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

Submitted: January 25^{th}, 2018

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 4.6
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 2.3 and 2.4. Updated type system and semantics with array specific rules in section 2.8 and 2.9. Started on section 4.9.
1.0	2017-12-29	Martin	Adressed feedback in section 2.6, 2.7 and 2.9. Finished section 2.10. Updated section 2.11 with new RTM implementation. Revised and rewrote large parts of chapter 3. Wrote section 4.1. Wrote section 4.8 and section 4.7 after finalizing compiler code. Finished section 4.9. Wrote section 4.10. Added conclusion in chapter 5. Added abstract and preface.
1.01	2018-01-16	Martin	Proofreading and minor changes. Added section 4.11

TODOs:

- Proofreading :-)

Abstract

The reversible object-oriented programming language (ROOPL) was presented in late 2016 and proved that object-oriented paradigms was feasible in the reversible setting. The language featured simple statically scoped objects which made non-trivial programs tedious, if not impossible to write using the limited tools provided. We introduce an extension to ROOPL in form the new language ROOPL++, featuring dynamic memory management and static arrays for increased language expressiveness. The language is a superset of ROOPL and has formally been defined in its language semantics, type system and computation universality. Considerations for reversible memory manager layouts designs were discussed and ultimately led to the selection of the Buddy Memory layout. Translations of the extensions presented in ROOPL++ to the reversible assembly language PISA are presented to provide garbage-free computations. The extension proved to be feasible in extending the expressiveness of ROOPL and as a result, showed that non-trivial reversible data structures, such as reversible binary trees and doubly-linked lists, are feasible and does not contradict the reversible computing paradigm.

Preface

This Master's Thesis presents a 30 ECTS workload as is submitted as the last part for the degree of Master of Science in Computer Science at the University of Copenhagen, Department of Computer Science.

The thesis consists of 94 pages and a ZIP archive containing source code developed as part of the thesis work. The thesis was submitted on the $25th$ of January, 2018 and will be subject to an oral defense no later than the 25^{th} of February, 2018.

I would like to thank my two supervisors, Robert Glück and Torben Mogensen, for their invaluable supervision and guidance throughout this project and introduction to the field of reversible computing. A big thanks to my university colleague and friend, Tue Haulund, for allowing me to continue his initial work on ROOPL and providing information, sparring and source code material and for being a great ally through our years at the University of Copenhagen. In addition, thanks to my dear aunt Doris, for financially supporting my studies by paying for all my books needed. Finally, a thanks to Jess, for all the love and support throughout the entire span of my thesis process.

Table of Contents

1 Introduction **9**
- 1.1 Reversible Computing . . . 9
- 1.2 Object-Oriented Programming . . . 10
- 1.3 Reversible Object-Oriented Programming . . . 10
- 1.4 Motivation . . . 11
- 1.5 Thesis Statement . . . 11
- 1.6 Outline . . . 11

2 The Roopl++ Language **12**
- 2.1 Syntax . . . 13
- 2.2 Object Instantiation . . . 14
- 2.3 Array Model . . . 16
- 2.4 Array Instantiation . . . 16
- 2.5 Referencing . . . 18
- 2.6 Local Blocks . . . 19
- 2.7 Roopl++ Expressiveness . . . 20
  - 2.7.1 Linked List . . . 20
  - 2.7.2 Binary Tree . . . 21
  - 2.7.3 Doubly Linked List . . . 23
- 2.8 Type System . . . 26
  - 2.8.1 Preliminaries . . . 26
  - 2.8.2 Expressions . . . 28
  - 2.8.3 Statements . . . 28
  - 2.8.4 Programs . . . 31
- 2.9 Language Semantics . . . 32
  - 2.9.1 Preliminaries . . . 32
  - 2.9.2 Expressions . . . 32
  - 2.9.3 Statements . . . 33
  - 2.9.4 Programs . . . 37
- 2.10 Program Inversion . . . 38
  - 2.10.1 Invertability of Statements . . . 39
  - 2.10.2 Type-Safe Statement Inversion . . . 39
- 2.11 Computational Strength . . . 42
  - 2.11.1 Reversible Turing Machines . . . 42
  - 2.11.2 Tape Representation . . . 43
  - 2.11.3 Reversible Turing Machine Simulation . . . 44

3 Dynamic Memory Management **46**
- 3.1 Fragmentation . . . 46
  - 3.1.1 Internal Fragmentation . . . 47

3.1.2 External Fragmentation . . . 47
3.2 Memory Garbage . . . 48
3.3 Linearity and Reference Counting . . . 50
3.4 Heap Manager Layouts . . . 50
3.4.1 Memory Pools . . . 51
3.4.2 One Heap Per Record Size . . . 52
3.4.3 One Heap Per Power-Of-Two . . . 53
3.4.4 Shared Heap, Record Size-Specific Free Lists . . . 55
3.4.5 Buddy Memory . . . 55

4 Compilation **58**
4.1 The ROOPL to PISA Compiler . . . 58
4.2 ROOPL++ Memory Layout . . . 59
4.3 Inherited ROOPL features . . . 60
4.4 Program Structure . . . 60
4.5 Buddy Memory Translation . . . 62
4.6 Object Allocation and Deallocation . . . 65
4.7 Referencing . . . 66
4.8 Arrays . . . 67
4.8.1 Construction and Destruction . . . 68
4.8.2 Array Element Access . . . 69
4.9 Error Handling . . . 69
4.10 Implementation . . . 70
4.11 Evaluation . . . 71

5 Conclusions **72**
5.1 Future Work . . . 72

References **74**

Appendix A Pisa Translated Buddy Memory **76**

Appendix B RoooplppC Source Code **78**

Appendix C Example Ouput **87**

List of Figures

2.1 Example ROOPL++ program implementing the Ackermann function 12
2.2 Syntax domains and EBNF grammar for ROOPL++ 13
2.3 Linked List cell class 20
2.4 Linked List class 21
2.5 Binary Tree class 22
2.6 Binary Tree node class (cont) 22
2.7 Binary Tree node class 23
2.8 Doubly Linked List class 24
2.9 Multiple identical reference are needed for a doubly linked list implementation . 24
2.10 Doubly Linked List Cell class 25
2.11 Doubly Linked List Cell class (cont) 25
2.12 Definition *gen* for constructing the finite class map Γ of a given program p, originally from [10] 27
2.13 Definition of fields and methods, originally from [10] 27
2.14 Definition *arrayType* for mapping types of arrays to either class types or the integer type 27
2.15 Typing rules for expressions in ROOPL, originally from [10] 28
2.16 Typing rule extension for the ROOPL typing rules 28
2.17 Typing rules for statements in ROOPL, originally from [10] 29
2.18 Typing rules extensions for statements in ROOPL++ 30
2.19 Typing rules for class methods, classes and programs, originally from [10] 31
2.20 Semantic values, originally from [10] 32
2.21 Semantic inference rules for expressions, originally from [10] 32
2.22 Extension to the semantic inference rules for expression in ROOPL++ 33
2.23 Definition of binary expression operator evaluation, originally from [10] 33
2.24a Semantic inference rules for statements, originally from [10] 34
2.24b Semantic inference rules for statements, originally from [10] (cont) 35
2.24c Semantic inference rules for statements, originally from [10] (cont) 35
2.24d Extension to the semantic inference rules for statements in ROOPL++ 36
2.24e Extension to the semantic inference rules for statements in ROOPL++ (cont) . . 36
2.25 Semantic inference rules for programs, originally from [10] 38
2.26 ROOPL++ statement inverter, extended from [10] 38
2.27 Modified statement inverter for statements, originally from [10] 39
2.28 Method for moving the tape head in the RTM simulation 44
2.29 Method for executing a single TM transition 45
2.30 Main RTM simulation method 45

3.1a Creation of internal fragmentation of size $n - m$ due to *over-allocation* 47
3.1b Creation of internal fragmentation of size $n - m \ mod \ n$ due to *over-allocation* . 47
3.2 Example of external fragmentation caused for allocation and deallocation order . 48

3.3 Example of avoiding external fragmentation using allocation and deallocation order 48
3.4 The "garbage" output of an injective function f is the input to its inverse function f' . . . 49
3.5 All free lists are considered equivalent "garbage" in terms of injective functions . 49
3.6 Memory layout using one heap per record size . . . 52
3.7 Memory layout using one heap per power-of-two . . . 54
3.8 Record size-specific free lists on a shared heap . . . 56
3.9 Buddy Memory block allocation example . . . 56

4.1 Memory layout of a ROOPL program, originally from [10] . . . 58
4.2 Illustration of object memory layout . . . 59
4.3 Memory layout of a ROOPL++ program . . . 59
4.4 Overall layout of a translated ROOPL++ program . . . 61
4.5 Dynamic dispatch approach for entering the allocation subroutine . . . 62
4.6 PISA translation of the nested conditionals in the Buddy Memory algorithm . . 63
4.7 PISA translation of the outer **if-then** statement for the Buddy Memory algorithm 63
4.8 PISA translation of the inner **if-then** statement for the Buddy Memory algorithm 64
4.9 PISA translation of the inner **else** statement for the Buddy Memory algorithm . 64
4.10 Non-oppsite deallocation results in a different free list after termination . . . 65
4.11 PISA translation of the **malloc** procedure entry point of Buddy Memory algorithm 65
4.12 PISA translation of heap allocation and deallocation for objects . . . 66
4.13 Illustration of object memory layout . . . 67
4.14 PISA translation of the reference copying and deletion statements . . . 67
4.15 Illustration of prefixing in the memory layout of two ROOPL++ arrays . . . 68
4.16 PISA translations of array allocation and deallocation statements . . . 68
4.17 Lines of code comparison between target and compiled ROOPL++ programs . . . 71

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. Most of 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 [5]. 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 [22].

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 [12].

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) [24] and the BobISA architecture and instruction set [21] and high level languages Janus [14, 28, 26] and R [6] exists.

While cryptocurrency mining and many other computations are not reversible, the area remains interesting in terms of its applications and gains.

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 of *one-to-one* functions, also known as *injective* functions, which results in a garbage-free computations, as garbage-generating functions simply can be unwinded to clean up.

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 by running the algorithm in backwards direction.

The reversibility is however not free at comes and 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, when 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. 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 [15].

1.3 Reversible Object-Oriented Programming

The high-level reversible language Roopl (Reversible Object-Oriented Programming Language) was introduced in late 2016 [10, 11]. 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.

Conceptualizations and ideas for the Joule language was also published in 2016 [20]. The language, a homonym of Janus Object-Oriented Language, Jool, presented an alternative OOP extension to Janus, differing from Roopl. The language featured heap allocated objects with constructors and multiple object references, as such also addressing the problems with

ROOPL. The language is still a work in progress, aiming to provide a useful, reversible object oriented-programming language.

1.4 Motivation

ROOPL's block defined objects and lack of references are problematic when write complex, reversible programs using OOP methodologies as they pose severe limitations on the expressiveness. 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, and ultimately increase the usability of reversible OOP. Work within the field of reversible computing related to heap manipulation [2], reference counting [17] and garbage collection [18] 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 feasible and effective. The resulting expressiveness allows non-trivial reversible programming previously unseen, such as reversible data structures, including linked lists, doubly linked lists and trees.

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 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 PISA translations of the reversible heap allocation algorithm, 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 ROOPL++ Language

With the design and implementation of the REVERSIBLE OBJECT-ORIENTED PROGRAMMING LANGUAGE (ROOPL) and the work-in-progress report of JOULE, 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.

```
1  class Program
2      int result
3      int n
4      int m
5
6      method main()
7          n ^= 3
8          m ^= 4
9          new Ackermann ack
10         call ack::find(m, n)
11         call ack::get(result)
12         delete Ackermann ack
```

```
1  class Ackermann
2      int tmp
3      int result
4
5      method find(int m, int n)
6          if m = 0 then
7              result ^= n
8              result += 1
9          else
10             if n = 0 then
11                 m -= 1
12                 tmp ^= 1
13                 call find(m, tmp)
14                 tmp ^= 1
15             else
16                 n -= 1
17                 call find(m, n)
18                 m -= 1
19                 call find(m, result)
20             fi n = 0
21         fi m = 0
22
23     method get(int out)
24         out ^= result
```

Figure 2.1: Example ROOPL++ program implementing the Ackermann function

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 [10] has become a standard part of the language. Class definitions remains unchanged, and consist 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 Array type. A class definition is required to include at least one method, defined by the **method** keyword followed by a method name, a comma-separated list of parameters and a body.

Reversible assignments for integer variables and integer array 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 array 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 statically-scoped 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 **new** statement.

Arrays are also instantiated by usage of **new** and **delete**. Assignment of array cells depends on the type of the arrays, which is further discussed in section 2.4.

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 largely unchanged in ROOPL++.

2.2 Object Instantiation

Object instantiation through the **new** statement, follows the pattern of the mechanics known from the **construct**/**destruct** blocks from ROOPL, but providing improved scoping and lifetime options 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 *dynamic* 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 address of the newly allocated block is stored in the previously defined and zero-cleared reference x.
3. The block statement s is executed.

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

1. The reference x may be modified by swapping its internal values with that of other references of the same type, but should be restored to their 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 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 [28].

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. Arrays were decided to be part of the core language for this reason, 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 arrays. 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. When swapping array elements from a base class array with object references the programmer must be careful not to swap the values of, say, and A object into a B reference.

2.4 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.
3. The address of the newly allocated memory is stored in the previously defined and zero-cleared reference x.

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 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    ...                          // Clear all array cells
12
13    delete Foo fooArray[0]       // Legal object array cell zero-clearing
14    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 of same type 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

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

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

2.5 Referencing

Besides the addition of dynamically lifetimed objects and arrays, ROOPL++ also increases program flexibility by allowing multiple references to objects and arrays through the usage of the **copy** statement. Once instantiated through either a **new** or **construct**/**destruct** block, an object or array reference can be copied into another zero-cleared variable. The reference acts as a regular 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 copies a reference of variable x, an instance of class or array 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 or array c.

