FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

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

Table of Contents

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 [2]. 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 [3] and garbage collection [4] 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.

ROOPL++ Grammar

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

Syntax Domains

$\mathit{prog} \in$ Programs	$s \in$ Statements	$n \in$ Constants
$\mathit{cl} \in$ 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 2.1: 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

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 needs 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, 5, 3].

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 blocks would be a non-reversible action.

Fixed-sized records

A simple layout for a heap manager would allocate memory by allocating the same amount (fixed-size) of memory regardless of the actual size of the record. This would simplify the implementation greatly as the free list simply is a linked list of free cells of identical size. However, a huge disadvantage to this approach is large amounts of garbage is generated if the difference in sizes between different types of records are significant.

`TODO: Visualization`

One heap per record

Another layout approach would be maintaining one heap per record 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.

`TODO: Visualization`

Variable-sized records

In this approach the free list is a linked list of small fixed-size blocks. When allocating space for a record, n blocks of at least the size of the record is allocated. This layout is simple to implement as we avoid having to determine free list block size dependent in the size of program's classes. Furthermore, maintenance of the free list is simple as the blocks are fixed size. Garbage is however guaranteed to build up as we always allocate memory at least the size of the record, thus in most cases we "over-allocate", which effectively leaves garbage in the memory as the space is allocated and never used.

```
TODO: Visualize
```

Shared heap, record-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 garbage build-up when allocating and freeing as the different free lists ensures that allocation of a record matches its optimal size, hence eliminating the "over-allocation" problem of the previous approach. 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.

```
TODO: Visualization
```

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

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

[4] 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.

[5] 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.