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 25th, 2018
Revision 1.01
### Revision History

| Revision | Date       | Author(s) | Description |
|----------|------------|-----------|-------------|
| 1.01     | 2018-04-11 | Martin    | Fixed incorrect subfigures (d, e, f) in Figure 4.11. Updated Figure 3.4 to better emphasize output copying. Updated Figure 3.5 to better reflect the harmless garbage flow through programs. |
Abstract

The reversible object-oriented programming language (ROOPL) was presented in late 2016 and proved that object-oriented programming paradigms works 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 fixed-sized arrays for increased language expressiveness. The language is a superset of ROOPL and has formally been defined by its language semantics, type system and computational universality. Considerations for reversible memory manager layouts are discussed and ultimately lead to the selection of the Buddy Memory layout. Translations of the extensions added in ROOPL++ to the reversible assembly language Pisa are presented to provide garbage-free computations. The dynamic memory management extension successfully increases the expressiveness of ROOPL and as a result, shows that non-trivial reversible data structures, such as binary trees and doubly-linked lists, are feasible and do not contradict the reversible computing paradigm.
Preface

This Master’s Thesis 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, presenting a 30 ECTS workload.

The thesis consists of 231 pages and a ZIP archive containing source code and test programs developed as part of the thesis work.

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 ROOP 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 8  
1.1 Reversible Computing 8  
1.2 Object-Oriented Programming 9  
1.3 Reversible Object-Oriented Programming 9  
1.4 Motivation 10  
1.5 Thesis Statement 10  
1.6 Outline 10

2 The ROOPL++ Language 11  
2.1 Syntax 12  
2.2 Object Instantiation 13  
2.3 Array Model 15  
2.4 Array Instantiation 15  
2.5 Referencing 17  
2.6 Local Blocks 18  
2.7 ROOPL++ Expressiveness 19  
2.7.1 Linked List 19  
2.7.2 Binary Tree 20  
2.7.3 Doubly Linked List 22  
2.8 Type System 25  
2.8.1 Preliminaries 25  
2.8.2 Expressions 27  
2.8.3 Statements 27  
2.8.4 Programs 30  
2.9 Language Semantics 31  
2.9.1 Preliminaries 31  
2.9.2 Expressions 31  
2.9.3 Statements 32  
2.9.4 Programs 36  
2.10 Program Inversion 37  
2.10.1 Invertibility of Statements 37  
2.10.2 Type-Safe Statement Inversion 38  
2.11 Computational Strength 41  
2.11.1 Reversible Turing Machines 41  
2.11.2 Tape Representation 42  
2.11.3 Reversible Turing Machine Simulation 43

3 Dynamic Memory Management 45  
3.1 Fragmentation 45  
3.1.1 Internal Fragmentation 46
# List of Figures

2.1 Example Roopl++ program implementing the Fibonacci function .................. 11
2.2 Syntax domains and EBNF grammar for Roopl++ ................................. 12
2.3 *construct/destruct*-blocks can be considered syntactic sugar ................ 18
2.4 Linked List cell class ................................................................. 19
2.5 Linked List class ........................................................................ 20
2.6 Binary Tree class ....................................................................... 21
2.7 Binary Tree node class (cont) ......................................................... 21
2.8 Binary Tree node class .................................................................. 22
2.9 Multiple identical reference are needed for a doubly linked list implementation . 22
2.10 Doubly Linked List class ................................................................. 23
2.11 Doubly Linked List Cell class ......................................................... 23
2.12 Doubly Linked List Cell class (cont) ................................................ 24
2.13 Definition *gen* for constructing the finite class map $\Gamma$ of a given program $p$, originally from [11] ............................................................... 26
2.14 Definition of fields and methods, originally from [11] ............................. 26
2.15 Definition *arrayType* for mapping types of arrays to either class types or the integer type .......................................................... 26
2.16 Typing rules for expressions in Roopl, originally from [11] ......................... 27
2.17 Typing rule extension for the Roopl typing rules ................................... 27
2.18 Typing rules for statements in Roopl, originally from [11] ......................... 28
2.19 Typing rules extensions for statements in Roopl++ ................................. 29
2.20 Typing rules for class methods, classes and programs, originally from [11] 30
2.21 Semantic values, originally from [11] .................................................. 31
2.22 Semantic inference rules for expressions, originally from [11] ..................... 31
2.23 Extension to the semantic inference rules for expression in Roopl++ ........... 32
2.24 Definition of binary expression operator evaluation, originally from [11] .... 32
2.25a Semantic inference rules for statements, modified from [11] .................... 33
2.25b Semantic inference rules for statements, modified from [11] (cont) ......... 34
2.25c Extension to the semantic inference rules for statements in Roopl++ .......... 35
2.26 Semantic inference rules for programs, originally from [11] ...................... 36
2.27 Roopl++ statement inverter, extended from [11] .................................... 37
2.28 Modified statement inverter for statements, originally from [11] ............... 38
2.29 Method for moving the tape head in the RTM simulation ......................... 43
2.30 Method for executing a single TM transition ....................................... 44
2.31 Main RTM simulation method ......................................................... 44

3.1a Creation of internal fragmentation of size $n - m$ due to over-allocation ....... 46
3.1b Creation of internal fragmentation of size $n - m \ mod \ n$ due to over-allocation .. 46
3.2 Example of external fragmentation caused for allocation and deallocation order . 47
3.3 Example of avoiding external fragmentation using allocation and deallocation order 47
3.4 The "garbage" output of an injective function \( f \) is the input to its inverse function \( f^{-1} \).

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

3.6 Memory layout using one heap per record size.

3.7 Memory layout using one heap per power-of-two.

3.8 Record size-specific free lists on a shared heap (powers of two).

3.9 Buddy Memory block allocation example.

4.1 Memory layout of a ROOPL program, originally from [11].

4.2 Illustration of object memory layout.

4.3 Memory layout of a ROOPL++ program.

4.4 Definition of pseudoinstructions \texttt{SUBI}, \texttt{PUSH} and \texttt{POP}, modified from [11].

4.5 Overall layout of a translated ROOPL++ program.

4.6 Dynamic dispatch approach for entering the allocation subroutine.

4.7 PISA translation of the nested conditionals in the Buddy Memory algorithm.

4.8 PISA translation of the inner \texttt{if-then} statement for the Buddy Memory algorithm.

4.9 PISA translation of the inner \texttt{else} statement for the Buddy Memory algorithm.

4.10 Non-opposite deallocation results in a different free list after termination.

4.11 PISA translation of the malloc procedure entry point of Buddy Memory algorithm.

4.12 PISA translation of heap allocation and deallocation for objects.

4.13 PISA translation of a ROOPL++ object block.

4.14 Illustration of object memory layout.

4.15 PISA translation of the reference copying and deletion statements.

4.16 Illustration of prefixing in the memory layout of two ROOPL++ arrays.

4.17 PISA translations of array allocation and deallocation statements.

4.18 Illustration of array memory storage layout.

4.19 Lines of code comparison between target and compiled ROOPL++ programs.
In recent years, technologies such as cloud-based services, deep learning, cryptocurrency mining and other services requiring large computational power and availability have 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 [6]. 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 [24].

Reversible computing presents a possible solution the problematic power consumption issues revolving around computations. Traditional, irreversible computers 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 [14].

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) [26, 3] and the BobISA architecture and instruction set [23] and high level languages Janus [16, 31, 29] and R [7] 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 model 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 result in a garbage-free computation, as garbage-generating functions simply can be unwinded to clean up.

Each step of a reversible program is locally invertible, meaning each step has exactly one inverse step. A reversible program can be inverted simply by computing the inverse of each of its steps, without any knowledge about the overall functionality or requirements of the program. 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 backwards.

The reversibility is however not free and comes and the cost of strictness when writing programs. Almost every popular, irreversible programming language features a conditional component in form of \texttt{if-else}-statements. In these languages, we only define the \textit{entry}-condition in the conditional, that is, the condition that determines which branch of the component we continue execution in. In reversible languages, we must also specify an \textit{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 [17].

### 1.3 Reversible Object-Oriented Programming

The high-level reversible language ROOPL (Reversible Object-Oriented Programming Language) was introduced in late 2016 [11, 12]. 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 \texttt{construct} / \texttt{deconstruct} blocks, which needs to be predefined, as the program call stack is required to be reset before program termination.

Conceptualizations and ideas for the \texttt{Joule} language was also published in 2016 [22]. 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

The block defined objects of ROOPL and lack of multiple references are problematic when writing 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 [18] and garbage collection [19] 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 exemplified by 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 were taken. In this chapter, we present ROOPL++, the natural successor to ROOPL, improving the object instantiation of the language 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 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.

```plaintext
1 class Fib
2   int[] xs
3
4 method init()
5   new int[2] xs
6
7 method fib(int n)
8   if n = 0 then
9      xs[0] ^= 1
10     xs[1] ^= 1
11   else
12      n -= 1
13      call fib(n)
14      xs[0] += xs[1]
15      xs[0] <=> xs[1]
16     fi
17
18 method get(int out)
19   out ^= xs[1]
20
21 class Program
22   int result
23   int n
24
25 method main()
26   n ^= 4
27
28 new Fib f
29   call f::init()
30   call f::fib(n)
31   call f::get(result)
32   uncall f::fib(n)
33   uncall f::init()
34   delete Fib f
```

Figure 2.1: Example ROOPL++ program implementing the Fibonacci 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 entry point of the program is a nullary main method, which is defined exactly once and is instantiated during program start-up. Fields of the main object will serve as output of the program, just as in ROOPL.

ROOPL++ Grammar

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

Syntax Domains

\[
\begin{align*}
\text{prog} & \in \text{Programs} \quad \text{s} \in \text{Statements} \quad \text{n} \in \text{Constants} \\
\text{cl} & \in \text{Classes} \quad \text{e} \in \text{Expressions} \quad \text{x} \in \text{VarIDs} \\
\text{t} & \in \text{Types} \quad \odot \in \text{ModOps} \quad \text{q} \in \text{MethodIDs} \\
\text{m} & \in \text{Methods} \quad \otimes \in \text{Operators} \quad \text{c} \in \text{ClassIDs}
\end{align*}
\]

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

The ROOPL++ grammar extends the grammar of ROOPL with a new static integer or class array type and a new object lifetime option in form of objects outside of blocks, using the \textbf{new} and
delete approach. Furthermore, the local block extension proposed in [11] has become a standard part of the language. Class definitions remains unchanged, and consists of a **class** keyword followed by a class name. Subclasses must be specified using the **inherits** keyword and a following parent class name. Classes can have any number of fields of any of the data types, including the new 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 (ˆ=) 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 is undefined. Variables and array elements of any type can be swapped using the $\langle\rangle$ operator as long as the variable is of same type as the array type. If an array is of a base class type, subclass variable values can be swapped in and out of the array, as long as the resulting value in the variable is still of the original subclass type.

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 depend 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 object model of ROOPL 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

\[
\text{construct } c \ x \quad s \quad \text{destruct } x
\]

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

\[
\text{new } c \ x \quad \ldots \quad \text{delete } c \ x
\]

could be considered a *dynamic* block, meaning we can have overlapping blocks. Compared to `construct/destruct` block consisting of a single statement, the `new/delete` block consist of two separate statements. 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 \( x \).

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

1. The reference \( x \) may be modified by swapping its internal field values with that of other references of the same type, but should be zero-cleared 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, as is a delete statement without a preceding `new` statement.

Note that variable scopes are always static, but object scopes can be either static (using `construct/destruct`) or dynamic (using `new/delete`).
2.3 Array Model

Besides asymmetric object lifetimes, ROOPL++ also introduces reversible, fixed-sized 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 [31].

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.

In ROOPL++, arrays expand upon the array model from JANUS. Arrays are indexed 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 dimension.

Array element accessing is accomplished using the bracket notation known from 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

\[
\text{new int}[e] \ x
\]

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

1. The expression \(e\) is evaluated
2. Memory equal to the integer value that \(e\) evaluates to and an additional small amount of 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 fixed-sized 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.

```cpp
define int[5] intArray // Init new integer array
define Foo[2] fooArray // init new Foo array
intArray[1] += 10 // Legal array integer assignment
intArray[1] -= 10 // Legal Zero-clearing for integer array cells
new Foo fooObject
fooArray[0] <=> fooObject // Legal object array cell assignment
new Foo fooArray[2] // Legal object array cell assignment
... // Clear all array cells
delete Foo fooArray[0] // Legal object array cell zero-clearing
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

```
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. The value of the expression `e` is evaluated and used to reclaim the allocated memory space.

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

```
x[5] += x[5] + 1
```

However, we do allow other initialized, non-zero-cleared array elements from the same array or arrays of same type to be referenced in the right hand side of the statement. As with regular assignment, we still prohibit the left side reference to occur in the ride side, meaning the following statements are also prohibited

```
x += y[x]
y[x] += x
```

Chapter 2  The ROOPL++ Language
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

```
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

```
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 do not require all fields or cells to be zero-cleared (as they are simple 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 do 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 [29] consisted of a local variable allocation, a statement and a local variable deallocation. These local variable blocks add immense programmer usability as they 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 language's variable types; integers, arrays and user-defined types in the form of objects.

Local integer blocks work 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 essentially becomes a reference copy, analogous to a variable initialized from a copy statement.

\[
\begin{align*}
\text{construct } c_x & \quad s \quad \text{destruct } x \quad \text{def} \\
\text{local } c_x &= \text{nil} \\
\text{new } c_x & \quad s \quad \text{delete } c_x \\
\text{delocal } c_x &= \text{nil}
\end{align*}
\]

Figure 2.3: construct/destruct-blocks can be considered syntactic sugar

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 shown in figure 2.3.

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 the 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 reversible programs. 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 [11]. The implementation featured a ListBuilder and a Sum class, required to determine and retain the sum of a constructed linked list as the statically scoped object blocks of ROOPL would deallocate automatically after building the full list. In ROOPL++, we do not face the same challenges and the implementation becomes much more straightforward. Figure 2.5 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.

```
  class Cell
    Cell next
    int data

    method constructor(int value)
      data ^= value

    method append(Cell cell)
      if next = nil & cell != nil then
        next <=> cell // Store as next cell if current cell is end of list
      else skip
      fi

      if next != nil then
        call next::append(cell) // Recursively search until we reach end of list
      else skip
      fi

      next != nil
```

Figure 2.4: Linked List cell class

Figure 2.4 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
```java
class LinkedList
    Cell head
    int listLength

    method insertHead(Cell cell)
        if head = nil & cell != nil then
            head <=> cell // Set cell as head of list if list is empty
        else skip
        fi
        if head != nil & cell = nil
            head = nil
        fi

    method appendCell(Cell cell)
        call insertHead(cell) // Insert as head if empty list
        if head != nil then
            cell head::append(cell) // Iterate until we hit end of list
        else skip
        fi
        listLength += 1 // Increment length

    method prependCell(Cell cell)
        call insertHead(cell) // Insert as head if empty list
        if cell != nil & head != nil then
            cell head::append(head) // Set cell.next = head. head = nil after execution
        else skip
        fi
        if cell != nil & head = nil then
            cell <=> head // Set head = cell. Cell is nil after execution
        else skip
        fi
        cell = nil & head != nil
        listLength += 1 // Increment length

    method length(int result)
        result ^= listLength
```

Figure 2.5: 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 2.6, 2.8 and 2.7 shows the implementation of a binary tree in form of a rooted, unbalanced, min-heap. The `Tree` class shown in figure 2.6 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 2.8 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.

```plaintext
class Tree
  Node root

  method insertNode(Node node, int value)
    if root = nil & node != nil then
      root <=> node
    else skip
    fi
    root != nil & node = nil
  end
  if root != nil then
    call root::insertNode(node, value)
  else skip
  fi

  method getSum(int result)
    result += value // Add the value of this node to the sum
    if left != nil then
      call left::getSum(result) // If we have a left child, follow that path
    else skip
    fi
    left != nil
  end
    if right != nil then
      call right::getSum(result) // If we have a right child, follow that path
    else skip
    fi
    right != nil
  end

  method mirror()
    left <=> right // Swap left and right children
    if left = nil then skip
    else call left::mirror() // Recursively swap children if left != nil
    fi
    left = nil
  end
    if right = nil then skip
    else call right::mirror() // Recursively swap children if right != nil
    fi
    right = nil
  end
```

Figure 2.6: 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 2.7.

```plaintext
method getSum(int result)
  result += value // Add the value of this node to the sum
  if left != nil then
    call left::getSum(result) // If we have a left child, follow that path
  else skip
  fi
  left != nil
  if right != nil then
    call right::getSum(result) // If we have a right child, follow that path
  else skip
  fi
  right != nil
end

method mirror()
  left <=> right // Swap left and right children
  if left = nil then skip
  else call left::mirror() // Recursively swap children if left != nil
  fi
  left = nil
  if right = nil then skip
  else call right::mirror() // Recursively swap children if right != nil
  fi
  right = nil
end
```

Figure 2.7: Binary Tree node class (cont)
class Node
    Node left
    Node right
    int value

    method setValue(int newValue)
        value ^= newValue
    end

    method insertNode(Node node, int nodeValue)
        // Determine if we insert left or right
        if nodeValue < value then
            if nodeValue < value then
                if node = nil & node != nil then
                    // If open left node, store here
                    left <=> node
                else skip
                fi
            fi
            else
                if right = nil & node != nil then
                    // If open right node spot, store here
                    right <=> node
                else skip
                fi
            fi
        else
            if right = nil & node != nil then
                // If open right node spot, store here
                right <=> node
            else skip
            fi
        fi
    fi
end

Figure 2.8: Binary Tree node class

2.7.3 Doubly Linked List

Finally, we present the reversible doubly linked list, shown in figures 2.10-2.12. 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.

![Diagram of a doubly linked list](image)

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

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

```
class DoublyLinkedList
  Cell head
  int length

  method appendCell(Cell cell)
    if head = nil & cell != nil then
      head <=> cell
    else skip
    fi
    if head != nil then
      call head::append(cell)
    else skip
    fi
    head != nil
  fi
  length += 1

Figure 2.10: Doubly Linked List class
```

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.

```
class Cell
  int data
  int index
  Cell left
  Cell right
  Cell self

  method setData(int value)
    data ^= value
  end

  method setIndex(int i)
    index ^= i
  end

  method setLeft(Cell cell)
    left <=> cell
  end

  method setRight(Cell cell)
    right <=> cell
  end

  method setSelf(Cell cell)
    self <=> cell
  end

Figure 2.11: Doubly Linked List Cell class
```

The data structure could relatively easily be extended to work as a dynamic array. Currently each cell contains an index field, specifying their 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
method append(Cell cell)
if right = nil & cell != nil then // If current cell does not have a right neighbour
    right <=> cell // Set new cell as right neighbour of current cell
    local Cell selfCopy = nil
    copy Cell self selfCopy // Copy reference to current cell
    call right::setLeft(selfCopy) // Set current as left of right neighbour
    delocal Cell selfCopy = nil
    local int cellIndex = index + 1
    call right::setIndex(cellIndex) // Set index in right neighbour of current
    delocal int cellIndex = index + 1
else skip
    if right != nil then
        call right::append(cell) // Keep searching for empty right neighbour
    else skip
fi right != nil & cell = nil
if right != nil then
    call right::append(cell) // Keep searching for empty right neighbour
else skip
fi right != nil

Figure 2.12: Doubly Linked List Cell class (cont)

the correct position, in such a way that the old cell at index n now is the new right neighbour of
the cell 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
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 [11] and is analogously described by syntax-directed inference typing rules in the style of Winskel [28]. 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:

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

The type environment \( \Pi \) is a finite map pairing 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 \( \Pi' \) 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 \( \text{vars} : \text{Expressions} \rightarrow \text{VarIDs} \) is described by the following definition:

\[
\begin{align*}
\text{vars}(\text{nil}) & = \emptyset \\
\text{vars}(\text{nil}) & = \emptyset \\
\text{vars}(x) & = \{ x \} \\
\text{vars}(x[e]) & = \{ x \} \cup \text{vars}(e) \\
\text{vars}(e_1 \otimes e_2) & = \text{vars}(e_1) \cup \text{vars}(e_2).
\end{align*}
\]

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

\[
\begin{align*}
c_1 \prec: c_2 & \quad \text{if } c_1 \text{ inherits from } c_2 \\
c \prec: c & \quad (\text{reflexivity}) \\
c_1 \prec: c_3 & \quad \text{if } c_1 \prec: c_2 \text{ and } c_2 \prec: c_3 \quad (\text{transitivity})
\end{align*}
\]

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 \( \Gamma \) to be the class map of a program \( p \), such that \( \Gamma \) is a finite map from class identifiers to tuples of methods and fields for the class \( p \). Application of a class map \( \Gamma \) to some class \( cl \) is denoted as \( \Gamma(cl) \). Construction of a class map is done through function \( \text{gen} \), as shown in figure 2.13. Figure 2.14 defines the \( \text{fields} \) and \( \text{methods} \) 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 2.13 and 2.14 are originally from [11].
\[ \text{gen}(p_{c_1, \ldots, c_n}) = \left[ \alpha(c_1) \mapsto \beta(c_1), \ldots, \alpha(c_n) \mapsto \beta(c_n) \right] \]

\[ \alpha(\text{class } c \cdots) = c \quad \beta(\text{cl}) = \left( \text{fields}(\text{cl}), \text{methods}(\text{cl}) \right) \]

**Figure 2.13:** Definition \( \text{gen} \) for constructing the finite class map \( \Gamma \) of a given program \( p \), originally from [11].

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

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

\[ A \uplus B \overset{\text{def}}{=} A \cup \left\{ m \in B \mid \exists m' \left( \zeta(m') = \zeta(m) \land m' \in A \right) \right\} \]

\[ \zeta\left( \text{method } q \left( \cdots \right) s \right) = q \quad \eta\left( \text{class } c \cdots t_1 f_1 \cdots t_n f_n \cdots \right) = f s \]

\[ \delta\left( \text{class } c \cdots \text{method } q_1 \left( \cdots \right) s_1 \cdots \text{method } q_n \left( \cdots \right) s_n \cdots \right) = m s \]

**Figure 2.14:** Definition of fields and methods, originally from [11].

Finally, we formally define a link between arrays of a given type and other types. The function \( \text{arrayType} \), defined in figure 2.15, is \( c \) if the passed array \( a \) is an array of class \( c \) instances.

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

**Figure 2.15:** Definition \( \text{arrayType} \) for mapping types of arrays to either class types or the integer type.
2.8.2 Expressions

The type judgment

\[ \Pi \vdash_{expr} e : \tau \]

defines the type of expressions. The judgment reads as: under type environment \( \Pi \), expression \( e \) has type \( \tau \).

- \( \Pi \vdash_{expr} n : \text{int} \) \hspace{2em} T-CON
- \( \Pi(x) = \tau \) \hspace{2em} T-VAR
- \( \tau \neq \text{int} \) \hspace{2em} T-NIL

\( \Pi \vdash_{expr} e_1 : \text{int} \quad \Pi \vdash_{expr} e_2 : \text{int} \)
\[ \Pi \vdash_{expr} e_1 \otimes e_2 : \text{int} \] \hspace{2em} T-BINOPINT

\( \Pi \vdash_{expr} e_1 : c \quad \Pi \vdash_{expr} e_2 : c \quad \otimes \in \{=, \neq\} \)
\[ \Pi \vdash_{expr} e_1 \otimes e_2 : \text{int} \] \hspace{2em} T-BINOPOBJ

Figure 2.16: Typing rules for expressions in ROOPL, originally from [11]

The original expression typing rules from ROOPL are shown in figure 2.16. The type rules T-CON, T-VAR and T-NIL defines typing of the simplest expressions. Numeric literals are of type \text{int}, typing of variable expressions depends on the type of the variable in the type environment and the \text{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-ARELEMVAR which defines typing for array element variables, shown in figure 2.17.

\( \Pi(x) = \tau[\ ] \quad \Pi \vdash_{expr} e : \text{int} \)
\[ \Pi \vdash_{expr} x[e] : \tau \] \hspace{2em} T-ARELEMVAR

Figure 2.17: Typing rule extension for the ROOPL typing rules

2.8.3 Statements

The type judgment

\[ (\Pi, c) \vdash_{stmt} s \]

defines well-typed statements. The judgment reads as under type environment \( \Pi \) within class \( c \), statement \( s \) is well-typed with class map \( \Gamma \).
As with Janus, where entry and exit conditions are integers and branch and loop statements are well-typed, the type rules 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 \( \Pi \).

As with ROOPL, type correctness of local method invocation is defined in rule T-Call iff:

\[
\frac{\Gamma(x) = \Pi(x_1) \land \cdots \land \Pi(x_n) \land \Pi(\Pi(c)) = \left(\text{fields, methods}\right) \left(\text{method } q(t_1 y_1, \ldots, t_n y_n) s\right) \in \text{methods}\}}{
\frac{\{x_1, \ldots, x_n\} \cap \text{fields} = \emptyset \quad i \neq j \implies x_i \neq x_j 
\Pi(x_1) \prec t_1 \cdots \Pi(x_n) \prec t_n}{\Gamma(c) \mid \Gamma_{\text{stmt}} \quad \Gamma_{\text{stmt}} s_1 s_2 \quad \Gamma_{\text{stmt}} s_1 \quad \Pi_{\text{stmt}} s_2 \quad \Pi_{\text{stmt}} s_1 \quad \Pi_{\text{stmt}} s_2}}\]
• 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.

\[
\Pi(x) = \text{int}[\ ] \quad \Pi \vdash_{\text{expr}} e_1 : \text{int} \\
\left( x \cup \text{vars}(e_1) \right) \cap \text{vars}(e_2) = \emptyset \quad \Pi \vdash_{\text{expr}} e_2 : \text{int} \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ x[e_1] \circ = e_2 \quad \text{T-ARRElemAss}
\]

\[
\Pi(x) = c' \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{new} \ c' \ x \quad \text{T-OBJNEW}
\]

\[
\Pi(x) = c' \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{delete} \ c' \ x \quad \text{T-OBJDLT}
\]

\[
\text{arrayType}(a) \in \{ \text{classIDs}, \text{int} \} \quad \Pi \vdash_{\text{expr}} \ e = \text{int} \quad \Pi(x) = a[\ ] \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{new} \ a[e] \ x \quad \text{T-ARRNEW}
\]

\[
\text{arrayType}(a) \in \{ \text{classIDs}, \text{int} \} \quad \Pi \vdash_{\text{expr}} \ e = \text{int} \quad \Pi(x) = a[\ ] \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{delete} \ a[e] \ x \quad \text{T-ARRDLT}
\]

\[
\Pi(x) = c' \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{copy} \ c' \ x \ x' \quad \text{T-CP}
\]

\[
\Pi(x) = c' \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{uncopy} \ c' \ x \ x' \quad \text{T-UCP}
\]

\[
\langle \Pi, c \rangle \vdash_{\text{expr}} \ e_1 : c' \\
\langle \Pi[x \mapsto c'], c \rangle \vdash_{\text{stmt}}^{\Gamma} s \\
\langle \Pi, c \rangle \vdash_{\text{expr}} \ e_2 : c' \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{local} \ c' \ x = e_1 \\
\langle \Pi, c \rangle \vdash_{\text{stmt}}^{\Gamma} \ \text{delocal} \ c' \ x = e_2 \quad \text{T-LOCALBLOCK}
\]

Figure 2.19: Typing rules extensions for statements in ROOPL++

Figure 2.19 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 \text{int}, the variable \( x[e_1] \) is not present in the right-hand
side of the statement, no variables in \( e_1 \) exist in \( e_2 \) and both expressions \( e_1 \) and \( e_2 \) evaluates to integers.

The T-ObjNew and T-ObjDlt rules define well-typed \textbf{new} and \textbf{delete} statements for dynamically lifetimed objects. The \textbf{new} statement is well-typed, as long as \( c' \in \text{classIDs} \) and the variable \( x \) is of type \( c' \) under type environment \( \Pi \) and \textbf{delete} is also well-typed if the type of \( x \) under type environment \( \Pi \) is equal to \( c' \).

The T-ArrNew and T-ArrDlt rules define well-type \textbf{new} and \textbf{delete} statement for ROOPL++ arrays. The \textbf{new} statement is well-typed, if the type of the array either is a classID or \texttt{int}, the length expression evaluates to an integer and \( x \) is of of type \( a[ ] \) under the type environment \( \Pi \), and \textbf{delete} is well-typed if the type of the array is either a classID or \texttt{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 \textbf{copy} or \textbf{uncopy} statement requires that the types of \( x \) and \( x' \) both are \( c' \) under type environment \( \Pi \).

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 method. Figure 2.20 shows the typing rules for class methods, classes and programs.

\[
\langle \Pi[x_1 \mapsto t_1, ..., x_n \mapsto t_n], c \rangle \vdash^\Gamma_{stmt} s \quad \text{T-METHOD}
\]

\[
\langle \Pi, c \rangle \vdash^\Gamma_{meth} \text{method } q(t_1x_1, ..., t_nx_n) s \quad \text{T-METHOD}
\]

\[
\Pi = [f_1 \mapsto t_1, ..., f_n \mapsto t_n] \langle \Pi, c \rangle \vdash^\Gamma_{meth} m_1 \cdots \langle \Pi, c \rangle \vdash^\Gamma_{meth} m_n \quad \text{T-CLASS}
\]

\[
\left( \text{method main ( ) } s \right) \in \bigcup_{i=1}^n \text{methods}(c_i)
\]

\[
\Gamma = \text{gen}(c_1, ..., c_n) \vdash^\Gamma_{class} c_1 \cdots \vdash^\Gamma_{class} c_n \quad \text{T-CLASS}
\]

\[
\vdash^\Gamma_{prog} c_1 \cdots c_n \quad \text{T-PROG}
\]

\textbf{Figure 2.20}: Typing rules for class methods, classes and programs, originally from [11]
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 \( l \) to be a location. We define a location for integer variables to bind to a single location in program memory and a vector of memory locations for object and array variables, where the vector is the size of the object or array. A memory location is in the set of non-negative integers, \( \mathbb{N}_0 \). An environment \( \gamma \) is a partial function mapping variables to memory locations. A store \( \mu \) 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, a location or a vector of locations.

Applications of environments \( \gamma \) and stores \( \mu \) are analogous to the type environment \( \Gamma \), defined in section 2.8.1.

\[
\begin{align*}
l & \in \text{Locs} = \mathbb{N}_0 \\
\gamma & \in \text{Envs} = \text{VarIDs} \rightarrow \text{Locs} \\
\mu & \in \text{Stores} = \text{Locs} \rightarrow \text{Values} \\
\text{Objects} & = \left\{ (c_f, \gamma_f) \mid c_f \in \text{ClassIDs} \land \gamma_f \in \text{Envs} \right\} \\
v & \in \text{Values} = \mathbb{Z} \cup \text{Objects} \cup \text{Locs} \cup \left[ \text{Locs} \right]
\end{align*}
\]

Figure 2.21: Semantic values, originally from [11]

2.9.2 Expressions

The judgment:

\[
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e \Rightarrow v
\]

defines the meaning of expressions. We say that under environment \( \gamma \) and store \( \mu \), expression \( e \) evaluates to value \( v \).

\[
\begin{align*}
\langle \gamma, \mu \rangle \vdash_{\text{expr}} n & \Rightarrow n \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} x & \Rightarrow \mu(\gamma(x)) \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} \text{nil} & \Rightarrow 0 \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e_1 & \Rightarrow v_1 \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e_2 & \Rightarrow v_2 \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e_1 \otimes e_2 & \Rightarrow v
\end{align*}
\]

Figure 2.22: Semantic inference rules for expressions, originally from [11]
As shown in figure 2.22, expression evaluation has no effects on the store. Logical values are represented by \textit{truthy} and \textit{falsy} values of any non-zero value and zero respectively. The evaluation of binary operators is presented in figure 2.24.

\[
\frac{\langle \gamma, \mu \rangle \vdash_{\text{expr}} e \Rightarrow v \quad \gamma(x) = l \quad \mu(l)[v] = l' \quad \mu(l') = w}{\langle \gamma, \mu \rangle \vdash_{\text{expr}} x[e] \Rightarrow w} \quad \text{ARRELEM}
\]

\textbf{Figure 2.23:} Extension to the semantic inference rules for expression in \textsc{Roopl++}

For \textsc{Roopl++}, we extend the expression ruleset with a single rule for array element variables shown in figure 2.23. As with the expressions inference rules in \textsc{Roopl}, this extension has no effect on the store.

\[
\begin{align*}
[+](v_1, v_2) &= v_1 + v_2 & [\%](v_1, v_2) &= v_1 \mod v_2 \\
[-](v_1, v_2) &= v_1 - v_2 & [\&](v_1, v_2) &= v_1 \land v_2 \ , \text{bitwise} \\
[*](v_1, v_2) &= v_1 \times v_2 & [\|](v_1, v_2) &= v_1 \lor v_2 \ , \text{bitwise} \\
[/](v_1, v_2) &= v_1 \div v_2 & [\hat{\hat{\&}}](v_1, v_2) &= v_1 \oplus v_2 \\
[\&\&](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = 0 \lor v_2 = 0 \\ 1 & \text{otherwise} \end{cases} & [<=](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \leq v_2 \\ 1 & \text{otherwise} \end{cases} \\
[\|\|](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 = 0 \\ 1 & \text{otherwise} \end{cases} & [>=](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \geq v_2 \\ 1 & \text{otherwise} \end{cases} \\
[<](v_1, v_2) &= \begin{cases} 1 & \text{if } v_1 < v_2 \\ 0 & \text{otherwise} \end{cases} & [=](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 = v_2 \\ 1 & \text{otherwise} \end{cases} \\
[>](v_1, v_2) &= \begin{cases} 1 & \text{if } v_1 > v_2 \\ 0 & \text{otherwise} \end{cases} & [\neq](v_1, v_2) &= \begin{cases} 0 & \text{if } v_1 \neq v_2 \\ 1 & \text{otherwise} \end{cases}
\end{align*}
\]

\textbf{Figure 2.24:} Definition of binary expression operator evaluation, originally from [11]

\section*{2.9.3 Statements}

The judgment

\[
\gamma \vdash_{\text{stmt}}^\Gamma s : \mu \Rightarrow \mu'
\]

defines the meaning of statements. We say that under environment \(\gamma\), statement \(s\) with class map \(\Gamma\) reversibly transforms store \(\mu\) to store \(\mu'\). Figure 2.25a, 2.25b and 2.25c defines the operational semantics of \textsc{Roopl++}.

