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 2.6. Addressed Chapter 4 feedback. Rewrote pseudo-code algorithms in EXTENDED JANUS.
0.5	2017-10-11	Martin	Finalized a few sections in chapter 1. Update chapter 4 to reflect work done on compiler.
0.6	2017-10-12	Martin	Added first version of section 4.5
0.7	2017-10-25	Martin	Moved Dynamic Memory Management sections into their own chapter 3. Added computation strength section 2.11
0.8	2017-11-07	Martin	Added section 2.8 and 2.9. Started on section 2.10. Updated RTM in section 2.11.
0.9	2017-12-15	Martin	Wrote section 2.6, finished remaining sections of chapter 1, addressed feedback in chapter 2 (sections 2.8, 2.9), updated syntax for updated arrays in section 2.1, revised array sections to current implementation in section 2.4 and 2.5. Updated type system and semantics with array specific rules in section 2.8 and 2.9. Started on section 4.8.

TODOs:

- Revise section 2.11 with updated RTM implementation and description
- Finalize section 2.10
- Revise chapter 3 and finish figures
- Update chapter 4 to reflect code updates
- Describe reference counting and array implementation in chapter 4

- Write chapter 5.

Table of Contents

List of Figures

CHAPTER 1

Introduction

In recent years, technologies such as cloud-based services, cryptocurrency mining and other services requiring large computational power and availability has been on the rise. These services are hosted on massive server parks, consuming immense amounts of electricity in order to power the machines and the cooling architectures as heat dissipates from the hardware. A recent study showed that the Bitcoin network including its mining processes' currently stands at 0.13% of the total global electricity consumption, rivaling the usage of a small country like Denmark's [2]. With the recent years focus on climate and particularly energy consumption, companies have started to attempt to reduce their power usage in these massive server farms. As an example, Facebook built new server park in the arctic circle in 2013, in an attempt to take advantage of the natural surroundings in the cooling architecture to reduce its power consumption [15].

Reversible computing presents a possible solution the problematic power consumption issues revolving around computations. Traditional, irreversible computations dissipates heat during their computation. Landauer's principle states that deletion of information in a system always results by an increase in energy consumption. In reversible computing, all information is preserved throughout the execution, and as such, the energy consumption theoretically should be smaller [8].

Currently, reversible computing is not commercially appealing, as it is an area which still is being actively researched. However, several steps has been taken in the direction of a fully reversible system, which some day might be applicable in a large setting. Reversible machine architectures have been presented such as the Pendulum architecture and its instruction set Pendulum ISA (Pisa) [17] and the BobISA architecture and instruction set [14] and high level languages Janus [10, 21, 19] and R [4] exists.

1.1 Reversible Computing

Reversible computing is a two-directional computational model in which all processes are time-invertible. This means, that at any time during execution, the computation can return to a former state. In order to maintain *reversibility*, the reversible computational modal cannot compute *many-to-one* functions, as the models requires an exact inverse f^{-1} of a function f in order to support backwards determinism. Therefore, reversible programs must only consist on *one-to-one* functions, also known as *injective* functions, which results in a garbage-free computations.

As each step of a reversible program is locally invertible, meaning each of its component has

exactly one inverse component, a reversible program can be inverted simply by computing the inverse of each of its components, without any knowledge about the overall program's functionality or requirements. This property immediately yields interesting consequences in terms of software development, as an encryption or compression algorithm implemented in a reversible language immediately yields the decryption or decompression algorithm.

The reversibility is however not free at comes at the cost of strictness when writing programs. Almost every popular, irreversible programming language features a conditional component in form of **if-else**-statements. In these languages, we only define the *entry*-condition in the condition, that is, the condition that determines which branch of the component we continue execution in. In reversible languages, we must also specify an *exit*-condition, such that we can determine which branch we should follow, we we are executing the program in reverse. In theory, this sounds trivial, but in practice it turns to add a new layer of complexity when writing programs.

1.2 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.

1.3 Reversible Object-Oriented Programming

The high-level reversible language Roopl (Reversible Object-Oriented Programming Language) was introduced in late 2016 [6, 7]. 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 [12] and garbage collection [13] suggests that a Roopl extension is feasible.

1.5 Thesis Statement

An extension of the reversible object-oriented programming language with dynamic memory management is possible and effective. The resulting expressiveness allows non-trivial reversible programming previously unseen, such as reversible data structures.

1.6 Outline

This Master's thesis consists of four chapters, besides the introductory chapter. The following summary describes the following chapters

- **Chapter 2** formally defines the ROOPL extension in form of the new language ROOPL++ , a superset of ROOPL.
- **Chapter 3** serves as a brief description of dynamic memory management along with a discussion of various reversible, dynamic memory management layouts.
- **Chapter 4** presents the translation techniques utilized in the compiling a ROOPL++ program to PISA instructions.
- **Chapter 5** presents the conclusions of the thesis and future work proposals.

Besides the five chapters, a number of appendices is supplied, containing the source code of the ROOPL++ to PISA compiler, the ROOPL++ source code for the example programs and their translated PISA versions.

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	::=	$c \mid c[e] \mid$ **int**$[e]$	(class and arrays)
t	::=	**int** $\mid c \mid$ **int**[] $\mid c[]$	(data type)
y	::=	$x \mid x[e]$	(variable identifiers)
m	::=	**method** q**(**$t\ x,\ \ldots,\ t\ x$**)** s	(method)
s	::=	$y\ \odot$ **=** $e \mid y$ **<=>** y	(assignment)
	$\mid$	**if** e **then** s **else** s **fi** e	(conditional)
	$\mid$	**from** e **do** s **loop** s **until** e	(loop)
	$\mid$	**construct** $d\ y$ s **destruct** y	(object block)
	$\mid$	**local** $t\ x = e$ s **delocal** $t\ x = e$	(local variable block)
	$\mid$	**new** $d\ y \mid$ **delete** $d\ y$	(object con- and destruction)
	$\mid$	**copy** $d\ y\ y \mid$ **uncopy** $d\ y\ y$	(reference con- and destruction)
	$\mid$	**call** q**(**$x,\ \ldots,\ x$**)** $\mid$ **uncall** q**(**$x,\ \ldots,\ x$**)**	(local method invocation)
	$\mid$	**call** y**::**q**(**$x,\ \ldots,\ x$**)** $\mid$ **uncall** y**::**q**(**$x,\ \ldots,\ x$**)**	(method invocation)
	$\mid$	**skip** $\mid s\ s$	(statement sequence)
e	::=	$\overline{n} \mid x \mid x[e] \mid$ **nil** $\mid e\ \otimes\ e$	(expression)
$\odot$	::=	**+** $\mid$ **-** $\mid$ **^**	(operator)
$\otimes$	::=	$\odot \mid$ ***** $\mid$ **/** $\mid$ **%** $\mid$ **&** $\mid$ **|** $\mid$ **&&** $\mid$ **||** $\mid$ **<** $\mid$ **>** $\mid$ **=** $\mid$ **!=** $\mid$ **<=** $\mid$ **>=**	(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 static integer or class array type 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 [6] 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**.

Arrays are also instantiated by usage of **new** and **delete**. Assignment to array cells depends on the type of the arrays, which is further discussed in section 2.5.

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

$$\textbf{new}\ c\ x \quad s \quad \textbf{delete}\ c\ x$$

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 Object Referencing

Besides dynamically scoped objects, ROOPL++ also increases program flexibility by allowing multiple references to objects through the usage of the **copy** statement. Once instantiated through either a **new** or **construct**/**destruct** block, an object reference can be copied into a new variable. The reference acts as a regular class instance and can be modified through methods as per usual. To delete a reference, the logical inverse statement **uncopy** must be used.

The syntax for referencing consists of the statement

$$\textbf{copy}\ c\ x\ x'$$

which makes a reference of variable x, an instance of class c, and stores the reference in variable x'.

For deleting copies, the following statement is used

$$\textbf{uncopy}\ c\ x\ x'$$

which simply zero-clears variable x', which is a reference to variable x, an instance of class x.

The mechanism of the **copy** statement is simply as follows

1. The memory address stored in variable x is copied into the zero-cleared variable x'. If x' is not zero-cleared or x is not a class instance, then **copy** is undefined.

The mechanism of the **uncopy** statement is simply as follows

1. The memory address stored in variable x' is zero-cleared if it matches the address stored in x. If x' is not a copy of x or x has been zero-cleared before the **uncopy** statement is executed, said statement is undefined.

As references does not require all fields to be zero-cleared (as they simply are pointers to existing objects), the reversible programmer should carefully ensure that all references to objects are un-copied before deleting said object, as copied references to cleared objects would be pointing to cleared memory, which might be used later by the system. These type of references are also known as *dangling pointers*.

It should be noted, that from a language design perspective, it is the programmer's responsibility to ensure such situations does not occur. From an implementation perspective, such situations are usually checked by the compiler either statically during compilation or during the actual runtime of the program. This is addressed later in sections 4.7 and 4.8.

2.4 Array Model

Besides asymmetric object lifetimes, ROOPL++ also introduces reversible, static arrays of either integer or object types. While ROOPL only featured integers and custom data types in form of classes, one of its main inspirations, JANUS, implemented static, reversible arrays [21].

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, as one of the main goals of ROOPL++ is increased expressiveness while implementing reversible programs.

ROOPL++ 's arrays expand upon the array model from JANUS. Arrays are index by integers, starting from 0. In JANUS, only integer arrays were allowed, while in ROOPL++ arrays of any type can be defined, meaning either integer arrays or custom data types in form of class array. They are however, still restricted to one-dimensionality.

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

Arrays can contain elements of different classes sharing a base class, that is, say class A and B both inherit from some class C and array x is of type $C[]$. In this case, the array can hold elements of type A, B, and C.

2.5 Array Instantiation

Array instantiation uses the **new** and **delete** keywords to reversibly construct and destruct array types. The mechanisms of the statement

$$\textbf{new int}[e]\ x$$

in which we reserved memory for an integer array are as follows

1. The expression e is evaluated
2. Memory equal to the integer value e evaluates to and an additional small amount memory for of overhead is reserved for the array.

