UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

# Master's Thesis

**Martin Holm Cservenka**
Department of Computer Science
University of Copenhagen
djp595@alumni.ku.dk

# Design and Implementation of Dynamic Memory Management in a Reversible Object-Oriented Programming Language

**Supervisors:** Robert Glück & Torben Ægidius Mogensen

# Revision History

| Revision | Date | Author(s) | Description |
|---|---|---|---|
| 0.1 | 2017-05-02 | Martin | ROOPL++ design and heap design discussion |
| 0.2 | 2017-05-16 | Martin | Updated front-page logo and expanded heap design section |
| 0.3 | 2017-05-24 | Martin | Added Fragmentation and Garbage subsection. Added assumption about available heap manipulation subroutines for heap layout section. Added explanations of listings. Expanded Buddy-memory `get_free` algorithm with early idea for heap splitting/merging and heap growth |
| 0.4 | 2017-06-24 | Martin | Extended grammar in section 2.1. Added sections 2.2 to 2.5. Addressed Chapter 3 feedback. Rewrote pseudo-code algorithms in EXTENDED JANUS. |
| 0.5 | 2017-10-11 | Martin | Finalized a few sections in chapter 1. Update chapter 3 to reflect work done on compiler. |
| 0.6 | 2017-10-12 | Martin | Added first version of section 3.5 |

# Table of Contents

# List of Figures

# CHAPTER 1

# Introduction

## 1.1 Object-Oriented Programming

Object-oriented programming (OOP) has for many years been the most widely used programming paradigm as reflected in the popular usage of object-oriented programming languages, such as the C-family languages, `Java`, `PHP` and in recent years `Javascript` and `Python`. Using the OOP core concepts such as *inheritance*, *encapsulation* and *polymorphism* allows complex systems to be modeled by breaking the system into smaller parts in form of abstract objects.

## 1.2 Reversible Computing

## 1.3 Reversible Object-Oriented Programming

The high-level reversible language ROOPL (Reversible Object-Oriented Programming Language) was introduced in late 2016 and is, to our knowledge, the first of its kind [4]. The language extends the design of previously existing reversible imperative languages with object-oriented programming language features such as user-defined data types, class inheritance and subtype-polymorphism. As a first, ROOPL successfully integrates the object-oriented programming (OOP) paradigms into the reversible computation setting using a static memory manager to maintain garbage-free computation, but at cost of "programmer usability" as objects only lives within `construct` / `deconstruct` blocks, which needs to be predefined, as the program call stacks is required to be reset before program termination.

## 1.4 Motivation

ROOPL's block defined objects and lack of references are problematic when write complex, reversible programs using OOP methodologies. It has therefore been proposed to extend and partially redesign the language with dynamic memory management in mind, such that these shortcomings can be addressed, ultimately increasing the usability of reversible OOP. Work within the field of reversible computing related to heap manipulation [1], reference counting [7] and garbage collection [8] suggests that a ROOPL extension is feasible.

## 1.5 Thesis Statement

TBD

## 1.6 Outline

TBC

# CHAPTER 2

## The ROOLP++ Language

With the design and implementation of the REVERSIBLE OBJECT-ORIENTED PROGRAMMING LANGUAGE (ROOPL), the first steps into the uncharted lands of Object-Oriented Programming (OOP) and reversibility was taken. In this chapter, we will present ROOPL++ , the natural successor of ROOPL, improving the language's object instantiation by letting objects live outside `construct / deconstruct` blocks, allowing complex, reversible programs to be written using OOP methodologies. As with its predecessor, ROOPL++ is purely reversible and each component of a program written in ROOPL++ is locally invertible. This ensures no computation history is required nor added program size for backwards direction program execution.

Inspired by other language successors such as C++ was to C, ROOPL++ is a superset of ROOPL, containing all original functionality of its predecessor, extended with new object instantiation methods for increased programming usability and an array type.

TODO

**Figure 2.1:** Example ROOPL++ program

## 2.1 Syntax

A ROOPL++ program consists, analogously to a ROOPL program, of one or more class definitions, each with a varying number of fields and class methods. The program's entry point is a nullary main method, defined exactly once and is instantiated during program start-up. Fields of the main object will serve as output of the program, exactly as in ROOPL.

**ROOPL++ Grammar**

| | | | |
|---|---|---|---|
| $prog$ | ::= | $cl^+$ | (program) |
| $cl$ | ::= | **class** $c$ (**inherits** $c$)$^?$ $(t\ x)^*$ $m^+$ | (class definition) |
| $d$ | ::= | $c$ \| **vector**<$t$> | (class and array) |
| $t$ | ::= | **int** \| $c$ \| **vector**<$t$> | (data type) |
| $m$ | ::= | **method** $q$ **(**$t\ x,\ \ldots,\ t\ x$**)** $s$ | (method) |
| $s$ | ::= | $x\ \odot$ **=** $e$ \| $x$ **<=>** $x$ \| $x[e]\ \odot$ **=** $e$ | (assignment) |
| | \| | **if** $e$ **then** $s$ **else** $s$ **fi** $e$ | (conditional) |
| | \| | **from** $e$ **do** $s$ **loop** $s$ **until** $e$ | (loop) |
| | \| | **construct** $d\ x\quad s\quad$ **destruct** $x$ | (object block) |
| | \| | **local** $t\ x = e\quad s\quad$ **delocal** $t\ x = e$ | (local variable block) |
| | \| | **new** $d\ x$ \| **delete** $d\ x$ | (object con- and destruction) |
| | \| | **call** $q$ **(**$x,\ \ldots,\ x$**)** \| **uncall** $q$ **(**$x,\ \ldots,\ x$**)** | (local method invocation) |
| | \| | **call** $x$**::**$q$ **(**$x,\ \ldots,\ x$**)** \| **uncall** $x$**::**$q$ **(**$x,\ \ldots,\ x$**)** | (method invocation) |
| | \| | **skip** \| $s\ s$ | (statement sequence) |
| $e$ | ::= | $\overline{n}$ \| $x$ \| $x[e]$ \| **nil** \| $e\ \otimes\ e$ | (expression) |
| $\odot$ | ::= | **+** \| **-** \| **^** | (operator) |
| $\otimes$ | ::= | $\odot$ \| ***** \| **/** \| **%** \| **&** \| **\|** \| **&&** \| **\|\|** \| **<** \| **>** \| **=** \| **!=** \| **<=** \| **>=** | (operator) |