The following semantic rules have been simplified from the original \textsc{Roopl} semantics [11] to better accommodate the extended language.

The inference rule \textbf{Skip} defines the operational semantics of \textbf{skip} statements and has no effects on the store \(\mu\).
\[
\begin{align*}
\gamma \vdash_{\text{stmt}} \text{skip} & : \mu \Rightarrow \mu \\
\gamma \vdash_{\text{stmt}} s_1 : \mu \Rightarrow \mu' & \quad \gamma \vdash_{\text{stmt}} s_2 : \mu' \Rightarrow \mu'' \quad \Rightarrow \quad \text{SEQ} \\
\langle \gamma, \mu \rangle \vdash_{\text{stmt}} e \Rightarrow v & \quad \langle \mathcal{O} \rangle (\mu(\gamma(x)), v) = v' \\
\gamma \vdash_{\text{stmt}} x \odot = e : \mu \Rightarrow \mu[\gamma(x) \mapsto v'] \quad \Rightarrow \quad \text{AssVar} \\
\mu(\gamma(x_1)) = v_1 & \quad \mu(\gamma(x_2)) = v_2 \\
\gamma \vdash_{\text{stmt}} x_1 \Leftrightarrow x_2 : \mu \Rightarrow \mu[\gamma(x_1) \mapsto v_2, \gamma(x_2) \mapsto v_1] \quad \Rightarrow \quad \text{SwpVar} \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e_1 \neq 0 & \quad \gamma \vdash_{\text{stmt}} s_1 : \mu \Rightarrow \mu' \quad \gamma \vdash_{\text{loop}} (e_1, s_1, s_2, e_2) : \mu' \Rightarrow \mu'' \quad \Rightarrow \quad \text{LOOPMain} \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e_2 \Rightarrow 0 & \quad \gamma \vdash_{\text{loop}} (e_1, s_1, s_2, e_2) : \mu \Rightarrow \mu \\
\langle \gamma, \mu' \rangle \vdash_{\text{expr}} e_1 \Rightarrow 0 & \quad \gamma \vdash_{\text{stmt}} s_1 : \mu' \Rightarrow \mu'' \quad \gamma \vdash_{\text{loop}} (e_1, s_1, s_2, e_2) : \mu'' \Rightarrow \mu''' \quad \Rightarrow \quad \text{LOOPRec} \\
\langle \gamma, \mu \rangle \vdash_{\text{expr}} e_1 \neq 0 & \quad \gamma \vdash_{\text{stmt}} s_1 : \mu \Rightarrow \mu' \quad \langle \gamma, \mu' \rangle \vdash_{\text{expr}} e_2 \neq 0 \quad \Rightarrow \quad \text{IFTrue} \\
\langle \gamma, \mu \rangle \vdash_{\text{stmt}} s_1 : \mu \Rightarrow \mu' & \quad \langle \gamma, \mu' \rangle \vdash_{\text{expr}} e_2 \Rightarrow 0 \quad \Rightarrow \quad \text{IFFalse} \\
\end{align*}
\]

Figure 2.25a: Semantic inference rules for statements, modified from [11]

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 \(\gamma\) is mapped to the value \(v'\) resulting in an updated store \(\mu'\). For variable swapping SwpVar defines how value mappings between two variables are exchanged in the updated store.

For loops and conditionals, Rules LOOPMAIN, LOOPBASE and LOOPREC define the meaning of loop statements and IfTrue and IFFalse, similarly to the operational semantics of Janus, as
\[
\gamma(\text{this}) = l \quad \mu(l) = l' \quad \mu(l') = \left\langle c, (l_1, \ldots, l_m) \right\rangle \quad \gamma(y_i) = l_i'
\]

\[
\Gamma(c) = \left\langle \left( x_1, \ldots, x_m \right), \left( \ldots, \text{method } q(t_1z_1, \ldots, t_i z_k) \ s, \ldots \right) \right\rangle
\]

\[
\gamma' = [\text{this} \mapsto l, x_1 \mapsto l_1, \ldots, l_m \mapsto v, z_1 \mapsto l'_1, \ldots, z_k \mapsto l'_k]
\]

\[
\gamma' \vdash^\Gamma_{\text{stmt}} \ s : \mu \equiv \mu'
\]

**CALL**

\[
\gamma \vdash^\Gamma_{\text{stmt}} \text{call } q(y_1, \ldots, y_n) : \mu' \equiv \mu
\]

**UNCALL**

\[
\gamma \vdash^\Gamma_{\text{stmt}} \text{uncall } q(y_1, \ldots, y_n) : \mu \equiv \mu'
\]

\[
\gamma(x_0) = l \quad \mu(l) = l' \quad \mu(l') = \left\langle c, (l_1, \ldots, l_m) \right\rangle \quad \gamma(y_i) = l_i'
\]

\[
\Gamma(c) = \left\langle \left( x_1, \ldots, x_m \right), \left( \ldots, \text{method } q(t_1z_1, \ldots, t_i z_k) \ s, \ldots \right) \right\rangle
\]

\[
\gamma' = [\text{this} \mapsto l, x_1 \mapsto l_1, \ldots, l_m \mapsto v, z_1 \mapsto l'_1, \ldots, z_k \mapsto l'_k]
\]

\[
\gamma' \vdash^\Gamma_{\text{stmt}} \ s : \mu \equiv \mu'
\]

**CALLOBJ**

\[
\gamma \vdash^\Gamma_{\text{stmt}} \text{call } x_0 :: q(y_1, \ldots, y_n) : \mu' \equiv \mu
\]

**OBJUNCALL**

\[
\gamma \vdash^\Gamma_{\text{stmt}} \text{uncall } x_0 :: q(y_1, \ldots, y_n) : \mu \equiv \mu'
\]

\[
\gamma(\text{this}) = l \quad \mu(l) = l' \quad \mu(l') = \left\langle c, (l_1, \ldots, l_m) \right\rangle \quad \gamma(y_i) = l_i'
\]

\[
\Gamma(c) = \left\langle \left( x_1, \ldots, x_m \right), \text{methods} \right\rangle
\]

\[
\gamma' = \gamma[x \mapsto l_0] \quad l_0 \notin \text{dom}(\mu) \quad l_0 \notin \text{dom}(\mu)
\]

\[
\mu' = \mu \left[ \gamma'(x) \mapsto l_0, 0 \mapsto \left\langle c, (l_1, \ldots, 1_m) \right\rangle, l_1 \mapsto 0, \ldots, l_m \mapsto 0 \right]
\]

\[
\gamma' \vdash^\Gamma_{\text{stmt}} \ s : \mu' \equiv \mu''
\]

\[
\mu'' = \mu''' \left[ \gamma'(x) \mapsto l_0, 0 \mapsto \left\langle c, (l_1, \ldots, 1_m) \right\rangle, l_1 \mapsto 0, \ldots, l_m \mapsto 0 \right]
\]

**OBJBLOCK**

\[
\gamma \vdash^\Gamma_{\text{stmt}} \text{construct } c \ x \quad s \quad \text{destruct } x : \mu \equiv \mu''
\]

Figure 2.25b: Semantic inference rules for statements, modified from [11] (cont)

presented in [29]. LOOPMAIN is entered if \(e_1\) is true and each iteration enters LOOPREC until \(e_2\) is false, in which case 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.

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 \(\mu'\) 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...
The statically scoped object blocks are defined in rule \texttt{ObjBlock}. The operation semantics of these blocks are similar to \texttt{local}-blocks from \textsc{Janus}. We add the reference \( x \) to a new environment and afterwards map the location of \( x \) to the object tuple at location \( l_0 \), containing the locations of all object fields, all of which, along with \( l_0 \), must be unused in \( \mu \). The result store \( \mu'' \) is obtained after executing the body statement \( s \) in store \( \mu' \) mapping \( x \) to object reference at \( l_0 \), as long as all object fields are zero-cleared in \( \mu'' \) afterwards. If any of these conditions fail, the object block statement is undefined.

Figure 2.25c shows the extensions to the semantics of ROOPL++ with rules for \texttt{new}/\texttt{delete} and 

\texttt{uncall} statement, similarly to techniques presented in [31, 29].

\[ \langle \gamma, \mu \rangle \vdash_{\text{stmt}} e_1 \Rightarrow v_1 \quad \langle \gamma, \mu \rangle \vdash_{\text{stmt}} e_2 \Rightarrow v_2 \]

\[ \gamma(x) = l \quad \mu(l)[v_1] = l' \quad \mu(l') = w \quad \lbrack \textcircled{\circ} \rbrack(w, v_2) = w' \]

\[ \gamma \vdash_{\text{stmt}} x[e_1] \circ= e_2 : \mu \Rightarrow \mu[l' \mapsto w'] \]

\textsc{AssArrElemVar}

\[ \Gamma(e) = \langle (x_1, \ldots, x_m), \text{methods} \rangle \]

\[ \gamma(\text{fields}) = l \quad l_0 \notin \text{dom}(\mu) \quad \ldots \quad l_m \notin \text{dom}(\mu) \]

\[ \gamma \vdash_{\text{stmt}} \text{new} \ c \ x : \mu[l \mapsto 0] \Rightarrow \mu[l \mapsto (c, (l_1, \ldots, l_m)), l_1 \mapsto 0, \ldots, l_m \mapsto 0] \]

\textsc{ObjNew}

\[ \langle l, \gamma \rangle \vdash_{\text{stmt}} \text{new} \ c \ x : \mu' \Rightarrow \mu \]

\[ \gamma \vdash_{\text{stmt}} \text{delete} \ c \ x : \mu \Rightarrow \mu' \]

\textsc{ObjDelete}

\[ \langle l, \gamma \rangle \vdash_{\text{stmt}} \text{new} \ a[e] \ x : \mu' \Rightarrow \mu \]

\[ \gamma \vdash_{\text{stmt}} \text{delete} \ a[e] \ x : \mu' \Rightarrow \mu' \]

\[ \gamma \vdash_{\text{stmt}} \text{copy} \ c \ x \ x' : \mu[l' \mapsto 0] \Rightarrow \mu[l' \mapsto v] \]

\[ \gamma \vdash_{\text{stmt}} \text{copy} \ c \ x \ x' : \mu' \Rightarrow \mu \]

\textsc{Copy}

\[ \gamma \vdash_{\text{stmt}} \text{uncopy} \ c \ x \ x' : \mu \Rightarrow \mu' \]

\textsc{Uncopy}

\[ \langle l, \gamma \rangle \vdash_{\text{stmt}} e_1 \Rightarrow v_1 \quad \langle l, \gamma \rangle \vdash_{\text{stmt}} e_2 \Rightarrow v_2 \]

\[ r \notin \text{dom}(\mu) \]

\[ \gamma[x \mapsto r] \vdash_{\text{stmt}} s : \mu[r \mapsto v_1] \Rightarrow \mu[r \mapsto v_2] \]

\[ \gamma \vdash_{\text{stmt}} \text{local} \ c \ x = e_1 \quad \text{delocal} \ x = e_2 : \mu \Rightarrow \mu' \]

\textsc{LocalBlock}
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 $v_2$, the value at the location of variable $x[v_1]$ under environment $\gamma$ is mapped to the value $v_3$ resulting in an updated store $\mu'$.

Dynamic object construction and destruction is defined by rules ObjNew and ObjDelete. For construction, $x$ must be bound to a location $l$. We then make location $l$ point to a new pair consisting of the class name and a vector of $m$ new locations mapping object fields to locations. For destruction, $x$ is still bound to $l$ return $l$ to a null pointer. 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 location to a vector of size equal to the evaluated expression result. For deletion, we return the location of $x$ to a null pointer and remove the binding to the vector from the store.

Local blocks are as previously mentioned, semantically similar to object blocks, where the memory location of variable $x$ is mapped to an unused reference $r$ in the store $\mu$. Before body statement execution, we let $r$ bind to the evaluated value of $e_1$, $v_1$. The result store after body statement execution, $\mu'$ must have $r$ mapped to the expression value of $e_2$, $v_2$. $r$ is then zero-cleared using the value of expression evaluation and becomes 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 $\sigma$ 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.
While the invertibility of statements remains untouched by the extensions made in Chapter 2, the following proof, originally presented in [11], has been included for completeness.

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 [11], which maps statements to their inverse counterparts. Figure 2.27 shows the statement inverter, extended with the new ROOPL++ statements for construction/destruction and referencing copying/copy removal.

![Figure 2.27: ROOPL++ statement inverter, extended from [11]](image)

Program inversion is conducted by recursive descent over components and statements. A proposed extension to the statement inverter for whole-program inversion is retained in the ROOPL++ statement inverter. The extension covers a case that reveals itself during method calling. As a method call is equivalent to an uncall with the inverse method we simply change calls to uncalls during inversion, the inversion of the method body cancels out. The proposed extension, presented in [31, 11], simply avoids inversion of calls and uncalls, as shown in figure 2.28.

2.10.1 Invertibility of Statements

While the invertibility of statements remains untouched by the extensions made in ROOPL++, the following proof, originally presented in [11], has been included for completeness.
\[ T'[\text{call } q(\ldots)] = \text{call } q(\ldots) \]

\[ T'[\text{uncall } q(\ldots)] = \text{uncall } q(\ldots) \]

\[ T'[s] = T[s] \]

Figure 2.28: Modified statement inverter for statements, originally from [11]

If execution of a statement \( s \) in store \( \mu \) yields \( \mu' \), then execution of the inverse statement, \( T[s] \) in store \( \mu' \) should yield \( \mu \). Theorem 2.1 shows that \( T \) is a statement inverter.

**Theorem 2.1. (Invertibility of statements, originally from [11])**

\[
\begin{align*}
\langle l, \gamma \rangle & \vdash_{\text{stmt}} s : \mu \iff \langle l, \gamma \rangle & \vdash_{\text{stmt}} T[s] : \mu' \implies \mu
\end{align*}
\]

*Proof.* By structural induction on the semantic derivation of \( S \) (omitted). It suffices to show that \( S \implies S' \), as this can serve as proof of \( S' \implies S \), as \( T \) is an involution.

### 2.10.2 Type-Safe Statement Inversion

Given a well-typed statement, the statement inverter \( T \) 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 [11])**

\[
\begin{align*}
\langle \Pi, c \rangle & \vdash_{\text{stmt}} T[s] \implies \langle \Pi, c \rangle & \vdash_{\text{stmt}} T'[s]
\end{align*}
\]

*Proof.* By structural induction on \( T \). Unmodified ROOPL statements retained in ROOPL++ has been omitted.

- Case \( T = T-\text{ARR_ELEM_ASS} \)

\[
\begin{array}{c}
\Pi(x) : \text{int} \\
\Pi \vdash_{\text{expr}} e_1 : \text{int} \\
\Pi \vdash_{\text{expr}} e_2 : \text{int} \\
\end{array}
\]

\[
\begin{align*}
\langle \Pi, c \rangle & \vdash_{\text{stmt}} x[e_1] \odot e_2 \\
\Pi(x) & = \text{int} \\
\odot & = \text{int} \\
\end{align*}
\]

In this case, we have \( T[x \odot e] = x \odot' e \), for some \( \odot' \). Therefore, \( T' \) will also be a derivation of rule \( T-\text{ARR_ELEM_ASS} \), and as such, we can simply reuse the conditions \( C_1, C_2 \).
and the expressions $E_1, E_2$ in construction of $T'$

$$T' = \frac{\Pi(x) = \text{int}\{\}}{\Pi \vdash \text{expr} e_1 : \text{int} \quad (x \cup \text{vars}(e_1)) \cap \text{vars}(e_2) = \emptyset \quad \Pi \vdash \text{expr} e_2 : \text{int}}$$

- Case $T = \frac{\Pi(x) = c'}{(\Pi, c) \vdash \text{stmt} \text{new} c' x}$, meaning $T'$ must be of the form:

$$T' = \frac{\Pi(x) = c}{(\Pi, c) \vdash \text{stmt} \text{delete} c' x}$$

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

$$T' = \frac{\Pi(x) = c}{(\Pi, c) \vdash \text{stmt} \text{delete} c' x}$$

- Case $T = \frac{\text{arrayType}(a) \in \{ \text{classIDs, int} \}}{\Pi \vdash \text{expr} e = \text{int} \quad \Pi(x) = a[\]}$ \quad T-ARRNEW

In this case we still have $I[\text{new} c x] = \text{delete} c x$. Using $C_1$ and $E$, $T'$ must be of the form:

$$T' = \frac{\text{arrayType}(a) \in \{ \text{classIDs, int} \}}{\Pi \vdash \text{stmt} \text{delete} a[e] x}$$

- Case $T = \frac{\text{arrayType}(a) \in \{ \text{classIDs, int} \}}{\Pi \vdash \text{stmt} \text{delete} a[e] x}$ \quad T-ARRDLT
Similar to the object deletion case, we still have $I[\text{delete } c \ X] = \text{new } c \ X$. Using $C_1$ and $E$, $T'$ must be of the form:

$$T' = \frac{c_1 \ E \ \Pi(x) = a[]}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{new } a[c] \ X}$$

- Case $T = \frac{\Pi(x) = c' \ E \ \Pi(x') = c'}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{copy } c' \ X \ X'}$ T-Cp

We have $I[\text{copy } c \ X \ X'] = \text{uncopy } c \ X \ X'$. Using $C_1$, $T'$ must as such be of the form:

$$T' = \frac{c_1 \ \Pi(x) = c' \ E \ \Pi(x') = c'}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{uncopy } c' \ X \ X'}$$

- Case $T = \frac{\Pi(x) = c' \ E \ \Pi(x') = c'}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{uncopy } c' \ X \ X'}$ T-Ucp

We have $I[\text{uncopy } c \ X \ X'] = \text{copy } c \ X \ X'$. Using $C_1$, $T'$ must as such be of the form:

$$T' = \frac{\Pi(x) = c' \ E \ \Pi(x') = c'}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{copy } c' \ X \ X'}$$

- Case $T = \frac{\Pi(x) = c' \ E \ \Pi(x') = c'}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{copy } c' \ X \ X'}$ T-LOCALBLOCK

We have $I[\text{local } t \ X = e \ S \ \text{delocal } t \ X = e] = \text{local } t \ X = e \ I[[s]] \ \text{delocal } t \ X = e$.

By the induction hypothesis on $S$, we obtain $S'$ of $\langle \Pi[x \mapsto c'], c \rangle \vdash_{\text{stmt}} I[[s]]$. Using $E_1$, $S'$ and $E_2$ we construct $T'$

$$T' = \frac{\Pi(x) = c' \ E \ \Pi(x') = c'}{\langle \Pi, c \rangle \vdash_{\text{stmt}} \text{local } c' \ X = e_1 \ S \ \text{delocal } c' \ X = e_2}$$

Using these added cases to the original proof provided in [11], 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 2.28.
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 [29]. As such, 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 [29].

**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
- $\Gamma$ 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

$$q_1 = q'_1 \implies (t_1 \neq / \land t'_1 \neq / \land t_1 \neq t'_1) \text{ (forward determinism)}$$

$$q_2 = q'_2 \implies (t_1 \neq / \land t'_1 \neq / \land t_2 \neq t'_2) \text{ (backward determinism)}$$

A RTM simulation implemented in ROOPL by representing the set of states $\{q_1, \ldots, q_n\}$ and the tape alphabet $\Gamma$ as integers and the rule / and direction symbols $L$ and $R$ as the uppercase integer literals SLASH, LEFT and RIGHT was presented in [11]. As ROOPL contains no array or stack primitives, the transition table $\delta$ was suggested to be represented as a linked list of objects containing four integers $q_1, s_1, s_2$ and $q_2$ each, where $s_1$ 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 $q_1, s_1, s_2$ and $q_2$ represented as arrays, where the number of cells in each array is PC_MAX, in a similar fashion as shown in [29].

### 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 ease of use, a doubly linked list implementation similar to the one presented in section 2.7.3, of simple cell objects containing value, left, right and self fields, is more intuitive.

As such, the tape head finds 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, blank cells are deallocated if we are moving the tape head away from the cell currently neighbouring either end of the tape.

Figure 2.29 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 2.30 shows the modified method `inst` from [29], 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 2.31 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 termination. As objects in ROOPL++ are not bound by this limitation, the TM tape will exist as the program output when the RTM halts.¹

Instantiating a RTM simulation consists of initializing an initial tape head cell, as well as the

¹We are here breaking the rule that a `new` statement must eventually be followed by a `delete` statement to free the data.
transition rule arrays. After initialization, the simulate method is simply called and the simulation begins.
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 irreversible, modern programming languages uses dynamic memory management in some form for allocating space for objects in memory.

However, reversible, native support for 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, 30, 18].

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 paradigm.

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 [20].
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](image)

![Figure 3.1b: Creation of internal fragmentation of size $n - m \mod n$ due to over-allocation](image)

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 values of $n$. 
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.

![Figure 3.2: Example of external fragmentation caused for allocation and deallocation order](image1)

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.

![Figure 3.3: Example of avoiding external fragmentation using allocation and deallocation order](image2)

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 \( f^{-1} \).

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.

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 \( f^{-1} \), 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 \( f^{-1} \). The output free list of \( f^{-1} \) will naturally be a further modified free list.

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 modified free list \( free list' \), obtained after storing or freeing data in the heap during the execution of \( f \) with an input free list. A future injective function in the program, function \( g \), must thus take any modification of the original

---

Figure 3.4: The "garbage" output of an injective function \( f \) is the input to its inverse function \( f^{-1} \)

Figure 3.5: All free lists are considered equivalent "garbage" in terms of injective functions
free list in addition to its input to produce its output $output_g$ and a potentially further modified free list, \textit{free list}′. However, in the context of reversible heaps, we must consider all free lists as of equivalent, harmless 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 a function $g$ further in the program.

### 3.3 Linearity and Reference Counting

Programming languages use different approaches for storing and synchronizing variables and objects in memory. Typing \textit{linearity} is a distinction, which can reduce storage management and synchronization costs [4].

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 setting 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. Copying is not an option as field modification in one copy does not affect the other copies.

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 [18]. In RCFUN, reference counting is used to manage and trace the number of pointer copies made by respectively incrementing and decrementing a \textit{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 different 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 can always query the operating system for more memory.

Historically, most object-oriented programming languages utilize a dynamic memory manager during program execution. In lower-level languages such as C or 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 Java and Python, automatically 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) [15, 25, 21]. 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, 10, 19].

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 [18]. 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 [19].

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 bookkeeping 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 47 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, known as a multi-heap layout. 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](image)

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 can easily be determined at compile time. The deallocation is simply adding a new head to the free list.

Listing 3.1 outlines the allocation algorithm for this layout written in extended JANUS from [29]. 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.

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 with a size 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.

```plaintext
procedure malloc(int p, int osize, int freelists[])

local int counter = 0
local int csize = 2

call malloc1(p, osize, freelists, counter, csize)
delocal int csize = 2
delocal int counter = 0

procedure malloc1(int p, int osize, int freelists[], int counter, int csize)

if (csize < osize) then
    counter += 1
    csize += 1
call malloc1(p, osize, freelists, counter, csize)
csize -= 1
counter -= 1

else
    p += freelists[counter]
freelists[counter] -= p
```  

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 bookkeeping 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 feature a small number of fields, very rarely more than 16.

Additionally, helper classes of other sizes would spawn additional heaps and bookkeeping 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. Heap of some sizes may be mostly empty when another is full, resulting in wasted memory.

### 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 $2^n$.

![Figure 3.7: Memory layout using one heap per power-of-two](image)

Internal fragmentation does become a problem for very large records, as blocks are only of size $2^n$. An object of size 65 would fit in 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 bookkeeping. For smaller records, very minimal amounts of internal fragmentation occur.

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 assumes each heap has been initialized along with the free lists.
```plaintext
procedure double(int target)
    local int current = target
    target += current
    delocal int current = target / 2

procedure malloc1(int p, int osize, int freelists[], int counter, int csize)
    if (csize < osize) then
        counter += 1
        call double(csize)
        call malloc1(p, osize, freelists, counter, csize)
        counter -= 1
    else
        if freelists[counter] != 0 then
            p += freelists[counter]
            freelists[counter] -= p
            // Swap head of free list with next block of p
            freelists[counter] ^= M(p)
            M(p) ^= freelists[counter]
            freelists[counter] ^= M(p)
        else
            counter += 1
            call double(csize)
            call malloc1(p, osize, freelists, counter, csize)
            uncall double(csize)
            counter -= 1
        fi
    fi
fi
```

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 designs, 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 bookkeeping involved in maintaining multiple free lists. Having mixed-size blocks in a single heap is also a task which might prove difficult to accomplish reversibly. 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 simply check the free lists for a free block of size $n$, where $n \geq m$. If 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 [13].

![Figure 3.8: Record size-specific free lists on a shared heap (powers of two)](image)

![Figure 3.9: Buddy Memory block allocation example](image)

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 and merge where possible when freed.

The main 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. 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 bookkeeping.

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.

Listing 3.3 shows the Buddy Memory algorithm implemented in the extended Janus variant with local blocks from [29]. For simplification, object sizes are rounded to the nearest power-of-two during compile-time. The algorithm extends on the one heap per power-of-two algorithm presented in listing 3.2, page 53. 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.

Since a split block is always added as two blocks to an empty free list, we can only merge adjacent
blocks if they are the only two blocks in a free list.
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 an extension of ROOPL, many techniques are carried directly over, and have as 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 [11]. 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 [11]

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.
4.2 ROOPL++ Memory Layout

ROOPL++ builds upon the memory layout of its predecessor 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 shows the full layout of a ROOPL++ program stored in memory.

- As with ROOPL, the static storage segment contains load-time labelled DATA pseudo-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 translated 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 $f_{lp}$ pointing at the first free list. The free lists are simply the address pointing to the first block of its respective size. The free lists are stored such that the free list at address $f_{lp} + i$ 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 ($h_{p}$).

---

Figure 4.2: Illustration of prefixing in the memory layout of 3 ROOPL objects, originally from [11]
Unlike in Roopl, where the stack grows upwards, the Roopl++ stack grows downwards and 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 defaults 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 the largest possible size and storing the address of the respective free list.

In the following sections of this chapter, we will present various translation techniques. In these translations, we will make use of a number of Pisa pseudo-instructions to subtract integer values from registers and pushing/popping to the program stack. The pseudo-instructions are shown in figure 4.4 and are modified from [11], as the direction of the program stack is flipped in Roopl++.

\[
\begin{align*}
\text{SUBI } r \ i & \quad \text{def} = \text{ADDI } r \ - \ i \\
\text{PUSH } r & \quad \text{def} = [\text{EXCH } r \ r_{sp}, \ \text{SUBI } r_{sp} \ 1] \\
\text{POP } r & \quad \text{def} = [\text{ADDI } r_{sp} \ 1, \ \text{EXCH } r \ r_{sp}] 
\end{align*}
\]

Figure 4.4: Definition of pseudoinstructions SUBI, PUSH and POP, modified from [11]

4.3 Inherited ROOPL features

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

The dynamic dispatching mechanism presented in [11] 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 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 4.5.

The following 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 4.5 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 list is the size of one word and the full size of the free lists is configured in the source code (defaulted to 10, as described earlier).

Once the heap pointer and free lists pointer is initialized, the initial block of free memory is placed in the largest free lists by indexing to said list, by adding the length of the list of free lists, subtracting 1, writing the address of the first block (which is the same address as the heap
pointer, which points to the beginning of the heap) to the last free list and then resetting the free lists pointer to point to the first 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 and free list pointer not intentionally not cleared to simulate future program simulation using these pointers. 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 the stack section of memory.

4.5 Buddy Memory Translation

As briefly mentioned in section 4.2, 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 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.

Before we go into depth with the translation of the algorithm, we consider the mechanism for triggering the allocation subroutine. Naively, we could generate the entire block of code required for triggering the allocation subroutine. However, it is possible to reduce the number of instructions by using dynamic dispatch. The following code shows how to jump to the malloc subroutine:

```
(1) malloc1top : BRA malloc1bot ; Receive jump
(2) POP r\textsubscript{ro} ; Pop return offset from the stack
(3) \ldots \ldots ; Inverse of (7)
(4) malloc1entry : SWAPBR r\textsubscript{ro} ; Malloc1 entry and exit point
(5) NEG r\textsubscript{ro} ; Negate return offset
(6) PUSH r\textsubscript{ro} ; Store return offset on stack
(7-63) \ldots \ldots ; Allocation code
(64) malloc1bot : BRA malloc1top ; Jump
```

Figure 4.6: Dynamic dispatch approach for entering the allocation subroutine
for allocation for every new or delete statement in the target program. This approach would
severely limit the amount of objects we could allocate as the register pressure of the Buddy
Memory implementation is quite high, as we be shown in this section. Instead, we can utilize
the dynamic dispatching technique, which also is used for method invocations. This way, we
only generate the allocation instructions once, and then simply jump to the entry point from
different locations in the program. Figure 4.6 outlines the structure for this approach. By using
the SWAPPB instruction we can jump from multiple points of origin in the compiled program
and internally for the recursive needs of the algorithm itself.

Figure 4.7: PISA translation of the nested conditionals in the Buddy Memory algorithm

The main recursion body of the algorithm, malloc1 from listing 3.3, page 56 consists of two
conditionals, in which one is nested in the else branch of the outer conditional. Figure 4.7 shows
the translation structure of the nested conditional pair, using the translation techniques for
conditionals presented in [1].

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, counter, csize)  
5 uncall double(csize)  
6 counter -= 1

(14) ADDI r, 1 ; Counter++  
(15) RL r, 1 ; Call double(osize)  
(16) .... ; Inverse of (7)  
(17) .... ; Code for pushing temp reg values to stack  
(18) BRA malloc1entry ; Call malloc1()  
(19) .... ; Inverse of (17)  
(20) RR r, -1 ; Inverse of (15)  
(21) SUBI r, 1 ; Inverse of (14)

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

Figure 4.8 shows the translation of the outer if-then statement. As briefly mentioned, we can utilize the right bit shift instruction of Pisa, 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 4.6. 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, r_block; Copy address of the current block to p  
(31) SUB r_block, r_p; Clear r_block  
(32) EXCH r_malloc, r_p; Load address of next block  
(33) EXCH r_malloc, 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 4.9:** Pisa translation of the inner if-then statement for the Buddy Memory algorithm

Figure 4.9 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, counter, csize)  
5 uncall double(csize)  
6 counter -= 1  
7 freelists[counter] += p  
8 p += csize

(38) ADDI r, 1 ; Counter++  
(39) RL r_c, 1 ; Call double(csize)  
(40) .... ; Push temp reg values to stack  
(41) BRA malloc1entry ; Call malloc1()  
(42) .... ; Inverse of (40)  
(43) RR r_c, 1 ; Inverse of (39)  
(44) SUBI r, 1 ; Inverse of (38)  
(45) XOR r_malloc, r_p; Copy current address of p  
(46) EXCH r_malloc, r_p; Store address of p in free list  
(47) ADD r, r_c; Split block by p = other half of block

**Figure 4.10:** Pisa translation of the inner else statement for the Buddy Memory algorithm

The last statement translation is the inner else statement shown in figure 4.10. 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 limitations on the merging and splitting conditions were required to ensure reversibility. Since a split block is always added as two blocks to an empty free list, we can only merge adjacent blocks if they are the only two blocks in a free list. In the irreversible Buddy Memory algorithm block merging can occur in any place of the free list, but in the reversible version, we can only merge blocks at the start of the free list to maintain reversibility. The effect of this limitation prevents us from returning to one final block of free memory, if the deallocation order is not exactly opposite of the allocation order.

Figure 4.11 shows how alternative deallocation orders results in different free lists, compared to the original given to some function. However, as discussed in 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.
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 56.