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 or cells to be zero-cleared (as they simply are pointers to existing objects or arrays), the reversible programmer should carefully ensure that all references are un-copied before deleting said object or array, as copied references to cleared objects or arrays 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 3.3 and 4.9.

2.6 Local Blocks

The local block presented in the extended JANUS in [26] 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 the local variable initialized will be set to the evaluated result of a given expression.

Local array and object blocks feature 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 accessible. If the block is initiated with an existing object or array reference, the local variable essencially becomes 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 blocks allow 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 at its end of the block definition. The value of the variable is a pointer to an object or an array. Said object or array must have all fields/cells zero-cleared before the 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 previously unseen reversible data structures, which now are feasible, written in ROOPL++.

2.7.1 Linked List

Haulund presented a linked list implemented in ROOPL in [10]. 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 implements a *LinkedList* class, which simply has the head of the list and the list 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

```
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

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

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 as a consequence of reversibility, 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. Conventional flattening is not reversible, but perhaps a simplified reversible version could be defined by applying some limitiations or restrictions. 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 2.8-2.11. 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 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 and to its left and right neighbours. Figure 2.9 shows the multiple

```
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

reference needed for the doubly linked list implementation denoted by the three different arrow types.

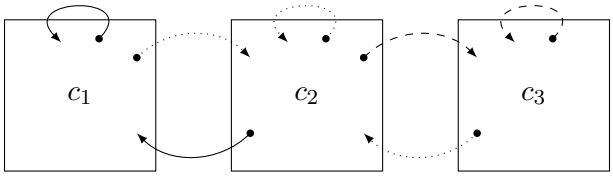


Figure 2.9: Multiple identical reference are needed for a doubly linked list implementation

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

The data structure could relatively easily be extended to work as a dynamic array. Currently each cell contains an *index* field, specifying their position 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. In reverse, this would remove a cell from the list. 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 are given an index that is out of 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 are 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

```
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 2.10: Doubly Linked List Cell class

```
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 2.11: Doubly Linked List Cell class (cont)

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 [10] and is analogously described by syntax-directed inference typing rules in the style of Winskel [25]. 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 2.12. Figure 2.13 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 2.12 and 2.13 are graciously borrowed from [10].

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

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

Figure 2.12: Definition *gen* for constructing the finite class map Γ of a given program p, originally from [10]

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

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

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

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

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

Figure 2.13: Definition of fields and methods, originally from [10]

Finally, we formally define a link between arrays of a given type and other types. The function *arrayType*, defined in figure 2.14, is c if the passed array a is an array of class c instances.

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

Figure 2.14: 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 2.15: Typing rules for expressions in Roopl, originally from [10]

The original expression typing rules from Roopl are shown in figure 2.15. 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 2.16.

$$\frac{\text{arrayType}(x) = \tau \qquad \Pi_{expr} \vdash \ e \ : \ \textbf{int} \qquad \Pi(x[e]) = \tau}{\Pi \vdash_{expr} \ x[e] : \tau} \text{ T-ArrElemVar}$$

Figure 2.16: 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 2.17: Typing rules for statements in Roopl, originally from [10]

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 2.18: Typing rules extensions for statements in Roopl++

Figure 2.18 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 lifetimed 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 2.19 shows the typing rules for class methods, classes and programs.

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

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

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

Figure 2.19: Typing rules for class methods, classes and programs, originally from [10]

2.9 Language Semantics

The following sections contain 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 2.20: Semantic values, originally from [10]

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 2.21: Semantic inference rules for expressions, originally from [10]

As shown in figure 2.21, 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 2.23.

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

Figure 2.22: Extension to the semantic inference rules for expression in Roopl++

For Roopl++, we extend the expression ruleset with a single rule for array element variables shown in figure 2.22. As with the expressions inference rules in Roopl, this extension has no effect on the store.

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

Figure 2.23: Definition of binary expression operator evaluation, originally from [10]

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 2.24a, 2.24b and 2.24c 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.

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

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

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

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

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

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

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

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

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

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

Figure 2.24a: Semantic inference rules for statements, originally from [10]

how value mappings between two variables are exchanged in the updated store.

For loops and conditionals, Rules LoopMain, LoopBase and LoopRec define the meaning of loop statements and IfTrue and IfFalse, similarly to the operational semantics of Janus, as presented in [26]. 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 and the else-branch is executed.

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

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

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

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

Figure 2.24b: Semantic inference rules for statements, originally from [10] (cont)

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

Figure 2.24c: Semantic inference rules for statements, originally from [10] (cont)

As presented in the operational semantics for ROOPL, rules CALL, UNCALL, CALLOBJ and UNCALLOBJ respectively define 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 [28, 26].

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 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

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

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

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

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

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c \ x \ : \ \mu' \rightleftharpoons \mu}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ c \ x \ : \ \mu \rightleftharpoons \mu'} \ \textsc{ObjDelete}$$

$$\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e \Rightarrow v \qquad \gamma' = [0 \mapsto a_1, \ ..., \ v \mapsto a_n] \qquad \{l', r, v', a_1, ..., a_n\} \cap \mathrm{dom}(\mu) = [r \mapsto 0]$$

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

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ a[e] \ x \ : \ \mu' \rightleftharpoons \mu}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ a[e] \ x \ : \ \mu \rightleftharpoons \mu'} \ \textsc{ArrDelete}$$

Figure 2.24d: Extension to the semantic inference rules for statements in Roopl++

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

$$\frac{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{copy} \ c \ x \ x' \ : \ \mu' \leftrightharpoons \mu}{\langle l, \gamma \rangle \vdash^{\Gamma}_{stmt} \ \textbf{uncopy} \ c \ x \ x' \ : \ \mu \rightleftharpoons \mu'} \ \textsc{Uncopy}$$

$$\langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_1 \Rightarrow v_1 \qquad \langle \gamma, \mu \rangle \vdash^{\Gamma}_{stmt} \ e_2 \Rightarrow v_2 \qquad \{l', r\} \cap \mathrm{dom}(\mu) = \emptyset$$

$$\mu' = \mu[l' \mapsto v_1, r \mapsto l']$$

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

Figure 2.24e: Extension to the semantic inference rules for statements in Roopl++ (cont)

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 2.24d and 2.24e 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 lifetimed 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 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 reference 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 reference 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 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.

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,

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

Figure 2.25: Semantic inference rules for programs, originally from [10]

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

In order to truly show that ROOPL++ in fact is a reversible language, we must demonstrate and prove local inversion of statements is possible, such that any program written in ROOPL++, regardless of context, can be executed in reverse. Haulund presented a statement inverter for ROOPL in [10], which maps statements to their inverse counterparts. Figure 2.26 shows the statement inverter, extended with the new ROOPL++ statements for construction/destruction and referencing copying/copy removal.

$$\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}[\![x[e_1]\ \textbf{+=}\ e_2]\!] = x[e_1]\ \textbf{-=}\ e_2 && \mathcal{I}[\![x[e_1]\ \textbf{-=}\ e_2]\!] = x[e_1]\ \textbf{+=}\ e_2\\
&\mathcal{I}[\![x[e_1]\ \textbf{\^{}=}\ e_2]\!] = x[e_1]\ \textbf{\^{}=}\ e_2 && \mathcal{I}[\![x[e_1]\ \textbf{<=>}\ e_2]\!] = x[e_1]\ \textbf{<=>}\ e_2\\
&\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 2.26: ROOPL++ statement inverter, extended from [10]

Program inversion is conducted by recursive descent over components and statements. A proposed extension to the statement inverted for whole-program inversion, is retained in the ROOPL++ statement inverter. The extension covers the case, which reveals itself during method calling. As a method call is equivalent to an uncall with the inverse method and we simply change calls to

uncalls during inversion, the inversion of the method body cancels out. The proposed extension, presented in [10], simply avoids inversion of calls and uncalls, as shown in figure 2.27.

$$\mathcal{I}'[\![\textbf{call } q(\ldots)]\!] = \textbf{call } q(\ldots) \qquad \mathcal{I}'[\![\textbf{call } x :: q(\ldots)]\!] = \textbf{call } x :: q(\ldots)$$
$$\mathcal{I}'[\![\textbf{uncall } q(\ldots)]\!] = \textbf{uncall } q(\ldots) \qquad \mathcal{I}'[\![\textbf{uncall } x :: q(\ldots)]\!] = \textbf{uncall } x :: q(\ldots)$$
$$\mathcal{I}'[\![s]\!] = \mathcal{I}[\![s]\!]$$

Figure 2.27: Modified statement inverter for statements, originally from [10]

2.10.1 Invertability of Statements

While the invertibility of statements remains untouched by the extensions made in ROOPL++, the following proof, originally presented in [10], has been included for completeness.

If execution of a statement s in store μ yields μ', then execution of the inverse statement, $\mathcal{I}[\![s]\!]$ in store μ' should yield μ. Theorem 2.1 shows that $\mathcal{I}$ is a statement inverter.

Theorem 2.1. *(Invertibility of statements, originally from [10])*

$$\overbrace{\langle l, \gamma\rangle \ \vdash^{\Gamma}_{stmt} \ s \ : \ \mu \rightleftharpoons \mu'}^{\mathcal{S}} \iff \overbrace{\langle l, \gamma\rangle \ \vdash^{\Gamma}_{stmt} \ \mathcal{I}[\![s]\!] \ : \ \mu' \rightleftharpoons \mu}^{\mathcal{S}'}$$

Proof. By structural induction on the semantic derivation of $\mathcal{S}$ (omitted). It suffices to show that $\mathcal{S} \implies \mathcal{S}'$, as this can serve as proof of $\mathcal{S}' \implies \mathcal{S}$, as $\mathcal{I}$ is an involution.

2.10.2 Type-Safe Statement Inversion

Given a well-typed statement, the statement inverter $\mathcal{I}$ should always produce a well-typed, inverse statement in order to correctly support backwards determinism of injective functions. Theorem 2.2 describes this.

Theorem 2.2. *(Inversion of well-typed statements, originally from [10])*

$$\overbrace{\langle \Pi, \ c\rangle \ \vdash^{\Gamma}_{stmt} \ s}^{\mathcal{T}} \implies \overbrace{\langle \Pi, \ c\rangle \ \vdash^{\Gamma}_{stmt} \ \mathcal{I}[\![s]\!]}^{\mathcal{T}'}$$

Proof. By structural induction on $\mathcal{T}$. Unmodified ROOPL statements retained in ROOPL++ has been omitted.

- Case $\mathcal{T} =$

$$\frac{\overbrace{x \in \text{IntegerArrayIDs}}^{\mathcal{C}_1} \quad \overbrace{\Pi \vdash_{expr} e_1 \ : \ \textbf{int}}^{\mathcal{E}_1} \quad \overbrace{x[e_1] \notin \text{vars}(e_2)}^{\mathcal{C}_2} \quad \overbrace{\Pi \vdash_{expr} \ e_2 : \textbf{int}}^{\mathcal{E}_2}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ x[e_1] \odot = e_2} \text{ T-ArrElemAss}$$

In this case, we have $\mathcal{I}[\![x \odot = \ e]\!] = x \odot' = \ e$, for some $\odot'$. Therefore, $\mathcal{T}'$ will also be a derivation of rule T-ArrElemAss, and as such, we can simply reuse the conditions $\mathcal{C}_1, \mathcal{C}_2$ and the expressions $\mathcal{E}_1.\mathcal{E}_2$ in construction of $\mathcal{T}'$

$$\mathcal{T}' = \frac{\overbrace{x \in \text{IntegerArrayIDs}}^{\mathcal{C}_1} \quad \overbrace{\Pi \vdash_{expr} e_1 \ : \ \textbf{int}}^{\mathcal{E}_1} \quad \overbrace{x[e_1] \notin \text{vars}(e_2)}^{\mathcal{C}_2} \quad \overbrace{\Pi \vdash_{expr} \ e_2 : \textbf{int}}^{\mathcal{E}_2}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ x[e_1] \odot' = e_2}$$

- Case $\mathcal{T} =$ $\dfrac{\overbrace{\Pi(x) = \textbf{nil}}^{\mathcal{C}_1}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c' \ x}$ T-ObjNew

In this case we have $\mathcal{I}[\![\textbf{new} \ c \ x]\!] \ = \textbf{delete} \ c \ x$, meaning $\mathcal{T}'$ must be of the form:

$$\mathcal{T}' = \frac{\overbrace{\Pi(x) = c'}^{\mathcal{C}_2}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ c' \ x}$$

- Case $\mathcal{T} =$ $\dfrac{\overbrace{\Pi(x) = c'}^{\mathcal{C}_1}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{delete} \ c' \ x}$ T-ObjDlt

Inverse of the previous case, we now have $\mathcal{I}[\![\textbf{delete} \ c \ x]\!] \ = \textbf{new} \ c \ x$, meaning $\mathcal{T}'$ must be of the form:

$$\mathcal{T}' = \frac{\overbrace{\Pi(x) = \textbf{nil}}^{\mathcal{C}_2}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ c' \ x}$$

- Case $\mathcal{T} =$ $\dfrac{\overbrace{\text{arrayType}(a) \in \left\{\text{classIDs}, \textbf{int}\right\}}^{\mathcal{C}_1} \quad \overbrace{\Pi \vdash_{expr} \ e = \textbf{int}}^{\mathcal{E}} \quad \overbrace{\Pi(x) = \textbf{nil}}^{\mathcal{C}_2}}{\langle \Pi, c \rangle \vdash^{\Gamma}_{stmt} \ \textbf{new} \ a[e] \ x}$ T-ArrNew