**Syntax Domains**

| | | |
|---|---|---|
| $prog \in$ Programs | $s \in$ Statements | $n \in$ Constants |
| $cl \in$ Classes | $e \in$ Expressions | $x \in$ VarIDs |
| $t \in$ Types | $\odot \in$ ModOps | $q \in$ MethodIDs |
| $m \in$ Methods | $\otimes \in$ Operators | $c \in$ ClassIDs |

**Figure 2.2:** Syntax domains and EBNF grammar for ROOPL++

The ROOPL++ grammar extends ROOPL's grammar with a new dynamic array type in form of **vector**s and a new object lifetime option in form of working with objects outside of blocks, using the **new** and **delete** approach. Furthermore, the local block extension proposed in [4] has become a standard part of the language. Class definitions remains unchanged, an consists of a **class** keyword followed by a class name. Subclasses must be specified using the **inherits**

keyword and a following parent class name. Classes can have any number of fields of any of the data types, including the new vector type. A class definition is required to include at least one method, defined by the **method** keyword followed by a a method name, a comma-separated list of parameters and a body.

Reversible assignments for integer variables and vector elements uses similar syntax as JANUS assignments, by updating a variable through any of the addition (+=), subtraction or bitwise XOR operators. As with JANUS, when updating a variable $x$ using any of said operators, the right-hand side of the operator argument must be entirely independent of $x$ to maintain reversibility. Usage of these reversible assignment operators for object or vector variables are undefined.

ROOPL++ objects can be instantiated in two ways. Either using object blocks known from ROOPL, or by using the **new** statement. The object-blocks have a limited lifetime, as the object only exists within the **construct** and **destruct** segments. Using **new** allows the object to live until program termination, if the program terminates with a **delete** call. By design, it is the programmers responsibility to deallocate objects instantiated by the usage of **new**.

The methodologies for argument aliasing and its restrictions on method on invocations from ROOPL carries over in ROOPL++ and object fields are as such disallowed as arguments to local methods to prevent irreversible updates and non-local method calls to a passed objects are prohibited. The parameter passing scheme remains call-by-reference and the ROOPL's object model remains unchanged in ROOPL++ .

## 2.2 Object Instantiation

Object instantiation through the **new** statement, follows the pattern of the mechanics of the **construct**/**destruct** blocks from ROOPL, except for the less limited scoping and lifetime of objects. The mechanisms of the statement

$$\textbf{construct}\ c\ x \quad s \quad \textbf{destruct}\ x$$

are as follows:

1. Memory for an object of class $c$ is allocated. All fields are automatically zero-initialized by virtue of residing in already zero-cleared memory.
2. The block statement $s$ is executed, with the name $x$ representing a reference to the newly allocated object.
3. The reference $x$ may be modified by swapping its value with that of other references of the same type, but it should be restored to its original value within the statement block $s$, otherwise the meaning of the object block is undefined.
4. Any state that is accumulated within the object should be cleared or uncomputed before the end of the statement is reached, otherwise the meaning of the object block is undefined.
5. The zero-cleared memory is reclaimed by the system.

The statement pair consisting of

**new** $c$ $x$ $\quad s \quad$ **delete** $c$ $x$

could be considered an *fluid* block, meaning we can have overlapping blocks. We can as such initialize an object $x$ of class $c$ and an object $y$ of class $d$ and destroy $x$ before we destroy $y$, a feature that was not possible in ROOPL. The mechanisms of the **new** statement are as follows

1. Memory for an object of class $c$ is allocated. All fields are automatically zero-initialized by virtue of residing in already zero-cleared memory.
2. The block statement $s$ is executed, with the name $x$ representing a reference to the newly allocated object.

and the mechanisms of the **delete** statement are as follow

1. The reference $x$ may be modified by swapping its value with that of other references of the same type, but it should be restored to its original value before a **delete** statement is called on $x$, otherwise the meaning of the object deletion is undefined.
2. Any state that is accumulated within the object should be cleared or uncomputed before the **delete** statement is executed, otherwise the meaning of the object block is undefined.
3. The zero-cleared memory is reclaimed by the system.

The mechanisms of the **new** and **delete** statements are, essentially, a split of the mechanisms of the **construct**/**destruct** blocks into two separate statements. As with ROOPL, fields must be zero-cleared after object deletion, otherwise it is impossible for the system to reclaim the memory reversibly. This is the responsibility of the of the programmer to maintain this, and to ensure that objects are indeed deleted in the first place. A **new** statement without a corresponding **delete** statement targeting the same object further ahead in the program is undefined.

## 2.3 Vector Model

Besides asymmetric object lifetimes, ROOPL++ also introduces reversible, dynamic arrays in form of the **vector** type. While ROOPL on featured integers and custom data types in form of classes, one of its main inspirations, JANUS, implemented static, reversible arrays [11].

While ROOPL by design did not include any data storage language constructs, as they are not especially noteworthy nor interesting from an OOP perspective, they do generally improve the expressiveness of the language. For ROOPL++ these were decided to be part of the core language as array, especially multi-dimensional, dynamic arrays become interesting from a memory management perspective.

ROOPL++ 's dynamic arrays expand upon the array model from JANUS. Arrays are index by integers, starting from 0. In JANUS, only integer arrays were allowed and arrays were one-dimensional. In ROOPL++ arrays of any type can be defined, meaning either integer arrays,

custom data types in form of class array or from other array types and have as such potential to be multi-dimensional.

Array element accessing is permitted using the bracket notation expression presented in JANUS. Accessing an out-of-bounds index is undefined. Vector instantiation and element assignments, aliasing and circularity is described in detail in the following section.

Vectors can contain elements of different classes if, say class $A$ and $B$ both inherit from some class $C$ and vector $x$ is a **vector**<$C$> type. In this case, the array can hold elements of type $A$, $B$, and $C$.

## 2.4 Vector Instantiation

Vector instantiation uses the **new** and **delete** keywords to reversibly construct and destruct vector types. In irreversible OOP languages, instantiation and insertion/deletion of elements are usually executed in the the following steps for empty, dynamic arrays

1. A small amount of memory is reserved for the overhead of an array
2. Once the array is non-empty, a pointer for the overhead place in memory points to the beginning of the data.
3. As elements are inserted, the data memory is expanded to fit new elements, if we need indexes higher than the current size of the array.
4. As the highest indexed elements are deleted, the data memory is collapsed to reduce the overall size of the array object.