Figure 4.12: PISA translation of the malloc procedure entry point of Buddy Memory algorithm

Figure 4.12 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 allocation 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 | delete c x |
|---|---|
| (1) \(\ldots\) | (1) \(\ldots\) |
| (2) \(\ldots\) | (2) \(\text{EXCH} \quad r_1 \quad r_p\) |
| (3) \(\text{PUSH} \quad r_1\) | (3) \(\text{XOR} \quad r_1 \quad \text{label}x\) |
| (4) \(\text{PUSH} \quad r_p\) | (4) \(\text{ADDI} \quad r_p \quad \text{offset}_c\) |
| (5) \(\text{BRA} \quad l_{\text{malloc}}\) | (5) \(\text{EXCH} \quad r_1 \quad r_p\) |
| (6) \(\text{POP} \quad r_p\) | (6) \(\text{XOR} \quad r_1 \quad 1\) |
| (7) \(\text{POP} \quad r_1\) | (7) \(\text{SUBI} \quad r_p \quad \text{offset}_c\) |
| (8) \(\ldots\) | (8) \(\ldots\) |
| (9) \(\ldots\) | (9) \(\ldots\) |
| (10) \(\ldots\) | (10) \(\text{PUSH} \quad r_1\) |
| (11) \(\text{XOR} \quad r_1 \quad \text{label}x\) | (11) \(\text{PUSH} \quad r_p\) |
| (12) \(\text{EXCH} \quad r_1 \quad r_p\) | (12) \(\text{BRA} \quad l_{\text{malloc}}\) |
| (13) \(\text{ADDI} \quad r_p \quad \text{offset}_c\) | (13) \(\text{POP} \quad r_p\) |
| (14) \(\text{XOR} \quad r_1 \quad 1\) | (14) \(\text{POP} \quad r_1\) |
| (15) \(\text{EXCH} \quad r_1 \quad r_p\) | (15) \(\ldots\) |
| (16) \(\text{SUBI} \quad r_p \quad \text{offset}_c\) | (16) \(\ldots\) |
| (17) \(\text{EXCH} \quad r_1 \quad r_p\) | (17) \(\ldots\) |
| (18) \(\ldots\) | (18) \(\ldots\) |

Figure 4.13: PISA translation of heap allocation and deallocation for objects
Figure 4.13 shows how each new and delete statement for objects are translated during compilation. They are simply inverse of each other. 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 size denotes the computed size of objects with class c, plus two, to account for the virtual table pointer and reference count space, rounded up to nearest power-of-two.

\[
\text{construct } c \ x \ s \ \text{destruct } x
\]

1. XOR \( r_x r_{sp} \) ; Store address of new object x in \( r_x \)
2. PUSH \( r_x \) ; Push r_x to the stack
3. \(
\) ; Code for new c x
4. \(
\) ; Code for statement s
5. \(
\) ; Code for delete c x
6. POP \( r_x \) ; Pop r_x from the stack
7. XOR \( r_x r_{sp} \) ; Clear r_x

Figure 4.14: PISA translation of a ROOPL++ object block

Figure 4.14 shows the updated translation technique for object blocks. In ROOPL, the object blocks allocated their objects on the stack, but in ROOPL++, we can now allocate them on the heap. To facilitate this, we simply execute the exact same instructions as in new and delete statements, with body statement execution code in between. As described in section 2.6, the construct/destruct block can be considered syntactic sugar, and its usage in a real world example would probably be limited.

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 one. When we uncopy a reference, the reference counter is decreased by one. 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 4.15 shows the object layout of ROOPL++ objects with the added space for the reference counting from the original ROOPL model in figure 4.2 on page 59.
Figure 4.15: Illustration of prefixing in the memory layout of three ROOPL++ objects

Figure 4.16 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.

\[
\text{copy} \ c \ x \ x' \\
\text{uncopy} \ c \ x \ x'
\]

Figure 4.16: PISA translation of the reference copying and deletion statements

4.8 Arrays

The fixed-sized arrays in ROOPL++ are also heap allocated to allow dynamic lifetime. The array memory layout is presented in figure 4.17. 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.

As the size of a ROOPL++ array is determined by a passed expression evaluation, it is unknown at compile time. This also means that out-of-bounds checking cannot be conducted during compilation. A possible solution for this is presented in section 4.9.
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 r1 ← [e] + 2
(3) PUSH r1 ; Push r1
(4) PUSH rp ; Push rp
(5) BRA malloc ; Allocate array
(6) POP rp ; Inverse of (4)
(7) POP r1 ; Inverse of (3)
(8) ... ; Inverse of (1)
(9) ... ; Code for rp ← [addr(x)]
(10) SUBI r1 2 ; r1 ← [e]
(11) EXCH r1 rp ; Store size in new array
(12) ADDI rp of setref ; Index to ref count pos
(13) XORI r1 1 ; Init ref count
(14) EXCH r1 rp ; Store ref count
(15) SUBI rp of setref ; Inverse of (13)
(16) EXCH rp r0 ; Store address in variable
(17) ... ; Inverse of (10)
(18) ... ; Inverse of (9)
(19) ... ; Inverse of (8)
(20) ... ; Inverse of (7)
(21) ... ; Inverse of (6)
(22) ... ; Inverse of (5)
(23) ... ; Inverse of (4)
(24) ... ; Inverse of (3)
(25) ... ; Inverse of (2)
(26) ... ; Inverse of (1)
```

delete a[e] x

```
delete a[e] x

(1) ... ; Code for rp ← [addr(x)]
(2) ... ; Code for r1 ← [e]
(3) ADDI rp of setref ; Index to ref count pos
(4) EXCH r1 rp ; Extract ref count
(5) XORI rp 1 ; Clear ref count
(6) SUBI rp of setref ; Inverse of (3)
(7) EXCH r1 rp ; extract size from object
(8) XORI rp r0 ; clear size in rp
(9) ... ; Push registers except rp, r0
(10) ADDI rp 2 ; Actual size of array
(11) PUSH rp ; Push rp
(12) PUSH rp ; Push rp
(13) BRA malloc ; Deallocate array
(14) POP rp ; Inverse of (12)
(15) POP rp ; Inverse of (11)
(16) SUBI rp 2 ; Inverse of (10)
(17) ... ; Inverse of (9)
(18) ... ; Inverse of (8)
(19) ... ; Inverse of (7)
```

Figure 4.18: Pisa translations of array allocation and deallocation statements

Figure 4.18 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 4.13 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. The
size of arrays cannot be 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. While the two blocks are code are not exact opposites they are functionally inverse of each other. An extra \texttt{XORI} instruction on line (8) in the deallocation block has been included to clear the stored array size using the value of the passed expression and further use this size for the inverse \texttt{malloc} subroutine.

\begin{equation}
\begin{array}{c}
  c[2] \\
  \downarrow \\
  \begin{array}{c|c|c}
    2 & 1 & \text{nil} \\
  \end{array}
\end{array}
\end{equation}

(a) Allocate an array of class \(c\) and size 2 in \(x\) by \texttt{new } \(c[2] \ x\)

\begin{equation}
\begin{array}{c}
  c[2] \\
  \downarrow \\
  \begin{array}{c|c|c|c}
    c & 2 & 1 & \text{nil} \\
  \end{array}
\end{array}
\end{equation}

(b) Assign first cell with \texttt{new } \(c \ x[0]\)

\begin{equation}
\begin{array}{c}
  c[2] \\
  \downarrow \\
  \begin{array}{c|c|c|c}
    c & 2 & 1 & \text{nil} \\
  \end{array}
\end{array}
\end{equation}

(e) Assign next cell with \texttt{new } \(c \ x[1]\)

Figure 4.19: Illustration of array memory storage layout

Figure 4.19 shows how object arrays simply contain pointers to allocated objects. For integer arrays, the cell values would stored directly in the allocated array space instead.

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 determines 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 \texttt{ROOPL++} might be syntactically valid and well-typed, this 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 executed code. Haulund describes the following conditions:

- If the entry expression of a conditional is \texttt{true}, then the exit assertion should also be \texttt{true} after executing the then-branch.
• If the entry expression of a conditional is \texttt{false}, then the exit assertion should also be \texttt{false} after executing the else-branch.

• The entry expression of a loop should initially be \texttt{true}.

• If the exit assertion of a loop is \texttt{false}, then the entry expression should also be \texttt{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 [11].

The extensions made to \texttt{ROOPL} in \texttt{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 \texttt{delete} statement.

• All cells of an instance should be zero-cleared before the array is deallocated using the \texttt{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 the value of a local object variable 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 the value of a local array variable 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 \texttt{new} statement must be zero-cleared beforehand.

• The variable in the \texttt{copy} statement must be zero-cleared beforehand.

• An object variable must be initialized using \texttt{new} or \texttt{copy} before its methods can be called.

• An array variable must be initialized using \texttt{new} or \texttt{copy} before its fields can be accessed.

• Array cell indices must be within bounds defined in the expression passed during initialization.

• Only one reference to an object or an array must exist when executing the \texttt{delete} statement.

• Swapping cell values between a subtype $A$ variable and parent-type $B$ array is only allowed if the value stored in the variable is also $A$ afterwards.

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 [11]. 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 [5].

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 1403 lines of HASKELL and the ROOPL++ compiler was extended to 2046 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 | Number of executed instructions |
|-------------------|-------------|----------|---------------------------------|
| Linked List       | 61          | 1280     | 18015                           |
| Doubly-Linked List| 66          | 1339     | 21825                           |
| Binary Tree       | 86          | 2056     | 6065                            |
| RTM Simulation    | 211         | 6716     | 64922                           |

Figure 4.20: Lines of code comparison between target and compiled ROOPL++ programs

The linked list test programs simply instantiates ten cells and links them in their respective lists. The binary tree test program instantiates three nodes and adds them to the tree structure, which afterwards is traversed to determine the sum of the nodes and finally mirroring the tree. The Reversible Turing Machine implementing incrementation of a non-negative \( n \)-bit binary number by 1 originally described in [29] has been implemented in ROOPL++ and successfully converts its initial tape value in little endian form of \( 1101 \) to \( 0011 \) after termination. It should be noted that these test programs require additional stack space during their lengthy computations and as such has been compiled with twice the length between the stack and heap to allow further stack growth.

As discussed, the compiler is considered proof-of-concept and no noteworthy optimizations has been implemented. However, for the sake of giving the reader an idea of the size blowup of a compiled ROOPL++ program, figure 4.20 details this difference. The lines of translated PISA instructions includes the 204 instructions needed for the \texttt{malloc} and \texttt{malloc1} PISA-equivalent mechanisms. The last row of the table shows how many instructions are execution during simulation using PendVM.
Conclusions

We formally presented a dynamic memory management extension for the reversible object-oriented programming language, ROOPL, in the 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 example programs such as a new reversible Turing machine simulator 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 also shown that complex dynamic data structures are not only feasible, but furthermore do not contradict the reversible computing paradigm.

We presented various dynamic memory management layouts and how each would translate into the reversible allocation algorithms. 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 the 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, the new fixed-sized array type and reference counting have been demonstrated and implemented in a proof-of-concept compiler for validation.

With the dynamic memory manager for reversible object-oriented programming languages allowing dynamic object-scopes and multiple references, exemplified by ROOPL++, we have successfully taking an additional step in the direction towards high-level abstractions reversible computations.

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 work could be made to extend the fixed-sized array type with a fully dynamic array supporting multiple dimensions. 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 for non-mutable objects has been designed and shown feasible for the reversible functional language RCFUN in [19]. 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 [8, 9].
References

[1] Axelsen, H. B. “Clean Translation of an Imperative Reversible Programming Language”. In: *Compiler Construction - 20th International Conference, CC 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26-April 3, 2011. Proceedings*. Ed. by J. Knoop. Vol. 6601. Lecture Notes in Computer Science. Springer, 2011, pp. 144–163.

[2] Axelsen, H. B. and Glück, R. “Reversible Representation and Manipulation of Constructor Terms in the Heap”. In: *Reversible Computation - 5th International Conference, RC 2013, Victoria, BC, Canada, July 4-5, 2013. Proceedings*. Ed. by G. W. Dueck and D. M. Miller. Vol. 7948. Lecture Notes in Computer Science. Springer, 2013, pp. 96–109.

[3] Axelsen, H. B., Glück, R., and Yokoyama, T. “Reversible Machine Code and Its Abstract Processor Architecture”. In: *Computer Science - Theory and Applications, Second International Symposium on Computer Science in Russia, CSR 2007, Ekaterinburg, Russia, September 3-7, 2007, Proceedings*. Ed. by V. Diekert, M. V. Volkov, and A. Voronkov. Vol. 4649. Lecture Notes in Computer Science. Springer, 2007, pp. 56–69.

[4] Baker, H. G. ‘Use-Once’ Variables and Linear Objects – Storage Management, Reflection and Multi-Threading. URL: http://home.pipeline.com/~hbaker1/UselVar.html (visited on 01/22/2018).

[5] 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/ (visited on 01/22/2018).

[6] Digiconomist. “Bitcoin Energy Consumption Index”. In: *Digiconomist* (2017-12-06). URL: https://digiconomist.net/bitcoin-energy-consumption (visited on 01/22/2018).

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

[8] Glück, R. and Yokoyama, T. “A Linear-Time Self-Interpreter of a Reversible Imperative Language”. In: *Computer Software 33.3* (2016), pp. 108–128.

[9] Glück, R. and Yokoyama, T. “A Minimalist’s Reversible While Language”. In: *IEICE Transactions 100-D.5* (2017), pp. 1026–1034.

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

[11] Haulund, T. “Design and Implementation of a Reversible Object-Oriented Programming Language”. In: *CoRR abs/1707.07845* (2017).

[12] Haulund, T., Mogensen, T. Å., 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. Vol. 10301. Lecture Notes in Computer Science. Springer, 2017, pp. 66–73.
References 77 of 231

[13] Knuth, D. E. *The Art of Computer Programming: Fundamental Algorithms*. The Art of Computer Programming. Addison-Wesley, 1998, pp. 435–455. ISBN: 0-201-89683-4.

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

[15] Lee, W. H. and Chang, M. “A study of dynamic memory management in C++ programs”. In: *Comput. Lang.* 28.3 (2002), pp. 237–272.

[16] Lutz, C. “Janus: a time-reversible language”. Letter to R. Landauer. 1986.

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

[18] Mogensen, T. Æ. “Reference Counting for Reversible Languages”. In: *Reversible Computation - 6th International Conference, RC 2014, Kyoto, Japan, July 10-11, 2014. Proceedings*. Ed. by S. Yamashita and S. Minato. Vol. 8507. Lecture Notes in Computer Science. Springer, 2014, pp. 82–94.

[19] Mogensen, T. Æ. “Garbage Collection for Reversible Functional Languages”. In: *Reversible Computation - 7th International Conference, RC 2015, Grenoble, France, July 16-17, 2015, Proceedings*. Ed. by J. Krivine and J. Stefani. Vol. 9138. Lecture Notes in Computer Science. Springer, 2015, pp. 79–94.

[20] Mogensen, T. Æ. “Programming Language Design and Implementation (Unpublished)”.

[21] *Python Software Foundation: Memory Management*. URL: https://docs.python.org/2/c-api/memory.html (visited on 01/22/2018).

[22] Schultz, U. P. and Axelsen, H. B. “Elements of a Reversible Object-Oriented Language - Work-in-Progress Report”. In: *Reversible Computation - 8th International Conference, RC 2016, Bologna, Italy, July 7-8, 2016, Proceedings*. Ed. by S. J. Devitt and I. Lanese. Vol. 9720. Lecture Notes in Computer Science. Springer, 2016, pp. 153–159.

[23] Thomsen, M. K., Axelsen, H. B., and Glück, R. “A Reversible Processor Architecture and Its Reversible Logic Design”. In: *Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers*. Ed. by A. D. Vos and R. Wille. Vol. 7165. Lecture Notes in Computer Science. Springer, 2011, pp. 30–42.

[24] 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 01/22/2018).

[25] Venners, B. *The Java Virtual Machine*. URL: http://www.artima.com/insidejvm/ed2/jvmP.html (visited on 01/22/2018).

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

[27] A. D. Vos and R. Wille, eds. *Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers*. Vol. 7165. Lecture Notes in Computer Science. Springer, 2012.

[28] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA, USA: MIT Press, 1993.
[29] Yokoyama, T., Axelsen, H. B., and Glück, R. “Principles of a reversible programming language”. In: Proceedings of the 5th Conference on Computing Frontiers, 2008, Ischia, Italy, May 5-7, 2008. Ed. by A. Ramírez, G. Bilardi, and M. Gschwind. ACM, 2008, pp. 43–54.

[30] Yokoyama, T., Axelsen, H. B., and Glück, R. “Towards a Reversible Functional Language”. In: Reversible Computation - Third International Workshop, RC 2011, Gent, Belgium, July 4-5, 2011. Revised Papers. Ed. by A. D. Vos and R. Wille. Vol. 7165. Lecture Notes in Computer Science. Springer, 2011, pp. 14–29.

[31] Yokoyama, T. and Glück, R. “A reversible programming language and its invertible self-interpreter”. In: Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007. Ed. by G. Ramalingam and E. Visser. ACM, 2007, pp. 144–153.
APPENDIX A

Pisa Translated Buddy Memory

(1) malloc1top : BRA malloc1bot ; Receive jump
(2) POP r0 ; Pop return offset from the stack
(3) mallocentry : SMAPBR r0 ; Malloc entry and exit point
(4) NEG r0 ; Negate return offset
(5) PUSH r0 ; Store return offset on stack
(6) ........ ; Code for $r_{ij} \leftarrow \text{addr}(\text{freelists}[\text{counter}])$
(7) ........ ; Code for $r_{block} \leftarrow [\text{freelists}[\text{counter}]]$
(8) ........ ; Code for $r_{i1} \leftarrow [\text{size} < \text{objectsize}]$
(9) .......... ; Inverse of (7)
(10) XOR r1 r0 ; Copy value of $c_{size} < \text{objectsize}$ into $r_{t}$
(11) .......... ; Inverse of (9)
(12) otest : BEQ r1 r0 otestf ; Receive jump
(13) XORI r1 1 ; Clear $r_{t}$
(14) ADDI r0 1 ; Counter++
(15) RL rsc 1 ; Call double($c_{size}$)
(16) .......... ; Inverse of (7)
(17) .......... ; Code for pushing temp reg values to stack
(18) BRA malloc1entry ; Call malloc1()
(19) .......... ; Inverse of (17)
(20) RR rsc 1 ; Inverse of (15)
(21) SUBI r0 1 ; Inverse of (14)
(22) XORI r0 1 ; Set $r_{t} = 1$
(23) oassertf : BRA oassert ; Jump
(24) otestf : BRA otest ; Receive jump
(25) .......... ; Code for $r_{i1} \leftarrow \text{addr}(\text{freelists}[\text{counter}]) \neq 0$
(26) XOR r12 r0 ; Copy value of $r_{i1}$ into $r_{12}$
(27) .......... ; Inverse of (25)
(28) iletst : BEQ r12 r0 iletstf ; Receive jump
(29) XORI r12 1 ; Clear $r_{12}$
(30) ADD r0 rblock ; Copy address of the current block to $p$
(31) SUB rblock r0 ; Clear $r_{block}$
(32) EXCH rtmp rp ; Load address of next block
(33) EXCH rtmp rti ; Set address of next block as new head of free list
(34) XOR rtmp rp ; Clear address of next block
(35) XORI r12 1 ; Set $r_{12} = 1$
(36) ilassertf : BRA ilassert ; Jump
(37) iltestf : BRA iltest ; Receive jump
(38) ADDI r0 1 ; Counter++
(39) RL rsc 1 ; Call double($c_{size}$)
(40) .......... ; Code for pushing temp reg values to stack
(41) BRA malloc1entry ; Call malloc1()
(42) .......... ; Inverse of (40)
(43) RR rsc 1 ; Inverse of (39)
(44) SUBI r0 1 ; Inverse of (38)
(45) \textbf{XOR} r_{\text{tmp}} \ r_p ; \text{Copy current address of } p

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

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

(48) \textbf{assert} : BNE r_{12} \ r_0 \ \textbf{assertt} ; \text{Receive jump}

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

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

(51) \ldots \ldots \ldots \text{; Code for } r_{a1} \leftarrow [p - \text{csize} \neq \text{addr}(\text{freelists}[\text{counter}])]

(52) \ldots \ldots \ldots \text{; Code for } r_{a2} \leftarrow [\text{addr}(\text{freelists}[\text{counter}]) = 0]

(53) \ldots \ldots \ldots \text{; Code for } r_{a3} \leftarrow [(p - \text{csize} \neq \text{addr}(\text{freelists}[\text{counter}])) \lor (\text{addr}(\text{freelists}[\text{counter}]) = 0)]

(54) \textbf{XOR} r_{t2} \ r_{a2} ; \text{Copy value of } r_{a2} \text{ into } r_{t2}

(55) \ldots \ldots \ldots \text{; Inverse of (52)}

(56) \ldots \ldots \ldots \text{; Inverse of (51)}

(57) \textbf{ADD} r_p \ r_{cs} ; \text{Inverse of (50)}

(58) \textbf{EXCH} r_{tmp} \ r_{fl} ; \text{Inverse of (49)}

(59) \textbf{assert} : BNE r_t \ r_0 \ \textbf{assertt} ; \text{Receive jump}

(60) \ldots \ldots \ldots \text{; Code for } r_{a2} \leftarrow [\text{csize} < \text{objectsize}]

(61) \ldots \ldots \ldots \text{; Code for } r_{t} \leftarrow [\text{csize} < \text{objectsize}]

(62) \textbf{XOR} r_t \ r_{a0} ; \text{Copy value of } c_{size} < \text{objectsize} \text{ into } r_t

(63) \ldots \ldots \ldots \text{; Inverse of (61)}

(64) \textbf{malloc}1 \ : \textbf{BRA malloc1top} ; \text{Jump}
# ROOPLPPC

**ROOPLPP** is a compiler translating source code written in **Reversible Object Oriented Programming Language++** (ROOPL++) to the reversible assembly language Pendulum ISA (PISA).

The compiler is to be considered proof-of-concept in connection with my Master’s Thesis on the ROOPL++ language.

## Requirements

ROOPLPPC uses [Stack](https://docs.haskellstack.org/en/stable/README/) to manage all dependencies and requirements.

## Building

Simply invoke

```
stack build
```

which compiles an executable into the `.stack-work` folder

## Usage

To compile a ROOPL++ program simply run

```
stack exec ROOPLPPC input.rplpp
```

which compiles the input program into Pisa and stores the compiled file as `input.pal` in the current directory.

To specify an output file name, simply provide it as an additional argument

