UNIVERSITY OF COPENHAGEN
FACULTY OF SCIENCE

Master's Thesis

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

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

Supervisors: Robert Glück & Torben Ægidius Mogensen

Revision History

Revision	**Date**	**Author(s)**	**Description**
0.1	2017-05-02	Martin	Roopl++ design and heap design discussion
0.2	2017-05-16	Martin	Updated front-page logo and expanded heap design section

Table of Contents

1 Introduction **3**
1.1 Reversible Computing . . . 3
1.2 Reversible Object-Oriented Programming . . . 3
1.3 Motivation . . . 3
1.4 Thesis Statement . . . 3
1.5 Outline . . . 3

2 The ROOLP++ Language **4**
2.1 Syntax . . . 5
2.2 Object Instantiation . . . 6
2.3 Vector Instantiation . . . 6

3 Compilation **7**
3.1 Dynamic Memory Management . . . 7
3.1.1 Heap manager layout . . . 7

References **12**

CHAPTER 1

Introduction

TODO

1.1 Reversible Computing

TODO

1.2 Reversible Object-Oriented Programming

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

1.3 Motivation

ROOPL's block defined objects and lack of references are problematic when write complex, reversible programs using OOP methodologies. It has therefore been proposed to extend and partially redesign the language with dynamic memory management in mind, such that these shortcomings can be addressed, ultimately increasing the usability of reversible OOP. Work within the field of reversible computing related to heap manipulation [1], reference counting [8] and garbage collection [9] suggests that a ROOPL extension is feasible.

1.4 Thesis Statement

TBD

1.5 Outline

TBC

CHAPTER 2

The ROOLP++ Language

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

`TODO: Arrays at some point`

`TODO: ROOPL++ example program`

`Idea: Let ROOPL++ be a superset of ROOPL`

`Idea: String type?`

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.

```
TODO: Local block
```

Roopl++ Grammar

$$
\begin{array}{rcll}
\mathit{prog} & ::= & \mathit{cl}^{+} & \text{(program)} \\
\mathit{cl} & ::= & \textbf{class}\ c\ (\textbf{inherits}\ c)^{?}\ (t\ x)^{*}\ m^{+} & \text{(class definition)} \\
t & ::= & \textbf{int} \mid c & \text{(data type)} \\
m & ::= & \textbf{method}\ q\,\texttt{(}t\ x\texttt{,}\ \ldots\texttt{,}\ t\ x\texttt{)}\ s & \text{(method)} \\
s & ::= & x\ \odot\texttt{=}\ e \mid x\ \texttt{<=>}\ x & \text{(assignment)} \\
 & \mid & \textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s\ \textbf{fi}\ e & \text{(conditional)} \\
 & \mid & \textbf{from}\ e\ \textbf{do}\ s\ \textbf{loop}\ s\ \textbf{until}\ e & \text{(loop)} \\
 & \mid & \textbf{new}\ c\ x & \text{(object construction)} \\
 & \mid & \textbf{delete}\ c\ x & \text{(object deconstruction)} \\
 & \mid & \textbf{call}\ q\,\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)} \mid \textbf{uncall}\ q\,\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)} & \text{(local method invocation)} \\
 & \mid & \textbf{call}\ x\texttt{::}q\,\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)} \mid \textbf{uncall}\ x\texttt{::}q\,\texttt{(}x\texttt{,}\ \ldots\texttt{,}\ x\texttt{)} & \text{(method invocation)} \\
 & \mid & \textbf{skip} \mid s\ s & \text{(statement sequence)} \\
e & ::= & \overline{n} \mid x \mid \texttt{nil} \mid e\ \otimes\ e & \text{(expression)} \\
\odot & ::= & \texttt{+} \mid \texttt{-} \mid \texttt{\^{}} & \text{(operator)} \\
\otimes & ::= & \odot \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{|} \mid \texttt{\&\&} \mid \texttt{||} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=} & \text{(operator)}
\end{array}
$$

Syntax Domains

$$
\begin{array}{lll}
\mathit{prog} \in \text{Programs} & s \in \text{Statements} & n \in \text{Constants} \\
\mathit{cl} \in \text{Classes} & e \in \text{Expressions} & x \in \text{VarIDs} \\
t \in \text{Types} & \odot \in \text{ModOps} & q \in \text{MethodIDs} \\
m \in \text{Methods} & \otimes \in \text{Operators} & c \in \text{ClassIDs}
\end{array}
$$

Figure 2.1: Syntax domains and EBNF grammar for Roopl++

2.2 Object Instantiation

TODO

2.3 Vector Instantiation

```
Idea:  Implement dynamic array (named vector).
Idea:  Instantiate with alloc call.
Idea:  Inverse operation:  dealloc
```

CHAPTER 3

Compilation

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

3.1 Dynamic Memory Management

Native support of complex data structures is not a trivial matter to implement in a reversible computing environment. Variable-sized records and frames need to be stored efficiently in a structured heap, while avoiding garbage build-up to maintain reversibility. A reversible heap manager layout has been proposed for a simplified version of the reversible functional language RFUN and later expanded to allow references to avoid deep copying values [1, 11, 8].