In ROOPL++ , we only allow instantiation of static arrays of a length defined in the given expression e. Array elements are assigned dependent on the type of the array. For integer arrays, any of the reversible assignment operators can be used to assign values to cells. For class arrays, we assign cell elements a little differently. We either 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 or we use the *swap* statement to swap values in and out of array cells. Usage of the **+=** or **-=** assignment operators on non-integer arrays is undefined.

```
1    new int[5] intArray          // Init new integer array
2    new Foo[2] fooArray          // init new Foo array
3
4    intArray[1] += 10            // Legal array integer assignment
5    intArray[1] -= 10            // Legal Zero-clearing for integer array cells
6
7    new Foo fooObject
8    fooArray[0] <=> fooObject    // Legal object array cell assignment
9    new Foo fooArray[2]          // Legal object array cell assignment
10
11   delete Foo fooArray[0]       // Legal object array cell zero-clearing
12   delete Foo fooArray[1]       // Legal object array cell zero-clearing
```

Listing 2.1: Assignment of array elements

As with ROOPL++ objects instantiated outside of **construct**/**destruct** blocks, arrays must be deleted before program termination to reversibly allow the system to reclaim the memory. Before deletion of an array, all its elements must be zero-cleared such that no garbage data resides in memory after erasure of the array reference.

Consider the statement

$$\textbf{delete int}[e]\ x$$

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 an array with fully zero-cleared cells before the **delete** statement. Otherwise, the meaning of the statement is undefined.
2. If the reference x is a fully zero-cleared array upon the **delete** statement execution, the zero-cleared memory is reclaimed by the system.

With reversible, static arrays of varying types, 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

$$x[5] += x[5] + 1$$

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

2.6 Local Blocks

The local block presented in the extended JANUS in [19] consisted of a local variable allocation, a statement and a local variable deallocation. These local variable blocks add immense programmer usability as the introduce a form of reversible temporary variable. The ROOPL compiler features support for local integer blocks, but not object blocks. In ROOPL++ , local blocks can be instantiated with all of the languages variable types; integers, arrays and user-defined types in form of objects.

Local integer blocks works exactly the same as in Roopl and JANUS, where in the local variable will be set to result a given expression.

Local array and object blocks features a number of different options. If a local array or object block is initialized with a **nil** value, the variable must afterwards be initialized using a new statement before any type-specific functionality is possible. If the block is initiated with an existing object or array reference, the local variable does in fact become a reference copy, analogous to a variable initialized from a **copy** statement.

For objects, the **construct**/**destruct**-blocks can be considered syntactic-sugar for a local block defined with a **nil** value, containing a **new** statement in the beginning of its statement block and a **delete** statement in the very end.

As local array and object block allow more freedom in terms of their interaction with other statements in the language, it is the programmer's responsibility that the local variable is deallocated using a correct expression in the end of the block definition. As the value of the variable is a pointer to an object or an array, said object or array must have all their fields/cells zero-cleared before the its pointer is zero-cleared at the end of the local block. If the pointer is at any point exchanged with the pointer of another object or array using the **swap** statement, the same conditions apply.

2.7 ROOPL++ Expressiveness

By introducing dynamic lifetime objects and by allowing objects to be referenced multiple times, we can express non-trivial programs in the reversible setting. To demonstrate the capacities, expressiveness and possibilities of ROOPL++ , the following section presents reversible data structures written in ROOPL++ .

2.7.1 Linked List

Haulund presented a Linked List implemented in ROOPL. The implementation featured a *ListBuilder* and a *Sum* class, required to determine and retain the sum of a constructed linked list as ROOPL's statically scoped object blocks would deallocate automatically after building the full list. In ROOPL++ we do not face the same challenges and the implementation becomes much more forward. Figure 2.4 implemented a *LinkedList* class, simply has the head of the list and the list of length as its internal fields. For demonstration, the class allows extension of the list by either appending or prepending cell elements to the list. In either case, we first check if the *head* field is initialized. If not, the cell we are either appending or prepending simply becomes the new head of the list. If we are appending a cell the Cell-class *append* method is called on the *head* cell with the new cell as its only argument. When prepending, the existing head is simply appended to the new cell and the new cell is set as head of the linked list.

```
1    class Cell
2        Cell next
3        int data
4
5        method constructor(int value)
6            data ^= value
7
8        method append(Cell cell)
9            if next = nil & cell != nil then
10               next <=> cell           // Store as next cell if current cell is end of list
11           else skip
12           fi next != nil & cell = nil
13
14           if next != nil then
15               call next::append(cell) // Recursively search until we reach end of list
16           else skip
17           fi next != nil
```

Figure 2.3: Linked List cell class

Figure 2.3 shows the *Cell* class of the linked list which has a *next* and a *data* field, a constructor and the *append* method. The append method works by recursively looking through the linked cell nodes until we reach the end of the free list, where the *next* field has not been initialized yet. When we find such a cell, we simply swap the contents of the *next* and *cell* variables, s.t. the cell becomes the new end of the linked list.

An interesting observation, is that the *append* method is called an additional time *after* setting the cell as the new end of the linked list. In a non-reversible programming language, we would simply call append in the else-branch of the first conditional. In the reversible setting, this is not an option, as the append call would modify the value of the *next* and *cell* variables and as such,

corrupt the control flow as the exit condition would be true after executing both the then- and else-branch of the conditional. This "wasted" additional call with a **nil** value *cell* is a recurring technique in the following presented reversible data structure implementations.

```
1   class LinkedList
2       Cell head
3       int listLength
4
5       method insertHead(Cell cell)
6           if head = nil & cell != nil then
7               head <=> cell               // Set cell as head of list if list is empty
8           else skip
9           fi head != nil & cell = nil
10
11      method appendCell(Cell cell)
12          call insertHead(cell)           // Insert as head if empty list
13
14          if head != nil then
15              call head::append(cell)     // Iterate until we hit end of list
16          else skip
17          fi head != nil
18
19          listLength += 1                 // Increment length
20
21      method prependCell(Cell cell)
22          call insertHead(cell)           // Insert as head if empty list
23
24          if cell != nil & head != nil then
25              call cell::append(head)     // Set cell.next = head. head = nil after execution
26          else skip
27          fi cell != nil & head = nil
28
29          if cell != nil & head = nil then
30              cell <=> head               // Set head = cell. Cell is nil after execution
31          else skip
32          fi cell = nil & head != nil
33
34          listLength += 1                 // Increment length
35
36      method length(int result)
37          result ^= listLength
```

Figure 2.4: Linked List class

2.7.2 Binary Tree

Figures 2.5, 2.7 and 2.6 shows the implementation of a binary tree in form of a rooted, unbalanced, min-heap. The *Tree* class shown in figure 2.5 has a single root node field and the three methods *insertNode, sum* and *mirror*. For insertion, the *insertNode* method is called from the *root*, if it is initialized and if not, the passed node parameter is simply set as the new root of the tree. The *insertNode* method implemented in the *Node* class shown in figure 2.7 first determines if we need to insert left or right but checking the passed value against the value of the current node. This is done recursively, until an uninitialized node in the correct subtree has been found. Note that the value of node we wish to insert must be passed separately in the method call as we otherwise cannot zero-clear it after swapping the node we are inserting with either the right or left child of the current cell.

```
1    class Tree
2        Node root
3
4        method insertNode(Node node, int value)
5            if root = nil & node != nil then
6                root <=> node
7            else skip
8            fi root != nil & node = nil
9
10           if root != nil then
11               call root::insertNode(node, value)
12           else skip
13           fi root != nil
14
15       method sum(int result)
16           if root != nil then
17               call root::getSum(result)
18           else skip
19           fi root != nil
20
21       method mirror()
22           if root != nil then
23               call root::mirror()
24           else skip
25           fi root != nil
```

Figure 2.5: Binary Tree class

Summing and mirroring the tree works in a similar fashion by recursively iterating each node of the tree. For summing we simply add the value of the node to the sum and for mirroring we swap the children of the node and then recursively swap the children of the left and right node, if initialized. The sum and mirror methods are implemented in figure 2.6.

```
1    method getSum(int result)
2        result += value                  // Add the value of this node to the sum
3
4        if left != nil then
5            call left::getSum(result)   // If we have a left child, follow that path
6        else skip                       // Else, skip
7        fi left != nil
8
9        if right != nil then
10           call right::getSum(result)  // If we have a right child, follow that path
11       else skip                       // Else, skip
12        fi right != nil
13
14   method mirror()
15       left <=> right                  // Swap left and right children
16
17       if left = nil then skip
18       else call left::mirror()        // Recursively swap children if left != nil
19       fi left = nil
20
21       if right = nil then skip
22       else call right::mirror()       // Recursively swap children if right != nil
23       fi right = nil
```

Figure 2.6: Binary Tree node class (cont)

The binary tree could be extended with a method for flattening into an array of size equal to

the number of tree nodes. The inverse of this method would be construction of a tree from a flattening method. This way, sorting an array could effectively be implemented by constructing a tree from an array, performing some recursive tree sorting method, and then flattening the tree into an array again.

```
1    class Node
2        Node left
3        Node right
4        int value
5
6        method setValue(int newValue)
7            value ^= newValue
8
9        method insertNode(Node node, int nodeValue)
10           // Determine if we insert left or right
11           if nodeValue < value then
12               if left = nil & node != nil then
13                   // If open left node, store here
14                   left <=> node
15               else skip
16               fi left != nil & node = nil
17
18               if left != nil then
19                   // If current node has left, continue iterating
20                   call left::insertNode(node, nodeValue)
21               else skip
22               fi left != nil
23           else
24               if right = nil & node != nil then
25                   // If open right node spot, store here
26                   right <=> node
27               else skip
28               fi right != nil & node = nil
29
30               if right != nil then
31                   // If current node has, continue searching
32                   call right::insertNode(node, nodeValue)
33               else skip
34               fi right != nil
35           fi nodeValue < value
```

Figure 2.7: Binary Tree node class

2.7.3 Doubly Linked List

Finally, we present the reversible doubly linked list, shown in figures 2.8-2.10. A *cell* in a doubly linked list contains a reference to itself named *self*, a reference to its left and right neighbours, a data and an index field. As with the linked list and binary tree implementation the *DoubleLinkedList* class has a a field referencing the head of the list and its *appendCell* method is identical to the one of the linked list.

This data structure is particularly interesting, as it, unlike the former two presented structures, cannot be expressed in ROOPL, as this requires multiple reference to objects, in order for an object to point to itself.