```
stack exec ROOPLPP input.rplpp output.pal
```

## Examples

To see usage examples, please refer to `test/` for example programs.

## Running compiled programs

The PendVM simulator executes compiled Pisa code and is hosted on Github [here](https://github.com/TueHaulund/PendVM).
module AST where

import Text.Show.Pretty

{- -- AST Primitives -- -}
type TypeName = String

type MethodName = String

data DataType = IntegerType

| ObjectType TypeName

| CopyType TypeName

| A ObjectArrayType TypeName

| IntegerArrayType

| ArrayType

| ArrayElementType

| NilType

deriving (Show)

-- Types
instance Eq DataType where

IntegerType == IntegerType = True

IntegerArrayType == IntegerArrayType = True

NilType == NilType = True

NilType == (ObjectType _) = True

(ObjectType t1) == (ObjectArrayType t2) = t1 == t2

(CopyType t1) == (CopyType t2) = t1 == t2

(ObjectArrayType t1) == (ObjectArrayType t2) = t1 == t2

(CopyType t1) == (ObjectArrayType t2) = t1 == t2

(ObjectType t1) == (CopyType t2) = t1 == t2

ArrayType == (ObjectArrayType _) = True

(ObjectArrayType _) == ArrayType = True

ArrayType == IntegerArrayType = True

IntegerArrayType == ArrayType = True

_ == _ = False

-- Binary Operators
data BinOp = Add

| Sub

| Xor

| Mul

| Div

| Mod

| BitAnd

| BitOr

| And

| Or

| Lt

| Gt

| Eq

| Neq

| Lte

| Gte

deriving (Show, Eq, Enum)

data ModOp = ModAdd

| ModSub

| ModXor

deriving (Show, Eq, Enum)

{- -- Generic AST Definitions --}

--Expressions
data GExpr v = Constant Integer
  | Variable v
  | ArrayElement (v, GExpr v)
  | Nil
  | Binary BinOp (GExpr v) (GExpr v)
  deriving (Show, Eq)

-- Statements
data GStmt m v = Assign v ModOp (GExpr v)
  | AssignArrElem (v, GExpr v) ModOp (GExpr v)
  | Swap (v, Maybe (GExpr v)) (v, Maybe (GExpr v))
  | Conditional (GExpr v) [GStmt m v] [GStmt m v] (GExpr v)
  | Loop (GExpr v) [GStmt m v] [GStmt m v] (GExpr v)
  | ObjectBlock TypeName v [GStmt m v]
  | LocalBlock DataType v (GExpr v) [GStmt m v]
  | ObjectCall (v, Maybe (GExpr v)) MethodName [(v, Maybe (GExpr v))]
  | ObjectUncall (v, Maybe (GExpr v)) MethodName [(v, Maybe (GExpr v))]
  | ObjectConstruction TypeName (v, Maybe (GExpr v))
  | ObjectDestruction TypeName (v, Maybe (GExpr v))
  | CopyReference DataType (v, Maybe (GExpr v)) (v, Maybe (GExpr v))
  | UnCopyReference DataType (v, Maybe (GExpr v)) (v, Maybe (GExpr v))
  | ArrayConstruction (TypeName, GExpr v) v
  | ArrayDestruction (TypeName, GExpr v) v
  | Skip
  deriving (Show, Eq)

-- Field/Parameter declarations
data GDecl v = GDecl DataType v
  deriving (Show, Eq)

-- Method: Name, parameters, body
data GMDecl m v = GMDecl m [GDecl v] [GStmt m v]
  deriving (Show, Eq)

-- Class: Name, fields, methods
data GCDecl m v = GCDecl TypeName (Maybe TypeName) [GDecl v] [GMDecl m v]
  deriving (Show, Eq)

-- Program
newtype GProg m v = GProg [GCDecl m v]
  deriving (Show, Eq)

{- Specific AST Definitions --}
-- Plain AST
type Identifier = String

type Expression = GExpr Identifier

type Statement = GStmt MethodName Identifier

type VariableDeclaration = GDecl Identifier

type MethodDeclaration = GMDecl MethodName Identifier

type ClassDeclaration = GCDecl MethodName Identifier

type Program = GProg MethodName Identifier

-- Scoped AST
type SIdentifier = Integer

type SExpression = GExpr SIdentifier

type SStatement = GStmt SIdentifier SIdentifier
type SVariableDeclaration = GDecl SIdentifier

type SMethodDeclaration = GMDecl SIdentifier SIdentifier

type SProgram = [(TypeName, GMDecl SIdentifier SIdentifier)]

|-- Other Definitions --|

type Offset = Integer

data Symbol = LocalVariable DataType Identifier
  | ClassField DataType Identifier TypeName Offset
  | MethodParameter DataType Identifier
  | Method [DataType] MethodName

  deriving (Show, Eq)

type SymbolTable = [(SIdentifier, Symbol)]

type Scope = [(Identifier, SIdentifier)]

printAST :: (Show t) => t -> String

printAST = ppShow
module PISA where

import Data.List (intercalate)
import Control.Arrow
import AST (TypeName, MethodName)

type Label = String
newtype Register = Reg Integer
  deriving (Eq)

{- Generic PISA Definitions --}

data GInstr i = ADD Register Register
  | ADDI Register i
  | ANDX Register Register Register
  | ANDIX Register Register i
  | NORX Register Register Register
  | NEG Register
  | ORX Register Register Register
  | ORIX Register Register i
  | RLX Register i
  | RLVX Register Register Register
  | SLVX Register Register i
  | SLLVX Register Register Register
  | SRAVLX Register Register Register
  | SRLVX Register Register Register
  | SUB Register Register
  | XOR Register Register
  | XORI Register i
  | BEQ Register Register Label
  | BGEZ Register Label
  | BGTZ Register Label
  | BLEZ Register Label
  | BLTZ Register Label
  | BNEE Register Register Label
  | BRA Label
  | EXCH Register Register
  | SWAPBR Register
  | RBRA Label
  | START
  | FINISH
  | DATA i
  | SUBI Register i --Pseudo
  deriving (Eq)

newtype GProg i = GProg [Maybe Label, GInstr i]

{- Macro PISA Definitions --}

data Macro = Immediate Integer
  | AddressMacro Label
  | SizeMacro TypeName
  | OffsetMacro TypeName MethodName
  | ProgramSize
  | FreeListsSize

Appendix B  ROOPLPPC Source Code  85 of 231
invertInstruction (RBRA l) = RBRA $ l ++ "_i"
invertInstruction (BRA l) = BRA $ l ++ "_i"
invertInstruction (BNE r1 r2 l) = BNE r1 r2 $ l ++ "_i"
invertInstruction (BLTZ r l) = BLTZ r $ l ++ "_i"
invertInstruction (BLEZ r l) = BLEZ r $ l ++ "_i"
invertInstruction (BGTZ r l) = BGTZ r $ l ++ "_i"
invertInstruction (BGEZ r l) = BGEZ r $ l ++ "_i"
invertInstruction (BEQ r1 r2 l) = BEQ r1 r2 $ l ++ "_i"
invertInstruction (RRV r1 r2) = RRV r1 r2
invertInstruction (RR r i) = RR r i
invertInstruction (RLV r1 r2) = RLV r1 r2
invertInstruction (RL r i) = RL r i
invertInstruction (SUBI r i) = ADDI r i
invertInstruction (SUB r1 r2) = ADD r1 r2

where
invertInstructions = [(Maybe Label, MInstruction)] -> [(Maybe Label, MInstruction)]
invertInstructions = reverse . map (second invertInstruction . first (fmap (++ "_i"))

instance Show Register where
show (Reg r) = ":" ++ show r

instance Show Instruction where
show (ADDI r i) = unwords "[ADDI ", show r, show i]"
show (ANDX r1 r2 r3) = unwords "[ANDX ", show r1, show r2, show r3]"
show (ANDINX r1 r2 i) = unwords "[ANDINX ", show r1, show r2, show i]"
show (NORX r1 r2 r3) = unwords "[NORX ", show r1, show r2, show r3]"
show (NEG r) = unwords "[NEG ", show r]
show (ORIX r1 r2 r3) = unwords "[ORIX ", show r1, show r2, show r3]"
show (ORX r1 r2 r3) = unwords "[ORX ", show r1, show r2, show r3]"
show (NEG r) = unwords "[NEG ", show r]
}

instance Show Program where
show (GProg Macro) = GProg SourceCode
show (GInstr Macro) = GInstr SourceCode

instance Show Instruction where
show (ADDI r i) = unwords "[ADDI ", show r, show i]"
show (ANDX r1 r2 r3) = unwords "[ANDX ", show r1, show r2, show r3]"
show (ANDINX r1 r2 i) = unwords "[ANDINX ", show r1, show r2, show i]"
show (NORX r1 r2 r3) = unwords "[NORX ", show r1, show r2, show r3]"
show (NEG r) = unwords "[NEG ", show r]
show (ORIX r1 r2 r3) = unwords "[ORIX ", show r1, show r2, show r3]"
show (ORX r1 r2 r3) = unwords "[ORX ", show r1, show r2, show r3]"
show (NEG r) = unwords "[NEG ", show r]

instance Show Program where
show (GProg Macro) = GProg SourceCode
show (GInstr Macro) = GInstr SourceCode
show (EXCH r1 r2) = unwords ["EXCH ", show r1, show r2]
show (SWAPBR r) = unwords ["SWAPBR", show r]
show (RBRA l) = unwords ["RBRA ", l]
show START = "START 
show FINISH = "FINISH"
show (DATA i) = unwords ["DATA ", show i]
show (SUBI r i) = unwords ["ADDI ", show r, show $ -i"] --Expand pseudo

showProgram :: Program -> String
showProgram (GProg p) = ";; pendulum pal file\n" ++ intercalate "\n" (map showLine p)
where showLine (Nothing, i) = spaces 25 ++ show i
  showLine (Just l, i) = l ++ ":" ++ spaces (24 - length l) ++ show i
spaces :: (Int -> String)
spaces n = [1..n] >>= ": 
writeProgram :: String -> Program -> IO ()
writeProgram output p = writeFile output $ showProgram p
module Parser (parseString) where

import Control.Monad.Except
import Data.Functor.Identity
import Data.Bifunctor
import Text.Parsec
import Text.Parsec.String
import Text.Parsec.Expr
import Text.Parsec.Language
import qualified Text.Parsec.Token as Token
import Debug.Trace (trace, traceShow)
import AST

{- Language Definition -}

keywords :: [String]
keywords =
  "class",
  "inherits",
  "method",
  "call",
  "uncall",
  "construct",
  "destruct",
  "skip",
  "from",
  "do",
  "loop",
  "until",
  "int",
  "nil",
  "if",
  "then",
  "else",
  "fi",
  "local",
  "delocal",
  "new",
  "delete",
  "copy",
  "uncopy"

-- Operator precedence identical to C
operatorTable :: [(String, BinOp)]
operatorTable =
  [ (*", Mul), ("/", Div), ("%", Mod)],
  [ ("+", Add), ("-", Sub)],
  [ ("<", Lt), ("<=", Lte), (">", Gt), (">=" , Gte) ],
  [ ("==", Eq), ("!=", Neq) ],
  [ ("&&", And)],
  [ ("||", Or) ],
  [ ("&", BitAnd)],
  [ ("|", BitOr)],
  [ ("^", Xor)]

languageDef :: Token.LanguageDef st
languageDef =
  emptyDef {
    Token.commentLine = "//",
    Token.nestedComments = False,
    Token.identStart = letter,
Token.identLetter = alphaNum <|> oneOf "-_'",
Token.reservedOpNames = concatMap (map fst) operatorTable,
Token.reservedNames = keywords,
Token.caseSensitive = True

tokenParser :: Token.TokenParser st
tokenParser = Token.makeTokenParser languageDef

{-- Parser Primitives --}
identifier :: Parser String
identifier = Token.identifier tokenParser

arrElemIdentifier :: Parser (String, Expression)
arrElemIdentifier = do
  x <- identifier
  y <- brackets expression
  return (x, y)

anyIdentifier :: Parser (String, Maybe Expression)
anyIdentifier = do
  x <- identifier
  y <- optionMaybe $ brackets expression
  return (x, y)

reserved :: String -> Parser ()
reserved = Token.reserved tokenParser

reservedOp :: String -> Parser ()
reservedOp = Token.reservedOp tokenParser

integer :: Parser Integer
integer = Token.integer tokenParser

symbol :: String -> Parser String
symbol = Token.symbol tokenParser

parens :: Parser a -> Parser a
parens = Token.parens tokenParser

brackets :: Parser a -> Parser a
brackets = Token.brackets tokenParser

colon :: Parser String
colon = Token.colon tokenParser

commaSep :: Parser a -> Parser [a]
commaSep = Token.commaSep tokenParser

typeName :: Parser TypeName
typeName = identifier

arrayTypeName :: Parser (TypeName, Expression)
arrayTypeName = do
  x <- try typeName <|> string "int"
  y <- brackets expression
  return (x, y)

methodName :: Parser MethodName
methodName = identifier

constant :: Parser Expression
constant = Constant <$> integer

variable :: Parser Expression
variable = Variable <$> identifier

arrayElementVariable :: Parser Expression
arrayElementVariable = ArrayElement <$> arrElemIdentifier
nil :: Parser Expression
nil = Nil <$> reserved "nil"

expression :: Parser Expression
expression = buildExpressionParser opTable <$> constant <|> try arrayElementVariable <|>
variable <$> nil

where binop (t, op) = Infix (Binary op <$> reservedOp t) AssocLeft

opTable = (map . map) binop operatorTable

{- Statement Parsers --}
modOp :: Parser ModOp
modOp = ModAdd <$> symbol "+=" <|> ModSub <$> symbol "-=" <|> ModXor <$> symbol "^="

assign :: Parser Statement
assign = Assign <$> identifier <*> modOp <*> expression

assignArrElem :: Parser Statement
assignArrElem = AssignArrElem <$> arrElemIdentifier <*> modOp <*> expression

swap :: Parser Statement
swap = Swap <$> anyIdentifier <* symbol "<=>" <*> anyIdentifier

conditional :: Parser Statement
conditional =
reserved "if" >> Conditional <$> expression <* reserved "then" <*> block <* reserved "else" <*> block <* reserved "fi" <*> expression

loop :: Parser Statement
loop =
reserved "from" >> Loop <$> expression <* reserved "do" <*> block <|> reserved "loop" <|> reserved "until" <*> expression

localCall :: Parser Statement
localCall =
reserved "call" >> LocalCall <$> methodName <* parens (commaSep anyIdentifier)

localUncall :: Parser Statement
localUncall =
reserved "uncall" >> LocalUncall <$> methodName <* parens (commaSep anyIdentifier)

objectCall :: Parser Statement
objectCall =
reserved "call"
195  >> ObjectCall
196  <$> anyIdentifier
197  <$> colon
198  <$> colon
199  <$> methodName
200  <$> parens {commaSep anyIdentifier}
201
202 objectUncall :: Parser Statement
203 objectUncall =
204  reserved "uncall"
205  >> ObjectUncall
206  <$> anyIdentifier
207  <$> colon
208  <$> colon
209  <$> methodName
210  <$> parens {commaSep anyIdentifier}
211
212 objectConstruction :: Parser Statement
213 objectConstruction =
214  reserved "new"
215  >> ObjectConstruction
216  <$> typeName
217  <$> anyIdentifier
218
219 objectDestruction :: Parser Statement
220 objectDestruction =
221  reserved "delete"
222  >> ObjectDestruction
223  <$> typeName
224  <$> anyIdentifier
225
226 localBlock :: Parser Statement
227 localBlock =
228  reserved "local"
229  >> LocalBlock
230  <$> dataType
231  <$> identifier
232  <$> symbol "="
233  <$> expression
234  <$> block
235  <$> reserved "delocal"
236  <$> dataType
237  <$> identifier
238  <$> symbol "="
239  <$> expression
240
241 objectBlock :: Parser Statement
242 objectBlock =
243  reserved "construct"
244  >> ObjectBlock
245  <$> typeName
246  <$> identifier
247  <$> block
248  <$> reserved "destruct"
249  <$> identifier
250
251 skip :: Parser Statement
252 skip = Skip <$ reserved "skip"
253
254 copyReference :: Parser Statement
255 copyReference =
256  reserved "copy"
257  >> CopyReference
258  <$> dataType
259  <$> anyIdentifier
260  <$> anyIdentifier
unCopyReference :: Parser Statement
unCopyReference =
  reserved "uncopy"
  >> UnCopyReference
  <$> dataType
  <$> anyIdentifier
  <$> anyIdentifier
arrayConstruction :: Parser Statement
arrayConstruction =
  reserved "new"
  >> ArrayConstruction
  <$> arrayTypeName
  <$> identifier
arrayDestruction :: Parser Statement
arrayDestruction =
  reserved "delete"
  >> ArrayDestruction
  <$> arrayTypeName
  <$> identifier
statement :: Parser Statement
statement =
  try assign <|>
  try assignArrElem <|> swap
  <|> conditional
  <|> loop
  <|> try localCall
  <|> try localUncall
  <|> objectCall
  <|> objectUncall
  <|> localBlock
  <|> objectBlock
  <|> try arrayConstruction <|> objectConstruction
  <|> try arrayDestruction <|> objectDestruction
  <|> skip
  <|> copyReference
  <|> unCopyReference
block :: Parser [Statement]
block = many1 statement
{| Top Level Parsers |
dataType :: Parser DataType
dataType =
  try (IntegerArrayType <$> reserved "int" <* symbol "\[" <* symbol "]")
  <|> IntegerType <$> reserved "int"
  <|> try (ObjectArrayType <$> typeName <* symbol "\[" <* symbol "]")
  <|> ObjectType <$> typeName
variableDeclaration :: Parser VariableDeclaration
variableDeclaration = GDecl <$> dataType <*> identifier
methodDeclaration :: Parser MethodDeclaration
methodDeclaration =
  reserved "method"
  >> GMDec1
  <$> methodName
  <$> parens (commaSep variableDeclaration)
  <$> block
classDeclaration :: Parser ClassDeclaration
classDeclaration =
  reserved "class"
  >> GCDec1
327  <$> typeName
328  <$> optionMaybe (reserved "inherits" >> typeName)
329  <$> many variableDeclaration
330  <$> many1 methodDeclaration
331
332 program :: Parser Program
333 program = spaces >> GProg <$> many1 classDeclaration <*> eof
334
335 parseString :: String -> Except String Program
336 parseString s = ExceptT (Identity $ first show $ parse program "" s)
 module ClassAnalyzer

 ( classAnalysis
 , printCAState
 , CAState(..)
 )

 where

 import Data.List
 import Data.Maybe
 import Control.Monad
 import Control.Monad.Except
 import Control.Monad.State
 import Text.Pretty.Simple (pPrint)
 import Debug.Trace (trace, traceShow)
 import AST

 type Size = Integer

 -- | The Class Analyzer State consists of a list of classes, sizes, methods
 -- and a main class
 data CAState = CAState {
   classes :: [(TypeName, ClassDeclaration)],
   subClasses :: [(TypeName, [TypeName])],
   superClasses :: [(TypeName, [TypeName])],
   classSize :: [(TypeName, Size)],
   classMethods :: [(TypeName, [MethodDeclaration])],
   mainClass :: Maybe TypeName
 }

 deriving (Show, Eq)

 -- | The Class Analyzer monad
 newtype ClassAnalyzer a = ClassAnalyzer { runCA :: StateT CAState (Except String) a }

 deriving (Functor, Applicative, Monad, MonadState CAState, MonadError String)

 -- | Initializes the Class Analyzer State with empty lists and Nothing for the mainClass
 initialState :: CAState

 initialState = CAState {
   classes = [],
   subClasses = [],
   superClasses = [],
   classSize = [],
   classMethods = [],
   mainClass = Nothing
 }

 -- | Returns a class from the Class Analyzer State if passed typename matches
 getClass :: TypeName -> ClassAnalyzer ClassDeclaration

 getClass n = gets classes >>= \cs ->
   case lookup n cs of
   (Just c) -> return c
   Nothing -> throwError $ "$ICE: Unknown class " ++ n

 -- | Returns the base class inherited from

 getBaseClass :: TypeName -> ClassAnalyzer (Maybe TypeName)

 getBaseClass n = getClass n >>= getBase

 where getBase (GCDecl _ b _ _) = return b

 -- | Throws error if class is defined multiple times

 checkDuplicateClasses :: ClassDeclaration -> ClassAnalyzer ()

 checkDuplicateClasses (GCDecl n _ _) = gets classes >>= \cs ->

when (count cs > 1) (throwError $ "Multiple definitions of class " ++ n)

where count = length . filter ((== n) . fst)

-- | Ensures legal inheritance
checkBaseClass :: ClassDeclaration -> ClassAnalyzer ()
checkBaseClass (GCDecl _ Nothing _ _) = return ()
checkBaseClass (GCDecl n (Just b) _ _) =
do when (n == b) (throwError $ "Class " ++ n ++ " cannot inherit from itself")
cs <=< gets classes
when (isNothing $ lookup b cs) (throwError $ "Class " ++ n ++ " cannot inherit from unknown class " ++ b)

-- | Checks duplicated field declarations
checkDuplicateFields :: ClassDeclaration -> ClassAnalyzer ()
checkDuplicateFields (GCDecl n _ fs _) =
mapM_ checkField fs

where count v = length . filter (\(GDecl _ v') -> v' == v) $ fs
checkField (GDecl _ v) =
when (count v > 1) (throwError $ "Multiple declarations of field " ++ v ++ " in class " ++ n)

-- | Checks duplicated method declaration in classes
checkDuplicateMethods :: ClassDeclaration -> ClassAnalyzer ()
checkDuplicateMethods (GCDecl n _ _ ms) =
mapM_ checkMethod ms'

where ms' = map (\(GMDecl n' _ _) -> n') ms
count m = length . filter (== m) $ ms'
checkMethod m =
when (count m > 1) (throwError $ "Multiple definitions of method " ++ m ++ " in class " ++ n)

-- | Checks cyclic inheritance
checkCyclicInheritance :: ClassDeclaration -> ClassAnalyzer ()
checkCyclicInheritance (GCDecl _ Nothing _ _) = return ()
checkCyclicInheritance (GCDecl n b _ _) = checkInheritance b [n]

where checkInheritance Nothing _ = return ()
checkInheritance (Just b') visited =
do when (b' `elem` visited) (throwError $ "Cyclic inheritance involving class " ++ n)
next <- getBaseClass b'
checkInheritance next (b' : visited)

-- | Sets the main class in the Class Analyzer State
setMainClass :: ClassDeclaration -> ClassAnalyzer ()
setMainClass (GCDecl n _ _ ms) =
when ("main" `elem` ms') (gets mainClass >>= set)

where
ms' = map (\(GMDecl n' _ _) -> n') ms
set (Just m) = throwError $ "Method main already defined in class " ++ m ++ " but redefined in class " ++ n
set Nothing = modify $ \s -> s {mainClass = Just n}

-- | Adds classes to the state
setClasses :: ClassDeclaration -> ClassAnalyzer ()
setClasses c@(GCDecl n _ _ _) = modify $ \s -> s {classes = (n, c) : classes s}

-- | Add subclasses to the state
setSubClasses :: ClassDeclaration -> ClassAnalyzer ()
setSubClasses (GCDecl n b _ _) = modify (\s -> s {subClasses = (n, []) : subClasses s}) >>
addSubClass n b

-- | Adds a subclass to the list of subclasses
addSubClass :: TypeName -> Maybe TypeName -> ClassAnalyzer ()

-- | Sets super classes in the state
setSuperClasses :: ClassDeclaration -> ClassAnalyzer ()
124 setSuperClasses (GCDecl n _ _ _) = gets subClasses >>= \sc ->
125     modify $ \s -> s { superClasses = (n, map fst $ filter (\( _, sub) -> n 'elem' sub) sc) :
126     superClasses s }
127
128    -- | Returns the nearest 2^n as size for given class
129    getClassSize :: ClassDeclaration -> ClassAnalyzer Size
130    getClassSize (GCDecl _ Nothing _ _ _) =
131    getSubClasses >>= \\sq ->
132    return $ 2 ^ (ceiling :: Double -> Integer) (logBase 2 (\sz ->
133    genericLength sz +
134    fromIntegral sz))
135
136    -- | Set class size in state
137    setClassSize :: ClassDeclaration -> ClassAnalyzer ()
138    setClassSize c@(GCDecl n _ _ _) =
139
140    -- | Performs Class Analysis on the program
141    classAnalysis :: Program -> Except String (Program, CAState)
142    classAnalysis p = runStateT (runCA $ caProgram p) initialState
143
144    -- | Pretty prints the Class Analyzer State
145    printCAState :: (Program, CAState) -> IO ()
146    printCAState (_, s) = pPrint s
ScopeAnalyzer.hs

{-# LANGUAGE GeneralizedNewtypeDeriving, FlexibleContexts #-}

module ScopeAnalyzer
  ( scopeAnalysis
  , printSAState
  , SAState(..)
  ) where

import Data.Maybe
import Data.List
import Data.Typeable
import Control.Monad.State
import Control.Monad.Except
import Debug.Trace (trace, traceShow)
import Text.Pretty.Simple (pPrint)
import AST
import ClassAnalyzer

data SAState =
  SAState {
    symbolIndex :: SIdentifier,
    symbolTable :: SymbolTable,
    scopeStack :: [Scope],
    virtualTables :: [(TypeName, [SIdentifier])],
    caState :: CAState,
    mainMethod :: SIdentifier
  }
  deriving (Show, Eq)

newtype ScopeAnalyzer a = ScopeAnalyzer { runSA :: StateT SAState (Except String) a }
  deriving (Functor, Applicative, Monad, MonadState SAState, MonadError String)

initialState :: CAState -> SAState
initialState s = SAState { symbolIndex = 0, symbolTable = [], scopeStack = [], virtualTables = [], caState = s, mainMethod = 0 }

enterScope :: ScopeAnalyzer ()
enterScope = modify $ \s -> s { scopeStack = [] : scopeStack s }

leaveScope :: ScopeAnalyzer ()
leaveScope = modify $ \s -> s { scopeStack = drop 1 $ scopeStack s }

topScope :: ScopeAnalyzer Scope
topScope = gets scopeStack >>= \ss ->
  case ss of
    [] -> return s
    _ -> throwError "ICE: Empty scope stack"

addToScope :: (Identifier, SIdentifier) -> ScopeAnalyzer ()
addToScope b =
  do ts <- topScope
     modify $ \s -> s { scopeStack = (b : ts) : drop 1 $ scopeStack s }

saInsert :: Symbol -> Identifier -> ScopeAnalyzer SIdentifier
saInsert sym n =
do ts <- topScope
  when (isJust $ lookup n ts) (throwError $ "Redeclaration of symbol: " ++ n)
  i <- gets symbolIndex
  modify $ \s -> s { symbolTable = (i, sym) : symbolTable s, symbolIndex = 1 + i }
  addToScope (n, i)
  return i

-- | Looks up an identifier in the scope
saLookup :: Identifier -> ScopeAnalyzer SIdentifier
saLookup n = gets scopeStack >>= \ss ->
  case listToMaybe $ mapMaybe (lookup n) ss of
    Nothing -> throwError $ "Undeclared symbol: " ++ n
    Just i -> return i

-- | Scope Analyses Expressions
saExpression :: Expression -> ScopeAnalyzer SExpression
saExpression (Constant v) = pure $ Constant v
saExpression (Variable n) = Variable <$> saLookup n
saExpression Nil = pure Nil
saExpression (ArrayElement (n, e)) =
  do
    n' <- saLookup n
    e' <- saExpression e
    return $ ArrayElement (n', e')
  saExpression (Binary binop e1 e2) =
    Binary binop
    <$> saExpression e1
    <*> saExpression e2

-- | Scope Analyses Statements
saStatement :: Statement -> ScopeAnalyzer SStatement
saStatement s =
  case s of
    (Assign n modop e) ->
      when (elem n $ var e) (throwError "Irreversible variable assignment")
      >> Assign
      <$> saLookup n
      <*> pure modop
      <*> saExpression e
    (AssignArrElem (n, e1) modop e2) ->
      when (elem (n, e1) $ varArr e2) (throwError "Irreversible variable assignment")
      >> AssignArrElem
      <$> saArrayCell n e1
      <*> pure modop
      <*> saExpression e2
    (Swap (n1, e1) (n2, e2)) ->
      Swap
      <$> maybeArrayCell n1 e1
      <*> maybeArrayCell n2 e2
    (Conditional e1 s1 s2 e2) ->
      Conditional
      <$> saExpression e1
      <*> mapM saStatement s1
      <*> mapM saStatement s2
      <*> saExpression e2
    (Loop e1 s1 s2 e2) ->
      Loop
      <$> saExpression e1
      <*> mapM saStatement s1
      <*> mapM saStatement s2
      <*> saExpression e2
    (LocalBlock t n el stmt e2) ->
do e1' <- saExpression e1
    enterScope
    n' <- saInsert (LocalVariable t n) n
    stmt' <- mapM saStatement stmt
    leaveScope
    e2' <- saExpression e2
    return $ LocalBlock t n' e1' stmt' e2'

(LocalCall m args) ->
    LocalCall
    <$> saLookup m
    <$> localCall m args

(LocalUncall m args) ->
    LocalUncall
    <$> saLookup m
    <$> localCall m args

(ObjectCall (o, e) m args) ->
do when (args /= nub args || (o, e) `elem` args) (throwError $ "Irreversible invocation of method " ++ m)
    >> ObjectCall
    <$> maybeArrayCell o e
    <$> pure m
    <$> saArgs args

(ObjectUncall (o, e) m args) ->
do when (args /= nub args || (o, e) `elem` args) (throwError $ "Irreversible invocation of method " ++ m)
    >> ObjectUncall
    <$> maybeArrayCell o e
    <$> pure m
    <$> saArgs args

(ObjectConstruction tp (n, e)) ->
    ObjectConstruction
    <$> pure tp
    <$> maybeArrayCell n e

(ObjectDestruction tp (n, e)) ->
    ObjectDestruction
    <$> pure tp
    <$> maybeArrayCell n e

(ObjectBlock tp n stmt) ->
do enterScope
    n' <- saInsert (LocalVariable (ObjectType tp) n) n
    stmt' <- mapM saStatement stmt
    leaveScope
    return $ ObjectBlock tp n' stmt'

Skip -> pure Skip

(CopyReference tp (n, e1) (m, e2)) ->
    CopyReference
    <$> pure tp
    <$> maybeArrayCell n e1
    <$> maybeArrayCell m e2

(UnCopyReference tp (n, e1) (m, e2)) ->
    UnCopyReference
    <$> pure tp
    <$> maybeArrayCell n e1
    <$> maybeArrayCell m e2

(ArrayConstruction (tp, e) n) ->
do n' <- saLookup n
  e' <- saExpression e
  return $ ArrayConstruction (tp, e') n'

(ArrayDestruction (tp, e) n) ->
do n' <- saLookup n
  e' <- saExpression e
  return $ ArrayDestruction (tp, e') n'

where var (Variable n) = [n]
  var (Binary _ e1 e2) = var e1 ++ var e2
  var _ = []
  varArr (ArrayElement (n, e)) = [(n, e)]
  varArr _ = []

isCF ClassField{} = True
isCF _ = False

rlookup = flip lookup

localCall :: MethodName -> [(Identifier, Maybe Expression)] -> ScopeAnalyzer [(SIdentifier, Maybe SExpression)]
localCall m args =
do when (args /= nub args) (throwError $ "Irreversible invocation of method " ++ m)
  args' <- saArgs args
st <- gets symbolTable
when (any isCF $ mapMaybe (rlookup st . fst) args') (throwError $ "Irreversible invocation of method " ++ m)
  return args'

saArgs :: [(Identifier, Maybe Expression)] -> ScopeAnalyzer [(SIdentifier, Maybe SExpression)]
saArgs args =
do (ns, es) <- pure $ unzip args
  ns' <- mapM saLookup ns
  es' <- mapM (mapM saExpression) es
  return $ zip ns' es'

maybeArrayCell :: Identifier -> Maybe Expression -> ScopeAnalyzer (SIdentifier, Maybe SExpression)
maybeArrayCell n e =
do n' <- saLookup n
  e' <- saExpression e
  return (n', e')

saArrayCell :: Identifier -> Expression -> ScopeAnalyzer (SIdentifier, SExpression)
saArrayCell n e =
do n' <- saLookup n
  e' <- saExpression e
  return (n', e')

-- | Set the main method in the Scope Analyzer state
setMainMethod :: SIdentifier -> ScopeAnalyzer ()
setMainMethod i = modify $ \s -> s { mainMethod = i }

-- | Scope Analyses Methods
saMethod :: (TypeName, MethodDeclaration) -> ScopeAnalyzer (TypeName, SMethodDeclaration)
saMethod (t, GMDecl m ps body) =
do m' <- saLookup m
  when (m == "main") (setMainMethod m')
enterScope
ps' <- mapM insertMethodParameter ps
  body' <- mapM saStatement body
leaveScope
return (t, GMDecl m' ps' body')
where insertMethodParameter (GDecl tp n) = GDecl tp <$> saInsert (MethodParameter tp n) n

-- | Returns subclasses for a given type name
getSubClasses :: TypeName -> ScopeAnalyzer [ClassDeclaration]
getSubClasses n =
do cs <- gets $ classes . caState
  sc <- gets $ subClasses . caState
  case lookup n sc of
    Nothing -> throwError $ "ICE: Unknown class " ++ n
    (Just sc') -> return $ mapMaybe (rlookup cs) sc'
  where rlookup = flip lookup

-- | Returns method name at given index
getMethodName :: SIdentifier -> ScopeAnalyzer (SIdentifier, MethodName)
getMethodName i = gets symbolTable >>= \st ->
do lookup i st of
  (Just (Method _ m)) -> return (i, m)
_ -> throwError $ "ICE: Invalid method index " ++ show i

-- | Prefixes the virtual table
prefixVtable :: [(SIdentifier, MethodName)] -> (SIdentifier, MethodName) -> [(SIdentifier, MethodName)]
prefixVtable [] m' = [m']
prefixVtable (m:ms) m' = if comp m m' then m':ms else m : prefixVtable ms m'
where comp (_, n) (_, n') = n == n'

-- | Scope Analyses a passed class
saClass :: Offset -> [SIdentifier] -> ClassDeclaration -> ScopeAnalyzer [(TypeName, SMethodDeclaration)]
saClass offset pids (GCDecl c _ fs ms) =
do enterScope
  mapM_ insertClassField $ zip [offset..] fs
  m1 <- mapM getMethodName pids
  m2 <- mapM insertMethod ms
  let m3 = map fst $ foldl prefixVtable m1 m2
  ms' <- getSubClasses c
  ms'' <- mapM saMethod $ zip (repeat c) ms
  leaveScope
  return $ ms'' ++ ms'
  where insertClassField (o, GDecl tl n) = saInsert (ClassField tl n c o) n
        insertMethod (GMDecl n ps _) = saInsert (Method (map getType ps) n) n >>=
                        getMethodName
                        getType (GDecl tl n) = tp

-- | Analyses Programs
saProgram :: Program -> ScopeAnalyzer SProgram
saProgram (GProg cs) = concat <$> mapM (saClass 2 []) cs

-- | Performs scope analysis on the entire program
scopeAnalysis :: (Program, CAState) -> Except String (SProgram, SAState)
scopeAnalysis (p, s) = runStateT (runSA $ saProgram p) $ initialSA s

-- | Pretty prints the current Scope Analysis State Monad
printSAState :: (Show a, MonadIO m) => (t, a) -> m ()
printSAState (_, s) = pPrint s
TypeChecker.hs

{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module TypeChecker (typeCheck) where

import Data.List
import Data.Maybe
import Control.Monad.Reader
import Control.Monad.Except
import Control.Exception
import Debug.Trace (trace, traceShow)
import AST
import ClassAnalyzer
import ScopeAnalyzer

newtype TypeChecker a = TypeChecker { runTC :: ReaderT SAState (Except String) a }
  deriving (Functor, Applicative, Monad, MonadReader SAState, MonadError String)

getType :: SIdentifier -> TypeChecker DataType
getType i = asks symbolTable >>= \\st ->
  case lookup i st of
    (Just (LocalVariable t _)) -> return t
    (Just (ClassField t _ _ _)) -> return t
    (Just (MethodParameter t _)) -> return t
    _ -> throwError $ "ICE: Invalid index " ++ show i

getParameterTypes :: SIdentifier -> TypeChecker [DataType]
getParameterTypes i = asks symbolTable >>= \\st ->
  case lookup i st of
    (Just (Method ps _)) -> return ps
    _ -> throwError $ "ICE: Invalid index " ++ show i

expectType :: DataType -> DataType -> TypeChecker ()
expectType t1 t2 = unless (t1 == t2) (throwError $ "Expected type: " ++ show t1 ++ "\nActual type: " ++ show t2)

getClassMethods :: TypeName -> TypeChecker [MethodDeclaration]
getClassMethods n = asks (classMethods . caState) >>= \\cm ->
  case lookup n cm of
    Nothing -> throwError $ "ICE: Unknown class " ++ n
    (Just ms) -> return ms

getDynamicParameterTypes :: TypeName -> MethodName -> TypeChecker [DataType]
getDynamicParameterTypes n m = getClassMethods n >>= \\ms ->
  case find (\(GMDecl m' _ _) -> m == m')) ms of
    Nothing -> throwError $ "Class " ++ n ++ " does not support method " ++ m
    (Just (GMDecl _ ps _)) -> return $ map (\(GDecl tp _) -> tp) ps

getArrayType :: DataType -> DataType
getArrayType tp = case tp of
  IntegerArrayType -> IntegerType
  ObjectArrayType t -> ObjectType t

checkCall :: [(SIdentifier, Maybe SExpression)] -> [DataType] -> TypeChecker ()
checkCall args ps =
  when (la /= lp) (throwError err)
  >>= mapM (mapM tcExpression . snd) args
  >>= mapM (getType . fst) args
  >>= \as -> mapM_ checkArgument (zip as ps)
  where la = length args
        lp = length ps

Appendix B  ROOPLPPC Source Code  102 of 231
err = "Passed " ++ show la ++ " argument(s) to method expecting " ++ show lp ++ " argument(s)"

checkArgument :: (DataType, DataType) -> TypeChecker ()
checkArgument (ObjectType ca, ObjectType cp) = asks (superClasses . caState) >>= \sc ->
  unless (ca == cp || maybe False (elem cp) (lookup ca sc)) (throwError $ "Class " ++ ca ++ " not a subtype of class " ++ cp)
checkArgument (ObjectType ca, ObjectArrayNodeType cp) = asks (superClasses . caState) >>= \sc ->
  unless (ca == cp || maybe False (elem cp) (lookup ca sc)) (throwError $ "Class " ++ ca ++ " not a subtype of class " ++ cp)
checkArgument (ObjectArrayNodeType ca, ObjectType cp) = asks (superClasses . caState) >>= \sc ->
  unless (ca == cp || maybe False (elem cp) (lookup ca sc)) (throwError $ "Class " ++ ca ++ " not a subtype of class " ++ cp)
checkArgument (IntegerArrayType, tp) = expectType (getArrayType IntegerArrayType) tp
checkArgument (ta, IntegerArrayType) = expectType (getArrayType IntegerArrayType) ta
checkArgument (ta, tp) = expectType tp ta

tcExpression :: SExpression -> TypeChecker DataType

tcExpression (Constant _) = pure IntegerType
tcExpression (Variable n) = getType n
tcExpression Nil = pure NilType

tcExpression (ArrayElement (n, e)) =
do t <- getType n
e' <- tcExpression e
expectType ArrayType t
tcExpression (Binary binop e1 e2) =
  | binop == Eq || binop == Neq =
do t1 <- tcExpression e1
t2 <- tcExpression e2
expectType t1 t2
pure IntegerType
  | otherwise =
do t1 <- tcExpression e1
t2 <- tcExpression e2
expectType t1 IntegerType
expectType t2 IntegerType
pure IntegerType

tcStatement :: SStatement -> TypeChecker ()
tcStatement s =
  case s of
    Assign n _ e ->
      getType n
        >>= expectType IntegerType
        >> tcExpression e
        >>= expectType IntegerType
    AssignArrElem (n, e1) _ e2) ->
      getType n
        >>= expectType IntegerArrayType
        >> tcExpression e1
        >>= expectType IntegerType
        >> tcExpression e2
        >>= expectType IntegerType
    Swap (n1, e1) (n2, e2)) ->
      do t1 <- getType n1
t2 <- getType n2
        if isNothing e1 /= isNothing e2
        then catchError (checkArgument (t1, t2)) (\_ -> checkArgument (t2, t1))
        else expectType (if isNothing e1 then t1 else getArrayType t1) (if isNothing e2 then t2 else getArrayType t2)
    Conditional e1 s1 s2 e2) ->
tcExpression e1
>>= expectType IntegerType
>> mapM_ tcStatement s1
>> tcExpression e2
>>= expectType IntegerType
(Loop e1 s1 s2 e2) ->
tcExpression e1.
>>= expectType IntegerType
>> mapM_ tcStatement s1
>> mapM_ tcStatement s2
>> tcExpression e2
>>= expectType IntegerType
(ObjectBlock _ _ stmt) ->
mapM_ tcStatement stmt
(LocalBlock t n el stmt e2) ->
getType n
>> tcExpression el
>>= expectType (if t == IntegerType then IntegerType else NilType)
>> mapM_ tcStatement stmt
>> tcExpression e2
>>= expectType (if t == IntegerType then IntegerType else NilType)
(LocalCall m args) ->
getParameterTypes m
>>= checkCall args
(LocalUncall m args) ->
getParameterTypes m
>>= checkCall args
(ObjectCall (o, e) m args) ->
do t <- getType o
e' <- mapM_ tcExpression e
case t of
  (ObjectType tn) -> getDynamicParameterTypes tn m >>= checkCall args
  (ObjectArrayType tn) ->
case e' of
    Nothing -> throwError $ "Non-object type " ++ show t ++ " does not support method invocation"
    _ -> getDynamicParameterTypes tn m >>= checkCall args
  _ -> throwError $ "Non-object type " ++ show t ++ " does not support method invocation"
(ObjectUncall (o, e) m args) ->
do t <- getType o
e' <- mapM_ tcExpression e
case t of
  (ObjectType tn) -> getDynamicParameterTypes tn m >>= checkCall args
  (ObjectArrayType tn) ->
case e' of
    Nothing -> throwError $ "Non-object type " ++ show t ++ " does not support method invocation"
    _ -> getDynamicParameterTypes tn m >>= checkCall args
  _ -> throwError $ "Non-object type " ++ show t ++ " does not support method invocation"
Skip -> pure ()
(ObjectConstruction tp (n, e)) ->
do t <- getType n
e' <- mapM_ tcExpression e
case e' of
Nothing -> expectType t (ObjectType tp)

_ -> checkArgument (ObjectType tp, t)

(ObjectDestruction tp (n, e)) ->
do t <- getType n
_ <- mapM tcExpression e

case t of
(ObjectType _) -> expectType t (ObjectType tp)
(ObjectArrayType _) -> checkArgument (ObjectType tp, t)
_ -> throwError $ "Expected type: " ++ show (ObjectType tp) ++ " Actual type: " ++ show t

-- Allow copying with a copy type
CopyReference _ (n, e1) (m, e2) ->
do t1 <- getType n
t2 <- getType m
e1' <- mapM tcExpression e1
e2' <- mapM tcExpression e2

when (t1 == IntegerType || t2 == IntegerType) (throwError "Integer types does not support reference copying")
if isNothing e1 /= isNothing e2
then catchError (checkArgument (t1, t2)) (\_ -> checkArgument (t2, t1))
else expectType (if isNothing e1 then t1 else getArrayType t1) (if isNothing e2 then t2 else getArrayType t2)

-- Allow uncopying with two identical copies
UnCopyReference _ (n, e1) (m, e2) ->
do t1 <- getType n
t2 <- getType m
e1' <- mapM tcExpression e1
e2' <- mapM tcExpression e2

when (t1 == IntegerType || t2 == IntegerType) (throwError "Integer types does not support reference copying")
if isNothing e1 /= isNothing e2
then catchError (checkArgument (t1, t2)) (\_ -> checkArgument (t2, t1))
else expectType (if isNothing e1 then t1 else getArrayType t1) (if isNothing e2 then t2 else getArrayType t2)

(ArrayConstruction (tp, e) n) ->
do t <- getType n
_ <- tcExpression e
case tp of
"int" -> expectType t IntegerArrayType
_ -> expectType t (ObjectArrayType tp)

(ArrayDestruction (tp, e) n) ->
do t <- getType n
_ <- tcExpression e
case tp of
"int" -> expectType t IntegerArrayType
_ -> checkArgument (ObjectArrayType tp, t)

getMethodName :: SIdentifier -> TypeChecker Identifier
getMethodName i = asks symbolTable >>= \st ->
case lookup i st of
(Just (Method _ n)) -> return n
_ -> throwError $ "ICE: Invalid index " ++ show i

tcMethod :: (TypeName, SMethodDeclaration) -> TypeChecker ()
tcMethod (_, GMDecl _ [] body) = mapM_ tcStatement body
tcMethod (_, GMDecl i (_:_) body) = getMethodName i >>= \n ->
  when (n == "main") (throwError "Method main has invalid signature")
  >> mapM_ tcStatement body

tcProgram :: SProgram -> TypeChecker (SProgram, SAState)
tcProgram p = (,) p <$> (mapM_ tcMethod p >> ask)
typeCheck :: (SProgram, SAState) -> Except String (SProgram, SAState)
typeCheck (p, s) = runReaderT (runTC $ tcProgram p) s
module CodeGenerator

generatePISA, showPISAProgram

import Data.List

import Control.Arrow
import Control.Monad.Except
import Control.Monad.State

import Debug.Trace (trace, traceShow)

import Text.Pretty.Simple (pPrint)

import AST
import ClassAnalyzer
import PISA
import ScopeAnalyzer

{-# ANN module "HLint: ignore Reduce duplication" #-}

-- | Register containing 0
registerZero = Reg 0

-- | Register containing Stack pointer
registerSP = Reg 1

-- | Register holding 'this'
registerThis = Reg 3

-- | Register containing Heap pointer
registerFLPs = Reg 4
registerHP :: Register
registerHP = Reg 5

-- | Pushes a new register to the register stack
pushRegister :: SIdentifier -> CodeGenerator Register
pushRegister i = do ri <- gets registerIndex
      modify $ \s -> s { registerIndex = 1 + ri, registerStack = (i, Reg ri) : registerStack s }
      return $ Reg ri

-- | Pop a register from the register stack
popRegister :: CodeGenerator ()
popRegister = modify $ \s -> s { registerIndex = (-1) + registerIndex s, registerStack = drop 1 $ registerStack s }

-- | Reserve a tmp register
tempRegister :: CodeGenerator Register
tempRegister = do ri <- gets registerIndex
      modify $ \s -> s { registerIndex = 1 + ri }
      return $ Reg ri

-- | Clear reserved tmp register
popTempRegister :: CodeGenerator ()
popTempRegister = modify $ \s -> s { registerIndex = (-1) + registerIndex s }