Consider the statement

**new vector**<**int**> $x$

in which we reserve memory for an **int vector** with the mechanics following mechanics

1. A small amount of memory is reserved for the overhead of an array.
2. On the first element insert, e.g. $x[n]$ += 1, a block of memory suitable for storing $|[0, 1, ..., n]|$ integer values are reserved, with the newly updated value of $x[n]$ inserted and uninitialized cells zero-cleared, and a pointer from the overhead to the data is linked.
3. If a value is inserted at, say, $x[n + 1]$, a similar action is dispatched, but deallocating the old block after allocating the new one first.
4. When the highest indexed cell element is zero-cleared, a allocation action is dispatched for obtaining a smaller block for our array data, while the old, larger block is freed.

In ROOPL++ , we only allow instantiation of empty, dynamic arrays. As shown above, Integer elements are assigned using one of the reversible assignment operators. For class **vector**s and **vector vectors**, we assign cell elements a little differently. We make use of the **new** and **delete** statements, but instead of specifying which variable should hold the newly created/deleted object or array, we specify which array cell it should be stored in. The mechanisms of these

assignments are equal to the example for **int vector**s; we grow the array on assignments to higher-than-current-max indexes and we shrink on zero-clearing of highest-indexed element.

`TODO: Code example of assigning` **new** $c$ $x[3]$ `to empty array`

As with ROOPL++ objects instantiated outside of **construct**/**destruct** blocks, **vector**s must be deleted before program termination to reversibly allow the system to reclaim the memory. Before deletion of a vector, all its elements must be zero-cleared such that the array is shrunk to the empty array.

Consider the statement

**delete vector**<**int**> $x$

with the following mechanics

1. The reference $x$ may be modified by swapping, assigning cell element values and zero-clearing cell element values, but must be restored to a 0-element sized array before the **delete** statement. Otherwise, the meaning of the statement is undefined.
2. If the reference $x$ is the empty array upon the **delete** statement execution, the zero-cleared memory is reclaimed by the system.

With reversible, dynamic arrays of varying types and dimensionality, we must be extremely careful when updating and assigning values, to ensure we maintain reversibility and avoid irreversible statements. Therefore, when assigning or updating integer elements with one of the reversible assignment operators, we prohibit the cell value from being reference on the right hand side, meaning the following statement is prohibited

$$x[5] \mathrel{+}= x[5] + 1$$

However, we do allow other, initialized (non-zero-cleared array elements) to be referenced in the right hand side of the statement.

## 2.5 Local blocks

# CHAPTER 3

# Compilation

The following chapter will present the considerations and designs for the reversible heap manager and the schemes used in the process of translating ROOPL++ to the reversible low-level machine language PISA.

## 3.1 Dynamic Memory Management

Native support of complex data structures is a non-trivial matter to implement in a reversible computing environment. Variable-sized records and frames need to be stored efficiently in a structured heap, while avoiding garbage build-up to maintain reversibility. A reversible heap manager layout has been proposed for a simplified version of the reversible functional language RFUN and later expanded to allow references to avoid deep copying values [1, 10, 7].

The following section presents a discussion of various heap manager layouts along with their advantages and disadvantages in terms of implementation difficulty, garbage build-up and the OOP domain.

### 3.1.1 Fragmentation

An important matter to consider when designing a heap layout for dynamic memory allocation is efficient memory usage. In a stack allocating memory layout, the stack discipline is in effect, meaning only the most recently allocated data can be freed. This is not the case with heap allocation, where data can be freed regardless of allocation order. This feature comes with the consequence of potential memory fragmentation, as blocks are being freed in any order [6].

We distinguish different types of fragmentation as internal or external fragmentation.

#### Internal Fragmentation

Internal fragmentation occurs in the memory heap when part of an allocated memory block is unused. This type of fragmentation can arise from a number of different scenarios, but mostly it originates from *over-allocating*.

An example of *over-allocating* would be a scenario where we are allocating memory for an object of size $m$ onto a simple, fixed-sized block heap, where the fixed block size is $n$ and $m \neq n$. If $n > m$, internal fragmentation would occur of size $n - m$ for every object of size $m$ allocated in said heap. If $n < m$, numerous blocks would be required for allocation to fit our object. In this case the internal fragmentation would be of size $m + n - (m + n) \mod n$ per allocated object of size $m$.

TODO

**Figure 3.1:** Example of internal fragmentation due to *over-allocation*

Figure 3.1 visualizes the examples of internal fragmentation build-up from *over-allocating* memory.

Intuitively, internal fragmentation can be prevented by ensuring that the size of the block(s) being used for allocating space for an object of size $m$ either match or sums to this exact size.

#### External Fragmentation

External fragmentation materializes in the memory heap when a freed block becomes partly or completely unusable for future allocation if it, say, is surrounded by allocated blocks but the size of the freed block is too small to contain any object on its own.

This type of fragmentation is generally a more substantial cause of problems than internal fragmentation, as the size of external fragmentation blocks usually are larger than the ones created from internal fragmentation, depending on the heap implementation (In a layout using variable-sized blocks of, say, size $2^n$, the internal fragment size becomes considerable for large $n$s).

Non-allocatable external fragments become a problem when there it is impossible to allocate space for a large object as a result of too many non-consecutive blocks scattered around the heap, caused by the external fragmentation. Physically, there is enough space to store the object, but not in the current heap state. In this scenario we would need a garbage collector to clean the heap and relocate blocks in such a manner that the fragmentation disperses.

TODO

**Figure 3.2:** Example of external fragmentation

### 3.1.2 Memory Garbage

In the reversible setting it should be our goal to return the memory to its original state after program termination.

Traditionally, in non-reversible programming languages, freed memory blocks are simply re-added to the free-list during deallocation and no modification of the actual data stored in the block is done. In the reversible setting we must return the memory block to its original state after the block has been freed (e.g. zero-cleared), to uphold the time-invertible and two-directional computational model.

In heap allocation, we maintain one or more free-lists to keep track of free blocks during program execution, which are stored in memory, besides the heap representation itself. These free-lists can essentially be considered garbage and as such, they must also be returned to their original state after execution. Furthermore, the if the heap grows during execution, it should be returned to its original size.

Returning the free-list(s) to their original states is a non-trivial matter, which is highly dependent on the heap layout and free-list design. Axelsen and Glück introduced a dynamic memory manager which allowed heap allocation and deallocation, but without restoring the free-list to its original state in [1]. Axelsen and Glück argue that an unrestored free-list is to be considered harmless garbage in the sense that the free-list left after termination is equivalent to a restored free-list, as it contains the same blocks, but linked in a different order, depending on the order of allocation and deallocation operations performed during program execution.

