and explain why do they behave in such a way.

Let us start with an outlier in the baseline benchmarks.

native-iter-by-region-baseline

The `native-iter-by-region-baseline` benchmark with FastR score 6.1497 and GNU-R score 24.723014 is the only baseline that is slower in FastR. Let us see the simplified C code of the benchmark with expanded macros:

```
int acc = 0;
for (int iter = 0; iter < ITERATIONS; iter++) {
  const int *dataptr = DATAPTR_OR_NULL(instance);
  // GNU-R falls into this branch
  if (px != NULL) {
    for (R_xlen_t k = 0; k < LENGTH(instance) - 1; k++) {
      acc += dataptr[k] - dataptr[k + 1];
    }
  }
  // Fast-R falls into this branch
  else {
```

```
    int __buff__[512];
    for (R_xlen_t outer_idx = 0; outer_idx < LENGTH(instance);
         outer_idx += 512)
    {
      INTEGER_GET_REGION(instance, outer_idx, 512, __buff__);
      int *dataptr = __buff__;
      for (R_xlen_t k = 0; k < 512 - 1; k++) {
        acc += dataptr[k] - dataptr[k + 1];
      }
    }
  }
}
```

where `ITERATION` is a parameter to the benchmark and `instance` is a standard integer vector.

The `DATAPTR_OR_NULL` function returns a pointer to data only if the data is already materialized, otherwise it returns `NULL`. In other words, the `DATAPTR_OR_NULL` function should not allocate any memory. This implies that in GNU-R, this function always returns a valid pointer for all the standard vectors, because the memory for such vectors had already been allocated. For FastR, this is not the case. We decided that, by default, the `DATAPTR_OR_NULL` function will return `NULL` for all the standard vectors that had not been allocated in the native heap before (see section 3.3.3).

The behavior of the `DATAPTR_OR_NULL` function in FastR and GNU-R implies that GNU-R executes the `if`-branch, and FastR the `else`-branch (as denoted in the snippet with comments). The for loop in the first branch is very simple and the C compiler will probably vectorize it. On the other hand, the second branch contains two nested for loops with calls to the `INTEGER_GET_REGION` function, and in FastR this is an upcall, which has some overhead even for Sulong.

iterate-native-mem-vec

The `iterate-native-mem-vec` benchmark with FastR score 16.797 and GNU-R score 1.034 is the greatest outlier in the standard benchmarks. This benchmark is written in R and this is the code (`iterate-native-mem-vec.r`):

```
acc <- 0L
for (i in 1:(length(instance) - 1)) {
    acc <- acc + instance[[i]] - instance[[i + 1L]]
}
```

Where `instance` is an ALTREP vector with `NativeMemVec` class (see 4.1.2). The `instance[[i]]` statement invokes the `Elt(i)` method, which is a C function that returns `((int *)native_ptr)[i]`, where `native_ptr` is a pointer to previously `malloc`-ed memory.

In FastR, the whole benchmark is inlined - the AST representing the `Elt` method consists of just a few nodes and is inlined into the AST representing the whole benchmarking function. At the runtime, Graal and Truffle do optimizations on the entire AST, no matter whether some part of the AST originates from C and the other part from R (see 2.4).

In GNU-R, the `instance[[i]]` statement is actually a double dispatch. Although the call target of this dispatch is always the same (the `NativeMemVec` class), GNU-R cannot optimize this at the runtime.

Note that `native-dataptr-before-native-mem-vec` (with FastR score 17.432 and GNU-R score 3.939) is faster in FastR because of the same reasoning, even though it is written in C.

native-iter-by-region-vec-wrapper-elt

The `native-iter-by-region-vec-wrapper-elt` benchmark with FastR score 0.442 and GNU-R score 4.134 is the greatest outlier for GNU-R - being almost ten times faster than FastR. This is not very surprising - we already explained why `native-iter-by-region-*` benchmarks are generally faster in GNU-R.

native-iter-by-region-vec-wrapper

The `native-iter-by-region-vec-wrapper` benchmark with FastR score 4.218 and GNU-R score 2.207 is the only `native-iter-by-region-*` benchmark that is faster in FastR. Remember that this benchmark iterates over the `VecWrapper` ALTREP class that has only `Dataptr` and `Length` methods registered. This means that we retrieve the elements of the ALTREP vector via the `Dataptr` method, rather than via the `Elt` method. And the result of the `Dataptr` method is cached (see section 3.4).