-- | Lookup register of given identifier
lookupRegister :: SIdentifier -> CodeGenerator Register
lookupRegister i = gets registerStack >>= \rs ->
      case lookup i rs of
        Nothing -> throwError $ "ICE: No register reserved for index " ++ show i
        (Just r) -> return r

-- | Returns the method name of a valid method identifier
getMethodName :: SIdentifier -> CodeGenerator MethodName
getMethodName i = gets (symbolTable . saState) >>= \st ->
      case lookup i st of
        (Just (Method _ n)) -> return n
        _ -> throwError $ "ICE: Invalid method index " ++ show i

-- | Inserts a unique method label in the label table for a given method identifier
insertMethodLabel :: SIdentifier -> CodeGenerator ()
insertMethodLabel m = do
      n <- getMethodName m
      i <- gets labelIndex
      modify $ \s -> s { labelIndex = 1 + i, labelTable = (m, "l_" ++ n ++ ":" ++ show i) : labelTable s }

-- | Returns the Method label for a method identifier
getMethodLabel :: SIdentifier -> CodeGenerator Label
getMethodLabel m = gets labelTable >>= \lt ->
      case lookup m lt of
        (Just l) -> return l
        Nothing -> insertMethodLabel m >> getMethodLabel m

-- | Returns a unique label by appending the label index to a passed label type
getUniqueLabel :: Label -> CodeGenerator Label
getUniqueLabel l = do
      i <- gets labelIndex
      modify $ \s -> s { labelIndex = 1 + i }
      return $ l ++ "_" ++ show i

-- | Loads the address to the variable of a given identifier
loadVariableAddress :: SIdentifier -> CodeGenerator (Register, [Maybe Label, MInstruction]), CodeGenerator ()
loadVariableAddress n = gets (symbolTable . saState) >>= \st ->
case lookup n st of
  (Just (ClassField _ _ _ o)) -> tempRegister >>=  ->
    return (r, [(Nothing, ADD r rt r2)],
      (Nothing, ADDI r $ Immediate o), popTempRegister)
  (Just (LocalVariable _ _)) -> lookupRegister n >>=  ->
    return (r, [],
      return ()
  (Just (MethodParameter _ _)) -> lookupRegister n >>=  ->
    return (r, [],
      return ()
  _ -> throwError $ "ICE: Invalid variable index " ++ show n

-- | Returns the value of a variable of given identifier
loadVariableValue :: SIdentifier -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  CodeGenerator ())
loadVariableValue n =
  do (ra, la, ua) <- loadVariableAddress n
  rv <- tempRegister
  return (rv, la ++ [(Nothing, EXCH rv ra)] ++ invertInstructions la, popTempRegister >>
    ua)

-- | Returns address an array element
loadArrayElementVariableAddress :: SIdentifier -> SExpression -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  CodeGenerator ())
loadArrayElementVariableAddress n e =
  do (ra, le, ue) <- cgExpression e
  (re, le, ue) <- cgExpression e
  rv <- tempRegister
  rt <- tempRegister
  return (rv, la ++ le ++ [(Nothing, EXCH rt ra), (Nothing, XOR rv rt), (Nothing, EXCH rt ra), (Nothing, ADDI rv ArrayElementOffset), (Nothing, ADD rv re)] ++
    invertInstructions (la ++ le), popTempRegister >> popTempRegister >> ue >> ua)

-- | Returns the value of an array element
loadArrayElementVariableValue :: SIdentifier -> SExpression -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  CodeGenerator ())
loadArrayElementVariableValue n e =
  do (ra, le, ue) <- loadArrayElementVariableAddress n e
  rv <- tempRegister
  return (rv, la ++ [(Nothing, EXCH rv ra)] ++ invertInstructions la, popTempRegister >>
    ua)

-- Â | Returns pointer to free list at given index
loadFreeListAddress :: Register -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  CodeGenerator ())
loadFreeListAddress index = tempRegister >>=  ->
  return (rt, [(Nothing, XOR rt registerFLPs), (Nothing, ADD rt index)], popTempRegister)

-- |Â Returns a copy of the pointer to the head of the free list at the given register
loadHeadAtFreeList :: Register -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  CodeGenerator ())
loadHeadAtFreeList rFreeList =
  do rv <- tempRegister
     rt <- tempRegister
     let copyAddress = [(Nothing, EXCH rt rFreeList),
       (Nothing, XOR rv rt), (Nothing, EXCH rt rFreeList)]
    return (rv, copyAddress, popTempRegister >> popTempRegister)

-- Â | Code generation for binary operators
cgBinOp :: BinOp -> Register -> Register -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  CodeGenerator ())
cgBinOp Add r1 r2 = tempRegister >>=  ->
  return (rt, [(Nothing, XOR rt r1), (Nothing, ADD rt r2)],
    popTempRegister)
cgBinOp Sub r1 r2 = tempRegister >>=  ->
  return (rt, [(Nothing, XOR rt r1), (Nothing, ADD rt r2)],
    popTempRegister)
cgBinOp Xor r1 r2 = tempRegister >>=  ->
  return (rt, [(Nothing, XOR rt r1), (Nothing, XOR rt r2)],
    popTempRegister)
cgBinOp BitAnd r1 r2 = tempRegister >>=  ->
  return (rt, [(Nothing, ANDX rt r1 r2)],
    popTempRegister)
cgBinOp BitOr r1 r2 = tempRegister >>= \rt -> return (rt, [(Nothing, ORX rt r1 r2)], popTempRegister)
cgBinOp Lt r1 r2 =
do rt <- tempRegister
rc <- tempRegister
l_top <- getUniqueLabel "cmp_top"
l_bot <- getUniqueLabel "cmp_bot"
let cmp = [(Nothing, XOR rt r1),
          (Nothing, SUB rt r2),
          (Just l_top, BGEZ rt l_bot),
          (Nothing, XORI rc $ Immediate 1),
          (Just l_bot, BGEZ rt l_top)]
return (rc, cmp, popTempRegister >> popTempRegister)
cgBinOp Gt r1 r2 =
do rt <- tempRegister
rc <- tempRegister
l_top <- getUniqueLabel "cmp_top"
l_bot <- getUniqueLabel "cmp_bot"
let cmp = [(Nothing, XOR rt r1),
          (Nothing, SUB rt r2),
          (Just l_top, BLEZ rt l_bot),
          (Nothing, XORI rc $ Immediate 1),
          (Just l_bot, BLEZ rt l_top)]
return (rc, cmp, popTempRegister >> popTempRegister)
cgBinOp Eq r1 r2 =
do rt <- tempRegister
l_top <- getUniqueLabel "cmp_top"
l_bot <- getUniqueLabel "cmp_bot"
let cmp = [(Just l_top, BNE r1 r2 l_bot),
          (Nothing, XORI rt $ Immediate 1),
          (Just l_bot, BNE r1 r2 l_top)]
return (rt, cmp, popTempRegister)
cgBinOp Neq r1 r2 =
do rt <- tempRegister
l_top <- getUniqueLabel "cmp_top"
l_bot <- getUniqueLabel "cmp_bot"
let cmp = [(Just l_top, BEQ r1 r2 l_bot),
          (Nothing, XORI rt $ Immediate 1),
          (Just l_bot, BEQ r1 r2 l_top)]
return (rt, cmp, popTempRegister)
cgBinOp Lte r1 r2 =
do rt <- tempRegister
rc <- tempRegister
l_top <- getUniqueLabel "cmp_top"
l_bot <- getUniqueLabel "cmp_bot"
let cmp = [(Nothing, XOR rt r1),
          (Nothing, SUB rt r2),
          (Just l_top, BGTZ rt l_bot),
          (Nothing, XORI rc $ Immediate 1),
          (Just l_bot, BGTZ rt l_top)]
return (rc, cmp, popTempRegister >> popTempRegister)
cgBinOp Gte r1 r2 =
do rt <- tempRegister
rc <- tempRegister
l_top <- getUniqueLabel "cmp_top"
l_bot <- getUniqueLabel "cmp_bot"
let cmp = [(Nothing, XOR rt r1),
          (Nothing, SUB rt r2),
          (Just l_top, BLTZ rt l_bot),
          (Nothing, XORI rc $ Immediate 1),
          (Just l_bot, BLTZ rt l_top)]
return (rc, cmp, popTempRegister >> popTempRegister)
cgBinOp _ _ _ = throwError "ICE: Binary operator not implemented"

Appendix B  ROOLPPC Source Code  110 of 231
238 \textbf{cgExpression} :: \textbf{SExpression} -> \textbf{CodeGen generator} (\textbf{Register}, [\textbf{Maybe Label}, \textbf{MInstruction}]),
\textbf{CodeGen generator} ()

239 \textbf{cgExpression} \ (\textbf{Constant} \ n) = \textbf{tempRegister} \ >>= \ \textbf{\_} \to \textbf{return} \ [(\textbf{Nothing}, \textbf{XORI} \ \textbf{rt} \ $ \ \textbf{Immediate} \ n), \ \textbf{popTempRegister}]

240 \textbf{cgExpression} \ (\textbf{Variable} \ i) = \textbf{loadVariableValue} \ i

241 \textbf{cgExpression} \ (\textbf{ArrayElement} \ (n, e)) = \textbf{loadArrayElementVariableValue} \ n \ e

242 \textbf{cgExpression} \ (\textbf{Nil}) = \textbf{return} \ [(\textbf{registerZero}, [], \textbf{popTempRegister})]

243 \textbf{cgExpression} \ (\textbf{Binary} \ \textbf{op} \ e1 \ e2) =

\begin{verbatim}
  do \ (r1, l1, u1) <- \textbf{cgExpression} \ e1
        (r2, l2, u2) <- \textbf{cgExpression} \ e2
        (ro, lo, uo) <- \textbf{cgBinOp} \ \textbf{op} \ r1 \ r2
        \textbf{return} \ (ro, l1 ++ l2 ++ lo, uo >> u2 >> u1)
\end{verbatim}

250 \textbf{-- \ Code generation for binary expressions}

251 \textbf{cgBinaryExpression} :: \textbf{SExpression} -> \textbf{CodeGen generator} (\textbf{Register}, [\textbf{Maybe Label}, \textbf{MInstruction}]),
\textbf{CodeGen generator} ()

252 \textbf{cgBinaryExpression} \ e =

\begin{verbatim}
do \ (re, le, ue) <- \textbf{cgExpression} \ e
  \textbf{rt} <- \textbf{tempRegister}
  \textbf{l_top} <- \textbf{getUniqueLabel} \ "f_top"
  \textbf{l_bot} <- \textbf{getUniqueLabel} \ "f_bot"
  \textbf{let} \ flatten = [(\textbf{Just} \ l_top, \textbf{BEQ} \ re \ \textbf{registerZero} \ l_bot),
                        (\textbf{Nothing}, \textbf{XORI} \ \textbf{rt} \ $ \ \textbf{Immediate} \ 1),
                        (\textbf{Just} \ l_bot, \textbf{BEQ} \ re \ \textbf{registerZero} \ l_top)]
  \textbf{return} \ (\textbf{rt}, \textbf{le} ++ flatten, \textbf{popTempRegister} >> \textbf{ue})
\end{verbatim}

262 \textbf{-- \ Code generation for assignments}

263 \textbf{cgAssign} :: \textbf{SIdentifier} -> \textbf{ModOp} -> \textbf{SExpression} -> \textbf{CodeGen generator} [\textbf{Maybe Label}, \textbf{MInstruction}]

264 \textbf{cgAssign} \ n \ modop \ e =

\begin{verbatim}
do \ (rt, lt, ut) <- \textbf{loadVariableValue} \ n
  (re, le, ue) <- \textbf{cgExpression} \ e
  \textbf{ue} >> \textbf{ut}
  \textbf{return} \ (\textbf{lt} ++ \textbf{le} ++ [(\textbf{Nothing}, \textbf{cgModOp} \ \textbf{modop} \ \textbf{rt} \ \textbf{re})] ++ \textbf{invertInstructions} \ (\textbf{lt} ++ \textbf{le})
\end{verbatim}

274 \textbf{-- \ Code generation for assignments}

275 \textbf{cgAssignArrayElem} :: (\textbf{SIdentifier}, \textbf{Maybe} \ \textbf{SExpression}) -> \textbf{ModOp} -> \textbf{SExpression} -> \textbf{CodeGen generator} [\textbf{Maybe Label}, \textbf{MInstruction}]

276 \textbf{cgAssignArrayElem} \ (n, e1) \ \textbf{modop} \ e2 =

\begin{verbatim}
do \ (rt, lt, ut) <- \textbf{loadArrayElementVariableValue} \ n \ e1
  (re, le, ue) <- \textbf{cgExpression} \ e2
  \textbf{l} <- \textbf{getUniqueLabel} \ "assArrElem"
  \textbf{ue} >> \textbf{ut}
  \textbf{return} \ (\textbf{lt} ++ \textbf{le} ++ [(\textbf{Just} \ l, \textbf{cgModOp} \ \textbf{modop} \ \textbf{rt} \ \textbf{re})] ++ \textbf{invertInstructions} \ (\textbf{lt} ++ \textbf{le})
\end{verbatim}

285 \textbf{-- \ Ensures correct loads for swapping}

286 \textbf{loadForSwap} :: (\textbf{SIdentifier}, \textbf{Maybe} \ \textbf{SExpression}) -> \textbf{CodeGen generator} (\textbf{Register}, [\textbf{Maybe Label}, \textbf{MInstruction}]), \textbf{CodeGen generator} ()

287 \textbf{loadForSwap} \ (n, x) = \textbf{gets} \ (\textbf{symbolTable} . \textbf{saState}) \ >>= \ \textbf{\_} \to

\begin{verbatim}
case \textbf{lookup} \ n \ \textbf{st} \ of
  (\textbf{Just} \ \textbf{ClassField} \ \textbf{IntegerArrayType} \ _ \ _ _) -> \textbf{case} \ x \ \textbf{of}
    \textbf{Just} \ x' ->
      \textbf{loadArrayElementVariableValue} \ n \ x'
      \_ -> \textbf{loadVariableValue} \ n
  (\textbf{Just} \ \textbf{ClassField} \ \textbf{ObjectArrayType} \ _ \ _ _) -> \textbf{case} \ x \ \textbf{of}
    \textbf{Just} \ x' ->
      \textbf{loadArrayElementVariableValue} \ n \ x'
      \_ -> \textbf{loadVariableValue} \ n
  (\textbf{Just} \ \textbf{ClassField} \ ()) -> \textbf{loadVariableValue} \ n
\end{verbatim}
(Just (LocalVariable (ObjectType _) _)) -> loadVariableValue n
(Just (LocalVariable (CopyType _) _)) -> loadVariableValue n
(Just (LocalVariable IntegerArrayType _)) ->
  case x of
    Just x' -> loadArrayElementVariableValue n x'
    _ -> loadVariableValue n
(Just (LocalVariable (ObjectArrayType _) _)) ->
  case x of
    Just x' -> loadArrayElementVariableValue n x'
    _ -> loadVariableValue n
(Just (MethodParameter IntegerType _)) -> loadVariableValue n
(Just (MethodParameter (ObjectType _) _)) -> loadVariableValue n
(Just (MethodParameter (CopyType _) _)) -> loadVariableValue n
(Just (MethodParameter IntegerArrayType _)) ->
  case x of
    Just x' -> loadArrayElementVariableValue n x'
    _ -> loadVariableValue n
(Just (MethodParameter (ObjectArrayType _) _)) ->
  case x of
    Just x' -> loadArrayElementVariableValue n x'
    _ -> loadVariableValue n
_ -> throwError $ "ICE: Invalid variable index " ++ show n

-- | Code generation for swaps
cgSwap :: (SIdentifier, Maybe SExpression) -> (SIdentifier, Maybe SExpression) ->
  CodeGenerator [(Maybe Label, MInstruction)]
cgSwap n1 n2 = if n1 == n2 then return [] else
do (r1, l1, u1) <- loadForSwap n1
do (r2, l2, u2) <- loadForSwap n2
    u2 >> u1
    l <- getUniqueLabel "swap"
   let swap = [(Just l, XOR r1 r2), (Nothing, XOR r2 r1), (Nothing, XOR r1 r2)]
return $ l1 ++ l2 ++ swap ++ invertInstructions (l1 ++ l2)

-- | Code generation for conditionals
cgConditional :: SExpression -> [SStatement] -> [SStatement] -> SExpression ->
  CodeGenerator [(Maybe Label, MInstruction)]
cgConditional e1 s1 s2 e2 =
do l_entry <- getUniqueLabel "entry"
do l_test <- getUniqueLabel "test"
do l_assert_t <- getUniqueLabel "assert_true"
do l_assert <- getUniqueLabel "assert_false"
do rt <- tempRegister
    (re1, le1, ue1) <- cgBinaryExpression e1
    uel <- concat <$> mapM cgStatement s1
    uel' <- concat <$> mapM cgStatement s2
    (re2, le2, ue2) <- cgBinaryExpression e2
    ue2 >> popTempRegister --rt
    return $ le1 ++ [(Nothing, XOR rt re1)] ++ invertInstructions le1 ++
        [(Just l_test, BEQ rt registerZero l_test_f), (Nothing, XORI rt $ Immediate 1)] ++
        [Just l_test_f, BRA l_test]
    ++
        [Nothing, XOR rt $ Immediate 1],
        [(Just l_assert, BNE rt registerZero l_assert_t)] ++
        le2 ++ [(Nothing, XOR rt re2)] ++ invertInstructions le2

-- | Code generation for loops
cgLoop :: SExpression -> [SStatement] -> [SStatement] -> SExpression ->
  CodeGenerator [(Maybe Label, MInstruction)]
cgLoop el s1 s2 e2 =
do l_entry <- getUniqueLabel "entry"
do l_test <- getUniqueLabel "test"
do l_test_f <- getUniqueLabel "test_false"
do rt <- tempRegister
    (re1, le1, ue1) <- cgBinaryExpression el
    uel <- concat <$> mapM cgStatement s1
    uel' <- concat <$> mapM cgStatement s2
    (re2, le2, ue2) <- cgBinaryExpression e2
    ue2 >> popTempRegister --rt
    return $ le1 ++ [(Nothing, XOR rt re1)] ++ invertInstructions le1 ++
        [(Just l_test, BEQ rt registerZero l_test_f), (Nothing, XORI rt $ Immediate 1)] ++
        [Nothing, XOR rt $ Immediate 1],
        [(Just l_assert, BNE rt registerZero l_assert_t)] ++
        le2 ++ [(Nothing, XOR rt re2)] ++ invertInstructions le2

Appendix B  ROOPLPPC Source Code  112 of 231
l_exit <- getUniqueLabel "exit"
rt <- tempRegister
(re1, le1, ue1) <- cgBinaryExpression e1
ue1
s1' <- concat <$> mapM cgStatement s1
s2' <- concat <$> mapM cgStatement s2
(re2, le2, ue2) <- cgBinaryExpression e2
ue2 >> popTempRegister --rt
return $ [(Nothing, XORI rt $ Immediate 1), (Just l_entry, BEQ rt registerZero l_assert) ++
le1 ++ [(Nothing, XOR rt re1)] ++ invertInstructions le1 ++ [(Just l_test, BNE rt registerZero l_exit)] ++ s2' ++
[(Just _assert, BRA l_entry), (Just l_exit, BRA l_test), (Nothing, XOR rt $ Immediate 1)]

-- | Code generation for object blocks
cgObjectBlock :: TypeName -> SIdentifier -> [SStatement] -> CodeGenerator [(Maybe Label, MInstruction)]
cgObjectBlock tp n stmt =
do rn <- pushRegister n
let push = [(Nothing, XOR rn registerSP), (Nothing, SUBI registerSP $ Immediate 1)]
malloc <- cgObjectConstruction tp (n, Nothing)
stmt' <- concat <$> mapM cgStatement stmt
free <- cgObjectDestruction tp (n, Nothing)
popRegister
return $ push ++ malloc ++ stmt' ++ free ++ invertInstructions push

-- | Code generation for local blocks
cgLocalBlock :: SIdentifier -> SExpression -> [SStatement] -> SExpression -> CodeGenerator [(Maybe Label, MInstruction)]
cgLocalBlock n e1 stmt e2 =
do rn <- pushRegister n
(re1, le1, ue1) <- cgExpression e1
rt1 <- tempRegister
popTempRegister >> ue1
stmt' <- concat <$> mapM cgStatement stmt
(re2, le2, ue2) <- cgExpression e2
rt2 <- tempRegister
popTempRegister >> ue2
popRegister --rn
l <- getUniqueLabel "localBlock"
let create re rt = [(Just l, XOR rn registerSP),
(Nothing, XOR rt re)],
(Nothing, SUBI registerSP $ Immediate 1)]
load = le1 ++ create re1 rt1 ++ invertInstructions le1
clear = le2 ++ invertInstructions (create re2 rt2) ++ invertInstructions le2
return $ load ++ stmt' ++ clear

-- | Code generation for calls
cgCall :: [(SIdentifier, Maybe SExpression)] -> [(Maybe Label, MInstruction)] -> Register ->
CodeGenerator [(Maybe Label, MInstruction)]
cgCall args jump this =
do (ra, la, ua) <- unzip3 <$> mapM loadAddr args
sequence_ ua
rs <$> gets registerStack
let rr = (registerThis : map snd rs) \ (this : ra)
store = concatMap push $ rr ++ ra ++ [this]
return $ concat la ++ store ++ jump ++ invertInstructions store ++ invertInstructions (concat la)
where push r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]
loadAddr (n, e) =
case e of
    Nothing -> loadVariableAddress n
    Just e' -> loadArrayElementVariableAddress n e'
412  -- | Code generation for local calling
413  cgLocalCall :: SIdentifier -> [(SIdentifier, Maybe SExpression)] -> CodeGenerator [(Maybe Label, MInstruction)]
414  cgLocalCall m args = getMethodLabel m >>= \l_m -> cgCall args [(Nothing, BRA l_m)]
415  registerThis
416
417  -- | Code generation for local uncalling
418  cgLocalUncall :: SIdentifier -> [(SIdentifier, Maybe SExpression)] -> CodeGenerator [(Maybe Label, MInstruction)]
419  cgLocalUncall m args = getMethodLabel m >>= \l_m -> cgCall args [(Nothing, RBRA l_m)]
420  registerThis
421
422  -- | Returns the type associated with a given identifier
423  getType :: SIdentifier -> CodeGenerator TypeName
424  getType i = gets (symbolTable . saState) >>= \st ->
425  case lookup i st of
426  (Just (LocalVariable (ObjectType tp) _) _) -> return tp
427  (Just (ClassField (ObjectType tp) _ _ _)) -> return tp
428  (Just (MethodParameter (ObjectType tp) _ _ _)) -> return tp
429  (Just (ClassField (ObjectArrayType tp) _ _ _)) -> return tp
430  (Just (MethodParameter (ObjectArrayType tp) _ _ _)) -> return tp
431  _ -> throwError $ "ICE: Invalid object variable index " ++ show i
432
433  -- | Load the return offset for methods
434  loadMethodAddress :: (SIdentifier, Register) -> MethodName -> CodeGenerator (Register, [(Maybe Label, MInstruction)])
435  loadMethodAddress (o, ro) m =
436  do rv <- tempRegister
437     rt <- tempRegister
438     rtgt <- tempRegister
439     offsetMacro <- OffsetMacro <$> getType o <*> pure m
440  1 <- getUniqueLabel "loadMetAdd"
441  let load = [(Just 1, EXCH rv ro),
442              (Nothing, ADDI rv offsetMacro),
443              (Nothing, EXCH rt rv),
444              (Nothing, XOR rtgt rt),
445              (Nothing, EXCH rt rv),
446              (Nothing, SUBI rv offsetMacro),
447              (Nothing, EXCH rv ro)]
448  return (rtgt, load)
449
450  -- | Load address or value needed for calls
451  loadForCall :: (SIdentifier, Maybe SExpression) -> CodeGenerator (Register, [(Maybe Label, MInstruction)], CodeGenerator ())
452  loadForCall (n, e) = gets (symbolTable . saState) >>= \st ->
453  case lookup n st of
454  (Just (ClassField (ObjectArrayType _) _ _ _)) ->
455    case e of
456      Just x' -> loadArrayElementVariableValue n x'
457      _ -> throwError $ "ICE: Invalid variable index " ++ show n
458  (Just ClassField (CopyType _) -> loadVariableValue n
459  (Just (LocalVariable (ObjectArrayType _) _ _ _)) -> loadVariableValue n
460  (Just (LocalVariable (CopyType _) _ _ _)) -> loadVariableValue n
461  (Just (LocalVariable (ObjectArrayType _) _ _ _)) ->
462    case e of
463      Just x' -> loadArrayElementVariableValue n x'
464      _ -> throwError $ "ICE: Invalid variable index " ++ show n
465  (Just _) -> loadVariableValue n
466      _ -> throwError $ "ICE: Invalid variable index " ++ show n
467
468  -- | Code generation for object calls
469  cgObjectCall :: (SIdentifier, Maybe SExpression) -> MethodName -> [(SIdentifier, Maybe SExpression)] -> CodeGenerator [(Maybe Label, MInstruction)]
470  cgObjectCall (o, e) m args =

Appendix B  ROOPLPPC Source Code 114 of 231
do (ro, lo, uo) <- loadForCall (o, e)
    rt <- tempRegister
    (rtgt, loadAddress) <- loadMethodAddress (o, rt) m
    l_jmp <- getUniqueLabel "l_jmp"
    let jp = [(Nothing, SUBI rtgt $ AddressMacro l_jmp),
              (Just l_jmp, SWAPBR rtgt),
              (Nothing, NEG rtgt),
              (Nothing, ADDI rtgt $ AddressMacro l_jmp)]
    call <- cgCall args jp rt
    popTempRegister >> popTempRegister >> popTempRegister -- rv, rt & rtgt from loadMethod Addr
    popTempRegister >> uo
    let load = lo ++ [(Nothing, XOR rt ro)] ++ loadAddress ++ invertInstructions lo
    return $ load ++ call ++ invertInstructions load

-- | Code generation for object uncalls
cgObjectUncall :: (SIdentifier, Maybe SExpression) -> MethodName -> [(SIdentifier, Maybe SExpression)] -> CodeGenerator [(Maybe Label, MInstruction)]
cgObjectUncall (o, e) m args =
do (ro, lo, uo) <- loadForCall (o, e)
    rt <- tempRegister
    (rtgt, loadAddress) <- loadMethodAddress (o, rt) m
    l_jmp <- getUniqueLabel "l_jmp"
    l_rjmp_top <- getUniqueLabel "l_rjmp_top"
    l_rjmp_bot <- getUniqueLabel "l_rjmp_bot"
    let jp = [(Nothing, SUBI rtgt $ AddressMacro l_jmp),
              (Just l_rjmp_top, RBRA l_rjmp_bot),
              (Just l_jmp, SWAPBR rtgt),
              (Nothing, NEG rtgt),
              (Nothing, ADDI rtgt $ AddressMacro l_jmp)]
    call <- cgCall args jp rt
    popTempRegister >> popTempRegister >> popTempRegister -- rv, rt & rtgt from loadMethod Addr
    popTempRegister >> uo
    let load = lo ++ [(Nothing, XOR rt ro)] ++ loadAddress ++ invertInstructions lo
    return $ load ++ call ++ invertInstructions load

-- | Code generation for object construction
cgObjectConstruction :: TypeName -> (SIdentifier, Maybe SExpression) -> CodeGenerator [(Maybe Label, MInstruction)]
cgObjectConstruction tp (n, e) =
do (rv, lv, uv) <- case e of
    Nothing -> loadVariableAddress n
    Just e' -> loadArrayElementVariableAddress n e'
    rp <- tempRegister
    rt <- tempRegister
    popTempRegister >> popTempRegister
    l <- getUniqueLabel "obj_con"
    rs <- gets registerStack
    let rr = (registerThis : map snd rs) \ [rp, rt]
        store = concatMap push rr
        malloc = [(Just 1, ADDI rt $ SizeMacro tp)] ++ push rt ++ push rp
        lb = 1 ++ "$bot"
        setVtable = [(Nothing, XORI rt $ AddressMacro "$l_" ++ tp ++ "$vt"),
                     (Nothing, EXCH rt rp),
                     (Nothing, ADDI rp ReferenceCounterIndex),
                     (Nothing, XORI rt $ Immediate 1),
                     (Nothing, EXCH rt rp),
                     (Just lb, SUBI rp ReferenceCounterIndex),
                     (Nothing, EXCH rp rv)]
        uv
        return $ store ++ malloc ++ [(Nothing, BRA "$l_malloc") ++ invertInstructions malloc ++ invertInstructions store ++ lv ++ setVtable ++ invertInstructions lv
                                        where push r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]
-- | Code generation for object destruction

cgObjectDestruction :: TypeName -> (SIdentifier, Maybe SExpression) -> CodeGenerator [(Maybe Label, MInstruction)]

cgObjectDestruction tp (n, e) =
do (rp, la, ua) <- case e of
  Nothing -> loadVariableValue n
  Just e' -> loadArrayElementVariableValue n e'
rt <- tempRegister
l <- getUniqueLabel "obj_des"
popTempRegister >> ua
rs <- gets registerStack
let
  removeVtable = [(Just lt, EXCH rt rp),
                   (Nothing, XORI rt $ AddressMacro $ "l_" ++ tp ++ ",_vt"),
                   (Nothing, ADDI rp ReferenceCounterIndex),
                   (Nothing, EXCH rt rp),
                   (Nothing, XORI rt $ Immediate 1),
                   (Nothing, SUBI rp ReferenceCounterIndex)]
  rr = (registerThis : map snd rs) \ [rp, rt]
store = concatMap push rr
free = [(Nothing, RBRA "l_malloc") ++ invertInstructions (la ++ store ++ free)]
return $ la ++ removeVtable ++ store ++ free ++ [(Nothing, RBRA "$malloc") ++ invertInstructions (la ++ store ++ free)]

where
  push r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]

-- | Code generation for reference construction
cgCopyReference :: (SIdentifier, Maybe SExpression) -> (SIdentifier, Maybe SExpression) -> CodeGenerator [(Maybe Label, MInstruction)]

cgCopyReference (n, e1) (m, e2) =
do (rcp, lp, up) <- case e2 of
  Nothing -> loadVariableValue m
  Just e2' -> loadArrayElementVariableValue m e2'
(rp, la, ua) <- case e1 of
  Nothing -> loadVariableValue n
  Just e1' -> loadArrayElementVariableValue n e1'
rt <- tempRegister
up >> ua >> popTempRegister
l <- getUniqueLabel "copy"
let
  reference = [(Just l, XOR rcp rp),
               (Nothing, ADDI rp ReferenceCounterIndex),
               (Nothing, EXCH rt rp),
               (Nothing, ADDI rt $ Immediate 1),
               (Nothing, EXCH rt rp),
               (Nothing, SUBI rp ReferenceCounterIndex)]
return $ lp ++ la ++ reference ++ invertInstructions (lp ++ la)

-- | Code generation for reference destruction
cgUnCopyReference :: (SIdentifier, Maybe SExpression) -> (SIdentifier, Maybe SExpression) -> CodeGenerator [(Maybe Label, MInstruction)]

cgUnCopyReference (n, e1) (m, e2) =
do (rcp, la1, ua1) <- case e2 of
  Nothing -> loadVariableValue m
  Just e2' -> loadArrayElementVariableValue m e2'
(rp, la2, ua2) <- case e1 of
  Nothing -> loadVariableValue n
  Just e1' -> loadArrayElementVariableValue n e1'
rt <- tempRegister
ua1 >> ua2 >> popTempRegister
l <- getUniqueLabel "uncopy"
ual => ua2 >> popTempRegister
let
  reference = [(Just l, XOR rcp rp),
               (Nothing, ADDI rp ReferenceCounterIndex),
               (Nothing, EXCH rt rp),
               (Nothing, SUBI rt $ Immediate 1),
               (Nothing, EXCH rt rp),
               (Nothing, SUBI rp ReferenceCounterIndex)]
return $ la1 ++ la2 ++ reference ++ invertInstructions (la1 ++ la2)

-- removeRegister (m, rcp)
return $ la1 ++ la2 ++ reference ++ invertInstructions (la1 ++ la2)

-- | Code generation for array construction
cgArrayConstruction :: SExpression -> SIdentifier -> CodeGenerator [(Maybe Label, MInstruction)]
cgArrayConstruction e n =
do (ra, la, wa) <- loadVariableAddress n
(re, le, ue) <- cgExpression e
rp <- tempRegister
rt <- tempRegister
popTempRegister >> popTempRegister
l <- getUniqueLabel "arr_con"
rs <- gets registerStack
let
  rr = (registerThis : map snd rs) \ [rp, rt]
  store = le ++ [(Just l, ADDI rt ArrayElementOffset), (Nothing, ADD rt re)] ++
    invertInstructions le ++ concatMap push rr
  malloc = push rt ++ push rp
  lb = l ++ ".bot"
  initArray = la ++ le ++
    (Nothing, XOR rt re),
    (Nothing, EXCH rt rp),
    (Nothing, ADDI rp ReferenceCounterIndex),
    (Nothing, XORI rt $ Immediate 1),
    (Nothing, EXCH rt rp),
    (Nothing, SUBI rp ReferenceCounterIndex),
    (Just lb, EXCH rp ra) ++
    invertInstructions (la ++ le)
  ue >> ua
return $ store ++ malloc ++ [(Nothing, BRA "l_malloc")]
where
  push r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]

-- | Code generation for array destruction
cgArrayDestruction :: SExpression -> SIdentifier -> CodeGenerator [(Maybe Label, MInstruction)]
cgArrayDestruction e n =
do (rp, lp, up) <- loadVariableValue n
(re, le, ue) <- cgExpression e
rt <- tempRegister
l <- getUniqueLabel "obj_des"
rs <- gets registerStack
let
  removeArray = [(Nothing, XCH rt rp),
                  (Nothing, ADDI rp ReferenceCounterIndex),
                  (Nothing, EXCH rt rp),
                  (Nothing, SUBI rp ReferenceCounterIndex),
                  (Nothing, XORI rt $ Immediate 1),
                  (Nothing, SUBI rp ReferenceCounterIndex)]
  rr = (registerThis : map snd rs) \ [rp, rt]
  store = concatMap push rr
  free = [(Just l, ADDI rt ArrayElementOffset), (Nothing, ADD rt re)] ++ push rt ++
    push rp
  lt = l ++ ".top"
return $ lp ++ le ++ removeArray ++ store ++ free ++ [(Nothing, RBRA "l_malloc")]
where
  push r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]

-- | Code generation for statements
cgStatement :: SStatement -> CodeGenerator [(Maybe Label, MInstruction)]
cgStatement (Assign n modop e) = cgAssign n modop e
cgStatement (AssignArrElem (n, e1) modop e2) = cgAssignArrElem (n, e1) modop e2
cgStatement (Conditional e1 s1 s2 e2) = cgConditional e1 s1 s2 e2
cgStatement (Loop e1 s1 s2 e2) = cgLoop e1 s1 s2 e2
cgStatement (ObjectBlock tp n stmt) = cgObjectBlock tp n stmt
cgStatement (LocalBlock _ n e1 stmt e2) = cgLocalBlock n e1 stmt e2
cgStatement (LocalCall m args) = cgLocalCall m args

655 cgStatement (LocalUncall m args) = cgLocalUncall m args

656 cgStatement (ObjectCall o m args) = cgObjectCall o m args

657 cgStatement (ObjectUncall o m args) = cgObjectUncall o m args

658 cgStatement (ObjectConstruction tp n) = cgObjectConstruction tp n

659 cgStatement (ObjectDestruction tp n) = cgObjectDestruction tp n

660 cgStatement Skip = return []

661 cgStatement (CopyReference _ n m) = cgCopyReference n m

662 cgStatement (UnCopyReference _ n m) = cgUnCopyReference n m

663 cgStatement (ArrayConstruction (_, e) n) = cgArrayConstruction e n

664 cgStatement (ArrayDestruction (_, e) n) = cgArrayDestruction e n

-- | Code generation for methods
666 cgMethod :: (TypeName, SMethodDeclaration) -> CodeGenerator [(Maybe Label, MInstruction)]
667 cgMethod (_, GMDecl m ps body) =
668 do l <- getMethodLabel m
669 rs <- addParameters
670 body' <- concat <$> mapM cgStatement body
671 clearParameters
672 let lt = l ++ "_top"
673 lb = l ++ "_bot"
674 mp = [(Just lt, BRA lb),
675 (Nothing, ADDI registerSP $ Immediate 1),
676 (Nothing, EXCH registerRO registerSP),
677 ++ concatMap pushParameter rs ++
678 [(Nothing, EXCH registerThis registerSP),
679 (Nothing, SUBI registerSP $ Immediate 1),
680 (Just l, SWAPBR registerRO),
681 (Nothing, NEG registerRO),
682 (Nothing, ADDI registerSP $ Immediate 1),
683 (Nothing, EXCH registerThis registerSP)],
684 ++ invertInstructions (concatMap pushParameter rs) ++
685 [(Nothing, EXCH registerRO registerSP),
686 (Nothing, SUBI registerSP $ Immediate 1)]
687 return $ mp ++ body' ++ [(Just lb, BRA lt)]
688 where addParameters = mapM (pushRegister . (\(GDecl _ p) -> p)) ps
689 clearParameters = replicateM_ (length ps) popRegister
690 pushParameter r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]

692 cgMalloc1 :: CodeGenerator [(Maybe Label, MInstruction)]
693 cgMalloc1 =
694 do -- Temp registers needed for malloc
695 r_p <- tempRegister -- Pointer to new obj
696 r_object_size <- tempRegister -- Object size
697 r_counter <- tempRegister -- Free list index
698 r_csize <- tempRegister -- Current cell size
699 rt <- tempRegister
700 rt2 <- tempRegister
701 rt_tmp <- tempRegister
702 -- Expressions and sub routines
703 (r_e1_outer, l_e1_outer, u_e1_outer) <- cgBinOp Lt r_csize r_object_size
704 (r_e2_outer, l_e2_outer, u_e2_outer) <- cgBinOp Lt r_csize r_object_size
705 (r_f1, l_f1, u_f1) <- loadFreeListAddress r_counter
706 (r_block, l_block, u_block) <- loadHeadAtFreeList r_f1
707 (r_e1_inner, l_e1_inner, u_e1_inner) <- cgBinOp Neq r_block registerZero
708 (r_e2_i1, l_e2_i1, u_e2_i1) <- cgBinOp Neq r_p r_tmp
709 (r_e2_i2, l_e2_i2, u_e2_i2) <- cgBinOp Eq r_tmp registerZero
710 (r_e2_i3, l_e2_i3, u_e2_i3) <- cgBinOp BitOr r_e2_i1 r_e2_i2
711 let tmpRegisterList = [rt, rt2, r_tmp, r_e1_outer, r_e2_outer, r_f1, r_block,
712 r_e1_inner, r_e2_i1, r_e2_i2, r_e2_i3]
713 -- Update state after evaluating expressions and subroutines
714 u_e2_i3 >> u_e2_i2 >> u_e2_i1 >> u_e1_inner

Appendix B  ROOPLPPC Source Code 118 of 231
u_block >> u_f1 >> u_e2_outer >> u_e1_outer
popTempRegister >> popTempRegister >> popTempRegister >> popTempRegister >>
popTempRegister >> popTempRegister >> popTempRegister

let l_o_test = "l_o_test"
l_o_assert_t = "l_o_assert_true"
l_o_assert_f = "l_o_assert_false"
l_i_test = "l_i_test"
l_i_assert_t = "l_i_assert_true"
l_i_assert_f = "l_i_assert_false"
l_m_top = "l_malloc1_top"
l_m_bot = "l_malloc1_bot"
l_e1_outer -- Set r_e1 -> c_size < obj_size
++ invertInstructions l_e1_outer ++
++ ([Just l_o_test, BRA l_o_assert_t], [Just l_o_test_f, BRA l_o_test])
++ invertInstructions l_block ++
++ ([Just l_i_test, BEQ rt2 registerZero l_i_test_f], [Just l_i_test_f, BRA l_i_test])
++ invertInstructions l_e1_inner ++
++ ([Nothing, XOR rt r_e1_inner])
++ invertInstructions (concatMap pushRegisterToStack tmpRegisterList)
++
++ ([Nothing, BRA l_m_entry])
++ invertInstructions (concatMap pushRegisterToStack tmpRegisterList)
++
++ ([Nothing, ADDI registerSP $ Immediate 1], [Nothing, EXCH registerRO registerSP]) -- Pop return offset from stack
++ invertInstructions l_f1 ++
++ ([Just l_m_entry, SWAPBR registerRO], [Nothing, ADDI r_counter $ Immediate 1]) -- counter++
++ invertInstructions l_block ++
++ ([Nothing, BRA l_m_entry], [Nothing, BR (S2_OUTER)])
Nothing, RR r_csize $ Immediate 1), -- uncall double(csize)
(Nothing, SUBI r_counter $ Immediate 1), -- counter == 1
(Nothing, XOR r_tmp r_p), -- Copy current address of p
(Nothing, EXCH r_tmp r_fl), -- Store address in current free list
(Nothing, ADD r_p r_csize), -- Set p to other half of the block we’re splitting
(Just 1_l_assert, BNE rt2 registerZero 1_l_assert_t),
(Nothing, EXCH r_tmp r_fl),
(Nothing, SUB r_p r_csize)
++ l_e2_i1 -- set r_e2_i1 <- p - csize != free_list[counter]
++ l_e2_i2 -- set r_e2_i2 <- free_list[counter] = 0
++ l_e2_i3 -- set r_e2_i3 <- r_e2_i1 || A r_e2_i2
++ [(Nothing, XOR r_p rt2 r_e2_i3)] -- set rt2 -> r_i_2
++ invertInstructions 1_e2_i3 -- Clear r_i_2
++ invertInstructions 1_e2_i2
++ invertInstructions 1_e2_i1 ++
(Nothing, ADD r_p r_csize),
(Nothing, EXCH r_tmp r_fl), -- S2_outer end
(Just 1_o_assert, BNE rt registerZero 1_o_assert_t)
++ l_e2_outer -- Set r_e2 -> c_size < obj_size
++ [(Nothing, XOR rt r_e2_outer)] -- r_t = r_e1_o
++ invertInstructions 1_e2_outer -- Clear r_e1_o
++ [(Just 1_m_bot, BRA 1_m_top)] -- Go to top

return malloc
where pushRegisterToStack r = [(Nothing, EXCH r registerSP), (Nothing, SUBI registerSP $ Immediate 1)]

cgMalloc :: CodeGenerator [(Maybe Label, MInstruction)]
cgMalloc =
do rp <- tempRegister -- Pointer to new obj
   ros <- tempRegister -- Object size
   rc <- tempRegister -- Free list index
   rs <- tempRegister -- Current cell size
popTempRegister >> popTempRegister >> popTempRegister >> popTempRegister
let malloc = [(Just "$l_malloc_top", BRA "$l_malloc_bot")]
++ [(Just "$l_malloc", SWAPBR registerRO),
   (Nothing, NEG registerRO),
   (Nothing, ADDI rs $ Immediate 2),
   (Nothing, XOR rc registerZero)]
++ concatMap pop [rp, ros]
++ push registerRO ++ [(Nothing, BRA "$l_malloc1")]
++ pop registerRO
++ concatMap push [ros, rp] ++ [(Nothing, XOR rc registerZero)],
   (Nothing, SUBI rs $ Immediate 2),
   (Just "$l_malloc_bot", BRA "$l_malloc_bot")]

return malloc
where pop r = [(Nothing, ADDI registerSP $ Immediate 1), (Nothing, EXCH r registerSP)]
push r = invertInstructions (pop r)

-- | Code generation for virtual tables
cgVirtualTables :: CodeGenerator [(Maybe Label, MInstruction)]
cgVirtualTables = concat <$> (gets (virtualTables . saState) >>= mapM vtInstructions)
   where vtInstructions (n, ms) = zip (vtLabel n) <$> mapM vtData ms
vtData m = DATA . AddressMacro <$> getMethodLabel m
vtLabel n = (Just "$l_" ++ n ++ "$_vt") : repeat Nothing

-- | Returns the main class label
getMainLabel :: CodeGenerator Label
getMainLabel = gets (mainMethod . saState) >>= getMethodLabel
-- | Fetches the main class from the class analysis state
getMainClass :: CodeGenerator TypeName
getMainClass = gets (mainClass . caState . saState) >>= \mc ->
case mc of
  (Just tp) -> return tp
  Nothing -> throwError "ICE: No main method defined"
-- | Fetches the field of a given type name
getFields :: TypeName -> CodeGenerator [VariableDeclaration]
getFields tp = do cs <- gets (classes . caState . saState)
case lookup tp cs of
  (Just (GCDecl _ _ fs _)) -> return fs
  Nothing -> throwError $ "ICE: Unknown class " ++ tp
-- | Code generation for output
cgOutput :: TypeName -> CodeGenerator ([Maybe Label, MInstruction]), ([Maybe Label, MInstruction])
cgOutput tp =
do mfs <- getFields tp
c o <- concat <$> mapM cgCopyOutput (zip [1..] mfs)
return (map cgStatic mfs, co)
where cgStatic (GDecl _ n) = (Just $ "l_r_" ++ n, DATA $ Immediate 0)
cgCopyOutput(o, GDecl _ n) =
do rt <- tempRegister
ra <- tempRegister
popTempRegister >> popTempRegister
let copy = [ADDI registerThis ReferenceCounterIndex,
ADDI registerThis $ Immediate o,
EXCH rt registerThis,
XORI ra $ AddressMacro $ "l_r_" ++ n, -- Init free list pointer list
EXCH rt ra,
XORI ra $ AddressMacro $ "l_r_" ++ n,
SUBI registerThis $ Immediate o,
SUBI registerThis ReferenceCounterIndex]
return $ zip (repeat Nothing) copy
-- | Generates code for the program entry point
cgProgram :: SProgram -> CodeGenerator PISA.MProgram
cgProgram p =
do vt <- cgVirtualTables
malloc <- cgMalloc
malloc1 <- cgMalloc1
rv <- tempRegister -- V table register
rb <- tempRegister -- Memory block register
popTempRegister >> popTempRegister
ms <- cgMethod p
l_main <- getMainLabel
mtp <- getMainClass
(out, co) <- cgOutput mtp
let mvt = "l_" ++ mtp ++ ",vt"
mn = [(Just "start", BRA "top"),
(Nothing, START),
(Nothing, ADDI registerFLPs ProgramSize), -- Init free list pointer list
(Nothing, XOR registerHP registerFLPs), -- Init heap pointer
(Nothing, ADDI registerHP FreeListsSize), -- Init space for FLPs list
(Nothing, XOR rb registerHP), -- Store address of initial memory block in rb
(Nothing, ADDI registerFLPs FreeListsSize), -- Index to end of free lists
(Nothing, SUBI registerFLPs $ Immediate 1), -- Index to last element of free lists
(Nothing, EXCH rb registerFLPs), -- Store address of first block in last element of free lists
(Nothing, ADDI registerFLPs $ Immediate 1), -- Index to end of free lists
900  (Nothing, SUBI registerFLPs FreeListsSize), -- Index to beginning of free lists
901  (Nothing, XOR registerSP registerHP), -- Init stack pointer 1/2
902  (Nothing, ADDI registerSP StackOffset), -- Init stack pointer 2/2
903  (Nothing, SUBI registerSP $ SizeMacro mtp), -- Allocate space for main on stack
904  (Nothing, XOR registerThis registerSP), -- Store address of main object
905  (Nothing, XORI rv $ AddressMacro mvt), -- Store address of vtable in rv
906  (Nothing, EXCH rv registerThis), -- Add address of vtable to stack
907  (Nothing, SUBI registerSP $ Immediate 1), -- Add address of vtable to stack
908  (Nothing, EXCH registerThis registerSP), -- Push 'this' to stack
909  (Nothing, SUBI registerSP $ Immediate 1), -- Push 'this' to stack
910  (Nothing, BRA l_main), -- Execute main
911  (Nothing, ADDI registerSP $ Immediate 1), -- Pop 'this'
912  (Nothing, EXCH registerThis registerSP) -- Pop 'this'
913  ++ co ++
914  (Nothing, ADDI registerSP $ Immediate 1), -- Pop vtable address
915  (Nothing, EXCH rv registerThis), -- Pop vtable address
916  (Nothing, XOR registerThis registerSP), -- Clear 'this'
917  (Nothing, ADDI registerSP $ SizeMacro mtp), -- Deallocate space for main
918  (Nothing, SUBI registerSP StackOffset), -- Clear stack pointer
919  (Nothing, XOR registerSP registerHP), -- Clear stack pointer
920  (Nothing, SUBI registerHP FreeListsSize), -- Reset Heap pointer (For pretty output)
921  (Nothing, XOR registerHP registerFLPs), -- Reset Heap pointer (For pretty output)
922  (Nothing, SUBI registerFLPs ProgramSize), -- Reset Free lists pointer (For pretty output)
923  (Just "finish", FINISH)
924  return $ PISA.GProg $ [(Just "top", BRA "start") ++ out ++ vt ++ malloc ++ malloc1 ++ ms ++ mn
925
926  -- | Generates code for a program
927  generatePISA :: (SProgram, SAState) -> Except String (PISA.MProgram, SAState)
928  generatePISA (p, s) = second saState <$> runStateT (runCG $ cgProgram p) (initialState s)
929  showPISAProgram :: (Show a, MonadIO m) => (t, a) -> m ()
930  showPISAProgram (_, s) = pPrint s
```
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module MacroExpander (expandMacros) where