In this case we still have $\mathcal{I}[\![\textbf{new} \ c \ x]\!] \ = \textbf{delete} \ c \ x$. Using $\mathcal{C}_1$ and $\mathcal{E}$, $\mathcal{T}'$ must be of the

form:

$$\mathcal{T}' = \frac{\overbrace{\text{arrayType}(a) \in \left\{\text{classIDs}, \textbf{int}\right\}}^{\mathcal{C}_1} \quad \overbrace{\Pi \vdash_{expr} e = \textbf{int}}^{\mathcal{E}} \quad \overbrace{\Pi(x) = a}^{\mathcal{C}_3}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{delete}\ a[e]\ x}$$

- Case $\mathcal{T} = \dfrac{\overbrace{\text{arrayType}(a) \in \left\{\text{classIDs}, \textbf{int}\right\}}^{\mathcal{C}_1} \quad \overbrace{\Pi \vdash_{expr} e = \textbf{int}}^{\mathcal{E}} \quad \overbrace{\Pi(x) = a}^{\mathcal{C}_2}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{delete}\ a[e]\ x}$ T-ArrDlt

Similar to the object deletion case, we still have $\mathcal{I}[\![\textbf{delete}\ c\ x]\!] = \textbf{new}\ c\ x$. Using $\mathcal{C}_1$ and $\mathcal{E}$, $\mathcal{T}'$ must be of the form:

$$\mathcal{T}' = \frac{\overbrace{\text{arrayType}(a) \in \left\{\text{classIDs}, \textbf{int}\right\}}^{\mathcal{C}_1} \quad \overbrace{\Pi \vdash_{expr} e = \textbf{int}}^{\mathcal{E}} \quad \overbrace{\Pi(x) = \textbf{nil}}^{\mathcal{C}_3}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{new}\ a[e]\ x}$$

- Case $\mathcal{T} = \dfrac{\overbrace{\Pi(x) = c'}^{\mathcal{C}_1} \quad \overbrace{\Pi(x') = \textbf{nil}}^{\mathcal{C}_2}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{copy}\ c'\ x\ x'}$ T-Cp

We have $\mathcal{I}[\![\textbf{copy}\ c\ x\ x']\!] = \textbf{uncopy}\ c\ x\ x'$. Using $\mathcal{C}_1$, $\mathcal{T}'$ must as such be of the form

$$\mathcal{T}' = \frac{\overbrace{\Pi(x) = c'}^{\mathcal{C}_1} \quad \overbrace{\Pi(x') = c'}^{\mathcal{C}_3}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{uncopy}\ c'\ x\ x'}$$

- Case $\mathcal{T} = \dfrac{\overbrace{\Pi(x) = c'}^{\mathcal{C}_1} \quad \overbrace{\Pi(x') = c'}^{\mathcal{C}_2}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{uncopy}\ c'\ x\ x'}$ T-UCp

We have $\mathcal{I}[\![\textbf{uncopy}\ c\ x\ x']\!] = \textbf{copy}\ c\ x\ x'$. Using $\mathcal{C}_1$, $\mathcal{T}'$ must as such be of the form

$$\mathcal{T}' = \frac{\overbrace{\Pi(x) = c'}^{\mathcal{C}_1} \quad \overbrace{\Pi(x') = \textbf{nil}}^{\mathcal{C}_3}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{copy}\ c'\ x\ x'}$$

- Case $\mathcal{T} = \dfrac{\overbrace{\langle \Pi, c\rangle \vdash^{\Gamma}_{expr} e_1}^{\mathcal{E}_1} \quad \overbrace{\langle \Pi[x \mapsto c'], c\rangle \vdash^{\Gamma}_{stmt} s}^{\mathcal{S}} \quad \overbrace{\langle \Pi, c\rangle \vdash^{\Gamma}_{expr} e_2}^{\mathcal{E}_2}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{local}\ c'\ x = e_1 \quad s \quad \textbf{delocal}\ c'\ x = e_2}$ T-LocalBlock

We have $\mathcal{I}[\![\textbf{local}\ t\ x\ = e \quad s \quad \textbf{delocal}\ t\ x\ = e]\!] = \textbf{local}\ t\ x = e \quad \mathcal{I}[\![s]\!] \quad \textbf{delocal}\ t\ x\ = e$.

By the induction hypothesis on $\mathcal{S}$, we obtain $\mathcal{S}'$ of $\langle \Pi[x \mapsto c'], c\rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s]\!]$. Using $\mathcal{E}_1$, $\mathcal{S}'$ and $\mathcal{E}_2$ we construct $\mathcal{T}'$

$$\mathcal{T}' = \frac{\overbrace{\langle \Pi, c\rangle \vdash^{\Gamma}_{expr} e_1}^{\mathcal{E}_1} \quad \overbrace{\langle \Pi[x \mapsto c'], c\rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s]\!]}^{\mathcal{S}'} \quad \overbrace{\langle \Pi, c\rangle \vdash^{\Gamma}_{expr} e_2}^{\mathcal{E}_2}}{\langle \Pi, c\rangle \vdash^{\Gamma}_{stmt} \textbf{local}\ c'\ x = e_1 \quad \mathcal{I}[\![s]\!] \quad \textbf{delocal}\ c'\ x = e_2}$$

Using these added cases to the original proof provided in [10], Theorem 2.2 shows that well-typedness is preserved over inversion of ROOPL++ methods. As methods are well-typed if their body statement is well-typed, inversion of classes and programs also preserve well-typedness, as classes consists of methods and programs of classes, by using the class inverter presented in figure 2.27.

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 [26]. 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 defined in [26].

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

$$q_1 = q_1' \implies (t_1 \neq / \ \wedge \ t_1' \neq / \ \wedge \ t_1 \neq t_1') \text{ (forward determinism)}$$

$$q_2 = q_2' \implies (t_1 \neq / \ \wedge \ t_1' \neq / \ \wedge \ t_2 \neq t_2') \text{ (backward determinism)}$$

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`** was presented in [10]. 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, have 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 shown in [26].

2.11.2 Tape Representation

As with regular Turing machines, the Reversible Turing machines also have tapes 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 similar to the one presented in section 2.7.3, of simple cell objects containing a *value*, *left*, *right* and *self* 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. When we move in either direction, we simply set the neighbour element as the new tape head and allocate a new neighbour for the new tape head cell, if we are at the end of the list, to simulate the infinitely-length tape. Reversibly, this means that when we move in the opposite direction, cells are deallocated if we are moving the tape head away from the cell currently neighbouring either end of the tape.

```
method moveRight(int symbol, Cell tapeHead)
    local Cell right = nil
    local Cell tmp = nil
    uncall tapeHead::getSymbol(symbol)   // Put symbol back in current cell
    call tapeHead::getRight(right)       // Get right neighbour

    if right = nil && symbol = BLANK then
        symbol ^= BLANK                  // Zero clear symbol
        new Cell right                   // Init new neighbour
        copy Cell right tmp              // Copy reference to self
        uncall right::getSelf(tmp)       // Store self reference
        uncall right::getLeft(tapeHead)  // Set tape head as left of new cell
        right <=> tapeHead
    else
        call right::getLeft(tmp)         // Get copy of tape head reference
        uncopy Cell tmp tapeHead         // Clear reference to tape head

        if tapeHead = nil && symbol = BLANK
            call tmp::getSelf(tapeHead) // rev: set self pointer
            uncopy Cell tmp tapeHead    // rev: new self pointer
            delete Cell tmp             // rev: new left neighbour
            symbol ^= BLANK
        else skip                       // In reverse:
        fi tmp = nil                    // Allocate new left if current is nil

        uncall right::getLeft(tmp)       // Put tape head reference back
        tapeHead <=> right
        call tapeHead::getRight(right)   // Get right of new tape head
        call tapeHead::getSymbol(symbol) // Get symbol of new tape head
    fi right = nil
    uncall tapeHead::getRight(right)    // Set right neighbour
    delocal Cell right = nil
    delocal Cell tmp = nil
```

Figure 2.28: Method for moving the tape head in the RTM simulation

Figure 2.28 shows the *moveRight* method for moving the tape head right. If the current tape head has no instantiated right neighbour we construct one using the **new** statement. Uncalling this method will move the tape head left. If the tape head is empty after moving left, we simply allocate a new cell, thus allowing tape growth in both directions.

2.11.3 Reversible Turing Machine Simulation

Figure 2.29 shows the modified method *inst* from [26], which executes a single instruction given the tape head, the current state, symbol, program counter and the four arrays representing the transition rules. As described above, we **call** *moveRight* to move the tape head right and **uncall** to move the tape head left.

Figure 2.30 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 is incremented.

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

```
method inst(int state, int symbol, int[] q1, int[] s1,
            int[] s2, int[] q2, int pc, Cell tapeHead)
    if state = q1[pc] && symbol = s1[pc] then   // Symbol rule:
        state += q2[pc]-q1[pc]                  // set state to q2[pc]
        symbol += s2[pc]-s1[pc]                 // set symbol to s2[pc]
    fi state = q2[pc] && symbol = s2[pc]
    if state = q1[pc] && s1[pc] = SLASH then    // Move rule:
        state += q2[pc]-q1[pc]                  // set state to q2[pc]
        if s2[pc] = RIGHT then
            call moveRight(symbol, tapeHead)    // Move tape head right
        fi s2[pc] = RIGHT
        if s2[pc] = LEFT then
            uncall moveRight(symbol, tapeHead)  // Move tape head left
        fi s2[pc] = LEFT
    fi state = q2[pc] && s1[pc] = SLASH
```

Figure 2.29: Method for executing a single TM transition

```
method simulate(Cell tapeHead, int state, int[] q1, int[] s1, int[] s2, int[] q2, int pc)
    from state = Qs do
        pc += 1                                 // Increment pc local int symbol = 0
        call tapeHead::getSymbol(symbol)        // Fetch current symbol
        call inst(state, symbol, q1, s1, s2, q2, pc, tapeHead)
        uncall tapeHead::getSymbol(symbol)      // Zero-clear symbol delocal symbol = 0
        if pc = PC_MAX then                     // Reset pc
            pc ^= PC_MAX
        else skip
        fi pc = 0
    loop skip
    until state = Qf
```

Figure 2.30: Main RTM simulation method

termination. As objects in ROOPL++ is not bound by this limitation, the TM tape will exist as the program output when the RTM halts.

Instantiating a RTM simulation consists of initializing an initial tape head cell, as well as the transition rule arrays. After initialization, the *simulate* method is simply called and the simulation begins.

CHAPTER 3

Dynamic Memory Management

In order to allow objects to live outside of static scopes, we need to utilize a different memory management technique, such that objects are not allocated on the stack. Dynamic memory management presents a method of storing objects in different memory structures, most commonly, a memory heap. Most irreversibly, modern programming languages uses dynamic memory management in some form for allocating space for objects in memory.

However, reversible, native support of complex data structures is a non-trivial matter to implement. 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 [2, 27, 17].

This chapter presents a brief introduction to fragmentation, garbage and linearity and how these respectively are handled reversibly, and a discussion of various heap manager layouts considered for Roopl++, along with their advantages and disadvantages in terms of implementation difficulty, garbage build-up and the OOP domain.

3.1 Fragmentation

Efficient memory usage is an important matter to consider when designing a heap layout for a dynamic memory manager. 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. A potential side effect of this freedom, comes as a consequence of memory fragmentation. We distinguish different types of fragmentation as internal or external fragmentation.

Internal fragmentation refers to unused space inside a memory block used to store an object, if, say, the object is smaller than the block it has been allocated to. External fragmentation occurs as blocks freed throughout execution are spread across the memory heap, resulting in *fragmented* free space [16].

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 cases of *over-allocation*, which occurs when the memory manager delegates memory larger than required to fit an object, due to e.g. fixed-block sizing.

For an example, consider a scenario, in which we allocate memory for an object of size m onto a simple, fixed-sized block heap. 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.

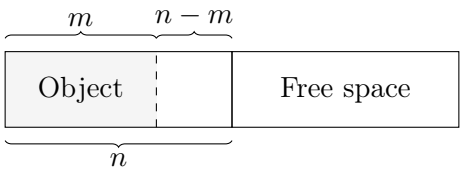


Figure 3.1a: Creation of internal fragmentation of size $n - m$ due to *over-allocation*

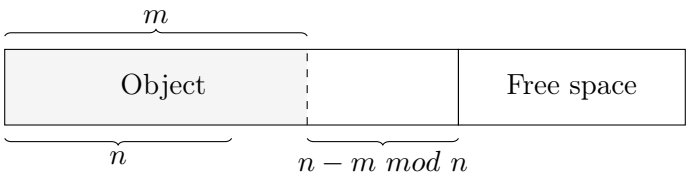


Figure 3.1b: Creation of internal fragmentation of size $n - m \ mod \ n$ due to *over-allocation*

Figure 3.1a and 3.1b visualize the examples of internal fragmentation build-up from *over-allocating* memory.

It is difficult for the memory manager to reclaim wasted memory caused by internal fragmentation, as it usually originates from a design choice. Intuitively, internal fragmentation can best be prevented by ensuring that the size of block(s) being used for allocating space for an object of size m either match or sums to this exact size, when designing the layout.

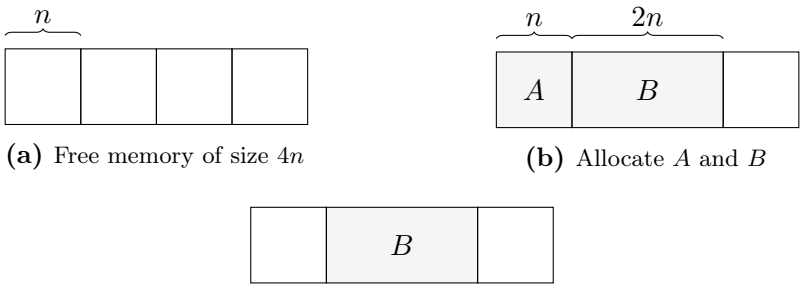
3.1.2 External Fragmentation

External fragmentation materializes in the memory heap when a freed block becomes partly or completely unusable for future allocation if, say, it 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 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, which is not possible to do reversibly.