When we append a cell to the list, we first search recursively through the list until we are at the end. The new cell is then set as *right* of the current cell. A reference to the current self is

```
1     class DoublyLinkedList
2         Cell head
3         int length
4
5         method appendCell(Cell cell)
6             if head = nil & cell != nil then
7                 head <=> cell
8             else skip
9             fi head != nil & cell = nil
10
11            if head != nil then
12                call head::append(cell)
13            else skip
14            fi head != nil
15
16            length += 1
```

Figure 2.8: Doubly Linked List class

created using the **copy** statement, and set as *left* of the new end of the list, thus resulting on the new cell being linked to list and now acting as end of the list.

```
1         class Cell
2         int data
3         int index
4         Cell left
5         Cell right
6         Cell self
7
8         method setData(int value)
9             data ^= value
10
11        method setIndex(int i)
12            index ^= i
13
14        method setLeft(Cell cell)
15            left <=> cell
16
17        method setRight(Cell cell)
18            right <=> cell
19
20        method setSelf(Cell cell)
21            self <=> cell
```

Figure 2.9: Doubly Linked List Cell class

The data structure could relatively easily be extended to work as a dynamic array. Currently each cell contains an *index* field, specifying their in the list. If, say, we wanted to insert some new data at index n, without updating the existing value, but essentially squeezing in a new cell, we could add a method to the *DoublyLinkedList* class taking a data value and an index. When executing this method, we could iterate the list until we reach the cell with index n, construct a new *cell* instance, update required *left* and *right* pointers to insert the new cell at the correct position, in such a way that the old cell at index n now is the new cell's right neighbour and finally recursively iterating the list, incrementing the index of cells to the right of the new cell by one. If we want to update an existing value at a index, a similar technique could be used, where we iterate through the cells until we find the correct index. If we're given an index that is out of

```
1     method append(Cell cell)
2         if right = nil & cell != nil then   // If current cell does not have a right neighbour
3             right <=> cell                  // Set new cell as right neighbour of current cell
4
5             local Cell selfCopy = nil
6             copy Cell self selfCopy         // Copy reference to current cell
7             call right::setLeft(selfCopy)   // Set current as left of  right neighbour
8             delocal Cell selfCopy = nil
9
10            local int cellIndex = index + 1
11            call right::setIndex(cellIndex) // Set index in right neighbour of current
12            delocal int cellIndex = index + 1
13        else skip
14        fi right != nil & cell = nil
15
16        if right != nil then
17            call right::append(cell)        // Keep searching for empty right neighbour
18        else skip
19        fi right != nil
```

Figure 2.10: Doubly Linked List Cell class (cont)

bounds in terms of the current length of the list, we could extend the tail on the list until reach a cell with the wanted index. When we're zero-clearing a value that is the furthest index, the inverse would apply, and a such we would zero-clear the cell, and the deallocate cells until we reach a cell which does not have a zero-cleared *data* field.

This extended doubly linked list would also allow lists of n-dimensional lists, as the type of the *data* field simply could be changed to, say, a *FooDoublyLinkedList*, resulting in an array of Foo arrays.

2.8 Type System

The type system of RoopL++ expands on the type system of RoopL presented by Haulund in [6] and is analogously described by syntax-directed inference typing rules in the style of Winskel in [18]. As RoopL++ introduces two new types in form of *references* and arrays, a few RoopL typing rules must be modified to accommodate these added types. For completeness all typing rules, including unmodified rules, are included in the following sections.

2.8.1 Preliminaries

The types in RoopL++ are given by the following grammar:

$$\tau ::= \textbf{int} \mid c \in \text{ClassIDs} \mid r \in \text{ReferenceIDs} \mid i \in \text{IntegerArrayIDs} \mid o \in \text{ClassArrayIDs}$$

The type environment Π is a finite map paring variables to types, which can be applied to an identifier x using the $\Pi(x)$ notation. Notation $\Pi' = \Pi[x \mapsto \tau]$ defines updates and creation of a new type environment Π' such that $\Pi'(x) = \tau$ and $\Pi'(y) = \Pi(y)$ if $x \neq y$, for some variable identifier x and y. The empty type environment is denoted as $[\,]$ and the function $vars : Expressions \rightarrow \text{VarIDs}$ is described by the following definition

$$\begin{aligned}
&\text{vars}(\bar{n}) && = \emptyset \\
&\text{vars}(\textbf{nil}) && = \emptyset \\
&\text{vars}(x) && = \{\ x\ \} \\
&\text{vars}(x[e]) && = \{\ x[e]\ \} \\
&\text{vars}(e_1 \otimes e_2) && = \text{vars}(e_1) \cup \text{vars}(e_2).
\end{aligned}$$

The binary subtype relation $c_1 \prec: c_2$ is required for supporting subtype polymorphism as is defined as follows

$$\begin{aligned}
&c_1 \prec: c_2 && \text{if } c_1 \text{ inherits from } c_2 \\
&c \prec: c && (reflexivity) \\
&c_1 \prec: c_3 && \text{if } c_1 \prec: c_2 \text{ and } c_2 \prec: c_3\ (transitivity)
\end{aligned}$$

Furthermore, we formally define object models, in such a way that inherited fields and methods are included, unless overridden by the derived fields. Therefore, we define Γ to be the class map of a program p, such that Γ is a finite map from class identifiers to tuples of methods and fields for the class p. Application of a class map Γ to some class cl is denoted as $\Gamma(cl)$. Construction of a class map is done through function *gen*, as shown in figure 2.11. Figure 2.12 defines the $fields$ and *methods* functions to determine these given a class. Set operation $\uplus$ defines method overloading by dropping base class methods if a similarly named method exists in the derived class. The definitions shown in Figure 2.11 and 2.12 are graciously borrowed from [6].

$$\text{gen}\Big(\overbrace{cl_1,\ ...,\ cl_n}^{p}\Big) = \overbrace{\Big[\alpha(cl_1) \mapsto \beta(cl_1),\ ...,\ \alpha(cl_n) \mapsto \beta(cl_n)\Big]}^{\Gamma}$$

$$\alpha\Big(\textbf{class } c\ \cdots\Big) = c \qquad \beta(cl) = \Big(\text{fields}(cl),\ \text{methods}(cl)\Big)$$

Figure 2.11: Definition *gen* for constructing the finite class map Γ of a given program p, originally from [6]

$$\text{fields}(cl) = \begin{cases} \eta(cl) & \text{if } cl \sim\ [\textbf{class } c\ \cdots] \\ \eta(cl) \cup \text{fields}\Big(\alpha^{-1}(c')\Big) & \text{if } cl \sim [\textbf{class } c \textbf{ inherits } c'\ \cdots] \end{cases}$$

$$\text{methods}(cl) = \begin{cases} \delta(cl) & \text{if } cl \sim [\textbf{class } c\ \cdots] \\ \delta(cl) \uplus \text{methods}\Big(\alpha^{-1}(c')\Big) & \text{if } cl \sim [\textbf{class } c \textbf{ inherits } c'\ \cdots] \end{cases}$$

$$A\ \uplus B \stackrel{def}{=}\ A \cup \left\{ m \in B \;\middle|\; \nexists\ m'\Big(\zeta(m') = \zeta(m) \wedge m' \in A\Big) \right\}$$

$$\zeta\Big(\textbf{method } q\ (\cdots)\ s\Big) = q \qquad \eta\Big(\textbf{class } c\ \cdots\ \overbrace{t_1 f_1\ \cdots\ t_n f_n}^{fs}\ \cdots\Big) = fs$$

$$\delta\Big(\textbf{class } c\ \cdots \overbrace{\textbf{method } q_1\ (\cdots)\ s_1\ \cdots\ \textbf{method } q_n\ (\cdots)\ s_n}^{ms}\ \cdots\Big) = ms$$

Figure 2.12: Definition of fields and methods, originally from [6]

Finally, we formally define a link between arrays of a given type and other types. That is, the function *arrayType* defined in figure 2.13 is class c if the passed array a is an array of c instances.

$$\text{arrayType}(a) = \begin{cases} c & \text{if } a \in o \text{ and } a \text{ is a } c \text{ array} \\ \textbf{int} & \text{if } a \in i \end{cases}$$

Figure 2.13: Definition *arrayType* for mapping types of arrays to either class types or the integer type

2.8.2 Expressions

The type judgment

$$\frac{}{\Pi \vdash_{expr} \ e : \tau}$$

defines the type of expressions. The judgment reads as: under type environment Π, expression e has type τ.

$$\frac{}{\Pi \vdash_{expr} \ n : \textbf{int}} \text{ T-Con} \qquad \frac{\Pi(x) = \tau}{\Pi \vdash_{expr} \ x : \tau} \text{ T-Var} \qquad \frac{\tau \neq \textbf{int}}{\Pi \vdash_{expr} \ \textbf{nil} \ : \tau} \text{ T-Nil}$$

$$\frac{\Pi \vdash_{expr} \ e_1 : \textbf{int} \qquad \Pi \vdash_{expr} \ e_2 : \textbf{int}}{\Pi \vdash_{expr} \ e_1 \otimes e_2 : \textbf{int}} \text{ T-BinOpInt}$$

$$\frac{\Pi \vdash_{expr} \ e_1 : \textbf{int} \qquad \Pi \vdash_{expr} \ e_2 : \textbf{int} \qquad \ominus \in \{\textbf{=}, \textbf{!=}\}}{\Pi \vdash_{expr} \ e_1 \otimes e_2 : \textbf{int}} \text{ T-BinOpObj}$$

Figure 2.14: Typing rules for expressions in Roopl, originally from [6]

The original typing rules from Roopl are shown in figure 2.14. The type rules T-Con, T-Var and T-Nil defines typing of the simplest expressions. Numeric literals are of type **int**, typing of variable expressions depends on the type of the variable in the type environment and the **nil** literal is a non-integer type. All Binary operations are defined for integers, while only equality-operators are defined for objects.

With the addition of the Roopl++ array type, we extend the expression typing rules with rule T-ArrElemVar which defines typing for array element variables, shown in figure 2.15.

$$\frac{\Pi(x) = \tau' \qquad \Pi_{expr} \vdash \ e \ : \ \textbf{int} \qquad \Pi(x) = \tau}{\Pi \vdash_{expr} \ x[e] : \tau} \text{ T-ArrElemVar}$$

Figure 2.15: Typing rule extension for the Roopl typing rules

2.8.3 Statements

The type judgment

$$\frac{}{\langle \Pi, \ c \rangle \ \vdash^{\Gamma}_{stmt} \ s}$$

defines well-typed statements. The judgment reads as under type environment Π within class c, statement s is well-typed with class map Γ.