import Data.Maybe
import Data.List
import Control.Monad.Reader
import Control.Monad.Except
import Control.Arrow
import AST hiding (Program, GProg, Offset)
import PISA
import Debug.Trace (trace, traceShow)
import ScopeAnalyzer
import ClassAnalyzer

-- | Returns the offset table generated from the an indexed virtual table
getOffsetTable :: SAState -> [(TypeName, [(MethodName, Offset)])]
getOffsetTable s = map (second (map toOffset)) indexedVT
  where indexedVT = map (second $ zip [0..]) $ virtualTables s
toOffset (i, m) = (getName $ lookup m $ symbolTable s, i)
getName (Just (Method _ n)) = n
getNothing = error "ICE: Invalid method index"

-- | Initializes the macro state containing the address, size, offset tables and the program size
initialState :: MProgram -> SAState -> MEState
initialState (GProg p) s = MEState {
  addressTable = mapMaybe toPair $ zip [0..] p,
  sizeTable = (classSize . caState) s,
  offsetTable = getOffsetTable s,
  programSize = genericLength p,
  freeListsSize = 10,
  stackOffset = 2048,
  initialMemoryBlockSize = 1024,
  referenceCounterIndex = 1,
  arrayElementOffset = 2
  }
  where toPair (a, (Just 1, _)) = Just (1, a)
        toPair _ = Nothing
```
getAddress :: Label -> MacroExpander Address
getAddress l = asks addressTable >>= \at ->
case lookup l at of
  (Just i) -> return i
  Nothing -> throwError $ "ICE: Unknown label " ++ l

getAddress :: Label -> MacroExpander Address
getAddress l = asks addressTable >>= \at ->
case lookup l at of
  (Just i) -> return i
  Nothing -> throwError $ "ICE: Unknown label " ++ l

getSize :: TypeName -> MacroExpander Size
getSize tn = asks sizeTable >>= \st ->
case lookup tn st of
  (Just s) -> return s
  Nothing -> throwError $ "ICE: Unknown type " ++ tn

getOffset :: TypeName -> MethodName -> MacroExpander Offset
getOffset tn mn = asks offsetTable >>= \ot ->
case lookup tn ot of
  Nothing -> throwError $ "ICE: Unknown type " ++ tn
  (Just mo) -> case lookup mn mo of
    Nothing -> throwError $ "ICE: Unknown method " ++ mn
    (Just o) -> return o

-- | Macro definitions
meMacro :: Macro -> MacroExpander Integer
meMacro (Immediate i) = return i
meMacro (AddressMacro l) = getAddress l
meMacro (SizeMacro tn) = getSize tn
meMacro (OffsetMacro tn mn) = getOffset tn mn
meMacro ProgramSize = asks programSize
meMacro FreeListsSize = asks freeListsSize
meMacro StackOffset = asks stackOffset
meMacro InitialMemoryBlockSize = asks initialMemoryBlockSize
meMacro ReferenceCounterIndex = asks referenceCounterIndex
meMacro ArrayElementOffset = asks arrayElementOffset

-- | Macro instructions
meInstruction :: MInstruction -> MacroExpander Instruction
meInstruction (ADD r1 r2) = return $ ADD r1 r2
meInstruction (ADDI r m) = ADDI r <$> meMacro m
meInstruction (ANDX r1 r2 r3) = return $ ANDX r1 r2 r3
meInstruction (ANDIX r1 r2 m) = ANDIX r1 r2 <$> meMacro m
meInstruction (NORX r1 r2 r3) = return $ NORX r1 r2 r3
meInstruction (NEG r) = return $ NEG r
meInstruction (ORX r1 r2 r3) = return $ ORX r1 r2 r3
meInstruction (ORIX r1 r2 m) = ORIX r1 r2 <$> meMacro m
meInstruction (RL r m) = RL r <$> meMacro m
meInstruction (RLV r1 r2) = return $ RLV r1 r2
meInstruction (RR r m) = RR r <$> meMacro m
meInstruction (RRV r1 r2) = return $ RRV r1 r2
meInstruction (SLLX r1 r2 m) = SLLX r1 r2 <$> meMacro m
meInstruction (SLLVX r1 r2 r3) = return $ SLLVX r1 r2 r3
meInstruction (SRAX r1 r2 m) = SRAX r1 r2 <$> meMacro m
meInstruction (SRAX r1 r2 r3) = return $ SRAX r1 r2 r3
meInstruction (SRLX r1 r2 m) = SRLX r1 r2 <$> meMacro m
meInstruction (SRLVX r1 r2 r3) = return $ SRLVX r1 r2 r3
meInstruction (SUB r1 r2) = return $ SUB r1 r2
meInstruction (XOR r1 r2) = return $ XOR r1 r2
meInstruction (XORI r m) = XORI r <$> meMacro m
meInstruction (BEQ r1 r2 l) = return $ BEQ r1 r2 l
meInstruction (BGEZ r l) = return $ BGEZ r l
meInstruction (BGTZ r l) = return $ BGTZ r l
meInstruction (BLEZ r l) = return $ BLEZ r l
meInstruction (BLTZ r l) = return $ BLTZ r l
meInstruction (BNE r1 r2 l) = return $ BNE r1 r2 l
meInstruction (BRA l) = return $ BRA l
meInstruction (EXCH r1 r2) = return $ EXCH r1 r2
meInstruction (SWAPBR r) = return $ SWAPBR r
meInstruction (RBRA l) = return $ RBRA l
meInstruction START = return START
meInstruction FINISH = return FINISH
meInstruction (DATA m) = DATA <$> meMacro m
meInstruction (SUBI r m) = SUBI r <$> meMacro m

-- | Macro Expand a program
meProgram :: MProgram -> MacroExpander Program
meProgram (GProg p) = GProg <$> mapM expandPair p
where expandPair (l, i) = (l, i) <$> meInstruction i

-- | Interface for starting the macro extension process
expandMacros :: (MProgram, SAState) -> Except String Program
expandMacros (p, s) = runReaderT (runME $ meProgram p) $ initialState p s
import Control.Monad.Except
import System.IO
import System.Environment

import PISA
import Parser
import ClassAnalyzer
import ScopeAnalyzer
import TypeChecker
import CodeGenerator
import MacroExpander

import Data.List.Split

type Error = String

main :: IO ()
main =
do args <- getArgs
  when (null args) (error "Supply input filename.\nUsage: ROOPLPPC input.rplpp output.pal\n")
  when (length args > 2) (error "Too many arguments.\nUsage: ROOPLPPC input.rplpp output.\npal\n")
handle <- openFile (head args) ReadMode
input <- hGetContents handle
let output = if length args == 2 then last args else head (splitOn "." (head args)) ++ ".pal"
  either (hPutStrLn stderr) (writeProgram output) $ compileProgram input

compileProgram :: String -> Either Error PISA.Program
compileProgram s =
  runExcept s
  >>= classAnalysis
  >>= scopeAnalysis
  >>= typeCheck
  >>= generatePISA
  >>= expandMacros
Example Output

LinkedList.rplpp

```rplpp
| class Cell         |
|--------------------|
| Cell next          |
| int data           |
|                    |
| method constructor(int value) |
| data ^= value      |
|                    |
| method append(Cell cell) |
| if next = nil & cell != nil then |
| next <= cell       |
|                    |
|                    |
| else skip          |
| fi next != nil & cell = nil |
|                    |
| if next != nil then |
| call next::append(cell) |
|                    |
| else skip          |
| fi next != nil     |
|                    |
|                    |
| class LinkedList   |
| Cell head          |
| int listLength     |
|                    |
| method insertHead(Cell cell) |
| if head = nil & cell != nil then |
| head <= cell       |
|                    |
| else skip          |
| fi head != nil & cell = nil |
|                    |
| method appendCell(Cell cell) |
| call insertHead(cell) |
|                    |
| if head != nil then |
| call head::append(cell) |
|                    |
| else skip          |
| fi head != nil     |
|                    |
|                    |
| listLength += 1    |
|                    |
| method prependCell(Cell cell) |
| call insertHead(cell) |
|                    |
| if cell != nil & head != nil then |
| call cell::append(head) |
|                    |
```