Allocation and deallocation order is important in order to combat external fragmentation. For example, if we have a class A, which fit on one memory block of size n, and we have a class B, which fit on two memory blocks of size n and limited memory space, we can easily reach a situation, where we cannot fit more B objects due to external fragmentation.



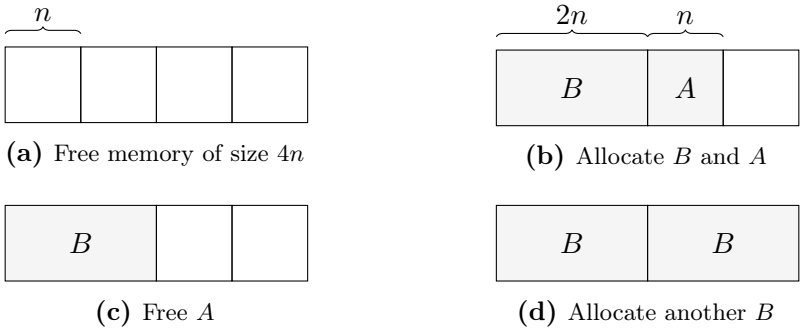
(a) Free memory of size $4n$

(b) Allocate A and B

(c) Free A. Cannot fit another B due to external fragmentation

Figure 3.2: Example of external fragmentation caused for allocation and deallocation order

Figure 3.2 shows this example, where the allocation and deallocation order causes a situation, in which we cannot allocate any more B objects, even though we physically have the required amount of free space in memory.



(a) Free memory of size $4n$

(b) Allocate B and A

(c) Free A

(d) Allocate another B

Figure 3.3: Example of avoiding external fragmentation using allocation and deallocation order

Figure 3.3 shows how changing allocation and deallocation order can combat external fragmentation.

3.2 Memory Garbage

A reversible computation should be garbage-free and as such 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 performed, as it simply is overwritten when the block is used later on. 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. Figure 3.4 illustrates how the output data (or garbage) of an injective function f is the input to its inverse function f'.

In heap allocation layouts, 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 heap itself can also be considered garbage and if it grows during execution, it should also be returned to its original size.

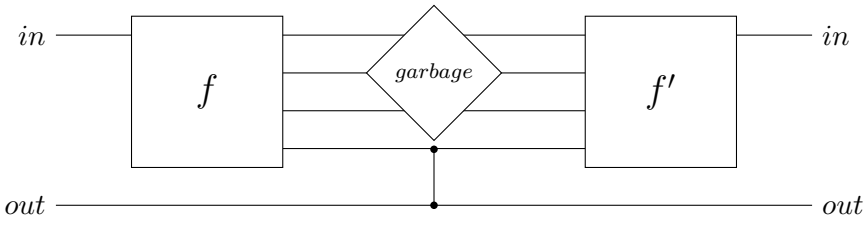


Figure 3.4: The "garbage" output of an injective function f is the input to its inverse function f'

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 [2]. Axelsen and Glück argue that an unrestored free list can be considered harmless garbage in the sense that the free list residing in memory 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. Figure 3.5 illustrates how an inverse, injective function f', whose non-inverse function f computes something which modifies a given free lists, does not require the *exact* output free list of f, but *any* free list of same layout as input for the inverse function f'. The output free list of f' will naturally be a further modified free list.

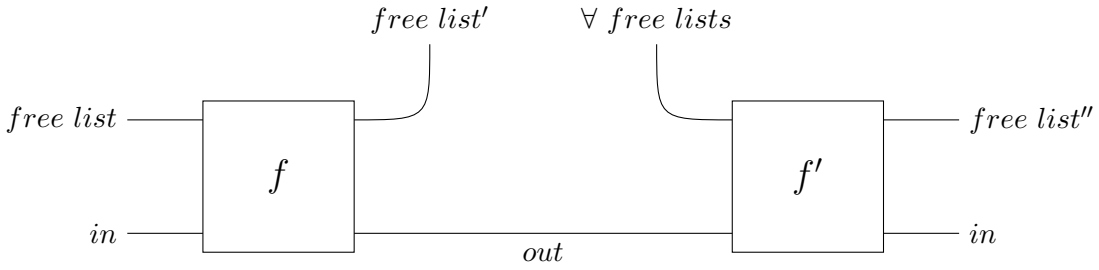


Figure 3.5: All free lists are considered equivalent "garbage" in terms of injective functions

This intuitively leads to the question of garbage classification. In the reversible setting all functions are injective. Thus, given some $input_f$, in a reversible computation using heap allocation, 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 3.5 shows how any free list can be passed between a function f and its inverse f^{-1}.

3.3 Linearity and Reference Counting

Programming languages use different approaches for storing and synchronizing variables and objects in memory. Typing *linearity* is a distinction, which can reduce storage management and synchronization costs [3].

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, at the cost of increased management in form of reference counting for e.g. objects. 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 use reference counting to allow multiple pointers to the same memory nodes as well as a translation from Rcfun into Janus in [17]. 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 and to avoid problematic fragmentation.

For the sake of simplicity, we will not consider the issue of retrieving memory pages reversibly. A reversible operating system is a long-term dream of the reversible researcher and as reversible programming language designers, we assume that Roopl++ will be running in an environment, in which an 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 management is manual and allocation has 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 and garbage collector to dispatch **malloc**- and **free**-like operations to the operating system and managing the obtained memory blocks in private heap(s) [13, 23, 19]. 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 [2, 9, 18].

Axelsen and Glück presented a static heap structure consisting of LISP-inspired constructor cells of fixed size and a single free list for the reversible function language RFUN in [2]. Mogensen presented an implementation in JANUS of reversible reference counting under the assumption of Axelsen and Glück's heap manager in [17]. Building on the previous work, Mogensen later presented a reversible intermediate language RIL and an implementation in RIL of a reversible heap manager, which uses reference counting and hash-consing to achieve garbage collection in [18].

We do not consider reference counting or garbage collection in the layouts presented in the following sections, but we later show how the selected layout for ROOPL++ is extended with reference counting in section 4.7.

3.4.1 Memory Pools

The simplest heap layout we can design uses fixed-sized blocks. This design is also known as memory pools, as memory is allocated from "pools" of fixed-sized blocks regardless of the record size. To model these pools of fixed-sized blocks, we simply use a linked list of identically sized free block cells, which we maintain over execution. While the fixed-block layout is simple and relatively easy in terms of implementation it is also largely uninteresting as it provides little to no options, besides sizing of the fixed-blocks, to combat fragmentation.

This layout comes with a few options in terms of the actual heap layout. If we only allow allocation of consecutive, adjacent free blocks, we should keep the free list sorted. If the free list is not sorted, and we have to allocate an object which requires n blocks, we have to iterate the free list n^2 times in the worst case to find a chain of consecutive blocks large enough to fit the object. The sorting part itself is non-trivial matter. Furthermore, we need some overhead storage inside the object to contains the references of the blocks occupied by the object, or some other structure which can be used when deallocating the object and returning all the blocks to the free list. If we allow allocation of non-consecutive blocks, larger amounts of book-keeping is required as we need to store knowledge of when and where the object is split.

Figures 3.2 and 3.3 from earlier in this chapter, in section 3.1.2 on page 48 illustrates examples with consecutive, fixed-sized block allocation.

3.4.2 One Heap Per Record Size

Instead of allocating space for objects from a single free list and heap, we could design an approach which uses one heap per record size. The respective classes and their sizes are easily identified during compile time from which the amount of heaps and free list will be initialized. This means the layout is very dynamic and potentially can change drastically in terms of the amount of heaps utilized depending on the input program.

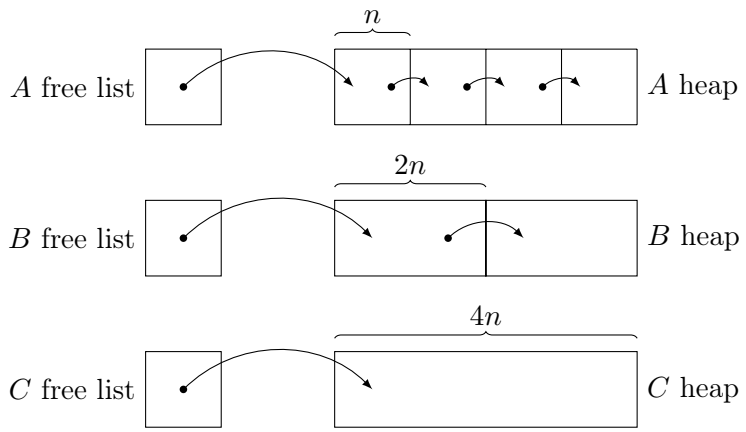


Figure 3.6: Memory layout using one heap per record size

Figure 3.6 illustrates three heaps with respective free lists for three classes A, B and C of size n, $2n$ and $4n$. Each heap is represented as a simple linked list with the free list simply being a pointer to the first free block in the heap.

The advantage of this approach would be effective elimination of internal and external fragmentation, as each heap fits their targeted record perfectly, making each allocation and deallocation tailored to the size of the record obtained from a static analysis during compilation, resulting in no over-allocation and no unusable chunks of freed memory appearing during varying deallocation order. Implementation-wise, allocation of an object of a given class simply becomes the task of popping the head of the respective free list, which easily can be determined at compile time. The inverse deallocation is simply added a new head to the free list.

Listing 3.1 outlines the allocation algorithm for this layout written in extended Janus from [26]. We assume that the heads of the free lists are stored in a single array primitive, such that the free list for records of size n are indexed at $n-2$ and $n > 2$ (as every record needs some overhead) and that we have heaps for continuous size range with no gaps. To maintain reversibility we only allow allocation from the head of the free list.

The algorithm consists of an entry point named **malloc** and a recursion body named **malloc1**. Given a zero-cleared pointer p, the size of the object we are allocating o_{size} and the array of free lists primitive, the recursion body is called after initializing a *counter*, which is an index into the free lists array and a counter size, c_{size}, which is the block size of the current free list the *counter* is indexed in. The recursion body first updates the free list index until we find a free list greater or equal to the size of the object we are allocating. Once such a free list has been found, the head of the free list is simply popped and the next block is set as the new head.

```
1  procedure malloc(int p, int osize, int freelists[])
2    local int counter = 0
3    local int csize = 2
4    call malloc1(p, osize, freelists, counter, csize)
5    delocal int csize = 2
6    delocal int counter = 0
7
8  procedure malloc1(int p, int osize, int freelists[], int counter, int csize)
9    if (csize < osize) then
10       counter += 1
11       csize += 1
12       call malloc1(p, osize, freelists, counter, csize)
13       csize -= 1
14       counter -= 1
15   else
16         p += freelists[counter]
17         freelists[counter] -= p
18
19         // Swap head of free list with p's next block
20         freelists[counter] ^= M(p)
21         M(p) ^= freelists[counter]
22         freelists[counter] ^= M(p)
23   fi csize < osize
```

Listing 3.1: Allocation algorithm for one heap per record size implemented in extended Janus

The obvious disadvantage to this layout 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.

Additional, helper classes of other sizes would spawn additional heaps and book-work, making the encapsulation concept of OOP rather unattractive, for the optimization-oriented reversible programmer.

Finally, while internal and external fragmentation is effectively eliminated, we are left with additional and considerable amounts of garbage in forms of all the heaps and free lists initialized in memory. If two record types only differ one word in size, two heaps would be initialized. Each heap intuitively need to be initialized with a chunk of memory from the underlying operating system such that objects can be allocated on their respective heaps, regardless of the number of times the heap is used during program execution. This is an obvious space requirement increase over the previously presented layout, and on average, the amount of required memory for a program compiled using this approach would probably be larger, than some of the following layouts, due to unoptimized heap utilization and sharing.

3.4.3 One Heap Per Power-Of-Two

To address the issues of the previous heap manager layout, we can optimize the amounts of heaps required by introducing a relatively small amount of internal fragmentation. Instead of having a heap per record size, we could have a heap per power-of-two. Records would be stored in the heap closest to their respective size and as such, we reduce the number of heaps needed, as many different records can be stored in the same heap. Records of size 5, 6, 7 and 8 would in the former layout be stored in four different heaps, where they would be stored in a single heap using this layout. Figure 3.7 illustrates the free lists and heaps up to n^m.

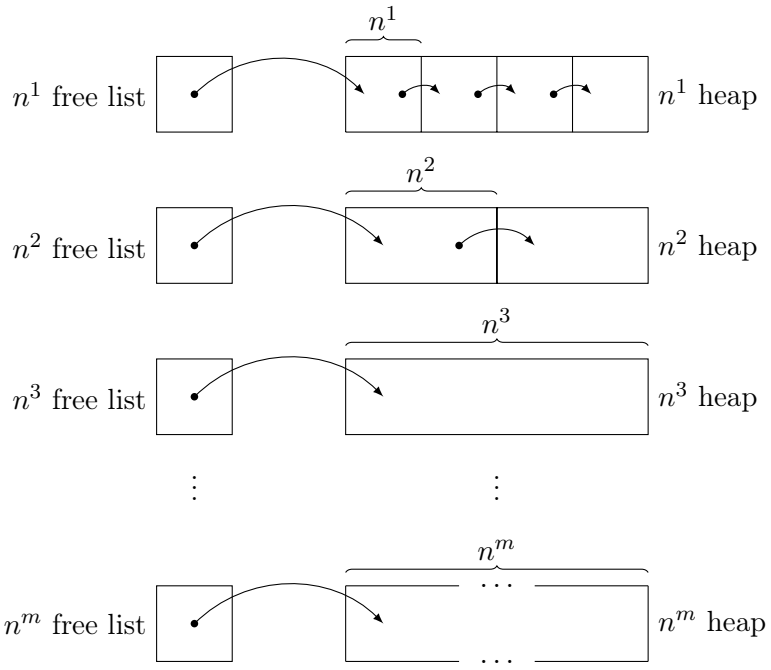


Figure 3.7: Memory layout using one heap per power-of-two

Internal fragmentation does become a problem for very large records, as blocks only are of size 2^n. An object of size 65 would fit on a 128 sized block, resulting in considerable amounts of wasted memory space in form of internal fragmentation. However, in the real world, most records are small and allocation of records causing this much amount of fragmentation is an unlikely scenario. To avoid large amounts of internal fragmentation building up when allocating large records, we could allocate space for large objects using smaller blocks. If a record exceeds some limit, which has been determined the cutoff point, one kilobyte for an example, we could split it into $\sqrt{n}$ sized chunks and use blocks of that size instead. This would reduce the amount of internal fragmentation at the cost of increased book-keeping. For smaller records, very minimal amounts of internal fragmentation appear.