4.2 Future work

- Support for `vroom` package, and other CRAN packages using ALTREP.
  - This is the most important future work as the compatibility of FastR with GNU-R is measured in the count of the packages that are supported on both platforms.
- Serialization and deserialization of ALTREP class descriptors and instances.
  - This functionality is not yet fully supported even in GNU-R.
- Add `Match` ALTREP method
  - As mentioned in section 2.2, the ALTREP classes are not final, some of them might be merged, and more methods might be added. The `Match` method is an example of an ALTREP method that was added after the initial ALTREP class hierarchy was introduced.

Conclusion

In this thesis, we designed and implemented ALTREP for FastR, and although the current implementation is not sufficiently complete to run the complex existing CRAN packages (notably `vroom`), it shows that supporting ALTREP in FastR is feasible and can give some benefits compared to the ALTREP implementation in GNU-R.

We managed to optimize the implementation of ALTREP, along with some other parts of FastR, so that it is faster than GNU-R in some benchmarks presented in this thesis. We concluded that the results are acceptable with respect to the current limitations of Sulong.

The modifications of the current codebase of FastR were minimal - the ALTREP vectors are simply new types of `VectorData`, therefore they share the same interface as other types of vectors. In GNU-R, on the other hand, the ALTREP vectors need special treatment since the dispatch to ALTREP methods has to be done explicitly. Every builtin function in GNU-R that supports ALTREP has the following scheme:

```
SEXP some_builtin_function(SEXP arg) {
    SEXP ans = R_NilValue;
    if (ALTREP(arg)) {
        // Dispatch to some ALTREP method.
        ans = ALTREP_DISPATCH(arg, Method);
        if (ans != R_NilValue) {
            return ans;
        }
    }
    // Do standard operation as for standard vectors.
}
```

This is not the case for FastR, where we do not have to introduce special handling for ALTREP objects.

We designed two native packages to test the ALTREP implementation. Both of these packages are standalone packages with minimal dependencies, usable by both GNU-R and FastR. One of the packages, `classtests`, takes an object, not necessarily an ALTREP object, and tests whether it conforms to certain R Internals API contracts. It serves both as a TCK for the ALTREP API, and as a documentation that is currently missing in the sources of GNU-R. It might be published to CRAN in the future, as we believe that it might provide valuable information to the R community. Another package, `altreprffitests`, creates many ALTREP instances with different methods and checks whether they behave correctly in some R statements. The package tests represent a large contribution to the thesis.

The overall count of Source lines of code (SLOC) is approximately 11000 with 7000 lines of Java code, 3000 lines of C++ code, and 1000 lines of R code, not counting some Python code for the Jupyter notebook and the benchmarking framework.

Glossary

class descriptor *Raw* vector containing pointers to methods of a specific altrep object. . 13, 15, 27–30, 35, 36, 38, 47

JVM Compiler Interface A Java API for querying JVM data structures and profile information. . 18, 19

Acronyms

AOT Ahead of Time. 18

AST Abstract syntax tree. 4, 19–21, 24, 32, 39, 46

DSL Domain specific language. 21, 23

GC Garbage collector. 8–10, 25, 32

JDK Java Development Kit. 18, 19

JIT Just in Time. 4

JNI Java Native Interface. 24, 32

JVM Java Virtual Machine. 19, 25, 32, 34

PGO Profile-guided optimizations. 19

SLOC Source lines of code. 48

STL C++ Standard Template Library. 40

TCK Test compatibility kit. 24, 40, 48

TLI Truffle language implementation. 19–21, 24

Bibliography

[1] Oracle. Fastr github repository. URL `https://github.com/oracle/fastr/`.

[2] Thomas Würthinger, Christian Wimmer, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Christian Humer, Gregor Richards, Doug Simon, and Mario Wolczko. One vm to rule them all. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*, Onward! 2013, page 187–204, New York, NY, USA, 2013. Association for Computing Machinery.