$$\frac{x \notin \text{vars}(e) \qquad \Pi \vdash_{expr} \ e : \textbf{int} \qquad \Pi(x) = \textbf{int}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ x \odot = e} \text{ T-AssVar}$$

$$\frac{\Pi \vdash_{expr} \ e_1 : \textbf{int} \qquad \langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_1 \qquad \langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_2 \qquad \Pi \vdash_{expr} \ e_2 : \textbf{int}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{if } e_1 \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi } e_2} \text{ T-If}$$

$$\frac{\Pi \vdash_{expr} \ e_1 : \textbf{int} \qquad \langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_1 \qquad \langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_2 \qquad \Pi \vdash_{expr} \ e_2 : \textbf{int}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{from } e_1 \textbf{ do } s_1 \textbf{ loop } s_2 \textbf{ until } e_2} \text{ T-Loop}$$

$$\frac{\langle \Pi[x \mapsto c'], c \rangle \vdash^{\Gamma}_{stmt} \ s}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{construct } c' \ x \quad s \quad \textbf{destruct } x} \text{ T-ObjBlock} \qquad \frac{}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{skip}} \text{ T-Skip}$$

$$\frac{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_1 \qquad \langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_2}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ s_1 \ s_2} \text{ T-Seq} \qquad \frac{\Pi(x_1) = \Pi(x_2)}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ x_1 \ \textbf{<=>} \ x_2} \text{ T-SwpVar}$$

$$\frac{\begin{array}{c}\Gamma(\Pi(c)) = \Big(fields, \ methods\Big) \quad \Big(\textbf{method } q(t_1 \ y_1, \ ..., \ t_n \ y_n) \ s\Big) \in methods \\ \{x_1, \ ..., \ x_n\} \cap fields = \emptyset \qquad i \neq j \implies x_i \neq x_j \qquad \Pi(x_1) \prec: t_1 \ \cdots \ \Pi(x_n) \prec: t_n\end{array}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{call } q(x_1, \ ..., \ x_n)} \text{ T-Call}$$

$$\frac{\begin{array}{c}\Gamma(\Pi(x_0)) = \Big(fields, \ methods\Big) \quad \Big(\textbf{method } q(t_1 \ y_1, \ ..., \ t_n \ y_n) \ s\Big) \in methods \\ i \neq j \implies x_i \neq x_j \qquad \Pi(x_1) \prec: t_1 \ \cdots \ \Pi(x_n) \prec: t_n\end{array}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{call } x_0 :: q(x_1, \ ..., \ x_n)} \text{ T-CallO}$$

$$\frac{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{call } q(x_1, \ ..., \ x_n)}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{uncall } q(x_1, \ ..., \ x_n)} \text{ T-Uc} \qquad \frac{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{call } x_0 :: q(x_1, \ ..., \ x_n)}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{uncall } x_0 :: q(x_1, \ ..., \ x_n)} \text{ T-Uco}$$

Figure 2.16: Typing rules for statements in Roopl, originally from [6]

Typing rule T-AssVar defines variable assignments for an integer variable and an integer expression result, given that the variable x does not occur in the expression e.

The type rules T-If and T-Loop defines reversible conditionals and loops as known from Janus, where entry and exit conditions are integers and branch and loop statements are well-typed statements.

The object block, introduced in Roopl, is only well-typed if its body statement is well-typed.

The **skip** statement is always well-typed, while a sequence of statements are well-typed if each of the provided statements are. Variable **swap** statements are well-typed if both operands are of the same type under type environment Π.

As with Roopl, well-typing of local method invocation is defined in rule T-Call iff:

- The number of arguments matches the method arity
- No class fields are present in the arguments passed to the method (To prevent irreversible updates)
- The argument list contains unique elements
- Each argument is a subtype of the type of the equivalent formal parameter.

For foreign method invocations, typing rule T-CallO. A foreign method invocation is well-typed using the same rules as for T-Call besides having no restrictions on class fields parameters in the arguments, but an added rule stating that the callee object x_0 must not be passed as an argument.

The typing rules T-UC and T-UCO defines uncalling of methods in terms of their respective inverse counterparts.

$$\frac{x \in \text{IntegerArrayIDs} \qquad \Pi \vdash_{expr} e_1 \ : \ \textbf{int} \qquad x[e_1] \notin \text{vars}(e_2) \qquad \Pi \vdash_{expr} \ e_2 : \textbf{int}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ x[e_1] \odot = e_2} \text{T-ArrElemAss}$$

$$\frac{\Pi(x) = \textbf{nil}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c' \ x} \text{T-ObjNew} \qquad \frac{\Pi(x) = c'}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ c' \ x} \text{T-ObjDlt}$$

$$\frac{\text{arrayType}(a) \in \left\{\text{classIDs}, \textbf{int}\right\} \qquad \Pi \vdash_{expr} \ e = \textbf{int} \qquad \Pi(x) = \textbf{nil}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ a[e] \ x} \text{T-ArrNew}$$

$$\frac{\text{arrayType}(a) \in \left\{\text{classIDs}, \textbf{int}\right\} \qquad \Pi \vdash_{expr} \ e = \textbf{int} \qquad \Pi(x) = a}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ a[e] \ x} \text{T-ArrDlt}$$

$$\frac{\Pi(x) = c' \qquad \Pi(x') = \textbf{nil}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{copy} \ c' \ x \ x'} \text{T-Cp} \qquad \frac{\Pi(x) = c' \qquad \Pi(x') = c'}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{uncopy} \ c' \ x \ x'} \text{T-Ucp}$$

$$\frac{\langle \Pi, c \rangle \vdash^{\Gamma}_{expr} \ e_1 \qquad \langle \Pi[x \mapsto c'], c \rangle \vdash^{\Gamma}_{stmt} \ s \qquad \langle \Pi, c \rangle \vdash^{\Gamma}_{expr} \ e_2}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{local} \ c' \ x = e_1 \quad s \quad \textbf{delocal} \ c' \ x = e_2} \text{T-LocalBlock}$$

Figure 2.17: Typing rules extensions for statements in Roopl++

Figure 2.17 shows the typing rules for the extensions made to Roopl in Roopl++ , covering the **new**/**delete** and **copy**/**uncopy** statements for objects and arrays and local blocks.

The typing rule T-ArrElemAss defines assignment to integer array element variables, and is well-typed when the type of array x is **int**, the variable $x[e_1]$ is not present in the right-hand side of the statement and both expressions e_1 and e_2 evaluates to integers.

The T-ObjNew and T-ObjDlt rules define well-typed **new** and **delete** statements for dynamically life-timed objects. The **new** statement is well-typed, as long as $c' \in$ classIDs and the variable x is a **nil**-type and the **delete** is well-typed if the type of x under type environment Π is equal to c'.

The T-ArrNew and T-ArrDlt rules define well-type **new** and **delete** statement for Roopl++ arrays. The **new** statement is well-typed, if the type of the array either is a classID or **int**, the length expression evaluates to an integer and x is zero-cleared, and **delete** is well-typed if the type of the array is either a classID or **int**, the length expression evaluates to an integer and x is equal to the array type a.

Typing rules T-Cp and T-Ucp define well-typed reference copy and un-copying statements. A well-typed **copy** statement requires that the type of x is c' under type environment Π, while a well-typed **uncopy** statement further requires that the type of x' is c' too.

The rule T-LocalBlock defines well-typed local blocks. A local block is well-typed if its two expression e_1 and e_2 are well-typed and its body statement s is well-typed.

2.8.4 Programs

As with Roopl, a Roopl++ program is well-typed if all of its classes and their respective methods are well-typed and if there exists a nullary main methods. Figure 2.18 shows the typing rules for class methods, classes and programs.

$$\frac{\langle \Pi[x_1 \mapsto t_1,\ ...,\ x_n \mapsto t_n],\ c\rangle \vdash^{\Gamma}_{stmt}\ \ s}{\langle \Pi, c\rangle \vdash^{\Gamma}_{meth}\ \ \textbf{method}\ q(t_1 x_1,\ ...,\ t_n x_n)\ s}\ \text{T-Method}$$

$$\frac{\Gamma(c) = \Big(\overbrace{\big\{\langle t_1, f_1\rangle,\ ...,\ \langle t_i, f_i\rangle\big\}}^{fields},\ \overbrace{\{m_1,\ ...,\ m_n\}}^{methods}\Big) \qquad \Pi = \big[f_1 \mapsto t_1,\ ...,\ f_n \mapsto t_n\big] \qquad \langle \Pi, c\rangle \vdash^{\Gamma}_{meth}\ m_1 \quad \cdots \quad \langle \Pi, c\rangle \vdash^{\Gamma}_{meth}\ m_n}{\vdash^{\Gamma}_{class}\ \ c}\ \text{T-Class}$$

$$\frac{\Big(\textbf{method main}\ ()\ s\Big) \in \bigcup_{i=1}^{n} \text{methods}(c_i) \qquad \Gamma = \text{gen}(c_1,\ ...,\ c_n) \qquad \vdash^{\Gamma}_{class}\ c_1 \quad \cdots \quad \vdash^{\Gamma}_{class}\ c_n}{\vdash_{prog}\ \ c_1\ \cdots\ c_n}\ \text{T-Prog}$$

Figure 2.18: Typing rules for class methods, classes and programs, originally from [6].

2.9 Language Semantics

The following sections contains the operational semantics of ROOPL++ , as specified by syntax-directed inference rules.

2.9.1 Preliminaries

We define a memory location l to be a single location in program memory, where a memory location is in the set of non-negative integers, $\mathbb{N}_0$. An environment γ is a partial function mapping variables to memory locations. A store μ is a partial function mapping memory locations to values. An object is a tuple of a class name and an environment mapping fields to memory locations. A value is either an integer, an object or a memory location.

Applications of environments γ and stores μ are analogous to the type environment Γ, defined in section 2.8.1.

$$
\begin{aligned}
l \in \text{Locs} &= \mathbb{N}_0 \\
\gamma \in \text{Envs} &= \text{VarIDs} \rightharpoonup \text{Locs} \\
\mu \in \text{Stores} &= \text{Locs} \rightharpoonup \text{Values} \\
\text{Objects} &= \Big\{ \langle c_f,\ \gamma_f \rangle \mid c_f \in \text{ClassIDs} \ \wedge \ \gamma_f \in \text{Envs} \Big\} \\
v \in \text{Values} &= \mathbb{Z} \cup \text{Objects} \ \cup \text{Locs}
\end{aligned}
$$

Figure 2.19: Semantic values, originally from [6].

2.9.2 Expressions

The judgment:

$$\langle \gamma, \mu \rangle \vdash_{expr} \ e \Rightarrow v$$

defines the meaning of expressions. We say that under environment γ and store μ, expression e evaluates to value v.

$$\frac{}{\langle \gamma, \mu \rangle \vdash_{expr} \ n \Rightarrow \bar{n}}\ \textsc{Con} \qquad \frac{}{\langle \gamma, \mu \rangle \vdash_{expr} \ x \Rightarrow \mu\Big(\gamma(x)\Big)}\ \textsc{Var} \qquad \frac{}{\langle \gamma, \mu \rangle \vdash_{expr} \ \texttt{nil} \Rightarrow 0}\ \textsc{Nil}$$

$$\frac{\langle \gamma, \mu \rangle \vdash_{expr} \ e_1 \Rightarrow v_1 \qquad \langle \gamma, \mu \rangle \vdash_{expr} \ e_2 \Rightarrow v_2 \qquad [\![\otimes]\!](v_1, v_2) = v}{\langle \gamma, \mu \rangle \vdash_{expr} \ e_1 \otimes e_2 \Rightarrow v}\ \textsc{BinOp}$$

Figure 2.20: Semantic inference rules for expressions, originally from [6]

As shown in figure 2.20, expression evaluation has no effects on the store. Logical values are represented by *truthy* and *falsy* values of any non-zero value and zero respectively. Evaluation of binary operators is presented in figure 2.22.