The number of heaps needed for a computation can be determined at compile time by finding the smallest and largest record sizes and ensuring we have heaps to fit these effectively. The allocation process consists of determining the closest 2^n to the size of the record we are allocating and then simply popping the head of the respective free list.

Listing 3.2 shows a modified **malloc1** recursion body for the power-of-two approach. Once again, we assume our array of free lists contains the head of each free list, such that index n is the head of the free list of size 2^{n+1}. Instead of incrementing the counter size by one, as in the former layout algorithm, we double it, using the shown **double** procedure. Besides this change, the algorithm remains unchanged and still assume each heap has been initialized along with the free lists.

```
1  procedure double(int target)
2    local int current = target
3    target += current
4    delocal int current = target / 2
5
```

```
6   procedure malloc1(int p, int osize, int freelists[], int counter, int csize)
7     if (csize < osize) then
8         counter += 1
9         call double(csize)
10        call malloc1(p, osize, freelists, counter, csize)
11        uncall double(csize)
12        counter -= 1
13    else
14        if freelists[counter] != 0 then
15            p += freelists[counter]
16            freelists[counter] -= p
17
18            // Swap head of free list with p's next block
19            freelists[counter] ^= M(p)
20            M(p) ^= freelists[counter]
21            freelists[counter] ^= M(p)
22        else
23            counter += 1
24            call double(csize)
25            call malloc1(p, osize, freelists, counter, csize)
26            uncall double(csize)
27            counter -= 1
28        fi freelists[counter] = 0 || p != freelists[counter]
29    fi csize < osize
```

Listing 3.2: Allocation algorithm for one heap per power-of-two implemented in extended Janus

3.4.4 Shared Heap, Record Size-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 ensure 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. Having mixed-sized blocks in a single heap is also a task which might prove difficult to maintain in a reversible setting. How initialization and destruction of said heap should work is not clear. As with the multiple heap version of this layout, we are still left with the issues surrounding two records which only differs one word in size. In the former layout, two heaps were required to store records of these types. In this layout, we need to store two block sizes in our heap to allocate these records, with no internal fragmentation. We could allow these objects to be allocated on similarly-sized blocks, if we round the calculated class sizes up to, say, a power-of-two. We would essentially have a shared heap, power-of-two-specific free lists layout.

As the only change in this design are the heaps themselves, the allocation process remains unchanged from the one presented in listing 3.1 or listing 3.2 if we use the power-of-two approach. Figure 3.8 visualizes the shared heap and the free lists of this layout.

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

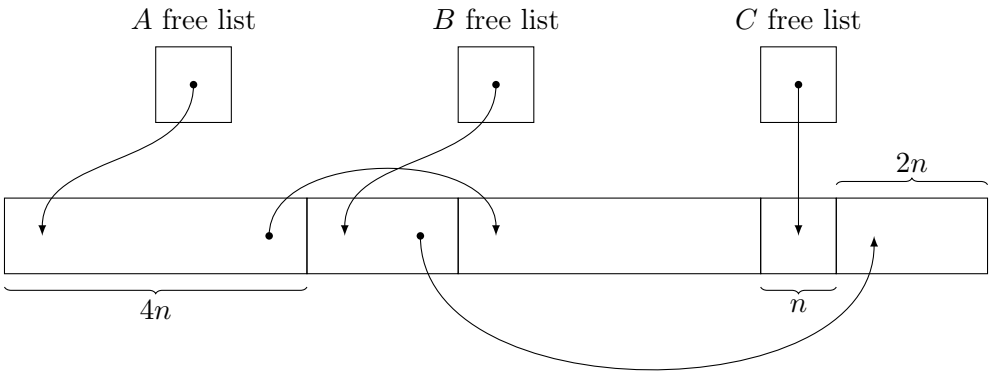
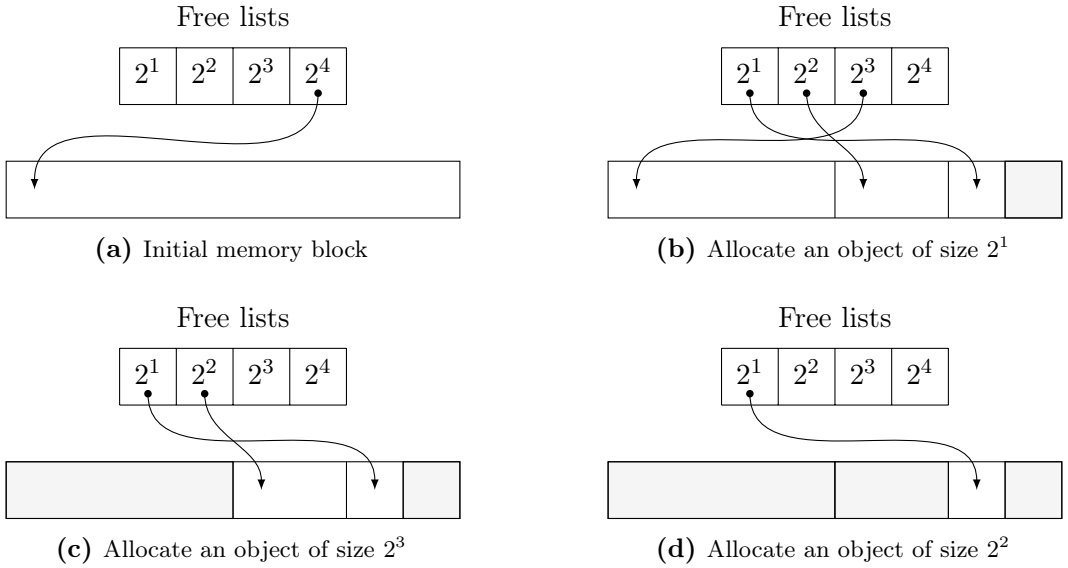


Figure 3.8: Record size-specific free lists on a shared heap

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, we do the action described above in reverse, thus merging the blocks again, where possible.



(a) Initial memory block

(b) Allocate an object of size 2^1

(c) Allocate an object of size 2^3

(d) Allocate an object of size 2^2

Figure 3.9: Buddy Memory block allocation example

Figure 3.9 illustrates an example of block splitting during allocation in the buddy system. Originally, one block of free memory is available. When allocating a record three factors smaller than the original block, three splits occurs.

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 optimize the problems around initializing a usable amount of variable-sized blocks in a shared heap, we simply initialize one large block in the buddy system, which we will split into smaller parts during execution.

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. As discussed in section 3.4.3, we would split large records into chunks of $\sqrt{n}$ at the cost of additional book-keeping.

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 malloc1(int p, int osize, int freelists[], int counter, int csize)
2    if (csize < osize) then
3        counter += 1
4        call double(csize)
5        call malloc1(p, osize, freelists, counter, csize)
6        uncall double(csize)
7        counter -= 1
8    else
9        if freelists[counter] != 0 then
10           p += freelists[counter]
11           freelists[counter] -= p
12
13           // Swap head of free list with p's next block
14           freelists[counter] ^= M(p)
15           M(p) ^= freelists[counter]
16           freelists[counter] ^= M(p)
17       else
18           counter += 1
19           call double(csize)
20           call malloc1(p, osize, freelists, counter, csize)
21           uncall double(csize)
22           counter -= 1
23           freelists[counter] += p
24           p += csize
25       fi freelists[counter] = 0 || p - csize != freelists[counter]
26   fi csize < osize
```

Listing 3.3: The Buddy Memory algorithm implemented in extended Janus

Listing 3.3 shows the Buddy Memory algorithm implemented in the extended Janus variant with local blocks from [26]. For simplification, object sizes are rounded to the nearest power-of-two during compile-time and we only allow allocations using the head of the free lists. The algorithm extends on the one heap per power-of-two algorithm presented in listing 3.2, page 54. The body of the allocation function is still executed recursively until a free list for a 2^n larger than the size of the object has been found. Once found, we continue searching until we have found a non-empty free list. If the non-empty free list for a 2^n larger than the object is found, the head of the list is popped and the popped block is split recursively, until a block the desired size is obtained. Throughout the splitting process, empty free lists are updated when a larger free block is split into a block which fits into those lists.

CHAPTER 4

Compilation

The following chapter presents the considerations and translation schemas used in the process of translating RoopL++ to the reversible low-level machine language Pisa. As RoopL++ is a continuation of RoopL, many techniques are carried directly over and have a such been left out.

Before presenting the RoopL++ compiler, a brief overview of the memory layout and modeling of the RoopL compiler, which the RoopL++ compiler is a continuation of, is provided.

4.1 The RoopL to Pisa Compiler

Haulund presented a proof-of-concept compiler along with the design for RoopL. The compiler translates well-typed RoopL programs into the reversible machine language Pisa in [10]. The RoopL compiler (RooplC) is written in Haskell and hosted at `https://github.com/TueHaulund/ROOPLC`.

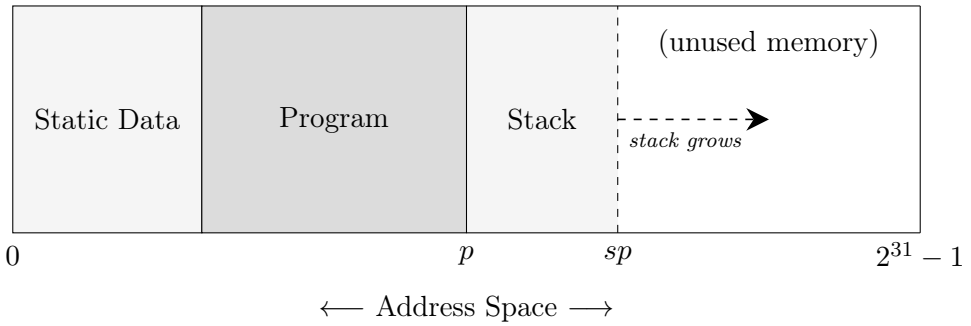


Figure 4.1: Memory layout of a RoopL program, originally from [10]

Figure 4.1 shows the memory layout of a compiled RoopL program. The layout consists of a static storage segment, the program segment and the stack.

The object model is simple and only features one additional word for storing the address of the virtual table for the object class. Figure 4.2 shows the prefixing for three simple classes modeling geometric shapes.

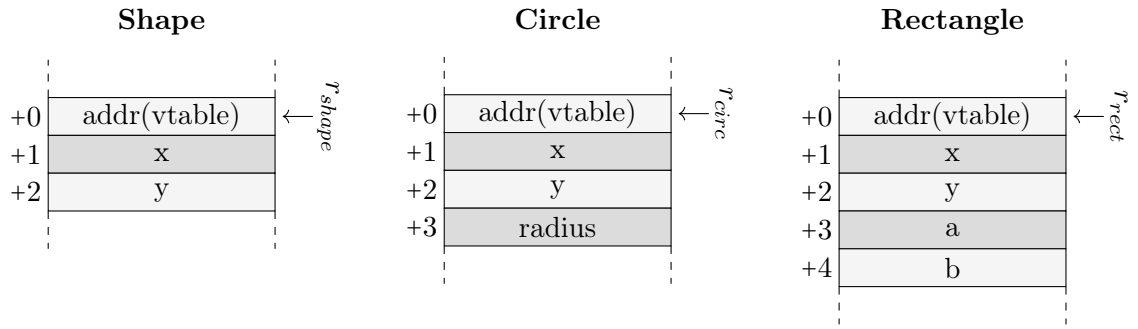


Figure 4.2: Illustration of prefixing in the memory layout of 3 Roopl objects, originally from [10]

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, addressing a number of disadvantages found in other considered layouts, 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.

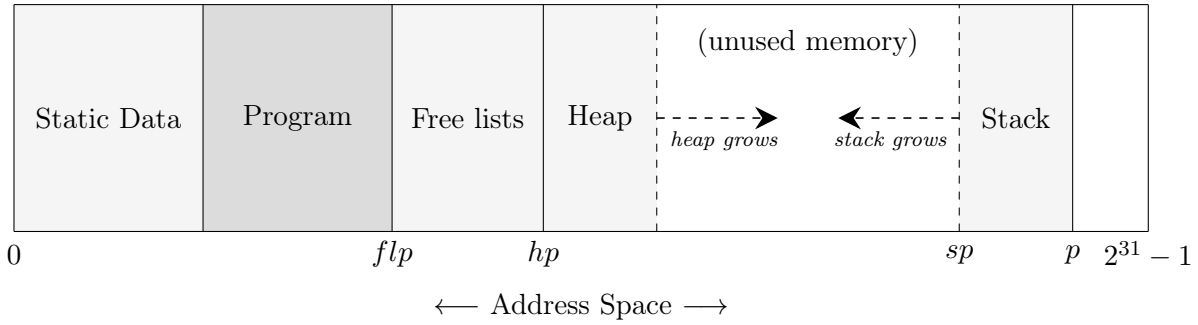


Figure 4.3: Memory layout of a Roopl++ program

Figure 4.3 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 mentioned in the previous chapter, 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 1024 words after the heap.

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.

The heap can simply be expanded by adding another block of largest possible size and storing the address to the respective free list.

4.3 Inherited ROOPL features

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

The dynamic dispatching mechanism presented in [10] 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.

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 4.4.

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

The free lists pointer is initialized by adding the base address, which varies with the size of the translated program, to the register r_{flps}. In figure 4.4 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).