[3] Christian Humer, Christian Wimmer, Christian Wirth, Andreas Wöß, and Thomas Würthinger. A domain-specific language for building self-optimizing ast interpreters. *SIGPLAN Not.*, 50(3):123–132, September 2014.

[4] R Core Team. R Language Definition. URL `https://cran.r-project.org/doc/manuals/r-devel/R-lang.html`.

[5] Jython. URL `https://en.wikipedia.org/wiki/Jython`.

[6] R Core Team. R Internals. URL `https://cran.r-project.org/doc/manuals/r-devel/R-ints.html`.

[7] CRAN repository. URL `https://cran.r-project.org/`.

[8] R Core Team. Write R Extensions. URL `https://cran.r-project.org/doc/manuals/r-devel/R-exts.html`.

[9] Tomas Kalibera Luke Tierney, Gabe Becker. ALTREP: Alternative representations for R objects. URL `https://svn.r-project.org/R/branches/ALTREP/ALTREP.html`.

[10] Luke Tierney. ALTREP: Alternative representations of Basic R Objects - a talk. URL `https://homepage.divms.uiowa.edu/~luke/talks/uiowa-2018.pdf`.

[11] Jim Hester. vroom package. URL `https://github.com/r-lib/vroom`.

[12] Jim Hester. vroom benchmark vignette. URL `https://vroom.r-lib.org/articles/benchmarks.html`.

[13] Want Jiefei. AltWrapper package. URL `https://github.com/Jiefei-Wang/AltWrapper`.

[14] Thomas Würthinger, Christian Wimmer, Christian Humer, Andreas Wöß, Lukas Stadler, Chris Seaton, Gilles Duboscq, Doug Simon, and Matthias Grimmer. Practical partial evaluation for high-performance dynamic language runtimes. *SIGPLAN Not.*, 52(6):662–676, June 2017.

[15] John Rose. JDK Enhancement Proposal 243: Java-Level JVM Compiler Interface. URL `https://openjdk.java.net/jeps/243`.

[16] Lukas Stadler, Thomas Würthinger, and Hanspeter Mössenböck. Partial escape analysis and scalar replacement for java. In *Proceedings of Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '14, page 165–174, New York, NY, USA, 2014. Association for Computing Machinery.

[17] Gilles Duboscq, Thomas Würthinger, Lukas Stadler, Christian Wimmer, Doug Simon, and Hanspeter Mössenböck. An intermediate representation for speculative optimizations in a dynamic compiler. In *Proceedings of the 7th ACM Workshop on Virtual Machines and Intermediate Languages*, VMIL '13, page 1–10, New York, NY, USA, 2013. Association for Computing Machinery.

[18] Google. V8 JavaScript engine. URL `https://v8.dev/`.

[19] Thomas Würthinger, Andreas Wöß, Lukas Stadler, Gilles Duboscq, Doug Simon, and Christian Wimmer. Self-optimizing ast interpreters. *SIGPLAN Not.*, 48(2):73–82, October 2012.

[20] Oracle. Truffle javadoc. URL `https://www.graalvm.org/truffle/javadoc/`.

[21] Cristian Esquivias. Writing a language in truffle - mumbler. URL `http://cesquivias.github.io/blog/2014/10/13/writing-a-language-in-truffle-part-1-a-simple-slow-interpreter/`.

[22] Stefan Marr. Add graal jit compilation to your jvm language in 5 easy steps. URL `https://stefan-marr.de/2015/11/add-graal-jit-compilation-to-your-jvm-language-in-5-easy-steps-step-1/`.

[23] Truffle library tutorial. URL `https://github.com/oracle/graal/blob/master/truffle/docs/TruffleLibraries.md`.

[24] Manuel Rigger, Matthias Grimmer, and Hanspeter Mössenböck. Sulong - execution of llvm-based languages on the jvm: Position paper. In *Proceedings of the 11th Workshop on Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems*, ICOOOLPS '16, New York, NY, USA, 2016. Association for Computing Machinery.