This intuitively leads to the question of garbage classification. In the reversible setting all functions are injective. Thus, given some $input_f$, the injective function $f$ produces some $output_f$ and some $garbage_f$ (e.g. garbage in form of storing data in the heap, so the free list changes, the heap grows, etc.). Its inverse function $f^{-1}$ must thus take $f$'s $output_f$ and $garbage_f$ as $input_{f^{-1}}$ to produce its output $output_{f^{-1}}$ which is $f$'s $input_f$. However, in the context of reversible heaps, we must consider all free-lists as of "equivalent garbage class" and thus freely substitutable with each other, as injective functions still can drastically change the block layout, free-list order, etc. during its execution in either direction. Figure 3.3 shows how any free-list can be passed between a function $f$ and its inverse $f^{-1}$.

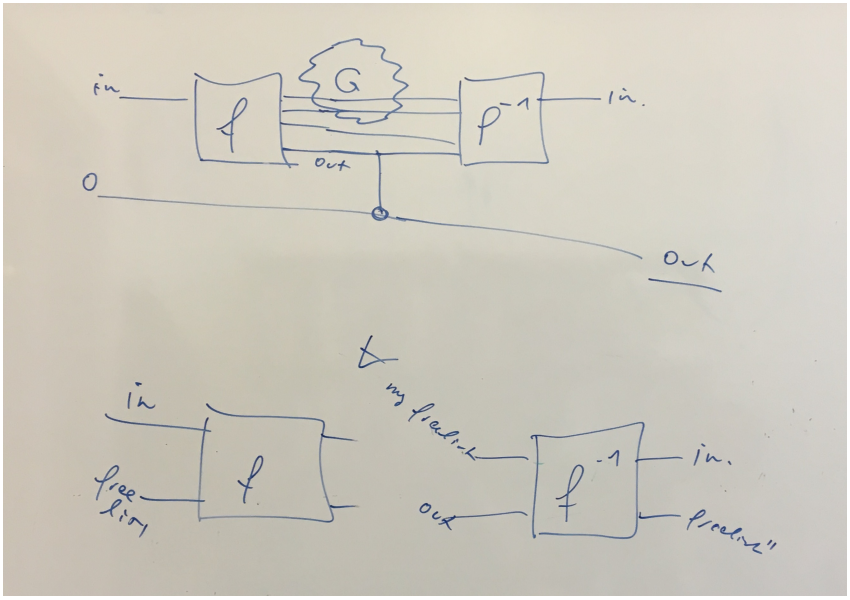


**Figure 3.3:** All free-lists are considered equivalent in terms of injective functions (Temporary photo)

### 3.1.3 Heap manager layouts

Heap managers can be implemented in numerous ways. Different layouts yield advantages when allocating memory, finding a free block or when collecting garbage. As our goal is to construct a

garbage-free heap manager, our finalized design should emphasize and reflect this objective in particular. Furthermore, we should attempt to allocate and deallocate memory as efficiently as possible, as merging and splitting of blocks is a non-trivial problem in a reversible setting.

For the sake of simplicity, we will not consider the the issue of retrieving memory pages reversibly. A reversible operating system is a long-term dream of the reversible programmer and as reversible programming language designers, we assume that ROOPL++ will be running in an environment, in which the operating system will be supplying memory pages and their mappings. As such, the following heap memory designs reflect this preliminary assumption, that we always can query the operating system for more memory.

Historically, most object-oriented programming languages utilize a dynamic memory manager during program execution. In older, lower-level languages such as C, memory allocation had to be stated explicitly and with the requested size through the `malloc` statement and deallocated using the `free` statement. Modern languages, such as C++, JAVA and PYTHON, *automagically* allocates and frees space for objects and variable-sized arrays by utilizing their dynamic memory manager to dispatch `malloc`- and `free`-like operations to the operating system and managing the obtained memory blocks in private heap(s) [5, 9, 2]. The heap layout of these managers vary from language to language and compiler to compiler.

Previous work on reversible heap manipulation has been done for reversible functional languages in [1, 3, 8].

For the sake of simplicity in the following heap layout pseudo-code outlines, we assume access to the following subroutines (with parameter passing), inspired by the body of Axelsen and Glück's `get_free` subroutine.

In order to reversibly allocate a block, we assume access to the subroutine `allocate_block` which, given the register $r_{cell}$ where we want the address of the allocated block stored and the register $r_{flp}$ containing the free-list pointer, effectively pops the head of the free-list to $r_{cell}$. Listing 3.1 shows the subroutine.

```
1 procedure allocate_block(r_cell, r_flp)
2     EXCH r_cell r_flp
3     SWAP r_cell r_flp
```

**Listing 3.1:** `allocate_block` subroutine

### Memory Pools

Perhaps the simplest layout for a heap manager would be memory pooling. In this design, memory is allocated from "pools" of fixed-size blocks regardless of the actual size of the record. The advantages of a heap layout following this approach would lie in the simplicity of its implementation, as a simple linked list of identical-sized free cells would need to be maintained.

If we assume a our fixed-sized blocks are of size $n$ machine words, the following `get_free` subroutine allocates and deallocates memory cells for a record of size $m$.

```
1 // Check if we have enough free blocks to hold the object
2 if (sizeof r_flp >= m + (n - (m % n)))
3 then
4     call allocate_block(r_cell, r_flp)
```

```
5     // Code for clearing m + (n - (m % n)) - n next blocks
6 else
7     // Code for growing heap
8 fi (r_cell != 0)
```

**Listing 3.2:** Allocating and deallocating records of size $m$ using block of a fixed size $n$. Code modified from [1]

Listing 3.2 shows the modified `get_free` subroutine for allocating and deallocating memory blocks in a memory pool layout. If the free-list has enough blocks available to hold the record of size $m$, the first $m + (n - (m\%n))$ ($m$ rounded up to nearest number dividable by $n$) blocks will be removed from the free list. If the free list is empty, the heap needs to be expended through some subroutine (Code not provided). The rounding of $m$ to nearest number dividable by $n$ could be computed at compile time.

A huge disadvantage to using fixed-sized memory blocks is the external fragmentation that occurs when freeing blocks, if the program's objects are not of the same size. When freeing a number of blocks in the middle of a section of allocated blocks, external fragmentation occurs, which becomes a problem if we need to allocate space for a large record but only have small sections of fixed-blocks available, scattered throughout the heap. A garbage collector could solve this issue, but is a non-trivial matter to implement.