```
(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)             XOR     r_sp    r_hp            ; Initialize stack pointer
(14)             ADDI    r_sp    offset_stack    ; Initialize stack pointer
(15)             XOR     r_m     r_sp            ; Store address of main object in r_m
(16)             XORI    r_v     label_vt        ; Store address of vtable in r_v
(17)             EXCH    r_v     r_sp            ; Push address of vtable onto stack
(18)             SUBI    r_sp    size_m          ; Allocate space for main object
(19)             PUSH    r_m                     ; Push 'this' onto stack
(20)             BRA     label_m                 ; Call main procedure
(21)             POP     r_m                     ; Pop 'this' from stack
(22)             SUBI    r_sp    size_m          ; Deallocate space of main object
(23)             EXCH    r_v     r_sp            ; Pop vtable address into r_v
(24)             XORI    r_v     label_vt        ; Clear r_v
(25)             XOR     r_m     r_sp            ; Clear r_m
(26)             SUBI    r_sp    offset_stack    ; Clear stack pointer
(27)             XOR     r_sp    r_hp            ; Clear stack pointer
(28)             SUBI    r_hp    size_fls        ; Clear heap pointer
(29)             XOR     r_hp    r_flsp          ; Clear heap pointer
(30)             SUBI    r_flps  p               ; Clear free lists pointer
(31)  finish :   FINISH                          ; Program exit point
```

Figure 4.4: Overall layout of a translated RoopL++ program

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 heap pointer register r_{hp}. The stack offset is configured in the source code and defaults to 1024, as described earlier in this chapter. As such, the heap and the stack each have 1024 words of space to utilize. 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 Buddy Memory Translation

As briefly mentioned, the Buddy Memory layout was selected as the memory manager layout as it addressed a number of problems related to fragmentation and initialization. The Buddy Memory layout could be converted to a reversible section with only a few restrictions and side effects, which will be described in this section. Firstly, we present the algorithm translated to PISA. As the algorithm is quite lengthy, it will be broken down into smaller chunks. The full translation is shown in a more readable version in appendix A.

The Buddy Memory algorithm consists of three JANUS procedures; the entry point **malloc**, the recursion body **malloc1** and a helper function **double**. The entry point is omitted for now, as it differs depending on which type of memory object we are allocating and will be presented in sections 4.6 and 4.8.1. The helper function can be implemented using a single instruction in PISA for our specific case of doubling number in the power-of-two, which we will show later.

```
(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
(7-63)                  ·····                  ; Allocation code
(64)     malloc1_bot :  BRA      malloc1_top   ; Jump
```

Figure 4.5: Dynamic dispatch approach for entering the allocation subroutine

Before we go into depth with the translation of the algorithm, we consider the mechanism for triggering the allocation mechanisms. Naively, we could generate the entire block of code required for allocation for every **new** or **delete** statement in the target program. This approach would severely limit the amounts of objects we can allocate as the register pressure of the Buddy Memory implementation is quite high, as will seen in this section. Instead, we can utilize the dynamic dispatching mechanisms, which also is used for method calling. This way, we only generate the allocation instructions once, and then simply jump to the entry point from different locations in the program. Figure 4.5 outlines the structure for this approach. By using the **SWAPBR** instruction we can jump from multiple points of origin in the compiled program and recursively for the algorithm's own recursive needs.

The main recursion body of the algorithm, **malloc1** from listing 3.3, page 57 consists of two conditionals, in which one is nested in the else branch of the outer conditional. Figure 4.6 shows the translation structure of the nested conditional pair, using the translation techniques for conditionals presented in [1].

```
1  if (csize < osize) then
2      // outer if-then
3  else
4      if freelists[counter] != 0 then
5          // inner if-then
6      else
7          // inner else
8      fi (freelists[counter] = 0 ||
9          p - csize != freelists[counter])
10 fi csize < osize
```

```
(7)                ·····                  ; Code for r_fl ← addr(freelists[counter])
(8)                ·····                  ; Code for r_block ← ⟦freelists[counter]⟧
(9)                ·····                  ; Code for r_c1_o ← ⟦c_size < object_size⟧
(10)               XOR  r_t   r_c1_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
(14-21)            ·····                  ; Code for outer if-then statement
(22)               XORI r_t   1           ; Set r_t = 1
(23)   o_assert_t : BRA  o_assert         ; Jump
(24)   o_test_f :  BRA  o_test            ; Receive jump
(25)               ·····                  ; Code for r_c1_i ← ⟦addr(freelists[counter]) ≠ 0⟧
(26)               XOR  r_t2  r_c1_i      ; Copy value of r_c1_i into r_t2
(27)               ·····                  ; Inverse of (25)
(28)   i_test :    BEQ  r_t2  r_0   i_test_f ; Receive jump
(29)               XORI r_t2  1           ; Clear r_t2
(30-34)            ·····                  ; Code for inner if-then statement
(35)               XORI r_t2  1           ; Set r_t2 = 1
(36)   i_assert_t : BRA  i_assert         ; Jump
(37)   i_test_f :  BRA  i_test            ; Receive jump
(38-47)            ·····                  ; Code for inner else statement
(48)   i_assert :  BNE  r_t2  r_0   i_assert_t ; Receive jump
(49)               EXCH r_tmp r_fl        ; Load address of head of current free list
(50)               SUB  r_p   r_cs        ; Set p to previous block address
(51)               ·····                  ; Code for r_c2_i1 ← ⟦p − c_size ≠ addr(freelists[counter])⟧
(52)               ·····                  ; Code for r_c2_i2 ← ⟦addr(freelists[counter]) = 0⟧
(53)               ·····                  ; Code for r_c2_i3 ← ⟦(p − c_size ≠ addr(freelists[counter])) ∨ (addr(freelists[counter]) = 0)⟧
(54)               XOR  r_r2  r_c2_i3     ; Copy value of r_c2_i3 into r_t2
(55)               ·····                  ; Inverse of (53)
(56)               ·····                  ; Inverse of (52)
(57)               ·····                  ; Inverse of (51)
(58)               ADD  r_p   r_cs        ; Inverse of (50)
(59)               EXCH r_tmp r_fl        ; Inverse of (49)
(60)   o_assert :  BNE  r_t   r_0   o_assert_t ; Receive jump
(61)               ·····                  ; Code for r_c2_o ← ⟦c_size < object_size⟧
(62)               XOR  r_t   r_c2_o      ; Copy value of c_size < object_size into r_t
(63)               ·····                  ; Inverse of (61)
```

Figure 4.6: PISA translation of the nested conditionals in the Buddy Memory algorithm

The nested conditionals contain large amounts of boilerplate code for evaluating the various expressions of the conditionals. As these conditionals requires comparisons with contents of the free lists, we must be careful with extracting and storing the values in the free list.

We have three statements to translate from here. The outer **if-then** statement, the inner **if-then** statement and the inner **else** statement.

```
1 counter += 1
2 call double(csize)
3 call malloc1(p, osize, freelists,
4              counter, csize)
5 uncall double(csize)
6 counter -= 1
```

```
(14)  ADDI r_c   1        ; Counter++
(15)  RL   r_sc  1        ; Call double(c_size)
(16)  ·····               ; Inverse of (7)
(17)  ·····               ; Code for pushing temp reg values to stack
(18)  BRA  malloc1_entry  ; Call malloc1())
(19)  ·····               ; Inverse of (17)
(20)  RR   r_sc  1        ; Inverse of (15)
(21)  SUBI r_c   1        ; Inverse of (14)
```

Figure 4.7: PISA translation of the outer **if-then** statement for the Buddy Memory algorithm

Figure 4.7 shows the translation of the outer **if-then** statement. As briefly mentioned, we can utilize PISA's right bit shift instruction, **RL**, in place of the **double** helper procedure from the JANUS implementation. By using a simple bit shift, we are able to maintain reversibility elegantly when doubling or halving numbers in the power-of-two. This statement also contains one of the careful storage operations of the free list values, in instruction **(16)**. Before we recursively branch to the entry point, we must place the previously extracted address of the head of the free list back into the free list. This is also the reason for instruction **(3)** in figure 4.5. Furthermore, we must push all temporary evaluated expression values to the stack, so they can be popped

when we return.

```
1 p += freelists[counter]
2 freelists[counter] -= p
3
4 // Swap head of free list
5 // with p's next block
6 freelists[counter] ^= M(p)
7 M(p) ^= freelists[counter]
8 freelists[counter] ^= M(p)
```

```
(30) ADD  r_p     r_block ; Copy address of the current block to p
(31) SUB  r_block r_p     ; Clear r_block
(32) EXCH r_tmp   r_p     ; Load address of next block
(33) EXCH r_tmp   r_fl    ; Set address of next block as new head of free list
(34) XOR  r_tmp   r_p     ; Clear address of next block
```

Figure 4.8: Pisa translation of the inner **if-then** statement for the Buddy Memory algorithm

Figure 4.8 shows the translation of the inner **if-then** statement. This statement translates easily using the **EXCH** instructions to swap with memory locations as simulated in the Janus code.

```
1 counter += 1
2 call double(csize)
3 call malloc1(p, osize, freelists,
4              counter, csize)
5 uncall double(csize)
6 counter -= 1
7 freelists[counter] += p
8 p += csize
```

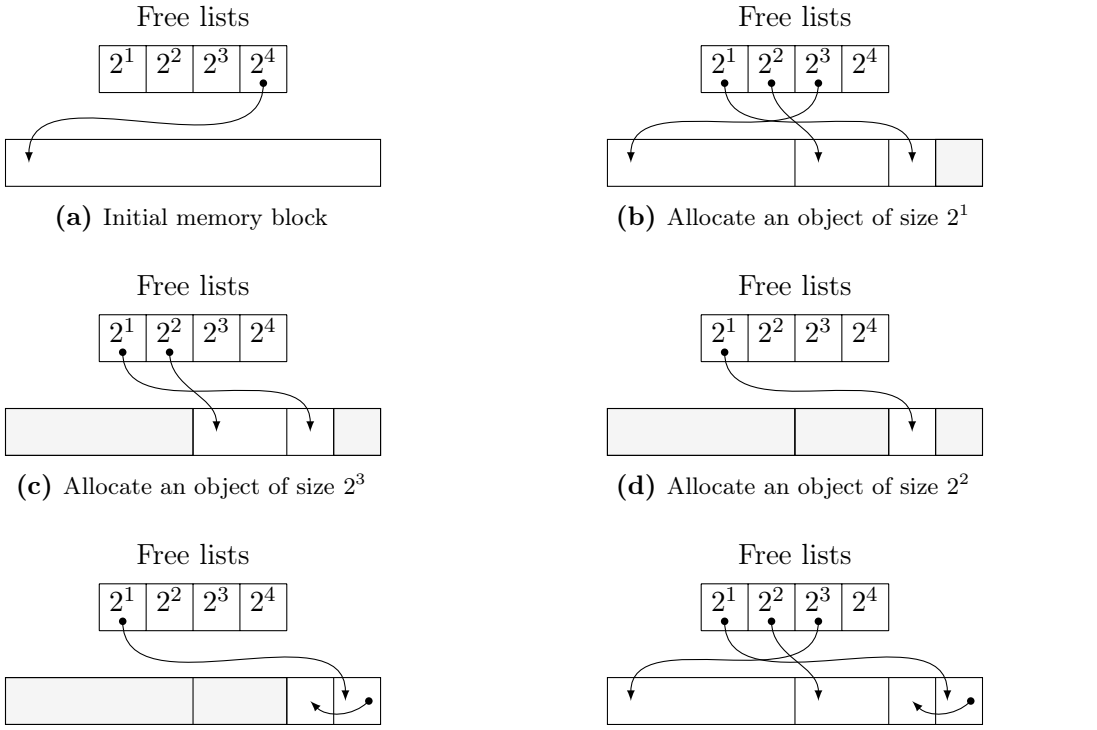
```
(38) ADDI r_c   1          ; Counter++
(39) RL   r_sc  1          ; Call double(c_size)
(40) ·····                 ; Code for pushing temp reg values to stack
(41) BRA  malloc1_entry    ; Call malloc1())
(42) ·····                 ; Inverse of (40)
(43) RR   r_sc  1          ; Inverse of (39)
(44) SUBI r_c   1          ; Inverse of (38)
(45) XOR  r_tmp r_p        ; Copy current address of p
(46) EXCH r_tmp r_fl       ; Store current address of p in current free list
(47) ADD  r_p   r_cs       ; Split block by setting p to second half of current block
```

Figure 4.9: Pisa translation of the inner **else** statement for the Buddy Memory algorithm

The last statement translation is the inner **else** statement shown in figure 4.9. This statement is almost identical to the outer **if-then** with the addition of the block splitting code. The block splitting is done in three instructions. First, the current block we are examining is set as the new head of the current free list. Afterwards the current free list block size is added to out pointer p, resulting in an effectively split block.

During the design of the reversible Buddy Memory algorithm a number of simplifications and limitations was required to ensure reversibility. Firstly, we only allow allocation from the head of a free list. This restriction ensures reversibility as we always can add a new head to a list, but not to the exact point in the linked list, where the block originally came from. Furthermore, we round all class sizes to the power-of-two. This simplifies the process of finding the right free list to allocate from, as the sizes we are comparing with always is of a power-of-two. The effects of these choices differs in severity. The latter, results in increased amounts of internal fragmentation, as discussed in the previous chapter. The former, however, prevents us from returning to one final block of free memory, if the deallocation order is not exactly opposite of the allocation order.

Figure 4.10 shows how alternative deallocation orders results in different free lists, compared to the original given to some function. However, as discussed in the section 3.2, we can consider every collection of Buddy Memory free lists equivalent, as a later computation can take another set of free lists and still execute its function, as long as the free lists have the required blocks available.



(a) Initial memory block

(b) Allocate an object of size 2^1

(c) Allocate an object of size 2^3

(d) Allocate an object of size 2^2

(e) Deallocate the object of size 2^1

(f) Deallocate the object of size 2^3, then the object of 2^2

Figure 4.10: Non-oppsite deallocation results in a different free list after termination

4.6 Object Allocation and Deallocation

Now that we have the main allocation mechanism in place and a method of accessing it through a label and a **SWAPBR** instruction, we can continue translating the **malloc** procedure entry point from listing 3.3 on page 57.