[25] Wikipedia: unit tests. URL `https://en.wikipedia.org/wiki/Unit_testing`.

[26] Mx github repository. URL `https://github.com/graalvm/mx`.

List of Figures

List of Tables

A. Attachments

A.1 Electronic attachments

The directories in the archive uploaded in Student Information System as electronic attachment of the thesis.

- `altrep-benchmarks` ... The benchmarking framework (integrated in `mx`) along with all the benchmarks used in the thesis.
- `altrep_demo_package` ... A rather small package containing just one simple ALTREP class for the purpose of demonstration.
- `altrep_tests_package` ... A `classtests` native package used for testing the ALTREP.
- `fastr` ... An author's fork of the FastR repository.
  - `fastr/documentation/dev/altrep.md` ... The documentation of the ALTREP implementation.
- `javadoc` ... The generated javadoc of FastR, Sulong, and Truffle.
- `notebooks` ... The directory containing the Jupyter notebook along with a script used by the Jupyter notebook and mesaured data.

Note that `fastr`, `altrep_tests_package`, and `altrep-benchmarks` directories are backed by Git VCS. All of these repositories are on the author's Github (Akirathan) and checked-out on `dipl-thesis` tag. Up until the date of the thesis submission, the ALTREP is not yet merged into FastR upstream. We expect that until the thesis defense, it will be merged into FastR.

The author's contribution to the FastR codebase can be seen with `git diff master HEAD`, or, more conveniently, at this link `https://github.com/oracle/fastr/compare/master...Akirathan:pm/altrep` (accessed 30.7.2020).

Dockerfile

Besides other build dependencies, FastR has GNU-R version 3.6.1 as a dependency and, by default, it builds the GNU-R from sources. And GNU-R version 3.6.1 requires specific versions of some libraries. Therefore, the simplest way to demonstrate the thesis is to use Docker. We have included the Dockerfile based on Ubuntu 20.04 that installs and downloads all the dependecies, builds FastR, and installs the ALTREP testing packages along with a `altrep_demo_package`. In the Dockerfile, the part that builds the Graal compiler is not included so the overall build takes less time. Note that even without the Graal compiler, the build of the whole Dockerfile takes approximately 18 minutes on the author's laptop. We need to build the Graal compiler only if we want to run the benchmarks.

The user can also download the Docker image from the author's DockerHub at `https://hub.docker.com/r/akirathan/fastr-altrep`. The image has 5GB.

The entry point of the Dockerfile is on purpose the default shell as there are more ways to start FastR - with Graal compiler or without Graal compiler. To start FastR without Graal compiler, simply run `mx r` - this will run FastR with LLVM backend. To run FastR with Graal compiler, we first have to build the compiler with:

```
cd $GRAAL_HOME/compiler
mx build
```

Then run `mx --dynamicimport graal/compiler r` to start FastR with Graal compiler.

Run demo

To run the demo, paste the following commands in R (run R with `mx r`):

```
library(altrepdemo)
demo("altrep_usage")
```

Run tests

There are `classtests.sh` and `altreprffitests.sh` scripts to run the `classtests` and `altreprffitests` package tests respectively.

Unit tests can be run with `mx rutdefault` to run most of the unit tests, or `mx rutsimple` to run just a subset of all the unit tests, which takes approximately 20 minutes.

All the package tests can be run with `mx pkgtest`, however we do not recommend to actually run them on a laptop as they take several dozens of hours to run for the first time, before the packages are cached.

Run benchmarks

For benchmarks, we first have to build the compiler with:

```
cd $GRAAL_HOME/compiler
mx build
```

then run some ALTREP benchmark with this form:

```
cd $FASTR_HOME
mx --dynamicimport altrep-benchmarks,graal/compiler \
        benchmark altrep:<benchmark_name>
```

Please refer to the

```
$PMAREK_HOME/dev/altrep-benchmarks
```

directory for the names and specifications of the benchmarks.

Note that because of the recent upstream changes to `mx` benchmarking suite, the ALTREP benchmarking framework does not currently work in its outdated version. However, we expect to update and fix the ALTREP benchmarking framework soon.