### One Heap Per Record Type

This layout uses multiple heaps, one per record type in the program. During compilation, classes would be analyzed and a heap for each class would be created.

The advantage of this approach would be less fragmentation, as each allocation is tailored as closely as possible to the size of the record obtained from a static analysis during compilation.

The obvious disadvantage is the amount of book-keeping and workload associated with growing and shrinking a heap and its neighbours, in case the program requests additional memory from the operating system. In real world object-oriented programming, most classes features a small number of fields, very rarely more than 16. As such, multiple heaps of same record size would exist, which intuitively seems inefficient. Additional, small helper classes would spawn additional heaps and additional book-work, making the encapsulation concept of OOP rather unattractive, for the optimization-oriented reversible programmer.

### One Heap Per Power-Of-Two

A different approach as to having one heap per record type, would be having one heap per power-of-two until some arbitrary size. Using this approach, records would be stored in the heap which has a block size of a power-of-two closes matching to the record's size. This layout is a distinction from the "one heap per record type" as it still retains the size-optimized storing idiom but allows the heaps to contain records of mixed types. For programs with a large amount of small, simple classes needed to model some system, where each class is roughly the same size, the amount of heaps constructed would be substantially smaller than using one heap per record type, as many records will fit within the same heap. Implementation wise, the number of heaps can be determined at compile time. Furthermore, to ensure we do not end up with heaps of very large

memory blocks, an arbitrary power-of-two size limit could be set at, say, 1 kb . If any record exceeds said limit, it could be split into $\sqrt{n}$ size chunks and stored in their respective heaps. This approach does, however, also suffer from large amount of book-keeping and fiddling when shrinking and growing adjacent heaps.

`Algorithm: Similar to buddy memory?`

`TODO: Graphics`

#### Shared Heap, Record Type-Specific Free Lists

A natural proposal, considering the disadvantages of the previously presented design, would be using a shared heap instead of record-specific heaps. This way, we ensure minimal fragmentation when allocating and freeing as the different free lists ensures that allocation of an object wastes as little memory as possible. By only keeping one heap, we eliminate the growth/shrinking issues of the multiple heap layout.

There is, however, still a considerable amount of book-keeping involved in maintaining multiple free-lists. The bigger the number of unique classes defined in a program, the more free-lists we need to maintain during execution. If the free-lists are not allowed to point at the same free block (which they intuitively shouldn't in order to ensure reversibility), a program with, say one hundred different classes of size 2, would require a hundred identical free lists.

#### Buddy Memory

The Buddy Memory layout utilizes blocks of variable-sizes of the power-of-two, typically with one free list per power-of-two using a shared heap. When allocating an object of size $m$, we simply check the free lists for a free block of size $n$, where $n \geq m$. Is such a block found and if $n > m$, we split the block into two halves recursively, until we obtain the smallest block capable of storing $m$. When deallocating a block of size $m$, do the action described above in reverse, thus merging the blocks again, where possible.

This layout is somewhat of a middle ground between the previous three designs, addressing a number of problems found in these. The Buddy Memory layout uses a single heap for all record-types, thus eliminating the problems related to moving adjacent heaps reversibly in a multi-heap layout. To prevent multiple, identical free-lists (e.g. free-lists pointing to same size blocks) occurring from having one free-list per record-type, we instead maintain free-lists per power-of-two.

The only drawback from this layout is the amount of internal fragmentation. As we only allocate blocks of a power-of-two size, substantial internal fragmentation follows when allocating large records, i.e. a allocating a block of size 128 for a record of size 65. However, as most real world programs uses much smaller sized records, we do not consider this a very frequent scenario.

Implementation-wise, this design would require doubling and halving of numbers related to the power-of-two. This action translates well into the reversible setting, as a simply bit-shifting directly gives us the desired result.

```
1  procedure double(int target)
2      local int current = target
3      target += current
4      delocal int current = target / 2
5
6
7  procedure malloc(int p, int object_size, int free_lists[], int next_block[])
8      local int counter = 0
9      local int csize = 2
10     call malloc1(p, object_size, free_lists, counter, csize, next_block)
11     delocal int csize = 2
12     delocal int counter = 0
13
14
15 procedure malloc1(int p, int object_size, int free_lists[], int counter, int csize, int next_block[])
16     if (csize < object_size) then
17         counter += 1
18         call double(csize)
19         call malloc1(p, object_size, free_lists, counter, csize, next_block)
20         uncall double(csize)
21         counter -= 1
22     else
23         if free_lists[counter] != 0 then
24             p += free_lists[counter]
25             free_lists[counter] -= p
26
27             // Swap head of free list with p's next block
28             free_lists[counter] ^= next_block[p]
29             next_block[p] ^= free_lists[counter]
30             free_lists[counter] ^= next_block[p]
31         else
32             counter += 1
33             call double(csize)
34             call malloc1(p, object_size, free_lists, counter, csize, next_block)
35             uncall double(csize)
36             counter -= 1
37             free_lists[counter] += p
38             p += csize
39         fi free_lists[counter] = 0 || p - csize != free_lists[counter]
40     fi csize < object_size
```

**Listing 3.3:** The Buddy Memory algorithm implemented in extended Janus.

Listing 3.3 shows the buddy memory algorithm implemented in extended Janus. For simplification in object sizes are rounded to the nearest power-of-two and we only allow allocations using the heads of the free lists. The body of the allocation function is executed recursively until a free block larger or equal to the size of the object has been found. Once found, said block is popped from the free list. If the block is larger than the object we are allocating (rounded to nearest power-of-two), the block is split recursively until a block the desired size is obtained.

**Figure 3.4:** Heap memory layouts (Draft)

## 3.2 ROOPL++ Memory Layout

ROOPL++ builds upon its predecessor's memory layout with dynamic memory management. The reversible Buddy Memory heap layout presented in section 3.1.3 is utilized in ROOPL++ as it is

an interesting layout, naturally translates into a reversible setting with one simple restriction (i.e only blocks which are heads of their respectable free lists are allocatable) and since its only drawback is dismissible in most real world scenarios.

Figure 3.5 shows the full layout of a ROOPL++ program stored in memory.