```
1 procedure malloc(int p, int osize,
2                  int freelists[])
3     local int counter = 0
4     local int csize = 2
5     call malloc1(p, osize, freelists,
6                  counter, csize)
7     delocal int csize = 2
8     delocal int counter = 0
```

```
(1)  l_malloc_top :  BRA     l_malloc_bot   ; Receive jump
(2)  l_malloc :      SWAPBR  r_o            ; Entry and exit point
(3)                  NEG     r_o            ; Negate return offset
(4)                  ADDI    c_size  2      ; Init c_size
(5)                  XOR     r_counter r_0  ; Init counter
(6)                  ···                    ; Pop r_p and object_size from stack
(7)                  PUSH    r_0            ; Push r_o
(8)                  BRA     l_malloc1      ; call malloc1()
(9)                  POP     r_0            ; Inverse of (7)
(10)                 ···     r_0            ; Inverse of (6)
(11)                 XOR     r_counter r_0  ; Inverse of (5)
(12)                 SUBI    c_size  2      ; Inverse of (4)
(13) l_malloc_bot :  BRA     l_malloc_top   ; Jump
```

Figure 4.11: PISA translation of the **malloc** procedure entry point of Buddy Memory algorithm

Figure 4.11 shows the translated **malloc** procedure. In addition to the original procedure, we also push the current return offset register value to the stack before we branch to the **malloc1** implementation, to ensure we have a zero-cleared register before starting the allocating process.

The translated procedure assumes that the pointer to the object we are allocating and its size are on top of the stack before entering the block. This translated procedure serves as the entry point for the allocation subroutine as it is also only generated once. Each **new** and **delete** statement branches to the l_{malloc} label to begin an allocation or a deallocation.

new c x

```
(1)  ···                                  ; Push registers
(2)  ···                                  ; Code for r_t ← x_size
(3)  PUSH  r_t                            ; Push r_t
(4)  PUSH  r_p                            ; Push r_p
(5)  BRA   l_malloc                       ; Allocate
(6)  POP   r_p                            ; Inverse of (4)
(7)  POP   r_t                            ; Inverse of (3)
(8)  ···                                  ; Inverse of (2)
(9)  ···                                  ; Inverse of (1)
(10) ···                                  ; Code for r_v ← [[addr(x)]]
(11) XORI  r_t   label_vt                 ; Store address of vtable in r_t
(12) EXCH  r_t   r_p                      ; Store vtable in new object
(13) ADDI  r_p   offset_ref               ; Index to ref count pos
(14) XORI  r_t   1                        ; Init ref count
(15) EXCH  r_t   r_p                      ; Store ref count
(16) SUBI  r_p   offset_ref               ; Inverse of (13)
(17) EXCH  r_p   r_v                      ; Store address in variable
(18) ···                                  ; Inverse of (10)
```

delete c x

```
(1)  ···                                  ; Code for r_p ← [[addr(x)]]
(2)  EXCH  r_t   r_p                      ; extract vtable from object
(3)  XORI  r_t   label_vt                 ; clear address of vtable in r_t
(4)  ADDI  r_p   offset_ref               ; Index to ref count pos
(5)  EXCH  r_t   r_p                      ; Extract ref count
(6)  XORI  r_t   1                        ; Clear ref count
(7)  SUBI  r_p   offset_ref               ; Inverse of (4)
(8)  ···                                  ; Push registers except r_p, r_t
(9)  ···                                  ; Code for r_t ← x_size
(10) PUSH  r_t                            ; Push r_t
(11) PUSH  r_p                            ; Push r_p
(12) RBRA  l_malloc                       ; Deallocate
(13) POP   r_p                            ; Inverse of (11)
(14) POP   r_t                            ; Inverse of (10)
(15) ···                                  ; Inverse of (9)
(16) ···                                  ; Inverse of (8)
(17) ···                                  ; Inverse of (1)
```

Figure 4.12: PISA translation of heap allocation and deallocation for objects

Figure 4.12 shows how each **new** and **delete** statement for objects are translated during compilation. They are simply each others inverse. For allocation, the object pointer and its size are pushed to the stack and then a jump to the malloc entry point is executed. After allocation, the virtual table and reference count are stored in the first two words of the allocated memory. Note how deallocation jumps and flips the direction of execution using the **RBRA** instruction, which then runs the allocation process in reverse. In the figure x_{size} denotes the computed size of objects with class c, plus two, to account for the virtual table and reference count space, rounded up to nearest power-of-two.

4.7 Referencing

As mentioned, one of the main strengths of Roopl++ in terms of increased expressiveness is allowance of multiple references to objects and arrays. When an object or array is constructed we allocate enough space to hold an additional *reference counter* which is initialized to 1. For each reference copied using the **copy**-statement, we incrementally increase the reference counter by 1. When we **uncopy** a reference, the reference counter is decreased. The object or array cannot be deconstructed until its reference counter has been returned to 1 as we would have a reference pointer to cleared memory in the heap. Such references are known as dangling pointers.

Figure 4.13 shows the object layout of Roopl++ objects with the added space for the reference

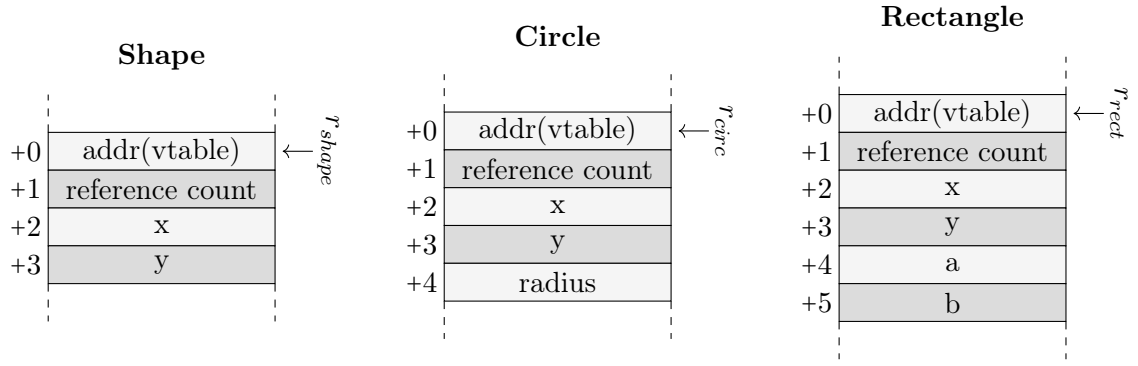


Figure 4.13: Illustration of prefixing in the memory layout of three ROOPL++ objects

counting from the original ROOPL model in figure 4.2 on page 59.

copy $c\ x\ x'$

```
(1)  ...                          ; Code for r_p ← addr(x)
(2)  ...                          ; Code for r_cp ← value(x')
(3)  XOR   r_cp   r_p             ; Copy address of x into x'
(4)  ADDI  r_p    offset_ref      ; Index to reference count address
(5)  EXCH  r_t    r_p             ; Extract reference count
(6)  ADDI  r_t    1               ; Increment reference count
(7)  EXCH  r_t    r_p             ; Store updated reference count
(8)  SUBI  r_p    offset_ref      ; Inverse of (3)
(9)  ...                          ; Inverse of (2)
(10) ...                          ; Inverse of (1)
```

uncopy $c\ x\ x'$