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

3.1.1 Heap manager layout

Heap managers can be implemented in numerous ways. Different layouts yield advantages when allocating memory, finding a free block or when collecting garbage. As our goal is to construct a garbage-free heap manager, our finalized design should emphasize and reflect this objective in particular. Furthermore, we should attempt to allocate and deallocate memory as efficiently as possible, as merging and splitting of blocks is a non-trivial problem in a reversible setting.

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

Historically, most object-oriented programming languages utilize a dynamic memory manager during program execution. In older, lower-level languages such as C, memory allocation had to be stated explicitly and with the requested size through the `malloc` statement and deallocated using the `free` statement. Modern languages, such as C++, JAVA and PYTHON, *automagically* allocates and frees space for objects and variable-sized arrays by utilizing their dynamic memory manager to dispatch `malloc`- and `free`-like operations to the operating system and managing the obtained memory blocks in private heap(s) [7, 10, 2]. The heap layout of these managers vary from language to language and compiler to compiler.

Previous work on reversible heap manipulation has been done for reversible functional languages in [1, 3, 9].

`TODO: Something about Cons/Nil heap from [1]`

`TODO: Something about ref count extension [8]`

Memory Pools

A simple layout for a heap manager would allocate memory using fixed-size blocks regardless of the actual size of the record, which is known as memory pooling [5]. The advantage of a heap layout following this approach would lie in the simplicity of its implementation, as simple a linked list of identical-sized free cells would need to be maintained. If we assume a our fixed-sized blocks are of size n machine words, the following `get_free` subroutine allocates and deallocates memory cells for a record of size m.

```
if ($r_{flp}$ == 0)
then
        XOR $r_{cell}$ $r_{hp}$
        ADDI $r_{hp}$ $m + (m\%n)$
else
        EXCH $r_{cell}$ $r_{flp}$
        SWAP $r_{cell}$ $r_{flp}$
fi ($r_{flp}$ == 0) && ($r_{cell}$ == $r_{hp}$ - $m + (m\%n)$)
```

Listing 3.1: Allocating and deallocating records of size m using block of a fixed size n. Code modified from [1]

A huge disadvantage to using fixed-sized memory blocks is the fragmentation that occurs when freeing blocks. When freeing a number of blocks in the middle of a section of allocated blocks, fragmentation occurs, which becomes a problem if we need to allocate space for a large record but only have small sections of fixed-blocks available, scattered throughout the heap. A garbage collector could solve this issue, but is a non-trivial matter to implement.

`TODO: Graphics`

Buddy Memory

In this approach we have all blocks be variable-size of the power-of-two. Having different block sizes and methods for splitting a larger block into smaller ones reduces fragmentation. In terms of implementation, reversible splitting and merging might be possible as we're always doubling the size when merging and halving when splitting, thus making the operations logically inverse of each other [6].

Assuming we have access to an array of pointers $flps[]$, pointing to the beginning of a free list, index at its power-of-two, we could allocate and deallocate a record of size n in the following manner

```
powerOfTwo = 2
while (powerOfTwo < n)
        powerOfTwo = powerOfTwo * 2

if (flps[powerOfTwo] == 0)
then
        XOR $r_{cell}^{powerOfTwo}$ $r_{hp}^{powerOfTwo}$
        ADDI $r_{hp}^{powerOfTwo}$ powerOfTwo
else
        EXCH  $r_{cell}^{powerOfTwo}$ flps[powerOfTwo]
        SWAP $r_{cell}^{powerOfTwo}$ flps[powerOfTwo]
fi (flps[powerOfTwo] == 0) && ($r_{cell}^{powerOfTwo}$ == $r_{hp}^{powerOfTwo}$ - powerOfTwo)
```

Listing 3.2: Allocating and deallocating records of size n using Buddy Memory

`TODO: Should growing/shrinking be handled here?`

`TODO: Graphics`

One heap per record type

Another layout approach would be maintaining one heap per record type in the program. During compilation, classes would be analysed and a heap for each class would be created. The advantage of this approach would be less garbage generation, as each allocation is tailored as closely as possible to the size of the record obtained from a static analysis during compilation. The obvious disadvantage is the amount of book-keeping and workload associated with growing and shrinking a heap and its neighbours.