For ROOPL++ , we extend the expression ruleset with a single rule for array element variables shown in figure 2.21. As with the expressions inference rules in ROOPL, this extension has no effects on the store.

$$\frac{\langle \gamma, \mu \rangle \vdash_{expr} \ e \Rightarrow v}{\langle \gamma, \mu \rangle \vdash_{expr} \ x[e] \Rightarrow \mu\Big(\gamma(x[v])\Big)} \ \textsc{ArrElemVar}$$

Figure 2.21: Extension to the semantic inference rules for expression in ROOPL++

$$\begin{aligned}
[\![\texttt{+}]\!](v_1, v_2) &= v_1 + v_2 & [\![\texttt{\%}]\!](v_1, v_2) &= v_1 \bmod v_2 \\
[\![\texttt{-}]\!](v_1, v_2) &= v_1 - v_2 & [\![\texttt{\&}]\!](v_1, v_2) &= v_1 \wedge v_2 \\
[\![\texttt{*}]\!](v_1, v_2) &= v_1 \times v_2 & [\![\texttt{|}]\!](v_1, v_2) &= v_1 \vee v_2 \\
[\![\texttt{/}]\!](v_1, v_2) &= v_1 \div v_2 & [\![\texttt{\^{}}]\!](v_1, v_2) &= v_1 \oplus v_2 \\
[\![\texttt{\&\&}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = 0 \vee v_2 = 0 \\ 1 & \text{otherwise} \end{cases} & [\![\texttt{<=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \leq v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{||}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 = 0 \\ 1 & \text{otherwise} \end{cases} & [\![\texttt{>=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \geq v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{<}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 < v_2 \\ 1 & \text{otherwise} \end{cases} & [\![\texttt{=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\![\texttt{>}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 > v_2 \\ 1 & \text{otherwise} \end{cases} & [\![\texttt{!=}]\!](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \neq v_2 \\ 1 & \text{otherwise} \end{cases}
\end{aligned}$$

Figure 2.22: Definition of binary expression operator evaluation

2.9.3 Statements

The judgment

$$\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ s : \mu \rightleftharpoons \mu'$$

defines the meaning of statements. We say that under environment γ and object l, statement s with class map Γ reversibly transforms store μ to store μ', where l is the location of the current object in the store. Figure 2.23a, 2.23b and 2.23c defines the operational semantics of ROOPL++ .

The inference rule SKIP defines the operational semantics of **skip** statements and has no effects on the store μ.

Rule SEQ defines statement sequences where the store potentially is updated between each statement execution.

$$\frac{}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ \textbf{skip} : \mu \rightleftharpoons \mu} \textsc{Skip}$$

$$\frac{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ s_1 : \mu \rightleftharpoons \mu' \qquad \langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ s_2 : \mu' \rightleftharpoons \mu''}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ s_1 \ s_2 : \mu \rightleftharpoons \mu''} \textsc{Seq}$$

$$\frac{\langle \gamma,\mu\rangle \vdash^{\Gamma}_{stmt} \ e \Rightarrow v \qquad [\![\odot]\!]\Big(\mu\Big(\gamma(x)\Big), v\Big) = v'}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ x \odot = e \ : \ \mu \rightleftharpoons \mu[\gamma(x) \mapsto v']} \textsc{AssVar}$$

$$\frac{\mu\Big(\gamma(x_1)\Big) = v_1 \qquad \mu\Big(\gamma(x_2)\Big) = v_2}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ x_1 \ \textbf{<=>} \ x_2 \ : \ \mu \rightleftharpoons \mu[\gamma(x_1) \mapsto v_2, \ \gamma(x_2) \mapsto v_1]} \textsc{SwpVar}$$

$$\frac{\langle \gamma,\mu\rangle \vdash^{\Gamma}_{expr} \ e_1 \not\Rightarrow 0 \qquad \langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ s_1 : \mu \rightleftharpoons \mu' \qquad \langle l,\gamma\rangle \vdash^{\Gamma}_{loop} \ (e_1, s_1, s_2, e_2) : \mu' \rightleftharpoons \mu''}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ \textbf{from} \ e_1 \ \textbf{do} \ s_1 \ \textbf{loop} \ s_2 \ \textbf{until} \ e_2 \ : \ \mu \rightleftharpoons \mu''} \textsc{LoopMain}$$

$$\frac{\langle \gamma,\mu\rangle \vdash^{\Gamma}_{expr} \ e_2 \not\Rightarrow 0}{\langle l,\gamma\rangle \vdash^{\Gamma}_{loop} \ (e_1, s_1, s_2, e_2) \ : \ \mu \rightleftharpoons \mu} \textsc{LoopBase}$$

$$\frac{\begin{array}{cc}\langle \gamma,\mu\rangle \vdash^{\Gamma}_{expr} e_2 \Rightarrow 0 & \langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} s_2 : \mu \rightleftharpoons \mu' \\ \langle \gamma,\mu'\rangle \vdash^{\Gamma}_{expr} e_1 \Rightarrow 0 \qquad \langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} s_1 : \mu' \rightleftharpoons \mu'' & \langle l,\gamma\rangle \vdash^{\Gamma}_{loop} (e_1, s_1, s_2, e_2) : \mu'' \rightleftharpoons \mu'''\end{array}}{\langle l,\gamma\rangle \vdash^{\Gamma}_{loop} \ (e_1, s_1, s_2, e_2) \ : \ \mu \rightleftharpoons \mu'''} \textsc{LoopRec}$$

$$\frac{\langle \gamma,\mu\rangle \vdash^{\Gamma}_{expr} e_1 \not\Rightarrow 0 \qquad \langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} s_1 : \mu \rightleftharpoons \mu' \qquad \langle \gamma,\mu'\rangle \vdash^{\Gamma}_{expr} e_2 \not\Rightarrow 0}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ \textbf{if} \ e_1 \ \textbf{then} \ s_1 \ \textbf{else} \ s_2 \ \textbf{fi} \ e_2 \ : \ \mu \rightleftharpoons \mu'} \textsc{IfTrue}$$

$$\frac{\langle \gamma,\mu\rangle \vdash^{\Gamma}_{expr} e_1 \Rightarrow 0 \qquad \langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} s_1 : \mu \rightleftharpoons \mu' \qquad \langle \gamma,\mu'\rangle \vdash^{\Gamma}_{expr} e_2 \Rightarrow 0}{\langle l,\gamma\rangle \vdash^{\Gamma}_{stmt} \ \textbf{if} \ e_1 \ \textbf{then} \ s_1 \ \textbf{else} \ s_2 \ \textbf{fi} \ e_2 \ : \ \mu \rightleftharpoons \mu'} \textsc{IfFalse}$$

Figure 2.23a: Semantic inference rules for statements, originally from [6]

Rule AssVar defines reversible assignment in which variable identifier x under environment γ is mapped to the value v' resulting in an updated store μ'. For variable swapping SwpVar defines how value mappings between two variables are exchanged in the updated store.

For loops and conditionals, Rules LoopMain, LoopBase and LoopRec defines the meaning of loop statements and IfTrue and IfFalse, similarly to the operational semantics of Janus, as presented in [19]. LoopMain is entered if e_1 is true and each iteration enters LoopRec until e_2 is false, in which LoopBase is executed. Similarly, if e_1 and e_2 are true, rule IfTrue is entered, executing the then-branch of the conditional. If e_1 and e_2 are false, the IfFalse rule is executed

$$\frac{\mu(l) = \langle c, \gamma' \rangle \quad \Gamma(c) = (fields, methods) \quad \Big(\textbf{method } q(t_1 y_1,\ ...,\ t_n y_n)\ s\Big) \in methods \quad \Big\langle l, \gamma'[y_1 \mapsto \gamma(x_1),\ ...,\ y_n \mapsto \gamma(x_n)]\Big\rangle \vdash^{\Gamma}_{stmt}\ s : \mu \rightleftharpoons \mu'}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \textbf{call } q(x_1,\ ...,\ x_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{Call}$$

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \textbf{call } q(x_1,\ ...,\ x_n)\ :\ \mu' \rightleftharpoons \mu}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \textbf{uncall } q(x_1,\ ...,\ x_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{Uncall}$$

$$\frac{l' = \mu\Big(\gamma(x_0)\Big) \quad \mu(l') = \langle c, \gamma' \rangle \quad \Gamma(c) = (fields, methods) \quad \Big(\textbf{method } q(t_1 y_1,\ ...,\ t_n y_n)\ s\Big) \in methods \quad \Big\langle l', \gamma'[y_1 \mapsto \gamma(x_1),\ ...,\ y_n \mapsto \gamma(x_n)]\Big\rangle \vdash^{\Gamma}_{stmt}\ s : \mu \rightleftharpoons \mu'}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \textbf{call } x_0 :: q(x_1,\ ...,\ x_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{CallObj}$$

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \text{call } x_0 :: q(x_1,\ ...,\ x_n)\ :\ \mu' \rightleftharpoons \mu}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \text{uncall } x_0 :: q(x_1,\ ...,\ x_n)\ :\ \mu \rightleftharpoons \mu'}\ \textsc{ObjUncall}$$

Figure 2.23b: Semantic inference rules for statements, originally from [6] (cont)

$$\frac{\Gamma(c) = \Big(\overbrace{\{\langle t_1, f_1\rangle,\ ...,\ \langle t_n, f_n\rangle\}}^{fields}, methods\Big) \quad \gamma' = [f_1 \mapsto a_1,\ ...,\ f_n \mapsto a_n] \quad \{l', r, a_1, ..., a_n\} \cap \text{dom}(\mu) = \emptyset \quad |\{l', r, a_1, ..., a_n\}| = n + 2 \quad \mu' = \mu\Big[a_1 \mapsto 0,\ ...,\ a_n \mapsto 0,\ l' \mapsto \langle c, \gamma' \rangle,\ r \mapsto l'\Big] \quad \langle l, \gamma[x \mapsto r] \rangle \vdash^{\Gamma}_{stmt}\ s : \mu' \rightleftharpoons \mu'' \quad \mu''(a_1) = 0\ \cdots\ \mu''(a_n) = 0}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt}\ \textbf{construct } c\ x\quad s\quad \textbf{destruct } x\ :\ \mu \rightleftharpoons \mu'' \restriction_{\text{dom}(\mu)}}\ \textsc{ObjBlock}$$

Figure 2.23c: Semantic inference rules for statements, originally from [6] (cont)

and the else-branch is executed.