```plaintext
44  else skip
45  fi cell != nil & head = nil
46
47  if cell != nil & head = nil then
48    cell <=> head          // Set head = cell. Cell is nil after execution
49  else skip
50  fi cell = nil & head != nil
51
52  listLength += 1        // Increment length
53
54  method length(int result)
55     result ^= listLength
56  end method
57
58  class Program
59    LinkedList linkedList
60    int sumResult
61    int listLength
62
63  method main()
64    new LinkedList linkedList       // Init new linked linkedList
65    listLength += 10
66
67    local int x = 0
68    from x = 0 do
69      skip
70    loop
71      local Cell cell = nil
72      new Cell cell              // Instantiate new cell
73      call cell::constructor(x)  // Set value of cell
74      call linkedList::appendCell(cell) // Append it to the linkedList
75      delocal Cell cell = nil
76      x += 1
77    until x = listLength
78    delocal int x = listLength
```

Appendix C  Example Output 128 of 231
LinkedList.pal

1 ;; pendulum pal file 60 XORI $10 1
2 top: BRA start 61 ADDI $8 1
3 l_r_linkedList: DATA 0 62 EXCH $19 $17
4 l_r_sumResult: DATA 0 63 XOR $18 $19
5 l_r_listLength: DATA 0 64 EXCH $19 $17
6 l_Program_vt: DATA 972 65 RL $9 1
7 l_LinkedList_vt: DATA 459 66 EXCH $10 $1
8 DATA 562 67 ADDI $1 -1
9 DATA 704 68 EXCH $11 $1
10 DATA 945 69 ADDI $1 -1
11 l_Cell_vt: DATA 223 70 EXCH $12 $1
12 DATA 252 71 ADDI $1 -1
13 l_malloc_top: BRA l_malloc_bot 72 EXCH $14 $1
14 l_malloc: SWAPBR $2 73 ADDI $1 -1
15 NEG $2 74 EXCH $16 $1
16 ADDI $9 2 75 ADDI $1 -1
17 XOR $8 0 76 EXCH $17 $1
18 ADDI $1 1 77 ADDI $1 -1
19 EXCH $6 $1 78 EXCH $18 $1
20 ADDI $1 1 79 ADDI $1 -1
21 EXCH $7 $1 80 EXCH $20 $1
22 EXCH $2 $1 81 ADDI $1 -1
23 ADDI $1 -1 82 EXCH $21 $1
24 BRA l_malloccl 83 ADDI $1 -1
25 ADDI $1 1 84 EXCH $22 $1
26 EXCH $2 $1 85 ADDI $1 -1
27 EXCH $7 $1 86 EXCH $23 $1
28 ADDI $1 -1 87 ADDI $1 -1
29 EXCH $6 $1 88 EXCH $24 $1
30 ADDI $1 -1 89 ADDI $1 -1
31 XOR $8 0 90 EXCH $23 $1
32 ADDI $9 -2 91 ADDI $1 -1
33 l_malloc_bot: BRA l_malloc_top 92 EXCH $22 $1
34 l_malloccl_top: BRA l_malloccl_bot 93 ADDI $1 -1
35 ADDI $1 1 94 EXCH $21 $1
36 EXCH $2 $1 95 ADDI $1 -1
37 SUB $17 $8 96 EXCH $20 $1
38 XOR $17 4 97 ADDI $1 -1
39 l_malloccl: SWAPBR $2 98 EXCH $18 $1
40 NEG $2 99 ADDI $1 -1
41 EXCH $2 $1 100 EXCH $17 $1
42 ADDI $1 -1 101 ADDI $1 -1
43 XOR $17 4 102 EXCH $16 $1
44 ADD $17 8 103 ADDI $1 -1
45 EXCH $19 17 104 EXCH $14 $1
46 XOR $18 19 105 ADDI $1 -1
47 EXCH $19 17 106 EXCH $12 $1
48 XOR $13 9 107 ADDI $1 -1
49 SUB $13 7 108 EXCH $11 $1
50 cmp_top_7: BGEZ $13 cmp_bot_8109 ADDI $1 1
51 XOR $14 1 110 EXCH $10 $1
52 cmp_bot_8: BGEZ $13 cmp_top_7111 RR $9 1
53 XOR $10 14 112 ADDI $8 -1
54 cmp_bot_8_i: BGEZ $13 113 XORI $10 1
55 cmp_top_7_i 114 l_o_assert_true: BRA l_o_assert
56 XORI $14 1 115 l_o_test_false: BRA l_o_test
57 cmp_top_7_i 116 cmp_top_11: BEQ $18 $0
58 cmp_bot_8_i: BGEZ $13 116 cmp_bot_12
59 ADD $13 7 117 cmp_bot_12: BEQ $18 $0
60 XOR $13 9 118 l_o_test_false
61 BEQ $10 0 119 l_o_test
62
Appendix C  Example Ouput 130 of 231
|   | cmp_top_21:  |
|---|-------------|
| 236 | ADDI $7 -3 |
| 237 | SUB $7 $3  |
| 238 | EXCH $9 $6 |
| 239 | XOR $8 $9  |
| 240 | EXCH $9 $6 |
| 241 | ADD $7 $3  |
| 242 | ADDI $7 $3 |
| 243 | EXCH $8 $7 |
| 244 | ADDI $7 $3 |

|   | cmp_bot_22_i: |
|---|--------------|
| 245 | SUB $7 $3  |
| 246 | ADDI $8 $3 |
| 247 | ADDI $8 $2 |
| 248 | EXCH $9 $8 |

|   | cmp_top_23:  |
|---|-------------|
| 249 | EXCH $2 $1 |
| 250 | EXCH $6 $1 |
| 251 | ADDI $1 -1 |
| 252 | EXCH $3 $1 |
| 253 | ADDI $1 -1 |
| 254 | EXCH $6 $1 |
| 255 | ADDI $1 -1 |
| 256 | EXCH $3 $1 |
| 257 | ADDI $1 -1 |
| 258 | EXCH $6 $1 |
| 259 | ADDI $1 -1 |
| 260 | EXCH $2 $1 |

|   | cmp_bot_24:  |
|---|-------------|
| 261 | EXCH $9 $8 |
| 262 | ADD $8 $3  |
| 263 | ADDI $8 $2 |
| 264 | EXCH $9 $8 |
| 265 | ADDI $8 -2 |
| 266 | SUB $8 $3 |
| 267 | BNE $9 $0  |

|   | cmp_top_22:  |
|---|-------------|
| 268 | XORI $10 1 |
| 269 | BNE $9 $0  |

|   | cmp_bot_22:  |
|---|-------------|
| 270 | EXCH $11 $6|
| 271 | BEQ $11 $0 |

|   | cmp_bot_23:  |
|---|-------------|
| 272 | XOR $12 1  |
| 273 | BEQ $11 $0 |

|   | cmp_bot_23:  |
|---|-------------|
| 274 | ANDX $13 $10 $12|
| 275 | BBQ $13 $0  |

|   | f_bot_26:    |
|---|-------------|
| 276 | XOR $14 1  |
| 277 | BEQ $13 $0 |

|   | f_bot_26:    |
|---|-------------|
| 278 | XOR $7 $14 |
| 279 | BEQ $13 $0 |

|   | f_bot_26_i:  |
|---|-------------|
| 280 | XOR $14 1  |
| 281 | BEQ $13 $0 |

|   | f_bot_26_i:  |
|---|-------------|
| 282 | ANDX $13 $10 $12|
| 283 | BBQ $11 $0  |

|   | cmp_bot_24_i: |
|---|--------------|
| 284 | BBQ $11 $0  |
| 285 | BBQ $11 $0  |

|   | cmp_bot_23_i: |
|---|--------------|
| 286 | EXCH $11 $6 |
| 287 | BNE $9 $0   |

|   | cmp_bot_22_i: |
|---|--------------|
| 288 | XORI $7 $3  |
| 289 | BNE $9 $0   |

|   | cmp_bot_21_i: |
|---|--------------|
| 290 | ADD $8 $3  |
| 291 | ADDI $8 $2  |
| 292 | EXCH $9 $8 |
| 293 | ADDI $8 $2  |
| 294 | SUB $8 $3  |
| 295 | BEQ $7 $0  |
| 296 | XOR $7 $1  |
| 297 | ADDI $8 $3 |
| 298 | ADDI $8 $2 |
| 299 | EXCH $9 $8 |
| 300 | ADDI $8 $2  |
| 301 | SUB $8 $3  |
| 302 | EXCH $10 $6|
| 303 | swap_27:    |
| 304 | XOR $9 $10 |
| 305 | XOR $9 $10 |
| 306 | EXCH $10 $6|
| 307 | ADD $8 $3  |
| 308 | ADDI $8 $2 |
| 309 | EXCH $9 $8 |
| 310 | ADDI $8 $2  |
| 311 | SUB $8 $3  |
| 312 | XORI $7 $1 |
| 313 | assert_true_18: |
| 314 | test_false_19: |
| 315 | assert_20:   |
| 316 | ADD $9 $0   |
| 317 | ADDI $8 $2  |
| 318 | EXCH $9 $8 |
| 319 | SUB $8 $3  |
| 320 | BNE $9 $0  |
| 321 | cmp_bot_28:  |
| 322 | BEQ $9 $0  |
| 323 | cmp_bot_29:  |
| 324 | EXCH $11 $6|
| 325 | cmp_top_30:  |
| 326 | XORI $12 1  |
| 327 | cmp_bot_31:  |
| 328 | ANDX $13 $10 $12|
| 329 | cmp_bot_32:  |
| 330 | f_bot_33:    |
| 331 | f_bot_33:    |
| 332 | f_bot_32:    |
| 333 | f_bot_33_i:  |
| 334 | f_bot_32_i:  |
| 335 | f_bot_32_i:  |
| 336 | f_bot_33_i:  |
| 337 | cmp_bot_31_i:|
| 338 | cmp_top_30_i:|
| 339 | cmp_top_30_i:|
| 340 | cmp_bot_31_i:|
| 341 | cmp_bot_29_i:|
| 342 | cmp_top_28_i:|

Appendix C  Example Output 131 of 231
l_insertHead_1_bot
455 ADDI $1 1
456 EXCH $2 $1
457 EXCH $6 $1
458 ADDI $1 -1
459 EXCH $3 $1
460 ADDI $1 -1
461 l_insertHead_1:
462 SWAPBR $2
463 NEG $2
464 ADDI $1 1
465 EXCH $3 $1
466 ADDI $1 1
467 EXCH $6 $1
468 EXCH $2 $1
469 ADDI $1 -1
470 ADDI $8 3
471 l_insertHead_1_bot:
472 ADDI $8 $3
473 SUB $3
474 cmp_top_52:
475 BEQ $9 $0
476 cmp_bot_53
477 XORI $10 1
478 cmp_bot_53:
479 BEQ $9 $0
480 cmp_top_54:
481 BEQ $11 $0
482 xor $7 $14
483 cmp_top_54:
484 BEQ $11 $0
485 xor $7 $14
486 cmp_top_54:
487 BEQ $11 $0
488 xor $7 $14
489 cmp_top_54:
490 BEQ $11 $0
491 xor $7 $14
492 cmp_top_54:
493 BEQ $11 $0
494 xor $7 $14
495 cmp_top_54:
496 BEQ $11 $0
497 xor $7 $14
498 cmp_top_54:
499 BEQ $11 $0
500 xor $7 $14
501 cmp_top_54:
502 test_48:
503 test_false_50
504 ADDI $9 $8
505 ADDI $9 $8
506 Appendix C  Example Output 133 of 231
ADDI $1 1 615 | XOR $13 $12

EXCH $2 $1 616 | EXCH $12 $11

EXCH $6 $1 617 | ADDI $11 -1

EXCH $1 -1 618 | EXCH $11 $10

EXCH $3 $1 619 | ADD $8 $3

EXCH $1 -1 620 | ADDI $8 2

SWAPBR $2 621 | EXCH $9 $8

NEG $2 622 | ADDI $8 -2

ADDI $1 1 623 | SUB $8 $3

EXCH $3 $1 624 | EXCH $3 $1

ADDI $1 1 625 | ADDI $1 -1

EXCH $6 $1 626 | EXCH $6 $1

EXCH $2 $1 627 | ADDI $1 -1

ADDI $1 -1 628 | EXCH $10 $1

EXCH $6 $1 629 | ADDI $1 -1

ADDI $1 -1 630 | ADDI $13 -629

EXCH $3 $1 631 | SWAPBR $13

ADDI $1 -1 632 | NEG $13

BRA l_insertHead_433 | ADDI $13 629

ADDI $1 1 634 | ADDI $1 1

EXCH $3 $1 635 | EXCH $10 $1

ADDI $1 1 636 | ADDI $1 1

EXCH $6 $1 637 | EXCH $6 $1

ADD $8 $3 638 | ADDI $1 1

ADDI $8 2 639 | EXCH $3 $1

EXCH $9 $8 640 | ADD $8 $3

ADDI $8 -2 641 | ADDI $8 2

ADDI $1 1 642 | EXCH $9 $8

SUB $8 $3 643 | ADDI $8 -2

BEQ $9 $0 644 | SUB $8 $3

XORI $10 1 645 | EXCH $11 $10

BEQ $9 $0 646 | ADDI $11 1

BEQ $10 $0 647 | EXCH $10 $1

SUB $8 $3 648 | XOR $11 1

XOR $9 $8 649 | ADDI $11 -1

EXCH $6 $1 650 | ADDI $11 -1

BEQ $10 $0 651 | loadMetAdd_73_i:

BEQ $9 $0 652 | EXCH $11 $10

loadMetAdd_73: | EXCH $11 $10

BEQ $9 $0 653 | test_false_67:

BEQ $10 $0 654 | BRA assert_68

BEQ $9 $0 655 | BRA test_65

BEQ $10 $0 656 | assert_true_66:

BEQ $9 $0 657 | BRA assert_68

BEQ $10 $0 658 | BRA test_65

BEQ $9 $0 659 | assert_true_66:

BEQ $9 $0 660 | loadMetAdd_73: | BRA assert_68

BEQ $9 $0 661 | BRA test_65

BEQ $10 $0 662 | assert_true_66:

ADD $8 $3 663 | BRA assert_68

ADDI $8 2 664 | BRA test_65

EXCH $9 $8 665 | assert_true_66:

ADDI $8 -2 666 | EXCH $11 $10

ADDI $8 -2 667 | EXCH $11 $10

SUB $8 $3 668 | EXCH $11 $10

SUB $8 $3 669 | EXCH $11 $10

CMP $8 $3 670 | EXCH $11 $10

ADD $8 $3 671 | EXCH $11 $10

CMP $8 $3 672 | EXCH $11 $10

CMP $9 $9 673 | EXCH $11 $10

CMP $9 $9 674 | EXCH $11 $10

CMP $9 $9 675 | EXCH $11 $10

ADDI $11 10 676 | EXCH $11 $10

ADDI $11 10 677 | EXCH $11 $10

EXCH $12 $11 678 | EXCH $12 $11
| Line | Instruction | Address | Raw Text |
|------|-------------|---------|----------|
| 1013 | XORI $8 10 | 1062 | ADDI $9 -4 |
| 1014 | ADD 6 53 | 1063 | SUB 9 $3 |
| 1015 | ADDI $6 4 | 1064 | EOR 8 $6 |
| 1016 | EXCH $7 06 | 1065 | test_116: BNE $7 00 exit_118 |
| 1017 | ADDI $6 -4 | 1066 | localBlock_128: XOR 8 $1 |
| 1018 | SUB 6 53 | 1067 | XOR 9 $0 |
| 1019 | localBlock_133: | 1068 | EXCH 9 $1 |
| 1020 | XOR 6 $1 | 1069 | ADDI 9 $1 -1 |
| 1021 | XOR 7 $0 | 1070 | EXCH 9 $3 $1 |
| 1022 | ADDI $1 -1 | 1071 | ADDI 1 -1 |
| 1023 | XORI $7 1 | 1072 | EXCH 9 $8 |
| 1024 | entry_115: | 1073 | ADDI 1 -1 |
| 1025 | BEQ $7 50 | 1074 | EXCH 6 $1 |
| 1026 | EXCH 6 50 | 1075 | ADDI 6 4 |
| 1027 | XORI $6 0 | 1076 | obj_con_123: ADDI 6 4 |
| 1028 | cmp_top_119: | 1077 | EXCH 10 $1 |
| 1029 | cmp_bot_120: | 1078 | ADDI 1 -1 |
| 1030 | f_top_121: | 1079 | EXCH 9 $1 |
| 1031 | f_bot_122: | 1080 | ADDI 1 -1 |
| 1032 | cmp_bot_120: | 1081 | BRA 1_malloc |
| 1033 | f_top_121: | 1082 | ADDI 1 1 |
| 1034 | f_bot_122: | 1083 | EXCH 9 $1 |
| 1035 | cmp_top_119: | 1084 | ADDI 1 1 |
| 1036 | cmp_bot_120: | 1085 | EXCH 9 $1 |
| 1037 | cmp_top_119: | 1086 | obj_con_123_bot: ADDI 10 -4 |
| 1038 | cmp_bot_120: | 1087 | ADDI 9 $1 |
| 1039 | cmp_top_118: | 1088 | EXCH 9 $1 |
| 1040 | cmp_bot_120: | 1089 | ADDI 9 $1 |
| 1041 | cmp_top_119: | 1090 | EXCH 9 $1 |
| 1042 | cmp_bot_120: | 1091 | ADDI 9 $1 |
| 1043 | cmp_top_119: | 1092 | EXCH 9 $1 |
| 1044 | cmp_bot_120: | 1093 | XORI 10 9 |
| 1045 | cmp_top_118: | 1094 | EXCH 9 $9 |
| 1046 | cmp_bot_120: | 1095 | ADDI 9 $9 |
| 1047 | cmp_top_119: | 1096 | XORI 10 1 |
| 1048 | cmp_bot_120: | 1097 | EXCH 9 $9 |
| 1049 | cmp_top_118: | 1098 | obj_con_123_i: ADDI 10 -4 |
| 1050 | cmp_bot_120: | 1099 | ADDI 9 $1 |
| 1051 | cmp_top_119: | 1100 | EXCH 9 $1 |
| 1052 | cmp_bot_120: | 1101 | ADDI 9 $1 |
| 1053 | cmp_top_118: | 1102 | EXCH 9 $1 |
| 1054 | cmp_bot_120: | 1103 | ADDI 9 $1 |
| 1055 | cmp_top_119: | 1104 | EXCH 9 $1 |
| 1056 | cmp_bot_120: | 1105 | ADDI 11 0 |
| 1057 | cmp_top_118: | 1106 | EXCH 11 $10 |
| 1058 | cmp_bot_120: | 1107 | ADDI 11 0 |
| 1059 | cmp_top_119: | 1108 | EXCH 11 $10 |
| 1060 | cmp_bot_120: | 1109 | ADDI 11 0 |
| 1061 | cmp_top_118: | 1110 | EXCH 11 $10 |
| | | 1111 | EXCH 9 $8 |
| | | 1112 | EXCH 9 $8 |
| | | 1113 | EXCH 9 $8 |
| | | 1114 | EXCH 9 $8 |
| | | 1115 | EXCH 9 $8 |
| | | 1116 | EXCH 9 $8 |
| | | 1117 | EXCH 9 $8 |
| | | 1118 | EXCH 9 $8 |
| | | 1119 | EXCH 9 $8 |
| | | 1120 | EXCH 9 $8 |
| | | 1121 | EXCH 9 $8 |
| | | 1122 | EXCH 9 $8 |
| | | 1123 | EXCH 9 $8 |
| | | 1124 | EXCH 9 $8 |
| | | 1125 | EXCH 9 $8 |
| | | 1126 | EXCH 9 $8 |
| | | 1127 | EXCH 9 $8 |
ADDI $1 1 1194
ADDI $9 -2
EXCH $3 $1 1195
SUB $9 $3
EXCH $9 $8 1196
ADDI $1 1
EXCH $11 $10 1197
EXCH $9 $1
ADDI $11 0 1198
XOR $9 $0
EXCH $12 $11 1199 localBlock_128_i:
XOR $8 $1
XOR $13 $12 1200
EXCH $8 $6
EXCH $12 $11 1201
XORI $9 1
ADDI $11 0 1202
ADD $8 $9
EXCH $11 $10 1203
XORI $9 1
XOR $10 $9 1204
EXCH $8 $6
EXCH $9 $8 1205 assert_l17:
ADDI $9 $3 1206 exit_l18:
BRA entry_l15
ADDI $9 2 1207
XORI $7 1
EXCH $10 $9 1208
ADD $7 $3
ADDI $9 -2 1209
ADDI $7 4
SUB $9 $3 1210
EXCH $8 $7
ADDI $12 1 1211
ADDI $1 1
EXCH $13 $12 1214
EXCH $9 $1
EXCH $14 $13 1215
XORI $9 $8
EXCH $13 $12 1216 localBlock_133_i:
XORI $6 $1
ADDI $12 -1 1217
ADDI $7 $3
EXCH $12 $11 1218
ADDI $7 4
ADDI $9 2 1219
ADDI $8 $7
ADDI $9 -2 1220
ADDI $7 -4
EXCH $10 $9 1221 SUB $7 $3
ADDI $9 -2 1222 l_main_0_bot:
ADDI $10 $9 1223 l_main_0_top:
SUB $9 $3 1224 l_main_0
SUB $9 $3 1225 start:
SUB $9 $3 1226 top
EXCH $3 $1 1227
START
ADDI $1 -1 1228
ADDI $4 1279
EXCH $6 $1 1229
XOR $5 $4
ADDI $1 -1 1230
ADDI $5 10
EXCH $8 $1 1231
XORI $7 $5
ADDI $12 -1 1232
ADDI $4 10
EXCH $11 $1 1233
ADDI $8 $4
EXCH $9 $1 1234
ADDI $4 $3
EXCH $10 $1 1235
ADDI $3 $1
ADDI $9 $3 1244
ADDI $1 1
EXCH $3 $1 1243
ADDI $9 $3
ADDI $2 1245
ADDI $5 $1
EXCH $10 $9 1246
ADDI $3 1
ADDI $9 -2 1247
ADDI $3 1
SUB $9 $3 1248
EXCH $6 $3
EXCH $12 $11 1249
XORI $7 1
ADDI $12 1 1250
EXCH $6 $7
EXCH $13 $12 1251
XORI $7 1
EXCH $14 $13 1252
ADDI $3 -1
EXCH $13 $12 1253
ADDI $3 -1
ADDI $12 -1 1254
ADDI $3 1
EXCH $12 $11 1255
ADDI $3 2
EXCH $11 $10 1256
ADDI $6 $3
ADDI $9 $3 1257
ADDI $7 2
ADDI $9 2 1258
ADDI $6 $7
EXCH $10 $9 1259
ADDI $7 2

Appendix C  Example Ouput 139 of 231
| Line | Instruction | Address |
|------|-------------|---------|
| 1260 | ADDI $3 -2  | 1271    |
| 1261 | ADDI $3 -1  | 1272    |
| 1262 | ADDI $3 1   | 1273    |
| 1263 | ADDI $3 3   | 1274    |
| 1264 | EXCH $6 $3  | 1275    |
| 1265 | XORI $7 3   | 1276    |
| 1266 | EXCH $6 $7  | 1277    |
| 1267 | XORI $7 3   | 1278    |
| 1268 | ADDI $3 -3  | 1279    |
| 1269 | ADDI $3 -1  | 1280 finish: |
| 1270 | ADDI $1 1   |         |

EXCH  $6 $3
XORI  $6 4
XOR   $6 $1
ADDI  $1 8
ADDI  $1 -2048
XORI  $1 5
ADDI  $5 -10
XOR   $5 $4
ADDI  $4 -1279
FINISH
class Node
    Node left
    Node right
    int value

method setValue(int newValue)
    value ^= newValue

method insertNode(Node node, int nodeValue)
    if nodeValue < value then
        if left = nil & node != nil then
            left <=> node // If open left node, store here
        else skip
            if left != nil then
                call left::insertNode(node, nodeValue) // If current node has left, continue iterating
            else skip
                if right != nil then
                    call right::insertNode(node, nodeValue) // If current node has right, continue searching
                else skip
                    fi
                fi
            else
                if right = nil & node != nil then
                    right <=> node // If open right node spot, store here
                else skip
                    if right != nil then
                        call right::insertNode(node, nodeValue) // If current node has right, continue searching
                    else skip
                        fi
                    fi
                else
                    fi
                fi
            fi
        else
            if right != nil then
                call right::insertNode(node, nodeValue) // If current node has right, continue searching
            else skip
                fi
            fi
        fi
    else
        if right = nil & node != nil then
            right <=> node // If open right node spot, store here
        else skip
            if right != nil then
                call right::insertNode(node, nodeValue) // If current node has right, continue searching
            else skip
                fi
            fi
        else
            fi
        fi
    fi

method getSum(int result)
    result += value // Add the value of this node to the sum
    if left != nil then
        call left::getSum(result) // If we have a left child, follow that path
    else skip // Else, skip
        fi
    if right != nil then
        call right::getSum(result) // If we have a right child, follow that path
    else skip // Else, skip
        fi
    fi

method mirror()
    left <=> right // Swap left and right children
    if left = nil then skip
    else call left::mirror() // Recursively swap children if left != nil
    fi
    if right = nil then skip
    else call right::mirror() // Recursively swap children if right != nil
    fi

class Tree
    Node root

method insertNode(Node node, int value)
    if root = nil & node != nil then
root <=> node
else skip
fi root != nil & node = nil
if root != nil then
call root::insertNode(node, value)
else skip
fi root != nil

method sum(int result)
if root != nil then
call root::getSum(result)
else skip
fi root != nil

method mirror()
if root != nil then
call root::mirror()
else skip
fi root != nil

class Program
int sumResult
Tree tree
int nodeCount
method main()
new Tree tree
nodeCount += 3
local int x = 0
from x = 0 do
skip
local Node node = nil
new Node node // Init new node
call node::setValue(x) // Set node value
call tree::insertNode(node, x) // Insert node in tree
delocal Node node = nil
x += 1
until x = nodeCount
delocal int x = nodeCount
call tree::sum(sumResult)
call tree::mirror()
BinaryTree.pal

1 ;; pendulum pal file
2 top: BRA start 61 l_o_test_false XORI $10 1
3 l_r_sumResult: DATA 0 62 ADDI $8 1
4 l_r_tree: DATA 0 63 EXCH $19 $17
5 l_r_nodeCount: DATA 0 64 XOR $18 $19
6 l_Program_vt: DATA 1644 65 EXCH $19 $17
7 l_Tree_vt: DATA 1201 66 RL $9 1
8 l_r_sumResult: DATA 1414 67 EXCH $10 $1
9 l_r_tree: DATA 1532 68 ADDI $1 -1
10 l_r_nodeCount: DATA 224 69 EXCH $11 $1
11 l_Program_vt: DATA 255 70 ADDI $1 -1
12 l_Tree_vt: DATA 727 71 EXCH $12 $1
13 DATA 962 72 ADDI $1 -1
14 l_malloc_top: BRA l_malloc_bot 73 EXCH $14 $1
15 l_malloc: SWAPBR $2 74 ADDI $1 -1
16 NEG $2 75 EXCH $16 $1
17 ADDI $9 2 76 ADDI $1 -1
18 XOR $8 0 77 EXCH $17 $1
19 ADDI $1 1 78 ADDI $1 -1
20 EXCH $6 0 79 EXCH $18 $1
21 ADDI $1 1 80 ADDI $1 -1
22 EXCH $7 1 81 EXCH $20 $1
23 EXCH $2 1 82 ADDI $1 -1
24 ADDI $1 -1 83 EXCH $21 $1
25 BRA l_malloccl 84 ADDI $1 -1
26 ADDI $1 1 85 EXCH $2 $1 86 ADDI $1 -1
27 EXCH $7 1 87 EXCH $23 $1
28 ADDI $1 -1 88 ADDI $1 -1
29 EXCH $6 1 89 BRA l_malloc
30 ADDI $1 -1 90 ADDI $1 -1
31 XOR $8 0 91 EXCH $23 $1
32 ADDI $9 -2 92 ADDI $1 1
33 l_malloccl: BRA l_malloccl_top 93 EXCH $22 $1
34 l_malloccl_top: BRA l_malloccl_bot 94 ADDI $1 1
35 ADDI $1 1 95 ADDI $1 1
36 EXCH $2 1 96 EXCH $21 $1
37 EXCH $7 1 97 ADDI $1 1
38 ADDI $1 1 98 EXCH $23 $1
39 EXCH $6 1 99 ADDI $1 1
40 ADDI $1 1 100 ADDI $1 1
41 ADDI $1 -1 101 ADDI $1 1
42 XOR $17 4 102 XOR $16 $1
43 ADDI $1 1 103 ADDI $1 1
44 XOR $17 4 104 ADDI $1 1
45 ADDI $1 1 105 ADDI $1 1
46 XOR $19 17 106 ADDI $1 1
47 XOR $18 19 107 ADDI $1 1
48 XOR $19 17 108 ADDI $1 1
49 XOR $13 9 109 ADDI $1 1
50 SUB $13 7 110 ADDI $1 1
51 cmp_top_8: BGEZ $13 cmp_bot_9 111 ADDI $1 1
52 cmp_bot_9: BGEZ $13 cmp_top_8 112 ADDI $1 1
53 cmp_bot_9_i: BGEZ $13 113 ADDI $8 -1
54 cmp_top_8_i: BGEZ $13 cmp_bot_9 114 XORI $10 $1
55 cmp_bot_9_i: cmp_bot_8_i: 115 l_o_assert_true: BRA l_o_assert
56 cmp_bot_8_i: XORI $14 1 116 l_o_test_false: BRA l_o_test
57 cmp_top_8_i: BEQ $18 $0
58 cmp_bot_9_i: cmp_top_12: BEQ $18 $0
59 ADD $13 7 118 cmp_bot_13: BEQ $20 $0
60 XOR $13 9 119 cmp_top_12

Appendix C Example Output 143 of 231
120  XOR $11 $20             183  XOR $11 $20
121 cmp_bot_13_i:              184  BEQ $18 $0
122 BEQ $18 $0               185  ADD $6 $9
123 cmp_top_12_i:             186  cmp_bot_15:
124 cmp_bot_13_i:             187  l_i_assert:
125 cmp_top_12_i:             188  cmp_bot_14:
126 cmp_top_13_i:             189  l_i_assert:
127 cmp_bot_13_i:             190  cmp_bot_16:
128 l_i_test:                191  cmp_top_14_i:
129 l_i_test_false:          192  cmp_top_16_i:
130 XORI $20 1                193  cmp_bot_17:
131 cmp_top_13_i:             194  cmp_bot_17:
132 cmp_bot_13_i:             195  cmp_bot_15:
133 l_i_assertrue:            196  cmp_bot_16:
134 l_i_asserfalse:           197  cmp_bot_16_i:
135 XORI $11 1                198  cmp_bot_17_i:
136 ADD $6 $18                199  cmp_bot_17:
137 SUB $18 $6                200  cmp_bot_15_i:
138 EXCH $12 $6               201  cmp_bot_15:
139 EXCH $12 $17              202  cmp_bot_16:
140 XOR $12 $6                203  cmp_bot_14_i:
141 XORI $11 1                204  cmp_bot_14:
142 ADD $6 $18                205  cmp_bot_15_i:
143 EXCH $12 $6               206  cmp_bot_15:
144 EXCH $12 $17              207  l_i_assertrue:
145 XORI $11 1                208  l_i_assertrue:
146 ADD $6 $9                 209  l_i_assertrue:
147 EXCH $12 $6               210  cmp_bot_15:
148 ADDI $1 -1                211  cmp_bot_16:
149 cmp_bot_17_i:             212  cmp_bot_17:
150 cmp_bot_16_i:             213  cmp_bot_17:
151 cmp_bot_16_i:             214  cmp_bot_17:
152 cmp_bot_15_i:             215  cmp_bot_15:
153 cmp_bot_15_i:             216  cmp_bot_15:
154 cmp_bot_15_i:             217  cmp_bot_15:
155 cmp_bot_15_i:             218  cmp_bot_15:
156 cmp_bot_15_i:             219  cmp_bot_15:
157 cmp_bot_15_i:             220  cmp_bot_15:
158 cmp_bot_15_i:             221  cmp_bot_15:
159 cmp_bot_15_i:             222  cmp_bot_15:
160 cmp_bot_15_i:             223  cmp_bot_15:
161 cmp_bot_15_i:             224  cmp_bot_15:
162 cmp_bot_15_i:             225  cmp_bot_15:
163 cmp_bot_15_i:             226  cmp_bot_15:
164 cmp_bot_15_i:             227  cmp_bot_15:
165 cmp_bot_15_i:             228  cmp_bot_15:
166 cmp_bot_15_i:             229  cmp_bot_15:
167 cmp_bot_15_i:             230  cmp_bot_15:
168 cmp_bot_15_i:             231  cmp_bot_15:
169 cmp_bot_15_i:             232  cmp_bot_15:
170 cmp_bot_15_i:             233  cmp_bot_15:
171 cmp_bot_15_i:             234  cmp_bot_15:
172 cmp_bot_15_i:             235  cmp_bot_15:

Appendix C  Example Output 144 of 231
Appendix C  Example Output 145 of 231
456  XOR $15 $14 509  f_top_65: BEQ $15 $0
457  EXCH $14 $13 510  f_bot_66  XORI $16 $1
458  ADDI $13 -1 511  f_bot_66: BEQ $15 $0
459  EXCH $13 $12 512  XOR $9 $16
460  XOR $12 $11 513  f_bot_66_i: BEQ $15 $0
461  ADD $10 $3 514  ADDI $16 $1
462  ADDI $10 2 515  f_top_65_i: XORI $16 $1
463  EXCH $11 $10 516  SUB $10 $3
464  ADDI $10 -2 517  f_top_65_i: BEQ $15 $0
465  SUB $10 $3 518  f_bot_66_i
466  XORI $9 1 519  assert_true_44: BRA assert_46
467  assert_true_44:
468  assert_false_45:
469  BRA test_43
470  assert_46:
471  ADDI $10 2
472  EXCH $11 $10
473  ADDI $10 -2
474  SUB $10 $3
475  BEQ $11 $0
476  cmp_top_53: 522  cmp_top_53:
477  cmp_bot_54:
478  cmp_top_53:
479  cmp_bot_54:
480  cmp_top_53:
481  cmp_bot_54:
482  cmp_top_53:
483  cmp_bot_54:
484  cmp_top_53:
485  cmp_bot_54:
486  cmp_top_53:
487  cmp_bot_54:
488  ADDI $10 3
489  EXCH $11 $10
490  ADDI $10 -2
491  SUB $10 $3
492  XORI $9 1
493  assert_true_58: 540
494  assert_true_58: 541  swap_67:
495  test_false_59: 542  test_false_59:
496  test_false_59:
497  ADDI $10 3
498  EXCH $11 $10
499  ADDI $10 -2
500  SUB $10 $3
501  cmp_top_53: 552  cmp_top_53:
502  cmp_bot_54:
503  cmp_bot_54:
504  cmp_top_53: 553  cmp_top_53:
505  cmp_bot_62: 554  cmp_bot_62:
506  cmp_bot_62:
507  cmp_bot_64: 555  cmp_bot_68:
508  cmp_top_63: 556  cmp_bot_69
509  cmp_bot_69
510  cmp_bot_68
511  cmp_bot_68
512  cmp_bot_68
513  cmp_bot_68
514  cmp_bot_68
515  cmp_bot_68
516  cmp_bot_68
517  cmp_bot_68
518  cmp_bot_68
519  cmp_bot_68
520  cmp_bot_68
521  cmp_bot_68
522  cmp_bot_68
523  cmp_bot_68
524  cmp_bot_68
525  cmp_bot_68
526  cmp_bot_68
527  cmp_bot_68
528  cmp_bot_68
529  cmp_bot_68
530  cmp_bot_68
531  cmp_bot_68
532  cmp_bot_68
533  cmp_bot_68
534  cmp_bot_68
535  cmp_bot_68
536  cmp_bot_68
537  cmp_bot_68
538  cmp_bot_68
539  cmp_bot_68
540  cmp_bot_68
541  cmp_bot_68
542  cmp_bot_68
543  cmp_bot_68
544  cmp_bot_68
545  cmp_bot_68
546  cmp_bot_68
547  cmp_bot_68
548  cmp_bot_68
549  cmp_bot_68
550  cmp_bot_68
551  cmp_bot_68
552  cmp_bot_68
553  cmp_bot_68
554  cmp_bot_68
555  cmp_bot_68
556  cmp_bot_68
557  cmp_bot_68
558  cmp_bot_68
559  cmp_bot_68
560  cmp_bot_68
561  cmp_bot_68
562  cmp_bot_68
Appendix C  Example Output 150 of 231
Appendix C  Example Output 151 of 231
APPENDIX C EXAMPLE OUTPUT 152 OF 231
| Address  | Instruction                              | Address  | Instruction                              |
|----------|------------------------------------------|----------|------------------------------------------|
| 1237     | cmp_bot_156_i                            | 1289     | XORI $13 1                               |
| 1238     | cmp_bot_154_i:                           | 1290     | BEQ $12 0                               |
|          | cmp_top_153_i                            | 1291     | EXCH $12 6                               |
| 1239     | XORI $11 1                               | 1292     | cmp_bot_161_i:                           |
| 1240     | cmp_top_153_i:                           |          | BEQ $10 0                               |
|          | cmp_bot_154_i                            | 1293     | EXCH $12 6                               |
| 1241     | ADD $9 3                                 | 1294     | cmp_bot_160_i:                           |
| 1242     | ADDI $9 2                                | 1295     | BEQ $10 0                               |
| 1243     | EXCH $10 9                               | 1296     | ADD $9 3                                 |
| 1244     | ADDI $9 -2                               | 1297     | ADDI $9 2                                |
| 1245     | SUB $9 3                                 | 1298     | EXCH $10 9                               |
| 1246     | BEQ $8 0                                 | 1299     | ADDI $9 -2                               |