`Algorithm: Determine heap for record type, call get_free on found heap`

`TODO: Graphics`

Shared heap, record type-specific free lists

A combination of the previous layouts would consist of a single heap, but record-specific free lists. This way, we ensure minimal fragmentation when allocating and freeing as the different free lists ensures that allocation of an object wastes as little memory as possible. By only keeping one heap, we eliminate the growth/shrinking issues of the multiple heap layout. There is, however, still a considerable amount of book-keeping involved in maintaining multiple free-lists. The bigger the number of unique classes defined in a program, the more free-lists we need to maintain during execution. If the free-lists are not allowed to point at the same free block (which they intuitively shouldn't in order to ensure reversibility), programs with many classes, say one hundred, could potentially fill up the heap with one hundred free lists, but only ever allocate objects for one of the classes, thus wasting a lot of memory on unused free lists. This scenario could potentially be avoided with compiler optimizations.

```
   Algorithm:  Find block size for record type-specific free list, call
get_free on shared heap
```

```
   TODO: Graphics
```

One heap per power-of-two

A different approach as to having one heap per record type, would be having one heap per power-of-two until some arbitrary size. Using this approach, records would be stored in the heap which has a block size of a power-of-two closes matching to the record's size. This layout is a distinction from the "one heap per record type" as it still retains the size-optimized storing idiom but allows the heaps to contain records of mixed types. For programs with a large amount of small, simple classes needed to model some system, where each class is roughly the same size, the amount of heaps constructed would be substantially smaller than using one heap per record type, as many records will fit within the same heap. Implementation wise, the number of heaps can be determined at compile time. Furthermore, to ensure we do not end up with heaps of very large memory blocks, an arbitrary power-of-two size limit could be set at, say, 1 kb . If any record exceeds said limit, it could be split into $\sqrt{n}$ size chunks and stored in their respective heaps. This approach does, however, also suffer from large amount of book-keeping and fiddling when shrinking and growing adjacent heaps.

```
   Algorithm:  Similar to buddy memory?
```

```
   TODO: Graphics
```

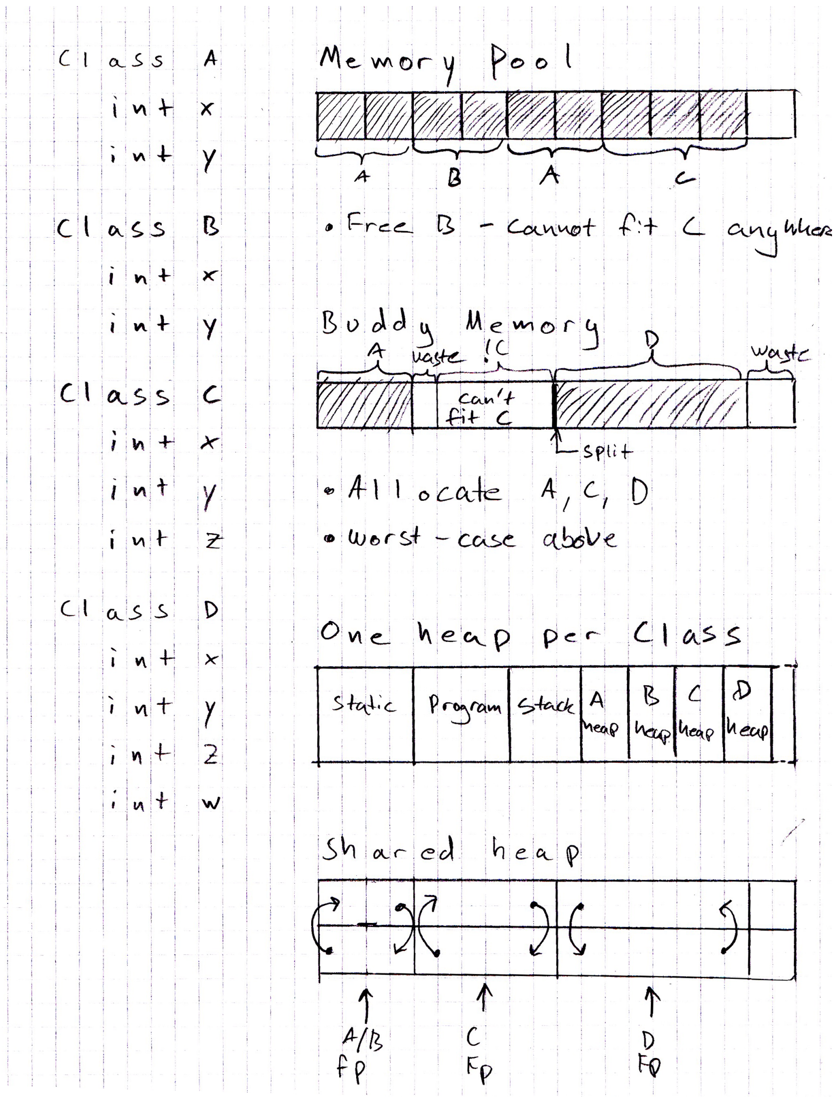

Figure 3.1: Heap memory layouts (Draft)

References

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

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

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

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

[5] Kenwright, B. “Fast Efficient Fixed-Size Memory Pool”. In: *COMPUTATION TOOLS 2012 : The Third International Conference on Computational Logics, Algebras, Programming, Tools, and Benchmarking* (2012), pp. 1–6.

[6] Knuth, D. *The Art of Computer Programming*. Vol. 1: Fundamental Algorithms. Reading, Massachusetts: Addison-Wesley, 1997, pp. 435–455. ISBN: 0-201-89683-4.

[7] 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.

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

[9] 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.

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

[11] 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.