As presented in the operational semantics for ROOPL, rules CALL, UNCALL, CALLOBJ and UNCALLOBJ respectively defines local and non-local method invocations. For local methods, method q in current class c should be of arity n matching the number of arguments. The updated store μ' is obtained after statement body execution in the object environment. As local uncalling is the inverse of local calling, the direction of execution is simply reversed, and as such the input store a **call** statement serves as the output store of the **uncall** statement, similarly to techniques presented in [21, 19].

The statically scoped object blocks are defined in rule OBJBLOCK. The operation semantics of these blocks are similar to **local**-blocks from JANUS. The new memory locations l', r and $a_1,\ ...,\ a_n$ must be unused in store μ. The updated store μ' contains location l' mapped to the

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_1 \Rightarrow v_1 \qquad \langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_2 \Rightarrow v_2 \qquad [\![\odot]\!]\Big(\mu\Big(\gamma(x[v_1])\Big), v_2\Big) = v_3}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ x[e_1] \odot = e_2 \ : \ \mu \rightleftharpoons \mu[\gamma(x[v_1]) \mapsto v_3]} \ \textsc{AssArrElemVar}$$

$$\Gamma(c) = \Big(\overbrace{\{\langle t_1, f_1 \rangle, \ ..., \ \langle t_n, f_n \rangle\}}^{fields}, methods\Big) \qquad \gamma' = [f_1 \mapsto a_1, \ ..., \ f_n \mapsto a_n]$$

$$\{l', r, a_1, ..., a_n\} \cap \mathrm{dom}(\mu) = [r \mapsto 0]$$

$$\frac{|\{l', r, a_1, ..., a_n\}| = n + 2 \qquad \mu' = \mu\Big[a_1 \mapsto 0, \ ..., \ a_n \mapsto 0, \ l' \mapsto \langle c, \gamma' \rangle, \ r \mapsto l'\Big]}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c \ x \ : \ \mu \rightleftharpoons \mu'[\gamma(x) \mapsto r]} \ \textsc{ObjNew}$$

$$\frac{\mu\Big(\gamma(x)\Big) = r \qquad \mu\Big(\gamma(r)\Big) = l' \qquad \mu' = \mu\Big[l' \mapsto 0, \ r \mapsto 0\Big]}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ c \ x \ : \ \mu \rightleftharpoons \mu' \setminus \{\gamma(x)\}} \ \textsc{ObjDelete}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e \Rightarrow v \qquad \gamma' = [0 \mapsto a_1, \ ..., \ v \mapsto a_n] \qquad \{l', r, v', a_1, ..., a_n\} \cap \mathrm{dom}(\mu) = [r \mapsto 0]}{|\{l', r, v', a_1, ..., a_n\}| = n + 3 \qquad \mu' = \mu\Big[a_1 \mapsto 0, \ ..., \ a_n \mapsto 0, \ l' \mapsto \langle a, \gamma' \rangle, \ r \mapsto l', x_s \mapsto v\Big]}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ a[e] \ x \ : \ \mu \rightleftharpoons \mu'[\gamma(x) \mapsto r]} \ \textsc{ArrNew}$$

$$\frac{\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e \Rightarrow v \qquad \mu\Big(\gamma(x_s)\Big) = v \qquad \mu\Big(\gamma(x)\Big) = r \qquad \mu\Big(\gamma(r)\Big) = l' \qquad \mu' = \mu\Big[l' \mapsto 0, r \mapsto 0, x_s \mapsto 0\Big]}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ a[e] \ x \ : \ \mu \rightleftharpoons \mu' \setminus \{\gamma(x)\}} \ \textsc{ArrDelete}$$

Figure 2.23d: Extension to the semantic inference rules for statements in Roopl++

object tuple $\langle c, \gamma' \rangle$, an object reference r mapped to l' and all object fields mapped to value 0. The result store μ'' is obtained after executing the body statement s in store μ' mapping x to object reference r, as long as all object fields are zero-cleared in μ'' afterwards. If any of these conditions fail, the object block statement is undefined.

Figures 2.23d and 2.23e show the extensions to the semantics of Roopl with rules for **new**/**delete** and **copy**/**uncopy** statements, array element assignment and local blocks.

Rule AssArrElemVar defines reversible assignment to array elements. After evaluating expressions e_1 to v_1 and e_2 to v_2, variable $x[v_1]$ under environment γ is mapped to the value v_3 resulting in an updated store μ'.

Dynamically life-timed object construction and destruction is defined by rules ObjNew and ObjDelete. For construction, location l' and $a_1, \ ..., \ a_n$ must once again be unused in the store. Unlike, in the object block rule, the reference r must be defined in the store, pointing to 0. Analogously to the object block, a new store is obtained by mapping location l' to the

$$\frac{\mu\Big(\gamma(x)\Big) = r \qquad \mu\Big(\gamma(r)\Big) = l \qquad \mu\Big(\gamma(l)\Big) = \langle c, \gamma' \rangle}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{copy}\ c\ x\ x' \ :\ \mu \leftrightharpoons \mu[x' \mapsto r]}\ \textsc{Copy}$$

$$\frac{\mu\Big(\gamma(x)\Big) = r \qquad \mu\Big(\gamma(x')\Big) = r}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{uncopy}\ c\ x\ x' \ :\ \mu \setminus \{\gamma(x')\}}\ \textsc{Uncopy}$$

$$\frac{\begin{array}{c}\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_1 \Rightarrow v_1 \qquad \langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_2 \Rightarrow v_2 \qquad \{l', r\} \cap \mathrm{dom}(\mu) = \emptyset \\ \mu' = \mu[l' \mapsto v_1, r \mapsto l'] \\ \langle l, \gamma[x \mapsto r] \rangle \vdash^{\Gamma}_{stmt} \ s : \mu' \rightleftharpoons \mu'' \qquad \{r\} \cap \mathrm{dom}(\mu'') = [r \mapsto v_2] \qquad \mu''' = \mu''[r \mapsto 0, l' \mapsto 0]\end{array}}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{local}\ c\ x = e_1 \quad s \quad \textbf{delocal}\ x = e_2 \ :\ \mu \rightleftharpoons \mu''' \restriction_{\mathrm{dom}(\mu, \mu'')}}\ \textsc{LocalBlock}$$

Figure 2.23e: Extension to the semantic inference rules for statements in ROOPL++ (cont)

object tuple, and r to l' and zero-initializing the object fields. Unlike object blocks, this is the resulting state of the construction statement. For destruction, x must map to a reference r which maps to a location l'. A new store μ' is obtained my resetting mappings of r and l' to be unused (zero-cleared). As with object blocks, it is the program itself responsible for zero-clearing object fields before destruction. If the object fields are not zero-cleared, the OBJDELETE statement is undefined.

Array construction and destruction is very similar to object construction and destruction. The major difference is we bind the evaluated expression size of the array we are constructing to the variable x_s in the store. For deletion, this x_s in the store must match the passed evaluated expression.

Object and array referencing is defined by rules COPY and UNCOPY. A references is created and a new store μ' obtained by mapping x' to the reference r which x current maps to, if c matches the tuple mapped to the location l. A references is removed and a new store μ' obtained if x and x' maps to the same reference r and x' then is removed from the store.

Local blocks are as previously mentioned, semantically similar to object blocks, where the memory locations l', r must be unused in the store μ. The updated store μ' contains location l' mapped to the evaluated value of e_1, v_1 and the reference r mapped to l'. The result store after body statement execution, μ'' must have l' mapped to the expression value of e_2, v_2. Before the local block terminates, a third store update is executed, clearing the used memory locations, such that l' and r are mapped to zero and as such become unused again.

2.9.4 Programs

The judgment

$$\vdash_{prog} p \Rightarrow \sigma$$

defines the meaning of programs. The class p containing the main method is instantiated and the main function is executed with the partial function σ as the result, mapping variable identifiers to values, correlating to the class fields of the main class.

$$\frac{\Gamma = \text{gen}(c_1,\ ...,\ c_n) \qquad \Gamma(c) = \Big(\overbrace{\{\langle t_1, f_1\rangle,\ ...,\ \langle t_n, f_n\rangle\}}^{fields},\ methods\Big) \qquad \Big(\textbf{method main ()}\ s\Big) \in methods \qquad \gamma = [f_1 \mapsto 1,\ ...,\ f_i \mapsto i] \qquad \mu = [1 \mapsto 0,\ ...,\ i \mapsto 0,\ i+1 \mapsto \langle c, \gamma\rangle] \qquad \langle i+1, \gamma\rangle \vdash^{\Gamma}_{stmt}\ s\ :\ \mu \rightleftharpoons \mu'}{\vdash_{prog}\ c_1\ \cdots\ c_n \Rightarrow (\mu' \circ \gamma)}\ \textsc{Main}$$

Figure 2.24: Semantic inference rules for programs, originally from [6]

As with ROOPL programs, the fields of the main method in the main class c are bound in a new environment, starting at memory address 1, as 0 is reserved for **nil**. The fields are zero-initialized in the new store μ and address $i + 1$ which maps to the new instance of c. After body execution, store μ' is obtained. The function $\mu' \circ \gamma$ maps class fields to their respective final values and serves as output of program p.