```
(1)  ...                          ; Code for r_p ← addr(x)
(2)  ...                          ; Code for r_cp ← value(x')
(3)  XOR   r_cp   r_p             ; Clear address of x from x'
(4)  ADDI  r_p    offset_ref      ; Index to reference count address
(5)  EXCH  r_t    r_p             ; Extract reference count
(6)  SUBI  r_t    1               ; Decrement reference count
(7)  EXCH  r_t    r_p             ; Store updated reference count
(8)  SUBI  r_p    offset_ref      ; Inverse of (3)
(9)  ...                          ; Inverse of (2)
(10) ...                          ; Inverse of (1)
```

Figure 4.14: PISA translation of the reference copying and deletion statements

Figure 4.14 shows the translated PISA code for the **copy** and **uncopy** statements. As shown, they are both very simple and each others inverse. For copying, the address of the passed variable x is simply copied into the zero-cleared value of x' and the reference count incremented by one. For deletion, the address is cleared and the reference count decremented. Copying and clearing is done through the **XOR** instruction. These translations features no error handling, but a solution is discussed in section 4.9.

4.8 Arrays

The static arrays in ROOPL++ are also heap allocated to allow dynamic lifetime. The array memory layout is presented in figure 4.15. As shown, the arrays feature two additional fields to store the size of the array and the reference count. Additionally, integer arrays store their values directly in the array while object arrays are a simple pointer stores.

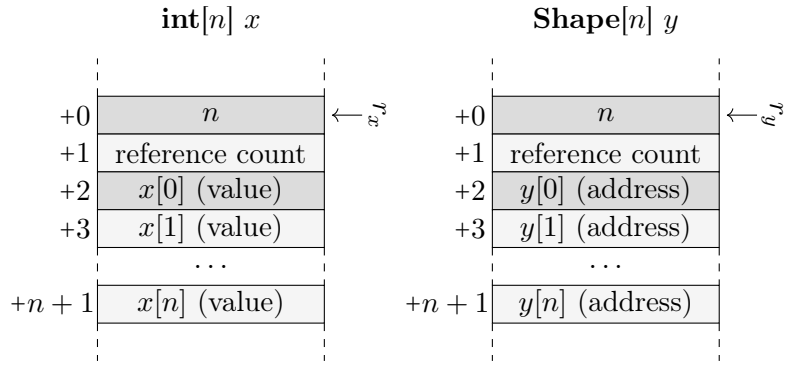


Figure 4.15: Illustration of prefixing in the memory layout of two Roopl++ arrays

4.8.1 Construction and Destruction

As Roopl++ arrays also are heap allocated, the buddy allocation implementation is also used for allocating arrays. The only difference between object and array allocation is that no virtual table is stored in the allocated space while the offsets for the reference counter are shared for both types. Due to this fact, **copy** and **uncopy** Pisa blocks generated during compile time are exactly the same for arrays and objects, as shown in the previous section.

new $a[e]$ x

(1)	···			; Push registers
(2)	···			; Code for $r_t \leftarrow [\![e]\!] + 2$
(3)	**PUSH**	r_t		; Push r_t
(4)	**PUSH**	r_p		; Push r_p
(5)	**BRA**	l_{malloc}		; Allocate array
(6)	**POP**	r_p		; Inverse of **(4)**
(7)	**POP**	r_t		; Inverse of **(3)**
(9)	···			; Inverse of **(1)**
(10)	···			; Code for $r_v \leftarrow [\![addr(x)]\!]$
(11)	**SUBI**	r_t	2	; $r_t \leftarrow [\![e]\!]$
(12)	**EXCH**	r_t	r_p	; Store size in new array
(13)	**ADDI**	r_p	$offset_{ref}$	; Index to ref count pos
(14)	**XORI**	r_t	1	; Init ref count
(15)	**EXCH**	r_t	r_p	; Store ref count
(16)	**SUBI**	r_p	$offset_{ref}$	; Inverse of **(13)**
(17)	**EXCH**	r_p	r_v	; Store address in variable
(18)	···			; Inverse of **(10)**

delete $a[e]$ x

(1)	···			; Code for $r_p \leftarrow [\![addr(x)]\!]$
(2)	**EXCH**	r_t	r_p	; extract size from object
(3)	···			; Code for $r_v \leftarrow [\![e]\!]$
(4)	**XORI**	r_t	r_v	; clear address of vtable in r_t
(5)	**ADDI**	r_p	$offset_{ref}$	; Index to ref count pos
(6)	**EXCH**	r_t	r_p	; Extract ref count
(7)	**XORI**	r_t	1	; Clear ref count
(8)	**SUBI**	r_p	$offset_{ref}$	; Inverse of **(5)**
(9)	···			; Push registers except r_p, r_v
(10)	**ADDI**	r_v	2	; Actual size of array
(11)	**PUSH**	r_v		; Push r_v
(12)	**PUSH**	r_p		; Push r_p
(13)	**RBRA**	l_{malloc}		; Deallocate array
(14)	**POP**	r_p		; Inverse of **(12)**
(15)	**POP**	r_v		; Inverse of **(11)**
(16)	**SUBI**	r_v	2	; Inverse of **(10)**
(17)	···			; Inverse of **(9)**
(18)	···			; Inverse of **(3)**
(19)	···			; Inverse of **(1)**

Figure 4.16: Pisa translations of array allocation and deallocation statements

Figure 4.16 shows the translation schemes used for array allocation and deallocation. As said, these are almost identical to the object allocation and deallocation schemes presented in figure 4.12 on page 66. Classes are analyzed during a compilation phase and their allocation size, the object size + 2 (for virtual table and reference counter) rounded up to nearest power-of-two. Arrays

cannot be sized determined during compilation, as that would require evaluating the expression passed to the initialization call, and as such, we add the overhead needed directly in the allocation and deallocation instructions.

4.8.2 Array Element Access

Array elements are simply passed as any other variable to methods or statements. Based on the variable type, compilation of various statements individually determine whether the address or the value of the passed variable should be used for the compiling the statement. For arrays, this is no different. If an integer array element is passed, it is treated just liked a regular integer variable. For an object array element, it is treated just like a regular object variable.

4.9 Error Handling

While a program written in RoopL++ might be syntactically valid and well-typed, it is not a guarantee that it executes successfully. A number of conditions exist, which cannot be determined at compile time, which in turn results in erroneous compiled and executed 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 [10].

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 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.
- An object variable must be initialized using **new** or **copy** before its methods can be called.
- An array variable must be initialized using **new** or **copy** before its fields can be accessed.
- Array cell indices must be within bounds defined in the expression passed during initialization.

It is the programmer's responsibility to meet these conditions. 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. We can insert run time error checks in the generated instructions such that the program is terminated if one of the conditions does not hold. The run time error checks can be added as dynamic error checks using error routines defined at labels, such as $label_{uninitialized\_object}$ which the program can jump to, if such a condition is unmet. Haulund presented an example for dynamic error checking for local blocks in [10]. Pisa and its simulator PendVM is, however, limited and does not support exit codes natively. To fully support dynamic error checking, PendVM could be extended to read from a value from a designated register to supply a more meaningful message for the programmer in the case of a run time exit.

4.10 Implementation

The Roopl++ compiler (Rooplppc) was implemented using techniques and translation schemes presented in this chapter, expanding upon the work of the original Roopl compiler (Rooplc). The compiler serves as a proof-of-concept and simply performs one-to-one translations of Roopl++ code to Pisa code without any optimizations along the way. The compiler is written in Haskell 7.10 and the translated output was tested on the Pendulum simulator, PendVM [4].

As with the Roopl compiler, the Roopl++ compiler is structured around the same six separate compilation phases.

1. **Parsing** consists of constructing an abstract syntax tree from the input program text using parser combinators from the Parsec library in Haskell.
2. **Class Analysis** verifies inheritance cycles, duplicated method names or fields and base classes. In this phase, we also compute the allocation size of each class
3. **Scope Analysis** constructs the virtual and symbol tables and maps every identifier to a unique variable or method.

4. **Type Checking** verifies that the parsed program is well-typed.
5. **Code Generation** translates the abstract syntax tree to blocks of PISA code in a recursive descent.
6. **Macro Expansion** expands macros left by the code generator for i.e. configuration variables, etc.

Compiled ROOPL programs have a size increase by a factor of 10 to 15 in terms of the lines of code. For ROOPL++ the size increase is much larger, partially due to the increase of static code included in form of the memory manager using the buddy layout described in this chapter and partially because heap allocations are more costly than stack allocations in terms of lines of code.

The ROOPL compiler was implemented in 1400 lines of HASKELL and the ROOPL++ compiler was extended to 2091 lines of HASKELL.

The entire compiler source code as well as example programs and their compiled versions are provided in the appendices and in the supplied ZIP archive. It is also hosted on Github as open source software under the MIT license at `https://github.com/cservenka/ROOPLPPC`.

Building and usage of the compiler is supplied in the README.md file found in the ZIP archive and in appendix B.

4.11 Evaluation

For evaluating the results of the implemented compiler, it was tested against example code provided throughout this thesis. Tests programs utilizing the linked list, doubly-linked list and binary tree data structures and the RTM implementation are found in appendix C.

Program	ROOPL++ LOC	PISA LOC
Linked List	5	6
Doubly-Linked List	8	9
Binary Tree	8	9

Figure 4.17: Lines of code comparison between target and compiled ROOPL++ programs

As discussed, while the compiler is considered proof-of-work and no mention-worthy optimizations has been implemented. However, for the sake of giving the reader and idea of the size blowup of a compiled ROOPL++ program, figure 4.17 details this difference. The lines of translated PISA instructions includes the 204 instructions needed for the **malloc** and **malloc1** PISA-equivalent mechanisms.

CHAPTER 5

Conclusions

We formally presented a dynamic memory management extension for the reversible object-oriented programing languages, ROOPL, in form of the superset language ROOPL++. The extension expands upon the previously presented static typing system defining well-typedness. The language successfully extends the expressiveness of its predecessor by allowing more flexibility within the domain of reversible object-oriented programming. With ROOPL++ we, as reversible programmers, can now define and model non-trivial dynamic data structures in a reversible setting, such as lists, trees and graphs. We illustrated this by examples programs such a new reversible Turing machine implementation along with implementations for linked lists, doubly-linked lists and binary trees as well as techniques for traversing these. Besides expanding the expressiveness of ROOPL, we have further shown that complex dynamic data structures are not only feasible, but furthermore does not contradict the reversible computing paradigm.

We presented various dynamic memory management layouts and how each would translate into the reversible setting. Weighing the advantages and disadvantages of each, the Buddy Memory layout was found to translate into reversible code very naturally with few side effects and addressed a number of disadvantages found in other considered layouts. With dynamically lifetimed objects allocation and deallocation order is important in terms of a entirely garbage-free computation. In most cases with ROOPL++, we only obtain partially garbage-free computations, as our free lists might not be restored to their original form, without an effective garbage collector design for the memory manager.

Techniques for clean translations of extended parts of the language, such as the memory manager and the new static array type have been demonstrated and implemented in a proof-of-concept compiler for validation.

With ROOPL++, we have successfully further strengthened the high-level layer of abstraction within the reversible programming paradigm.

5.1 Future Work

Naturally with the discovery of feasibility of non-trivial, reversible data structures with the introduction of ROOPL++, further study of design and implementation of reversible algorithms working with these data structures are an obvious contender for future research. Data structures such as lists, graphs and trees could potentially provide very interesting future reversible programs.

In terms of the future of reversible object-oriented languages, additional works could be made to extend the static array type with a fully dynamic array supporting multiple dimensionality. This addition could further help the discovery and research of reversible data structures such as trees and graphs. Such an extension could perhaps be added via a **put** and **take** statement pair, being each others inverse. After a dynamic array has been declared, it could automatically reallocate or upscale its internal space when putting new data outside of its current bounds. In reverse, the space could shrink or reallocate when removing the largest indexed value. The current memory management layout will still suffice for this extension.

Finally, more research could be conducted into reversible heap managers. We provided a simple manager which translated to our problem domain naturally. To obtain completely garbage free computations, a garbage collector could be designed to work with the reversible Buddy Memory memory manager. A reversible garbage collector has also been designed and shown feasible for the reversible functional language RCFUN in [18]. This garbage collector could perhaps be converted to an object-oriented setting. Additionally, experimentation with implementing the Buddy Memory layout into other reversible languages with dynamic allocation and deallocation such as R-WHILE and R-CORE provides an interesting opportunity [7, 8].

References

[1] Axelsen, H. B. "Clean Translation of an Imperative Reversible Programming Language". In: *CC 2011/ETAPS 2011: Proceedings of the 20th International Conference on Compiler Construction* (2011), pp. 144–163.

[2] 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.

[3] Baker, H. G. *'Use-Once' Variables and Linear Objects – Storage Management, Reflection and Multi-Threading.* URL: http://home.pipeline.com/~hbaker1/Use1Var.html.

[4] Clark, C. R. *Improving the Reversible Programming Language R and its Supporting Tools.* URL: http://www.cise.ufl.edu/research/revcomp/users/cclark/pendvm-fall2001/.

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

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

[7] Glück, R. and Yokoyama, T. "A Linear-Time Self-Interpreter of a Reversible Imperative Language". In: *Computer Software* 33.3 (2016), pp. 108–128. DOI: 10.11309/jssst.33.3_10.

[8] Glück, R. and Yokoyama, T. "A Minimalist's Reversible While Language". In: *IEICE Transactions on Information and Systems* E100.D.5 (2017), pp. 1026–1034. DOI: 10.1587/transinf.2016EDP727.

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

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

[11] 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.

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

[13] 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.

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

[15] Mitchell, J. C. and Apt, K. *Concepts in Programming Languages*. New York, NY, USA: Cambridge University Press, 2001. ISBN: 0521780985.

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

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

[18] 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.

[19] *Python Software Foundation: Memory Management*. URL: https://docs.python.org/2/c-api/memory.html.

[20] Schultz, U. P. and Axelsen, H. B. "Elements of a Reversible Object-Oriented Language". In: *Reversible Computation: 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*. Ed. by S. Devitt and I. Lanese. Cham: Springer International Publishing, 2016, pp. 153–159. ISBN: 978-3-319-40578-0. DOI: 10.1007/978-3-319-40578-0_10. URL: https://doi.org/10.1007/978-3-319-40578-0_10.

[21] 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.

[22] 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).

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

[24] Vieri, C. J. "Pendulum: A Reversible Computer Architecture". Master's Thesis. University of California at Berkeley, 1993.

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

[26] 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.

[27] 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.

[28] 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.

APPENDIX A

Pisa Translated Buddy Memory

(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 $r_{fl} \leftarrow addr(freelists[counter])$
(8)		.....				; Code for $r_{block} \leftarrow [\![freelists[counter]]\!]$
(9)		.....				; Code for $r_{e1_o} \leftarrow [\![c_{size} < object_{size}]\!]$
(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)		.....				; Inverse of **(7)**
(17)		.....				; Code for pushing temp reg values to stack
(18)		**BRA**	$malloc1_{entry}$			; Call $malloc1())$
(19)		.....				; Inverse of **(17)**
(20)		**RR**	r_{sc}	1		; Inverse of **(15)**
(21)		**SUBI**	r_c	1		; Inverse of **(14)**
(22)		**XORI**	r_t	1		; Set $r_t = 1$
(23)	o_{assert_t} :	**BRA**	o_{assert}			; Jump
(24)	o_{test_f} :	**BRA**	o_{test}			; Receive jump
(25)		.....				; Code for $r_{e1_i} \leftarrow [\![addr(freelists[counter]) \neq 0]\!]$
(26)		**XOR**	r_{t2}	r_{e1_i}		; Copy value of r_{e1_i} into r_{t2}
(27)		.....				; Inverse of **(25)**
(28)	i_{test} :	**BEQ**	r_{t2}	r_0	i_{test_f}	; Receive jump
(29)		**XORI**	r_{t2}	1		; Clear r_{t2}
(30)		**ADD**	r_p	r_{block}		; Copy address of the current block to p
(31)		**SUB**	r_{block}	r_p		; Clear r_{block}
(32)		**EXCH**	r_{tmp}	r_p		; Load address of next block
(33)		**EXCH**	r_{tmp}	r_{fl}		; Set address of next block as new head of free list
(34)		**XOR**	r_{tmp}	r_p		; Clear address of next block
(35)		**XORI**	r_{t2}	1		; Set $r_{t2} = 1$
(36)	i_{assert_t} :	**BRA**	i_{assert}			; Jump
(37)	i_{test_f} :	**BRA**	i_{test}			; Receive jump
(38)		**ADDI**	r_c	1		; $Counter$++
(39)		**RL**	r_{sc}	1		; Call $double(c_{size})$
(40)		.....				; Code for pushing temp reg values to stack
(41)		**BRA**	$malloc1_{entry}$			; Call $malloc1())$
(42)		.....				; Inverse of **(40)**
(43)		**RR**	r_{sc}	1		; Inverse of **(39)**
(44)		**SUBI**	r_c	1		; Inverse of **(38)**

(45) **XOR** r_{tmp} r_p ; Copy current address of p

(46) **EXCH** r_{tmp} r_{fl} ; Store current address of p in current free list

(47) **ADD** r_p r_{cs} ; Split block by setting p to second half of current block

(48) i_{assert} : **BNE** r_{t2} r_0 i_{assert_t} ; Receive jump

(49) **EXCH** r_{tmp} r_{fl} ; Load address of head of current free list

(50) **SUB** r_p r_{cs} ; Set p to previous block address

(51) ..... ; Code for $r_{e2_{i1}} \leftarrow [\![p - c_{size} \neq addr(freelists[counter])]\!]$

(52) ..... ; Code for $r_{e2_{i2}} \leftarrow [\![addr(freelists[counter]) = 0]\!]$

(53) ..... ; Code for $r_{e2_{i3}} \leftarrow [\![(p - c_{size} \neq addr(freelists[counter])) \vee (addr(freelists[counter]) = 0)]\!]$

(54) **XOR** r_{r2} $r_{e2_{i3}}$; Copy value of $r_{e2_{i3}}$ into r_{t2}

(55) ..... ; Inverse of **(53)**

(56) ..... ; Inverse of **(52)**

(57) ..... ; Inverse of **(51)**

(58) **ADD** r_p r_{cs} ; Inverse of **(50)**

(59) **EXCH** r_{tmp} r_{fl} ; Inverse of **(49)**

(60) o_{assert} : **BNE** r_t r_0 o_{assert_t} ; Receive jump

(61) ..... ; Code for $r_{e2_o} \leftarrow [\![c_{size} < object_{size}]\!]$

(62) **XOR** r_t r_{e2_o} ; Copy value of $c_{size} < object_{size}$ into r_t

(63) ..... ; Inverse of **(61)**

(64) $malloc1_{bot}$: **BRA** $malloc1_{top}$; Jump

APPENDIX B

ROOPLPPC Source Code

AST.hs

PISA.hs

Parser.hs

ClassAnalyzer.hs

ScopeAnalyzer.hs

TypeChecker.hs

CodeGenerator.hs

MacroExpander.hs

ROOPLPPC.hs

APPENDIX C

Example Ouput

LinkedList.rplpp

LinkedList.pal

BinaryTree.rplpp

BinaryTree.pal

DoublyLinkedList.rplpp

DoublyLinkedList.pal

RTM.rplpp

RTM.pal