- As with ROOPL, the static storage segment contains load-time labelled **DATA** instructions, initialized with virtual function tables and other static data needed by the translated program.
- The program segment is stored right after the static storage and contains the translate ROOPL++ program instructions.
- The free lists maintained by the Buddy Memory heap layout is placed right after the program segment, with the *free list pointer* $flp$ pointing at the first free list. The free lists are simple the address to the first block of its respective size. The free lists are stores such that the free list at address $flp + i$ corresponds to the free list of size $2^{i+1}$.
- The heap begins directly following the free lists. Its beginning is marked by the *heap pointer* ($hp$).
- Unlike in ROOPL, where the stack grows upwards, the ROOPL++ stack grows downwards and is begins at address $p$. The stack remains a LIFO structure, analogously to ROOPL.

As denoted in the previous section, we assume an underlying reversible operating system providing us with additional memory when needed. With no real way of simulating this, the ROOPL++ compiler places the stack at a fixed address $p$ and sets one free block in the largest $2^n$ free list initially. The number of free lists and the address $p$ is configurable in the source code, but is defaulted to 10 free lists, meaning initially one block of size 1024 is available and the stack is placed at address 2048.

In traditional compilers, the heap pointer usually points to the end of the heap. For reasons stated above, we never grow the heap as we start with a heap of fixed size. As such, the heap pointer simply points to the beginning of the heap.

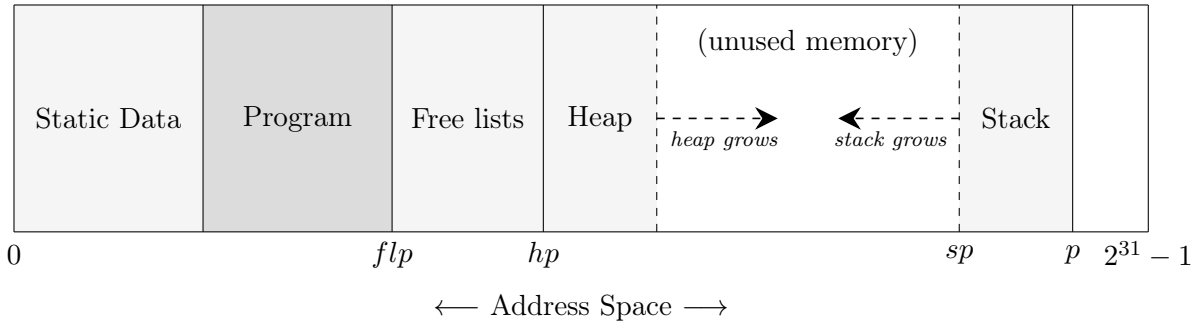


**Figure 3.5:** Memory layout of a ROOPL++ program

## 3.3 Inherited Roopl features

As mentioned, a number of features from Roopl carries over in Roopl++ .

The dynamic dispatching mechanism presented in [4] is inherited. As such, the invocation of a method implementation is based on the type of the object at run time. Virtual function tables are still the implementation strategy used in the dynamic dispatching implementation.

Evaluation of expressions and control flow remains unchanged.

For completeness, object blocks are included and still stack allocated as their life time is limited to the scope of their block and the dynamic allocation process is quite expensive in terms of register pressure and number of instructions compared to the stack allocated method presented implemented in the Roopl compiler.

The object layout remains unchanged (*for now, until reference counting is added*) and is shown in figure 3.6

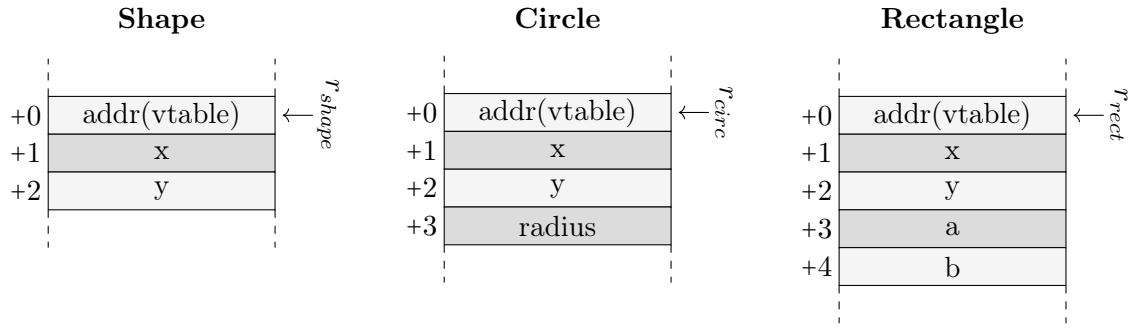


**Figure 3.6:** Illustration of prefixing in the memory layout of 3 Roopl++ objects

## 3.4 Program Structure

The program structure of a translated Roopl++ is analogous to the program structure of a Roopl program with the addition of free lists and heap initialization. The full structure is shown in figure 3.7.

This PISA code block initializes the free lists pointer, the heap pointer, the stack pointer, allocates the main object on the stack, calls the main method, deallocates the main object and finally clears the free lists, heap and stack pointers.

The free lists pointer is initialized by adding the base address, which varies with the size of the translated program, to the register $r_{flps}$. In figure 3.7 the base address is denoted by $p$.

The heap pointer is initialized directly after the free lists pointer by adding the size of the free lists. One free lists is the size of one word and the full size of the free lists is configured in the source code (defaulted to 10, as described earlier).

Once the heap pointer and free lists pointer is initialized, the initial block of free memory is placed in the largest free lists by indexing to said list, by adding the length of the list of free

```
(1)                 ......                          ; Static data declarations
(2)                 ......                          ; Code for program class methods
(3)  start :        START                           ; Program starting point
(4)                 ADDI    r_flps  p               ; Initialize free lists pointer
(5)                 XOR     r_hp    r_flps          ; Initialize heap pointer
(6)                 ADDI    r_hp    size_fls        ; Initialize heap pointer
(7)                 XOR     r_b     r_hp            ; Store address of initial free memory block in r_b
(8)                 ADDI    r_flps  size_fls        ; Index to end of free lists
(9)                 SUBI    r_flps  1               ; Index to last element of free lists
(10)                EXCH    rb      r_flps          ; Store address of first block in last element of free lists
(11)                ADDI    r_flps  1               ; Index to end of free lists
(12)                SUBI    r_flps  s               ; Index to beginning of free lists
(13)                ADDI    r_sp    offset_stack    ; Initialize stack pointer
(14)                XOR     r_m     r_sp            ; Store address of main object in r_m
(15)                XORI    r_v     label_vt        ; Store address of vtable in r_v
(16)                EXCH    r_v     r_sp            ; Push address of vtable onto stack
(17)                SUBI    r_sp    size_m          ; Allocate space for main object
(18)                PUSH    r_m                     ; Push 'this' onto stack
(19)                BRA     label_m                 ; Call main procedure
(20)                POP     r_m                     ; Pop 'this' from stack
(21)                SUBI    r_sp    size_m          ; Deallocate space of main object
(22)                EXCH    r_v     r_sp            ; Pop vtable address into r_v
(23)                XORI    r_v     label_vt        ; Clear r_v
(24)                XOR     r_m     r_sp            ; Clear r_m
(25)                SUBI    r_sp    offset_stack    ; Clear stack pointer
(26)                SUBI    r_hp    size_fls        ; Clear heap pointer
(27)                XOR     r_hp    r_flsp          ; Clear heap pointer
(28)                SUBI    r_flps  p               ; Clear free lists pointer
(29)  finish :      FINISH                          ; Program exit point
```