2.10 Program Inversion

$$\begin{aligned}
&\mathcal{I}[\![\textbf{skip}]\!] = \textbf{skip} && \mathcal{I}[\![s_1\ s_2]\!] = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]\\
&\mathcal{I}[\![x\ \textbf{+=}\ e]\!] = x\ \textbf{-=}\ e && \mathcal{I}[\![x\ \textbf{-=}\ e]\!] = x\ \textbf{+=}\ e\\
&\mathcal{I}[\![x\ \textbf{\^{}=}\ e]\!] = x\ \textbf{\^{}=}\ e && \mathcal{I}[\![x\ \textbf{<=>}\ e]\!] = x\ \textbf{<=>}\ e\\
&\mathcal{I}[\![\textbf{new}\ c\ x]\!] = \textbf{delete}\ c\ x && \mathcal{I}[\![\textbf{copy}\ c\ x\ x']\!] = \textbf{uncopy}\ c\ x\ x'\\
&\mathcal{I}[\![\textbf{delete}\ c\ x]\!] = \textbf{new}\ c\ x && \mathcal{I}[\![\textbf{uncopy}\ c\ x\ x']\!] = \textbf{copy}\ c\ x\ x'\\
&\mathcal{I}[\![\textbf{call}\ q(\ldots)]\!] = \textbf{uncall}\ q(\ldots) && \mathcal{I}[\![\textbf{call}\ x :: q(\ldots)]\!] = \textbf{uncall}\ x :: q(\ldots)\\
&\mathcal{I}[\![\textbf{uncall}\ q(\ldots)]\!] = \textbf{call}\ q(\ldots) && \mathcal{I}[\![\textbf{uncall}\ x :: q(\ldots)]\!] = \textbf{call}\ x :: q(\ldots)\\
&\mathcal{I}[\![\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ e_2]\!] && = \textbf{if}\ e_1\ \textbf{then}\ \mathcal{I}[\![s_1]\!]\ \textbf{else}\ \mathcal{I}[\![s_2]\!]\ \textbf{fi}\ e_2\\
&\mathcal{I}[\![\textbf{from}\ e_1\ \textbf{do}\ s_1\ \textbf{loop}\ s_2\ \textbf{until}\ e_2]\!] && = \textbf{from}\ e_1\ \textbf{do}\ \mathcal{I}[\![s_1]\!]\ \textbf{loop}\ \mathcal{I}[\![s_2]\!]\ \textbf{until}\ e_2\\
&\mathcal{I}[\![\textbf{construct}\ c\ x\quad s\quad \textbf{destruct}\ x]\!] && = \textbf{construct}\ c\ x\quad \mathcal{I}[\![s]\!]\quad \textbf{destruct}\ x\\
&\mathcal{I}[\![\textbf{local}\ t\ x\ = e\quad s\quad \textbf{delocal}\ t\ x\ = e]\!] && = \textbf{local}\ t\ x = e\quad \mathcal{I}[\![s]\!]\quad \textbf{delocal}\ t\ x\ = e
\end{aligned}$$

Figure 2.25: ROOPL++ statement inverter

2.11 Computational Strength

Traditional, non-reversible programming languages have their computational strength measured in terms of their abilities to simulate the Turing machine (TM). If any arbitrary Turing machine can be implemented in some programming language, the language is said to be computationally

universal or Turing-complete. In essence, Turing-completeness marks when a language can compute all computable functions. Reversible programming languages, like JANUS, ROOPL and ROOPL++ , are not Turing-complete as they only are capable of computing injective, computable functions.

For determining computing strength of reversible programming languages, Yokoyama et al. suggests that the reversible Turing machine (RTM) could serve as the baseline criterion [19]. As such, if a reversible programming language is reversibly universal or r-Turing complete if it is able to simulate a reversible Turing machine cleanly, i.e. without generating garbage data. If garbage was on the tape, the function simulated by the machine would not be an injective function and as such, no garbage should be left after termination of the simulation.

2.11.1 Reversible Turing Machines

Before we show that ROOPL++ in fact is r-Turing complete, we present the formalized reversible Turing machine definition, as used in [19].

Definition 2.1. *(Quadruple Turing Machine)*

A TM T is a tuple $(Q,\ \Gamma,\ b,\ \delta,\ q_s,\ q_f)$ where

Q *is the finite non-empty set of states*

Γ *is the finite non-empty set of tape alphabet symbols*

$b\ \in\ \Gamma$ *is the blank symbol*

$\delta\ :\ (Q\ \times\ \Gamma\ \times\ \Gamma\ \times\ Q)\ \cup\ (Q\ \times\ \{/\}\ \times\ \{L,\ R\}\ \times\ Q)$ *is the partial function representing the transitions*

$q_s\ \in\ Q$ *is the starting state*

$q_f\ \in\ Q$ *is the final state*

The symbols L and R represent the tape head shift-directions left and right. A quadruple is either a symbol rule of the form $(q_1,\ s_1,\ s_2,\ q_2)$ or a shift rule of the form $(q_1,\ /,\ d,\ q_2)$ where $q_1 \in Q$, $q_2 \in Q$, $s_1 \in \Gamma$, $s_2 \in \Gamma$ and d being either L or R.

A symbol rule $(q_1,\ s_1,\ s_2,\ q_2)$ means that in state q_1, when reading s_1 from the tape, write s_2 to the tape and change to state q_2. A shift rule $(q_1,\ /,\ d,\ q_2)$ means that in state q_1, move the tape head in direction d and change to state q_2.

Definition 2.2. *(Reversible Turing Machine)*

A TM T is a reversible TM iff, for any distinct pair of quadruples $(q_1,\ s_1,\ s_2,\ q_2)\ \in\ \delta_T$ and $(q_1',\ s_1',\ s_2',\ q_2')\ \in\ \delta_T$, we have

$$q_1\ =\ q_1'\ \implies\ (t_1\ \neq\ /\quad\wedge\quad t_1'\ \neq\ /\quad\wedge\quad t_1\ \neq\ t_1')$$ *(forward determinism)*

$$q_2\ =\ q_2'\ \implies\ (t_1\ \neq\ /\quad\wedge\quad t_1'\ \neq\ /\quad\wedge\quad t_2\ \neq\ t_2')$$ *(backward determinism)*

Haulund describe a RTM simulation implemented in ROOPL by representing the set of states $\{q_1, \ldots, q_n\}$ and the tape alphabet Γ as integers and the rule / and direction symbols L and R as the uppercase integer literals **SLASH**, **LEFT** and **RIGHT**. As ROOPL contains no array or stack primitives, the transition table δ was suggested to be represented as a linked list of objects containing four integers **q1**, **s1**, **s2** and **q2** each, where **s1** equals **SLASH** for shift rules. In ROOPL++ , we do, however, has an array primitive and as such, we can simply simulate transitions, by having rules **q1**, **s1**, **s2** and **q2** represented as arrays, where the number of cells in each array is **PC_MAX**, in a similar fashion as in [19].

2.11.2 Tape Representation

As with regular Turing machines, the tape of the machine is of infinite length. Therefore, we must simulate tape growth in either direction. Yokoyama et al. represented the tape using two stack primitives in the Janus RTM interpreter and Haulund used list of objects. In ROOPL++ , we could implement a stack, as objects are not statically scoped as in ROOPL. However, in terms of easy of use, a doubly linked-list implementation, of simple objects containing a *value* and *left* and *right* field, is more intuitive.

As such, the tape head hovers a tape cell by inspecting a specific element of the doubly linked-list tape representation. If the tape head needs in a direction, but the current cell has no neighbours in said direction, we simply allocate a new cell object and attach as the neighbour. Checking if a cell object is either end of the list is simply checking if its *left* or *right* fields are uninitialized, resulting in the **nil** value for objects.

2.11.3 Reversible Turing Machine Simulation

Figure 2.26 shows the modified method *inst* from [19], which executes a single instruction given a the current state, symbol, program counter and the four arrays representing the transition rules.

```
1    method inst(int state, int symbol, int q1[], int s1[], int s2[],
2                int q2[], int pc, Cell tape, Cell left, Cell right)
3        if (state = q1[pc]) && (symbol = s1[pc]) then   // Symbol rule:
4            state += q2[pc]-q1[pc]                      // set state to q2[pc]
5            symbol += s2[pc]-s1[pc]                     // set symbol to s2[pc]
6        fi (state = q2[pc]) && (symbol = s2[pc])
7        if (state = q1[pc]) && (s1[pc] = SLASH) then    // Move rule:
8            state += q2[pc]-q1[pc]                      // set state to q2[pc]
9            if s2[pc] = RIGHT then
10               call move_tape(tape, left, right, 1)    // Move tape head right
11           fi s2[pc] = RIGHT
12           if s2[pc] = LEFT then
13           call move_tape(tape, left, right -1)        // Move tape head left
14           fi s2[pc] = LEFT
15       fi (state = q2[pc]) && (s1[pc] = SLASH)
```

Figure 2.26: Method for executing a single TM transition

Figure 2.28 shows the simulate method which is the main method responsible for running the RTM simulation. The tape is extended in either direction when needed and the program counter it incremented.

```
1     method move_tape(Cell tape, Cell left, Cell right, int direction)
2         if (direction = 1) then                 // Right direction
3             local Cell tmp = nil
4                 if (right = nil) then           // expand tape
5                     new Cell cell               // Create new cell. This reference is not used again
6                     copy Cell cell r1           // create reference to new cell
7                     copy Cell cell r2           // create reference to new cell
8                     uncall tape::getRight(r1)   // Set right cell reference
9                     right <=> r2                // set right = r, r = nil
10                fi (right != nil)
11
12                call tape::getLeft(tmp)         // Fetch original Left object
13                if (tmp != nil) then
14                    left ^= tmp                 // Destroy reference to left (note: impossible currently)
15                fi (tmp != nil)
16                uncall tape::getLeft(tmp)       // Put original left object back
17                tape <=> left                   // left = tape, tape = nil
18                left <=> right                  // tape = right, right = nil
19
20                call tape::getRight(tmp)        // Fetch right object of tape head cell
21                if (tmp != nil) then
22                    copy Cell right_new tmp     // Create reference to new right
23                    right <=> right_new         // set right to point at new right neighbour
24                fi (tmp != nil)
25                uncall tape:getRight(tmp)       // Put object back
26            delocal Cell tmp = nil
27        fi (direction = 1)
28
29        if (direction = -1) then                // Left direction
30            ...                                 // Similar to above, with all left and rights swapped
31        fi (direction = -1)
```

Figure 2.27: Method for moving the tape head in the RTM simulation

Unlike the ROOPL simulation, ROOPL++ is not limited by stack allocated, statically-scoped objects. Due to this limitation, the ROOPL RTM simulator cannot finish with the TM tape as its program output when the RTM halts, as the call stack of the simulation must unwind before termination. As ROOPL++ has dynamically-scoped objects, this limitation is lifted and as such, the TM tape will exist as the program out when the RTM halts.

The main object initializes a Cell which is the initial tape. References to the neighbours of the cell which currently is right under the tape head is passed along with the transition rule variable. Initially the neighbour references are **nil**. The tape head is moved by the method shown in figure 2.27. The method takes the current cell under the tape head, its neighbours and a direction. From here, it then begins moves the tape head by updating references to the neighbours and the current cell. If a neighbour in the direction we want to move in is **nil**, a new cell is created and references to it created.

```
1     method simulate(Cell tape, Cell left, Cell right int pos, int state, int q1[],
2                     int s1[], int s2[], int q2[], int pc)
3         from state = Qs do
4             pc += 1                                          // Increment pc
5             local int symbol = 0
6             call tape::lookup(pos, symbol)                   //Fetch current symbol
7             call inst(state, symbol, q1, s1, s2, q2, pc, tape, left, right)
8             uncall tape::lookup(pos, symbol)                 //Zero-clear symbol
9             delocal symbol = 0
10            if pc = PC_MAX then                              // Reset pc
11                pc ^= PC_MAX
12            fi pc = 0
13        until state = Qf
```

Figure 2.28: Main RTM simulation method

CHAPTER 3

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, 20, 12].