**Figure 3.7:** Overall layout of a translated RoopL++ program

lists, subtracting 1, writing the address of the first block (which is the same address as the heap pointer, which points to the beginning of the heap) to the last free list and then resetting the free lists pointer to point to the 1st list again, afterwards.

The stack pointer is initialized simply by adding the stack offset to the stack register $r_{sp}$. The stack offset is configured in the source code and defaults to 2048, as described earlier in this chapter. Once the stack pointer has been initialized, the main object is allocated on the stack and the main method called, analogously to the RoopL program structure.

When the program terminates and the main method returns, the main object is popped from the stack and deallocated and the stack pointer is cleared. The heap pointer is then cleared followed by the free lists pointer. The contents of the free lists and whatever is left on the heap is untouched at this point. It is the programmers responsibility to free dynamically allocated objects in their RoopL++ program. Furthermore, depending on the deallocation order, we might not end up with exactly one fully merged block in the end and as such, we do not invert the steps taken to initialize this initial free memory block. Analogously to RoopL, the values of the main object are left in stack section of memory.

## 3.5 Object Allocation and Deallocation

As allocation and deallocation intuitively should be each other's inverse, numerous instructions are shared between the two, mainly the core of the reversible buddy algorithm shown in listing 3.3. The PISA translated version of the `malloc1` function from listing 3.3 is shown in figure 3.9.

In the translated `malloc1` function, the **SWAPBR** functions as entry and exit point, as PISA's paired jumps could not work here, as we need to jump the entry point of the function from multiple locations, so support the recursiveness of the function. Once the entry point has been reached, the return offset obtained from the **SWAPBR** instruction is negated and stored on the heap. When the function reaches the bottom of the instruction block, a jump to the top is performed where the negated return offset is popped from the stack before the **SWAPBR** instruction is called again, resulting in a jump to the location which originally branched to the $malloc1_{entry}$ label. As with the Janus implementation of the algorithm, the PISA version is dependent on a number of arguments, which should be stored in registers. Register $r_{sc}$ hold the block size counter, $r_c$ the free list index counter and $r_{size_c}$ the size of the class. Once the `malloc1` instructions has completed $r_p$ contains the address of the allocated object. The allocation and deallocation entry points are responsible for settings these, as seen in figure 3.8

**new** $c$ $x$

```
(1)  ADDI  r_sc        2                  ; Init block size counter
(2)  XOR   r_c         r_0                ; Init free list index counter
(3)  ADDI  r_size_c    size_c             ; Init class size
(4)  BRA   malloc_entry_q                 ; Jump to malloc entry
(5)  ......                               ; Code for malloc1 (Fig ??)
(6)  XORI  r_t         label_vt           ; Store address of vtable in r_t
(7)  EXCH  r_t         r_p                ; Write vtable address in object
(8)  SUBI  r_size_c    size_c             ; Inverse of (3)
(9)  XOR   r_c         r_0                ; Inverse of (2)
(10) SUBI  r_sc        2                  ; Inverse of (1)
```

**delete** $c$ $x$

```
(1)  EXCH  r_t         r_p                ; Clear vtable in object
(2)  XORI  r_t         label_vt           ; Clear r_t
(3)  ADDI  r_sc        2                  ; Init block size counter
(4)  XOR   r_c         r_0                ; Init free list index counter
(5)  ADDI  r_size_c    size_c             ; Init class size
(6)  BRA   malloc_entry_q                 ; Jump to inverted malloc entry
(7)  ......                               ; Code for inverted malloc1 (Fig ??)
(8)  SUBI  r_size_c    size_c             ; Inverse of (5)
(9)  XOR   r_c         r_0                ; Inverse of (4)
(10) SUBI  r_sc        2                  ; Inverse of (3)
```

**Figure 3.8:** PISA translation of heap allocation and deallocation for objects

As mentioned, the `malloc(1)` is executed recursively, scanning through free blocks in free lists holding blocks of equal or greater size of the $size_c$ we want to allocate. Once such a block has been found, it is recursively split, if its size isn't equal to $size_c$. Before each recursive call (or rather branching in PISA), we must push a number of temporary register values to the stack, to ensure we can re-obtain theres values once the next recursive call returns. These temporary values includes the address of the current free list and the address of its first block, the expression evaluation results needed for the 2 conditional statements along with a temporary register, holding different date depending on where the algorithm branches from. As can be seen in 3.9, the register pressure is quite high for the heap allocation and deallocation translations. In fact, 11 free registers besides the 4 registers initiated in the code generation for $new$ and $free$ are needed for the translation to succeed. This number of register should obviously be optimized to reduce the register pressure, as taking up 15 registers for allocating or deallocating an object, would only leave 10 free registers available in the current scope, as the free lists, heap and stack pointer further take up 3 registers, the return offset register $r_{ro}$ and register 0 takes up an additional 2. This effectively means that no more than 10 objects

can be heap allocation in the same scope, as of writing. The main register hogging part of the translation scheme is the expression evaluation required for the two conditionals (Line 16, 23, 39 and 40 in listing 3.3) as the composite expressions such as `free_lists[counter] = 0 || p - csize != free_lists[counter]` currently requires 3 temporary registers for evaluating `free_lists[counter] = 0`, `p - csize != free_lists[counter]` and finally `free_lists[counter] = 0 || p - csize != free_lists[counter]` separately.