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 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 [11].

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

3.1.1 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 $n - m \ mod \ n$ per allocated object of size m.

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

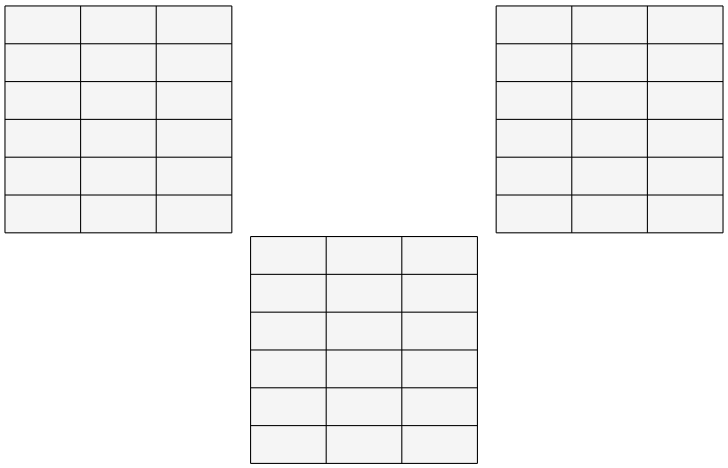

Figure 3.1: Example of internal fragmentation due to *over-allocation* (Work in progress)

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.

3.1.2 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 objects on its own.

This type of fragmentation is generally a more substantial cause of problems than internal fragmentation, as the amount of wasted memory typically is larger and less predictable in external fragmentation blocks than in internal fragmentation blocks. Depending on the heap implementation, i.e. a layout using variable-sized blocks of, say, size 2^n, the internal fragment size becomes considerable for large ns.

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 to relocate blocks in such a manner that the fragmentation disperses.

TODO

Figure 3.2: Example of external fragmentation

3.2 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)

3.3 Linearity and reference counting

Reversible programming languages such as Janus and Roopl are linear in the sense that object and variable pointers cannot be copied and are only deleted during deallocation. Pointer copying greatly increases the flexibility of programming, especially in a reversible settings where zero-clearing is critical. For variables, pointer copying is not particular interesting, nor would it add much flexibility as the values of a variable simply can be copied into statically-scoped local blocks. For objects however, tedious amounts of boilerplate work must be done if object A and B need to work on the same object C and only one reference to each object is allowed.

Mogensen presented the reversible functional language Rcfun which used reference counting to allow multiple pointers to the same memory nodes as well as a translation from Rcfun into Janus. In Rcfun, reference counting is used to manage and trace the number of pointer copies made by respectively incrementing and decrementing a *reference count* stored in the memory

node, whenever the original node pointer is copied or a copy pointer is deleted. For the presented heap manage, deletion of object nodes was only allowed when no references to a node remained.

In non-reversible languages, reference counting is also used in garbage collection by automatically deallocating unreachable objects and variables which contains no referencing.

3.4 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) [9, 16, 3]. 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, 5, 13].

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

3.4.1 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.

3.4.2 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.

3.4.3 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`

3.4.4 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.

3.4.5 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)

CHAPTER 4

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.

4.1 The ROOPL to PISA compiler

4.2 ROOPL++ Memory Layout

ROOPL++ builds upon its predecessor's memory layout with dynamic memory management. The reversible Buddy Memory heap layout presented in section 3.4.5 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 4.1 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.

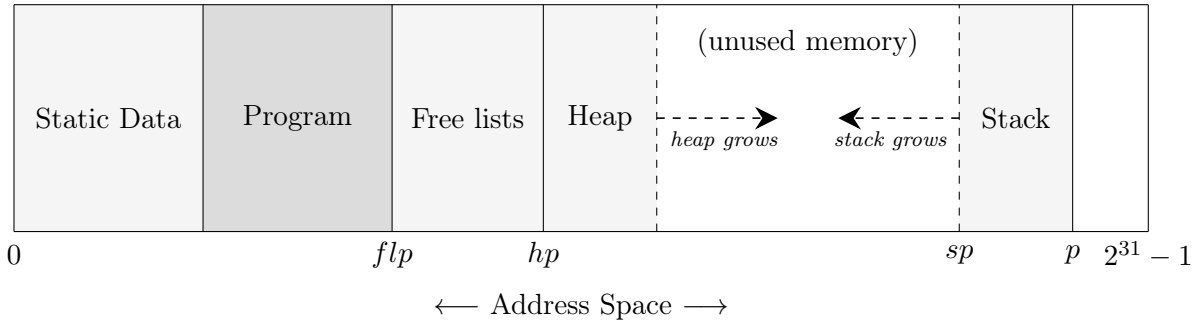

Figure 4.1: Memory layout of a RoopL++ program

4.3 Inherited RoopL features

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

The dynamic dispatching mechanism presented in [6] 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 4.2

4.4 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 4.3.

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

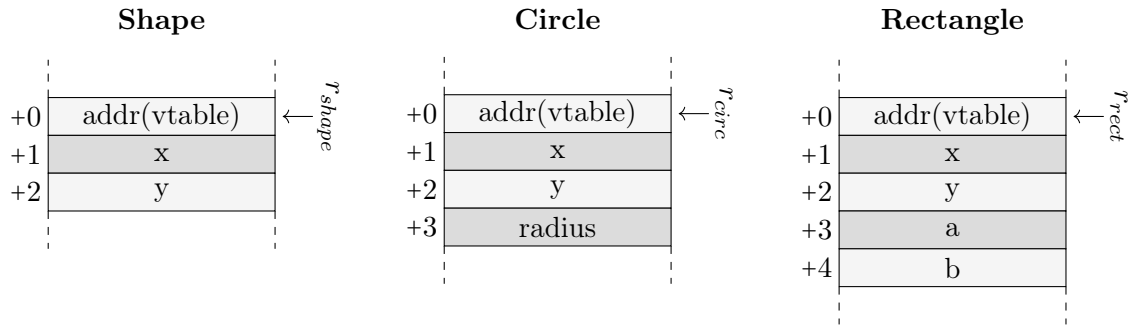

Figure 4.2: Illustration of prefixing in the memory layout of 3 Roopl++ objects

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 4.3 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 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.

4.5 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 4.5.

In the translated `malloc1` function, the **`SWAPBR`** functions as entry and exit point, as PISA's

```
(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 4.3: Overall layout of a translated Roopl++ program

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 4.4

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

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 4.4: PISA translation of heap allocation and deallocation for objects

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 4.5, 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.

4.6 Arrays

4.6.1 Construction and destruction

4.6.2 Array Element Access

4.7 Referencing

4.8 Error Handling

While a program written in ROOPL++ might be syntactically valid and well-typed, it is not a guarantee that it compiles successfully. A number of conditions exists, which cannot be determined at compile time, which results in erroneous compiled code. Haulund describes the following conditions:

- If the entry expression of a conditional is **true**, then the exit assertion should also be **true** after executing the then-branch.
- If the entry expression of a conditional is **false**, then the exit assertion should also be **false** after executing the else-branch.
- The entry expression of a loop should initially be **true**.
- If the exit assertion of a loop is **false**, then the entry expression should also be **false** after executing the loop-statement.
- All instance variables should be zero-cleared within an object block, before the object is deallocated.
- The value of a local variable should always match the value of the delocal-expression after the block statement has executed.

The extensions made to ROOPL in ROOPL++ brings forth a number of additional conditions:

- All fields of an object instance should be zero-cleared before the object is deallocated using the **delete** statement.
- All cells of an an instance should be zero-cleared before the array is deallocated using the **delete** statement.
- Local object blocks should have their fields zero-cleared after the execution of the block statement.
- Local array blocks should have their cells zero-cleared after the execution of the block statement.
- If a local object variable's value is exchanged during its block statement and the new value is an object reference, this object must have its fields zero-cleared after the execution of the block statement.

- If a local array variable's value is exchanged during its block statement and the new value is an array reference, this array must have its cell zero-cleared after the execution of the block statement.
- The variable in the **new** statement must be zero-cleared beforehand.
- The variable in the **copy** statement must be zero-cleared beforehand.
- The variable in the **copy** statement must be zero-cleared beforehand.
- The variable in the **new** statement must be zero-cleared beforehand.

It is the programmer's responsibility to meet these conditions are met. As these conditions, in general, cannot be determined at compile time, undefined program behaviour will occur as the termination will continue silently, resulting in erroneous program state.

4.9 Implementation

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

CHAPTER 5

Conclusions

5.1 Future work

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] Digiconomist. "Bitcoin Energy Consumption Index". In: *Digiconomist* (2017-12-06). URL: https://digiconomist.net/bitcoin-energy-consumption (visited on 12/12/2017).

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

[4] Frank, M. P. "The R Programming Language and Compiler". MIT Reversible Computing Project Memo #M8. 1997.

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

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

[7] Haulund, T., Mogensen, T. Æ., and Glück, R. "Implementing Reversible Object-Oriented Language Features on Reversible Machines". In: *Reversible Computation: 9th International Conference, RC 2017, Kolkata, India, July 6-7, 2017, Proceedings*. Ed. by I. Phillips and H. Rahaman. Cham: Springer International Publishing, 2017, pp. 66–73. ISBN: 978-3-319-59936-6. DOI: 10.1007/978-3-319-59936-6_5. URL: https://doi.org/10.1007/978-3-319-59936-6_5.

[8] Landauer, R. "Irreversibility and Heat Generation in the Computing Process". In: *IBM Journal of Research and Development* 5.3 (1961), pp. 183–191.

[9] 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.

[10] Lutz, C. "Janus: a time-reversible language". Letter to R. Landauer. 1986.

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

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

[13] 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.

[14] Thomsen, M. K., Axelsen, H. B., and Glück, R. "A Reversible Processor Architecture and Its Reversible Logic Design". In: *RC 11: Proceedings of the Third International Conference on Reversible Computation* (2011), pp. 30–42.

[15] Vance, A. “Inside the Arctic Circle, Where Your Facebook Data Lives”. In: *Bloomberg* (2013-12-04). URL: https://www.bloomberg.com/news/articles/2013-10-04/facebooks-new-data-center-in-sweden-puts-the-heat-on-hardware-makers (visited on 12/12/2017).

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

[17] Vieri, C. J. “Pendulum: A Reversible Computer Architecture”. Master’s Thesis. University of California at Berkeley, 1993.

[18] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993. ISBN: 0-262-23169-7.

[19] Yokoyama, T., Axelsen, H. B., and Glück, R. “Principles of a Reversible Programming Language”. In: *CF 2008: Proceedings of the 2008 Conference on Computing Frontiers* (2008), pp. 43–54.

[20] 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.

[21] 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.