## 3.6 Vectors

TBD

## 3.7 Error Handling

TBD

## 3.8 Implementation

TBD

```
(1)  malloc1_top :  BRA    malloc1_bot          ; Receive jump
(2)                 POP    r_ro                 ; Pop return offset from the stack
(3)                 ......                      ; Inverse of (7)
(4)  malloc1_entry : SWAPBR r_ro                ; Malloc1 entry and exit point
(5)                 NEG    r_ro                 ; Negate return offset
(6)                 PUSH   r_ro                 ; Store return offset on stack
(7)                 ......                      ; Code for r_fl ← addr(free_lists[counter])
(8)                 ......                      ; Code for r_block ← value(free_lists[counter])
(9)                 ......                      ; Code for r_e1_o ← [[c_size < object_size]]
(10)                XOR    r_t   r_e1_o         ; Copy value of c_size < object_size into r_t
(11)                ......                      ; Inverse of (9)
(12) o_test :       BEQ    r_t   r_0   o_test_f ; Receive jump
(13)                XORI   r_t   1              ; Clear r_t
(14)                ADDI   r_c   1              ; Counter++
(15)                RL     r_sc  1              ; Call double(c_size)
(16)                ......                      ; Code for pushing temp reg values to stack
(17)                BRA    malloc1_entry        ; Call malloc1())
(18)                ......                      ; Inverse of (16)
(19)                RR     r_sc  1              ; Inverse of (15)
(20)                SUBI   r_c   1              ; Inverse of (14)
(21)                XORI   r_t   1              ; Set r_t = 1
(22) o_assert_t :   BRA    o_assert             ; Jump
(23) o_test_f :     BRA    o_test               ; Receive jump
(24)                ......                      ; Code for r_e1_i ← [[addr(free_lists[counter]) ≠ 0]]
(25)                XOR    r_t2  r_e1_i         ; Copy value of r_e1_i into r_t2
(26)                ......                      ; Inverse of (24)
(27) i_test :       BEQ    r_t2  r_0   i_test_f ; Receive jump
(28)                XORI   r_t2  1              ; Clear r_t2
(29)                ADD    r_p   r_block        ; Copy address of the current block to p
(30)                SUB    r_block r_p          ; Clear r_block
(31)                EXCH   r_tmp r_p            ; Load address of next block
(32)                EXCH   r_tmp r_fl           ; Set address of next block as new head of free list
(33)                XOR    r_tmp r_p            ; Clear address of next block
(34)                XORI   r_t2  1              ; Set r_t2 = 1
(35) i_assert_t :   BRA    i_assert             ; Jump
(36) i_test_f :     BRA    i_test               ; Receive jump
(37)                ADDI   r_c   1              ; Counter++
(38)                RL     r_sc  1              ; Call double(c_size)
(39)                ......                      ; Code for pushing temp reg values to stack
(40)                BRA    malloc1_entry        ; Call malloc1())
(41)                ......                      ; Inverse of (39)
(42)                RR     r_sc  1              ; Inverse of (38)
(43)                SUBI   r_c   1              ; Inverse of (37)
(44)                XOR    r_tmp r_p            ; Copy current address of p
(45)                EXCH   r_tmp r_fl           ; Store current address of p in current free list
(46)                ADD    r_p   r_cs           ; Split block by setting p to the second half of the current block
(47) i_assert :     BNE    r_t2  r_0   i_assert_t; Receive jump
(48)                EXCH   r_tmp r_fl           ; Load address of head of current free list
(49)                SUB    r_p   r_cs           ; Set p to previous block address
(50)                ......                      ; Code for r_e2_i1 ← [[p − c_size ≠ addr(free_lists[counter])]]
(51)                ......                      ; Code for r_e2_i2 ← [[addr(free_lists[counter]) = 0]]
(52)                ......                      ; Code for r_e2_i3 ← [[(p − c_size ≠ addr(free_lists[counter])) ∨ (addr(free_lists[counter]) = 0)]]
(53)                XOR    r_t2  r_e2_i3        ; Copy value of r_e2_i3 into r_t2
(54)                ......                      ; Inverse of (52)
(55)                ......                      ; Inverse of (51)
(56)                ......                      ; Inverse of (50)
(57)                ADD    r_p   r_cs           ; Inverse of (49)
(58)                EXCH   r_tmp r_fl           ; Inverse of (48)
(59) o_assert :     BNE    r_t   r_0   o_assert_t; Receive jump
(60)                ......                      ; Code for r_e2_o ← [[c_size < object_size]]
(61)                XOR    r_t   r_e2_o         ; Copy value of c_size < object_size into r_t
(62)                ......                      ; Inverse of (60)
(63) malloc1_bot :  BRA    malloc1_top          ; Jump
```

**Figure 3.9:** The core of the reversible buddy memory algorithm translated into PISA. For deallocation the inverse of this block is generated during compile-time.

# References


[1] Axelsen, H. B. and Glück, R. "Reversible Representation and Manipulation of Constructor Terms in the Heap". In: *RC '13: Proceedings of the 5th International Conference on Reversible Computation* (2013), pp. 96–109.

[2] Foundation, P. S. *Memory Management*. URL: `https://docs.python.org/2/c-api/memory.html`.

[3] Hansen, J. S. K. "Translation of a Reversible Functional Programming Language". Master's Thesis. University of Copenhagen, DIKU, 2014.

[4] Haulund, T. "Design and Implementation of a Reversible Object-Oriented Programming Language". Master's Thesis. University of Copenhagen, DIKU, 2016.

[5] Lee, W. H. and Chang, M. "A study of dynamic memory management in C++ programs". In: *Computer Languages, Systems and Structures* 23 (3 2002), pp. 237–272.

[6] Mogensen, T. Æ. "Programming Language Design and Implementation (Unpublished)".

[7] Mogensen, T. Æ. "Reference Counting for Reversible Languages". In: *RC '14: Proceedings of the 6th International Conference on Reversible Computation* (2014), pp. 82–94.

[8] Mogensen, T. Æ. "Garbage Collection for Reversible Functional Languages". In: *RC '15: Proceedings of the 7th International Conference on Reversible Computation* (2015), pp. 79–94.

[9] Venners, B. *The Java Virtual Machine*. URL: `http://www.artima.com/insidejvm/ed2/jvmP.html`.

[10] Yokoyama, T., Axelsen, H. B., and Glück, R. "Towards a Reversible Functional Language". In: *RC 11: Proceedings of the Third International Conference on Reversible Computation* (2011), pp. 14–29.

[11] Yokoyama, T. and Glück, R. "A Reversible Programming Language and its Invertible Self-Interpreter". In: *PEPM 2007: Proceedings of the Workshop on Partial Evaluation and Program Manipulation* (2007), pp. 144–153.