|          | test_149:                                | 1300     | SUB $9 $3                                |
| 1247     | XORI $8 1                                | 1301     | ADD $9 $3                                |
| 1248     | ADD $9 3                                 | 1302     | ADDI $9 2                                |
| 1249     | ADDI $9 2                                | 1303     | EXCH $10 9                               |
| 1250     | EXCH $10 9                               | 1304     | ADDI $9 -2                               |
| 1251     | ADDI $9 -2                               | 1305     | SUB $9 $3                                |
| 1252     | SUB $9 3                                 |          | cmp_bot_170:                             |
| 1253     | EXCH $11 6                               |          | BEQ $10 0                               |
| 1254     | XOR $10 11                               | 1306     | cmp_bot_171:                             |
| 1255     | XOR $11 10                               | 1307     | BEQ $10 0                               |
| 1256     | XOR $10 11                               |          | cmp_top_170:                             |
| 1257     | EXCH $11 6                               | 1308     | f_top_172:                               |
| 1258     | ADD $9 3                                 | 1309     | f_bot_173:                               |
| 1259     | ADDI $9 2                                |          | XORI $12 1                               |
| 1260     | EXCH $10 9                               | 1310     | f_bot_172:                               |
| 1261     | ADDI $9 -2                               |          | BEQ $11 0                               |
| 1262     | SUB $9 3                                 | 1311     | f_bot_173:                               |
| 1263     | XORI $8 1                                | 1312     | f_bot_172:                               |
| 1264     | assert_true_150:                          |          | XOR $8 12                                |
|          | BRA assert_152                           |          | BEQ $11 0                               |
| 1265     | test_false_151:                          |          | assert_true_150                          |
|          | BRA test_169                             |          | BEQ $11 0                               |
| 1266     | assert_152:                              |          | assert_true_150                          |
|          | assert_true_150                          |          | BEQ $10 0                               |
| 1267     | ADD $9 3                                 | 1315     | cmp_bot_171_i:                           |
| 1268     | ADDI $9 2                                | 1316     | cmp_top_170_i:                           |
| 1269     | EXCH $10 9                               | 1317     | cmp_bot_171_i:                           |
| 1270     | ADDI $9 -2                               | 1318     | cmp_top_170_i:                           |
| 1271     | SUB $9 3                                 |          | test_166:                                |
| 1272     | BEQ $10 0                               |          | BEQ $8 0                                |
| 1273     | cmp_top_160:                             | 1319     | test_false_168                           |
|          | cmp_bot_161                             | 1320     | XOR $8 1                                 |
| 1274     | cmp_bot_161:                             | 1321     | ADD $9 3                                 |
|          | cmp_top_160                             | 1322     | ADD $9 2                                 |
| 1275     | EXCH $12 6                               | 1323     | EXCH $10 9                               |
| 1276     | cmp_top_160:                             |          | ADDI $9 -2                               |
|          | cmp_bot_162:                             |          | SUB $9 $3                                |
|          | cmp_bot_163                             |          | cmp_bot_163:                             |
| 1277     | XORI $13 1                               | 1324     | XORI $11 10                              |
| 1278     | cmp_bot_163:                             | 1325     | ADD $9 3                                 |
|          | cmp_top_162                             | 1326     | ADDI $9 2                                 |
| 1279     | ANDX $14 $11 $13                        | 1327     | EXCH $10 9                               |
| 1280     | f_top_164:                               | 1328     | ADDI $9 -2                               |
|          | f_bot_165:                               |          | SUB $9 $3                                |
| 1281     | BEQ $14 0                               |          | xor $11 10                               |
| 1282     | f_bot_165:                               | 1329     | loadMetaAdd_174:                         |
|          | f_bot_164:                               |          | EXCH $12 11                              |
| 1283     | XOR $8 15                                | 1330     | ADD $9 3                                 |
| 1284     | f_bot_165_i:                             | 1331     | ADDI $9 2                                 |
|          | f_top_164_i:                             |          | SUB $9 $3                                |
| 1285     | XOR $14 15                               | 1332     | xor $14 13                               |
| 1286     | f_top_164_i:                             | 1333     | EXCH $13 12                              |
|          | f_bot_165_i:                             |          | ADDI $9 -2                               |
| 1287     | ANDX $14 $11 $13                        | 1334     | SUB $9 $3                                |
| 1288     | cmp_bot_163_i:                           | 1335     | xor $14 13                               |
|          | cmp_top_162_i                           | 1336     | EXCH $12 -1                              |
|          |                                              |          | EXCH $12 11                              |
|          |                                              |          | ADDI $9 $3                                |

Appendix C Example Output
XOR $9 $8 1630
EXCH $10 $9 1631  
f_top_206_i:  
BEQ $9 $0
ADDI $10 3 1632  
f_bot_207_i:  
BEQ $8 $0
EXCH $11 $10 1633  
cmp_top_204_i:  
BEQ $8 $0
EXCH $12 $11 1634  
XORI $10 1
EXCH $11 $10 1635  
ADDI $10 -3 1636  
cmp_top_204_i:  
XORI $9 1
EXCH $10 $9 1637  
ADD $7 3 1638  
ADDI $7 3 1639
EXCH $8 $7 1640  
ADD $8 7 $7
EXCH $7 -2 1641  
ADDI $7 -2
SUB $7 $3 1642  
SUB $7 3 $3
EXCH $3 $1 1643  
l_mirror_3_bot:  
BRA l_mirror_3_top
ADD $3 $1 1644  
l_main_0_top:  
BRA l_main_0_bot
EXCH $9 $1 1645  
ADDI $9 1
ADDI $1 -1 1646  
ADDI $1 -1
ADDI $10 -3 1647  
ADDI $10 1
ADDI $12 1586 1648  
ADDI $1 1
EXCH $9 $1 1649  
EXCH $9 1 $1
ADDI $1 1 1650  
EXCH $1 1
EXCH $3 $1 1651  
EXCH $3 1 $1
ADDI $7 $3 1652  
EXCH $7 3 $3
ADDI $7 3 1653  
ADDI $7 -2 1654  
ADDI $7 1
ADDI $8 $7 1655  
l_mirror_3_bot:  
BRA l_mirror_3_top
ADDI $8 -4 1656  
l_main_0_bot:  
ADD $8 4
ADDI $7 -2 1657  
ADD $7 -2
SUB $7 $3 1658  
SUB $7 3 $3
EXCH $9 $1 1659  
EXCH $9 1
ADDI $11 $10 1660  
ADDI $11 10
ADDI $12 $11 1661  
ADDI $12 11
ADDI $11 $10 1662  
ADDI $11 10
ADDI $10 -3 1663  
ADDI $10 3
ADDI $12 1586 1664  
ADDI $1 1
EXCH $9 $1 1665  
EXCH $9 1
ADDI $1 $1 1666  
EXCH $1 1
EXCH $8 $7 1667  
ADD $8 7
ADDI $7 2 1668  
ADD $7 2
SUB $7 $3 1669  
SUB $7 3 $3
EXCH $9 $1 1670  
XOR $9 1
ADDI $9 $1 1671  
ADDI $9 1
ADDI $17 -1 1672  
ADDI $17 1
BRA assert_197:  
ADDI $1 -1
ADDI $12 -1586 1673  
ADDI $12 1586
ADDI $10 3 1674  
ADDI $10 3
ADDI $11 $10 1675  
ADDI $11 10
ADDI $10 -3 1676  
ADDI $10 1
ADDI $10 -3 1677  
ADDI $10 3
ADDI $10 3 1678  
ADDI $10 3
ADDI $10 -3 1679  
ADDI $10 3
ADDI $10 -3 1680  
ADDI $10 3
ADDI $7 $3 1681  
ADDI $7 3
ADDI $7 $3 1682  
ADDI $7 3
ADDI $7 $3 1683  
ADDI $7 3
ADDI $7 $3 1684  
ADDI $7 3
ADDI $7 $3 1685  
ADDI $7 3
ADDI $7 $3 1686  
ADDI $7 3
ADDI $7 $3 1687  
ADDI $7 3
ADDI $7 $3 1688  
ADDI $7 3
ADDI $7 $3 1689  
ADDI $7 3
ADDI $7 $3 1690  
ADDI $7 3
ADDI $7 $3 1691  
ADDI $7 3
ADDI $7 $3 1692  
ADDI $7 3
1693 | EXCH $7 $1 | 1742 | EXCH $3 $1
1694 | ADDI $1 -1 | 1743 | ADDI $1 -1
1695 | XORI $7 1 | 1744 | ADDI $8 $1
1696 | entry_209: BEQ $7 $0 | 1745 | ADDI $1 -1
          assert_211 | 1746 | EXCH $6 $1
1697 | EXCH $8 $6 | 1747 | ADDI $1 -1
1698 | cmp_top_213: BNE $8 $0 | 1748 | obj_con_217: ADDI $10 8
          cmp_bot_214 | 1749 | EXCH $10 $1
1699 | XORI $9 1 | 1750 | ADDI $1 -1
1700 | cmp_bot_214: BEQ $8 $0 | 1751 | ADDI $9 $1
          cmp_top_213 | 1752 | ADDI $1 -1
1701 | f_top_215: BEQ $9 $0 | 1753 | ADDI $1 -1
          f_bot_216 | 1754 | BRA l_malloc
1702 | XORI $9 1 | 1755 | EXCH $9 $1
1703 | cmp_bot_214: BEQ $8 $0 | 1756 | ADDI $1 -1
          cmp_top_213 | 1757 | EXCH $10 $1
1704 | XORI $9 1 | 1758 | obj_con_217_i: ADDI $10 -8
1705 | cmp_bot_214_i: BEQ $9 $0 | 1759 | ADDI $1 -1
          cmp_top_213_i | 1760 | EXCH $6 $1
1706 | XORI $9 1 | 1761 | ADDI $1 -1
1707 | cmp_bot_215_i: BEQ $9 $0 | 1762 | EXCH $8 $1
          cmp_top_215_i | 1763 | ADDI $1 -1
1708 | XORI $10 1 | 1764 | EXCH $3 $1
1709 | cmp_top_213_i: BEQ $8 $0 | 1765 | XORI $10 8
          cmp_bot_214_i | 1766 | EXCH $10 $9
1710 | XORI $10 1 | 1767 | ADDI $9 1
1711 | EXCH $8 $6 | 1768 | XORI $10 10
1712 | EXCH $8 $6 | 1769 | EXCH $10 $9
1713 | ADD $9 $3 | 1770 | obj_con_217_bot: ADDI $9 -1
1714 | ADD $9 4 | 1771 | EXCH $9 $8
1715 | EXCH $10 $9 | 1772 | EXCH $9 $8
1716 | ADD $9 -4 | 1773 | XOR $10 $9
1717 | SUB $9 $3 | 1774 | loadMetAdd_218: EXCH $11 $10
1718 | cmp_bot_223: BEQ $8 $0 | 1775 | ADDI $11 0
          cmp_bot_224 | 1776 | EXCH $12 $11
1719 | XORI $9 1 | 1777 | XORI $13 $12
1720 | cmp_bot_224: BEQ $8 $0 | 1778 | EXCH $12 $11
          cmp_top_223 | 1779 | ADDI $11 0
1721 | f_top_225: BEQ $11 $0 | 1780 | EXCH $11 $10
          f_bot_226 | 1781 | EXCH $9 $8
1722 | XORI $12 1 | 1782 | EXCH $9 $8
1723 | f_bot_226: BEQ $11 $0 | 1783 | ADDI $1 -1
          f_top_225 | 1784 | EXCH $8 $1
1724 | XOR $7 $12 | 1785 | ADDI $1 -1
1725 | f_bot_226_i: BEQ $11 $0 | 1786 | EXCH $6 $1
          f_top_225_i | 1787 | ADDI $1 -1
1726 | XORI $12 1 | 1788 | EXCH $10 $1
1727 | f_top_225_i: BEQ $11 $0 | 1789 | ADDI $1 -1
          f_bot_226_i | 1790 | ADDI $13 -1789
1728 | cmp_bot_224_i: BEQ $8 $0 | 1791 | l_jmp_219: SWAPBR $13
          cmp_top_223_i | 1792 | NEG $13
1729 | XORI $11 1 | 1793 | ADDI $13 1789
1730 | cmp_top_223_i: BEQ $8 $0 | 1794 | ADDI $1 1
          cmp_bot_224_i | 1795 | EXCH $10 $1
1731 | ADD $9 $3 | 1796 | ADDI $1 1
1732 | ADDI $9 4 | 1797 | EXCH $6 $1
1733 | EXCH $10 $9 | 1798 | ADDI $1 1
1734 | ADDI $9 -4 | 1800 | EXCH $8 $1
1735 | SUB $9 $3 | 1801 | EXCH $3 $1
1736 | EXCH $8 $6 | 1802 | EXCH $9 $8
1737 | test_210: BNE $7 $0 | 1803 | l_jmp_210: ADDI $11 0
1738 | localBlock_222: XOR $8 $1 | 1804 | ADDI $11 0
1739 | XOR $9 $0 | 1805 | EXCH $12 $11
1740 | EXCH $9 $1 | 1806 | XOR $13 $12
1741 | ADDI $1 -1 | 1807 | EXCH $12 $11

Appendix C  Example Output 158 of 231
1808 | loadMetAdd_218_i:          | ADDI $11 0 1874 | ADD $8 $9
1809 | EXCH $11 $10 1875 | XORI $9 1
1810 | XOR $10 $9 1876 | EXCH $8 $6
1811 | EXCH $9 $8 1877 | assert_211: BRA entry_209
1812 | ADD $9 $3 1878 | EXCH $6 $3
1813 | ADDI $9 3 1879 | XORI $7 1
1814 | EXCH $10 $9 1880 | ADD $7 $3
1815 | ADDI $9 $3 1881 | ADDI $7 $4
1816 | SUB $9 $3 1882 | EXCH $6 $7
1817 | XOR $10 $10 1883 | ADDI $7 $4
1818 | EXCH $12 $11 1884 | SUB $7 $3
1819 | ADDI $12 0 1885 | ADDI $1 1
1820 | EXCH $13 $12 1886 | EXCH $9 $1
1821 | XOR $14 $13 1887 | XOR $9 $8
1822 | EXCH $13 $12 1888 | localBlock_227_i:
1823 | ADDI $12 0 1889 | ADD $7 $3
1824 | EXCH $12 $11 1890 | ADDI $7 $4
1825 | ADDI $9 $3 1891 | EXCH $8 $7
1826 | ADDI $9 3 1892 | ADDI $7 $4
1827 | EXCH $10 $9 1893 | SUB $7 $3
1828 | ADDI $9 $3 1894 | ADD $6 $3
1829 | SUB $9 $3 1895 | ADDI $6 $3
1830 | EXCH $3 $1 1896 | EXCH $7 $6
1831 | ADDI $1 $1 1897 | ADD $6 $3
1832 | EXCH $8 $1 1898 | SUB $6 $3
1833 | ADDI $1 $1 1899 | XOR $8 $7
1834 | EXCH $6 $1 1900 | ADD $9 $8
1835 | ADDI $1 $1 1901 | ADDI $9 $1
1836 | EXCH $11 $1 1902 | EXCH $10 $9
1837 | ADDI $1 $1 1903 | XOR $11 $10
1838 | ADDI $14 $1837 1904 | EXCH $10 $9
1839 | l_jump_221: | SWAPBR $14 1905 | ADDI $9 $1
1840 | NEG $14 1906 | EXCH $9 $8
1841 | ADDI $14 1837 1907 | ADD $6 $3
1842 | ADDI $1 $1 1908 | ADD $6 $3
1843 | EXCH $11 $1 1909 | EXCH $7 $6
1844 | ADDI $1 $1 1910 | ADD $6 $3
1845 | EXCH $6 $1 1911 | SUB $6 $3
1846 | ADDI $1 $1 1912 | ADD $6 $3
1847 | EXCH $8 $1 1913 | ADDI $12 $3
1848 | ADDI $1 $1 1914 | EXCH $3 $1
1849 | EXCH $3 $1 1915 | ADD $1 $1
1850 | ADDI $9 $3 1916 | EXCH $12 $1
1851 | ADDI $9 $3 1917 | ADD $1 $1
1852 | EXCH $10 $9 1918 | EXCH $8 $1
1853 | ADDI $9 $3 1919 | ADD $1 $1
1854 | SUB $9 $3 1920 | ADDI $11 $19
1855 | EXCH $12 $11 1921 l_jump_229: | SWAPBR $11
1856 | ADDI $12 0 1922 | NEG $11
1857 | ADDI $12 0 1922 | ADDI $11 $19
1858 | EXCH $13 $12 1923 | ADDI $1 $1
1859 | EXCH $13 $12 1924 | ADD $12 $2
1860 | ADDI $12 0 1926 | ADDI $1 $1
1861 | loadMetAdd_220_i:          | EXCH $12 $11 1927 | EXCH $12 $1
1862 | XOR $11 $10 1928 | ADDI $1 $1
1863 | ADD $9 $3 1929 | EXCH $3 $1
1864 | ADDI $9 $3 1930 | ADDI $12 $2
1865 | EXCH $10 $9 1931 | SUB $12 $3
1866 | ADDI $9 $3 1932 | ADD $6 $3
1867 | SUB $9 $3 1933 | ADD $6 $3
1868 | ADDI $1 $1 1934 | EXCH $7 $6
1869 | EXCH $9 $1 1935 | ADD $6 $3
1870 | XOR $9 $10 1936 | SUB $6 $3
1871 | localBlock_222_i:          | XOR $8 $1 1937 | EXCH $9 $8
1872 | EXCH $8 $6 1938 | ADDI $9 $1
1873 | XORI $9 1 1939 | EXCH $10 $9

Appendix C Example Output 159 of 231
1940 | XOR $11 $10 1999 | start: | BRA top
1941 | EXCH $10 $9 2000 | START |
1942 | ADDI $9 -1 2001 | ADDI | $4 2055
1943 | loadMetAdd_228_i: | EXCH $9 $8 2002 | XOR | $5 $4
1944 | EXCH $9 $7 2003 | ADDI | $5 10
1945 | ADD $6 $3 2004 | XOR | $7 $5
1946 | ADD $6 $3 2005 | ADDI | $4 10
1947 | EXCH $7 $6 2006 | ADDI | $4 -1
1948 | ADDI $6 -3 2007 | EXCH | $7 $4
1949 | SUB $6 $3 2008 | ADDI | $4 1
1950 | ADD $6 $3 2009 | ADDI | $4 -10
1951 | ADDI $6 3 2010 | XOR | $1 5
1952 | EXCH $7 $6 2011 | ADDI | $1 2048
1953 | ADDI $6 -3 2012 | ADDI | $1 -8
1954 | SUB $6 $3 2013 | XOR | $3 $1
1955 | XOR $8 $7 2014 | XORI | $6 4
1956 | loadMetAdd_230: | EXCH $9 $8 2015 | EXCH | $6 $3
1957 | ADDI $9 2 2016 | ADDI | $1 -1
1958 | EXCH $10 $9 2017 | EXCH | $3 $1
1959 | XOR $11 $10 2018 | ADDI | $1 -1
1960 | EXCH $10 $9 2019 | BRA l_main_0
1961 | ADDI $9 -2 2020 | ADDI | $1 1
1962 | EXCH $9 $8 2021 | EXCH | $3 $1
1963 | ADD $6 $3 2022 | ADDI | $3 1
1964 | ADDI $6 3 2023 | ADDI | $3 1
1965 | EXCH $7 $6 2024 | EXCH | $6 $3
1966 | ADDI $6 -3 2025 | XORI | $7 1
1967 | SUB $6 $3 2026 | EXCH | $6 $7
1968 | EXCH $3 $1 2027 | XORI | $7 1
1969 | ADDI $1 -1 2028 | ADDI | $3 -1
1970 | EXCH $8 $1 2029 | ADDI | $3 -1
1971 | ADDI $1 -1 2030 | ADDI | $3 1
1972 | ADDI $11 -1971 2031 | ADDI | $3 2
1973 | l_jmp_231: | SWAPBR $11 2032 | EXCH | $6 $3
1974 | NEG $11 2033 | XORI | $7 2
1975 | ADDI $11 1971 2034 | EXCH | $6 $7
1976 | ADDI $1 2035 | XORI | $7 2
1977 | EXCH $8 $1 2036 | ADDI | $3 -2
1978 | ADDI $1 2037 | ADDI | $3 -1
1979 | EXCH $3 $1 2038 | ADDI | $3 1
1980 | ADD $6 $3 2039 | ADDI | $3 3
1981 | ADDI $6 3 2040 | EXCH | $6 $3
1982 | EXCH $7 $6 2041 | XORI | $7 3
1983 | ADDI $6 -3 2042 | EXCH | $6 $7
1984 | SUB $6 $3 2043 | XORI | $7 3
1985 | EXCH $9 $8 2044 | ADDI | $3 -3
1986 | ADDI $9 2 2045 | ADDI | $3 -1
1987 | EXCH $10 $9 2046 | ADDI | $1 1
1988 | XOR $11 $10 2047 | EXCH | $6 $3
1989 | EXCH $10 $9 2048 | XORI | $6 4
1990 | ADDI $9 -2 2049 | XOR | $3 $1
1991 | loadMetAdd_230_i: | EXCH $9 $8 2050 | ADDI | $1 8
1992 | XOR $8 $7 2051 | ADDI | $1 -2048
1993 | ADD $6 $3 2052 | XOR | $1 5
1994 | ADDI $6 3 2053 | ADDI | $5 -10
1995 | EXCH $7 $6 2054 | XOR | $5 $4
1996 | ADDI $6 -3 2055 | ADDI | $4 -2055
1997 | SUB $6 $3 2056 | finish: FINISH
1998 | l_main_0_bot: | BRA l_main_0_top
DoublyLinkedList.rplpp

class Cell
  int data
  int index
  Cell left
  Cell right
  Cell self

method setData(int value)
  data ^= value

method setIndex(int i)
  index ^= i

method setLeft(Cell cell)
  left <=> cell

method setRight(Cell cell)
  right <=> cell

method setSelf(Cell cell)
  self <=> cell

method append(Cell cell)
  if right = nil & cell != nil then // If current cell does not have a right neighbour
    right <=> cell // Set new cell as right neighbour of current cell
  else skip
    fi right != nil & cell = nil

  local Cell selfCopy = nil
  copy Cell self selfCopy // Copy reference to current cell
  call right::setLeft(selfCopy) // Set current cell as left neighbour of newly
    added right neighbour
  delocal Cell selfCopy = nil

  local int cellIndex = index + 1
  call right::setIndex(cellIndex) // Set cell index in newly added right neighbour
    of current cell
  delocal int cellIndex = index + 1

  else skip
    fi right != nil then
      call right::append(cell) // Keep searching for empty right neighbour
    else skip
      fi right != nil

class DoublyLinkedList
  Cell head
  int length

method appendCell(Cell cell)
  if head = nil & cell != nil then
    head <=> cell
  else skip
    fi head != nil & cell = nil

  if head != nil then
    call head::append(cell)
  else skip
    fi head != nil

  length += 1

class Program
  DoublyLinkedList list
```plaintext
method main()
new DoublyLinkedList list
listLength += 10
local int x = 0
from x = 0 do skip
loop 
local Cell cell = nil
new Cell cell
local Cell cellCopy = nil
call cell::setSelf(cellCopy)
delocal Cell cellCopy = nil
call cell::setData(x)
call list::appendCell(cell)
delocal Cell cell = nil
x += 1
until x = listLength
```
null
cmp_bot_13_i:  
  cmp_top_12_i  
XORI $20 1  
  BEQ $18 0  
  l_i_assert:  
  l_i_assert_true  
ADDI $17 9  
BEQ $11 0  
  cmp_top_14:  
  cmp_bot_15  
ADDI $11 1  
ADD $6 $18  
SUB $18 $6  
  cmp_bot_15:  
BEQ $12 $6  
ADDI $12 $17  
BNE $12 $0  
ADDI $11 1  
XORI $11 1  
  cmp_bot_17:  
BRA l_i_assert  
l_i_assert_true:  
BRA l_i_test_false  
l_i_test_false:  
ADDI $8 1  
ADDI $10 $1  
ADDI $11 1  
ADDI $1 -1  
ADDI $12 $1  
ADDI $1 -1  
ADDI $12 $1  
ADDI $1 -1  
ADDI $16 $1  
ADDI $1 -1  
ADDI $17 $1  
ADDI $1 -1  
EXCH $10 $1  
EXCH $11 l_i_setData_2:  
EXCH $12 $1  
EXCH $14 $1  
ADDI $1 -1  
ADDI $18 $1  
ADDI $1 -1  
ADDI $17 $1  
ADDI $1 -1  
ADDI $21 $1  
ADDI $1 -1  
ADDI $20 $1  
ADDI $1 -1  
ADDI $1 -1  
ADDI $21 $1  
ADDI $1 -1  
ADDI $22 $1  
ADDI $22 $1  
ADDI $1 -1  
ADDI $1 -1  
ADDI $1 -1  
ADDI $1 -1  
ADDI $1 -1  
ADDI $1 -1  
ADDI $20 $1  
ADDI $1 -1  
l_o_assert:  
BNE $10 $0  
BRA l_malloc1  
cmp_top_10:  
BGEZ $15 cmp_bot_11  
XORI $16 $1  
cmp_bot_11:  
BGEZ $15 cmp_top_10  
ADD $15 $7  
XOR $15 $9  
l_malloc1_bot:  
BRA l_malloc1_top  
l_setData_2_top:  
BRA l_setData_2_bot  
l_setData_2:  
SWAPBR $2  
ADDI $14 $1  
NEG $2  
ADDI $1 219  
ADDI $18 $1  
ADDI $1 220  
ADDI $1 221  
ADD $17 $1  
ADDI $1 222  
ADDI $1 223  
ADDI $16 $1  
ADDI $1 224  
ADDI $1 225  
ADDI $1 226  
ADDI $1 227  
ADDI $12 $1  
ADDI $1 228  
ADD $1 229  
ADDI $1 230  
ADDI $1 231  
ADDI $10 $1  
ADDI $1 232  
ADDI $9 1  
ADDI $1 233  
ADDI $8 -1  
ADDI $12 $6  
XOR $12 $6  
XORI $130 1  
XOR $12 $6  
ADDI $16 $1  
ADDI $12 $1  
ADDI $11 1  
ADDI $10 $1  
l_test_false:  
BRA l_i_test_false
Appendix C  Example Output 166 of 231
loadMetAdd_33_i:

localBlock_35_i:

loadMetAdd_36_i:

localBlock_38:

loadMetAdd_36:

Appendix C  Example Output  167 of 231
Appendix C  Example Output 170 of 231
1178 | ADDI $1 -1 1244 | EXCH $3 $1
1179 | EXCH $8 $1 1245 | ADD $9 $3
1180 | ADDI $1 -1 1246 | ADDI $9 2
1181 | EXCH $6 $1 1247 | EXCH $10 $9
1182 | ADDI $1 -1 1248 | ADDI $9 -2
1183 | EXCH $10 $1 1249 | SUB $9 $3
1184 | ADDI $1 -1 1250 | EXCH $12 $11
1185 | ADDI $13 -1184 1251 | ADDI $12 0
1186 | l_jump_105: | SWAPBR $13 1252 | EXCH $13 $12
1187 | NEG $13 1253 | XOR $14 $13
1188 | ADDI $13 1184 1254 | EXCH $13 $12
1189 | ADDI $1 1 1255 | ADDI $12 0
1190 | EXCH $10 $1 1256 | loadMetAdd_106_i: | EXCH $12 $11
1191 | ADDI $1 1 1257 | XOR $11 $10
1192 | EXCH $6 $1 1258 | ADD $9 $3
1193 | ADDI $1 1 1259 | ADDI $9 2
1194 | EXCH $8 $1 1260 | EXCH $10 $9
1195 | ADDI $1 1 1261 | ADDI $9 -2
1196 | EXCH $3 $1 1262 | SUB $9 $3
1197 | EXCH $9 $8 1263 | ADDI $1 1
1198 | EXCH $11 $10 1264 | EXCH $9 $1
1199 | ADDI $11 0 1265 | XOR $9 0
1200 | EXCH $12 $11 1266 | localBlock_108_i: | XOR $8 $1
1201 | XOR $13 $12 1267 | EXCH $8 $6
1202 | EXCH $12 $11 1268 | XORI $9 1
1203 | ADDI $11 0 1269 | ADD $8 $9
1204 | EXCH $11 $10 1270 | loadMetAdd_104_i: | XOR $9 1
1205 | XOR $10 $9 1271 | EXCH $8 $6
1206 | EXCH $9 $8 1272 | assert_93: | BRA entry_91
1207 | ADD $9 $3 1273 | exit_94: | BRA test_92
1208 | ADDI $9 2 1274 | XORI $7 1
1209 | EXCH $10 $9 1275 | ADD $7 $3
1210 | ADDI $9 $2 1276 | ADDI $7 3
1211 | SUB $9 $3 1277 | EXCH $8 $7
1212 | XOR $11 $10 1278 | ADDI $7 -3
1213 | EXCH $12 $11 1279 | SUB $7 $3
1214 | ADDI $12 0 1280 | ADDI $1 1
1215 | EXCH $13 $12 1281 | EXCH $9 $1
1216 | XOR $14 $13 1282 | XOR $9 $8
1217 | EXCH $13 $12 1283 | localBlock_113_i: | XOR $6 $1
1218 | ADDI $12 0 1284 | ADD $7 $3
1219 | EXCH $12 $11 1285 | ADDI $7 3
1220 | ADDI $9 $3 1286 | EXCH $8 $7
1221 | ADDI $9 2 1287 | ADDI $7 -3
1222 | EXCH $10 $9 1288 | SUB $7 $3
1223 | ADDI $9 -2 1289 l_main_0_bot: | BRA l_main_0_top
1224 | SUB $9 $3 1290 start: | BRA top
1225 | EXCH $3 $1 1291 | START
1226 | ADDI $1 -1 1292 | ADDI $4 1338
1227 | EXCH $6 $1 1293 | XOR $5 $4
1228 | ADDI $1 -1 1294 | ADDI $5 10
1229 | EXCH $8 $1 1295 | XOR $7 $5
1230 | ADDI $1 -1 1296 | ADDI $4 10
1231 | EXCH $11 $1 1297 | ADDI $4 -1
1232 | ADDI $1 -1 1298 | EXCH $7 $4
1233 | l_jump_107: | ADDI $4 -1
1234 | SWAPBR $14 1300 | ADDI $4 -10
1235 | NEG $14 1301 | XOR $1 $5
1236 | ADDI $14 1232 1302 | ADDI $1 2048
1237 | ADDI $1 1 1303 | ADDI $1 -4
1238 | EXCH $11 $1 1304 | XOR $3 $1
1239 | ADDI $1 1 1305 | XOR $6 $3
1240 | EXCH $8 $1 1306 | EXCH $6 $3
1241 | ADDI $1 1 1307 | ADDI $1 -1
1242 | EXCH $6 $1 1308 | EXCH $3 $1
1243 | ADDI $1 1 1309 | ADDI $1 -1

Appendix C  Example Output 173 of 231
BRA  l_main_0
ADDI $1 1
EXCH $3 $1
ADDI $3 1
ADDI $3 1
EXCH $6 $3
ADDI $3 -1
ADDI $3 -1
ADDI $3 1
EXCH $6 $3
ADDI $3 2
EXCH $6 $3
XORI $7 1
ADDI $7 1
ADDI $7 1
ADDI $3 -1
ADDI $3 -1
ADDI $3 1
ADDI $3 1
ADDI $3 1
ADDI $3 -1
ADDI $3 -1
ADDI $3 1
ADDI $3 2
EXCH $6 $3
XORI $7 2
XORI $7 2
ADDI $3 -1
ADDI $3 1
ADDI $1 1
ADDI $1 1
EXCH $6 $3
EXCH $6 $3
XORI $6 3
EXCH $6 $7
EXCH $6 $7
XORI $7 1
ADDI $7 1
ADDI $7 1
ADDI $3 -1
ADDI $3 -1
ADDI $3 1
ADDI $3 1
ADDI $3 1
ADDI $3 -1
ADDI $3 -1
ADDI $3 1
ADDI $3 2
EXCH $6 $3
XORI $6 3
XOR $3 $1
ADDI $1 4
ADDI $1 4
ADDI $1 -2048
XOR $1 $5
ADDI $1 -10
ADDI $5 -10
ADDI $5 $4
ADDI $4 -1338
XOR $7 2
XOR $7 2
finish:
FINISH
class Cell
    Cell self
    Cell right
    Cell left
    int data

    method getLeft(Cell cell)
        right <=> cell
    end

    method getRight(Cell cell)
        left <=> cell
    end

    method getSelf(Cell cell)
        self <=> cell
    end

    method getSymbol(int symbol)
        symbol <=> data
    end

class RTM
    Cell tapeHead
    int[] q1
    int[] q2
    int[] s1
    int[] s2
    int SLASH
    int LEFT
    int RIGHT
    int BLANK
    int state
    int Qs
    int Qf
    int symbol
    int PC_MAX
    int pc

    method initLiterals()
        // Initialize string literals
        SLASH += 9999
        LEFT += 9998
        RIGHT += 9997
        BLANK += 9996
        // Set max program counter
        PC_MAX += 7
    end

    method initRules()
        // Initialize transition rule arrays
        new int[8] q1
        new int[8] q2
        new int[8] s1
        new int[8] s2
        // Define transition rules for binary number incrementation
        q1[0] += 1
        s1[0] += BLANK
        s2[0] += BLANK
        q2[0] += 2
        q1[1] += 2
        s1[1] += SLASH
        s2[1] += RIGHT
        q2[1] += 3
    end
method initTape()
  local Cell cell0 = nil
  local Cell cell1 = nil
  local Cell cell2 = nil
  local Cell cell3 = nil
  local Cell cell4 = nil

  // Init cells
  new Cell cell0
  new Cell cell1
  new Cell cell2
  new Cell cell3
  new Cell cell4

  // Write 1 1 0 1 on tape
  symbol += BLANK
  uncall cell0::getSymbol(symbol)
  symbol += 1
  uncall cell1::getSymbol(symbol)
  symbol += 1
  uncall cell2::getSymbol(symbol)
  symbol += 1
  uncall cell4::getSymbol(symbol)

  // Set tape head
  tapeHead <=> cell0

  // Set self pointers
  copy Cell tapeHead cell0
  uncall tapeHead::getSelf(cell0)
  copy Cell cell1 cell0
  uncall cell1::getSelf(cell0)
  copy Cell cell2 cell10
  uncall cell12::getSelf(cell10)
  copy Cell cell13 cell10
  uncall cell13::getSelf(cell10)
copy Cell cell14 cell10
uncall cell14::getSelf(cell10)

// Link cell 3 and 4
copy Cell cell13 cell10
uncall cell14::getLeft(cell10)
uncall cell13::getRight(cell14)

// Link cell 2 and 3
copy Cell cell12 cell10
uncall cell13::getLeft(cell10)
uncall cell12::getRight(cell13)

// Link cell11 and cell 2
copy Cell cell11 cell10
uncall cell12::getLeft(cell10)
uncall cell11::getRight(cell12)

// Link tapeHead and cell 1
copy Cell tapeHead cell10
uncall cell11::getLeft(cell10)
uncall tapeHead::getRight(cell11)

delocal Cell cell14 = nil
delocal Cell cell13 = nil
delocal Cell cell12 = nil
delocal Cell cell11 = nil
delocal Cell cell10 = nil

method init()
    // Prepare for simulation
    call initLiterals()
call initRules()
call initTape()

    // Init pc, start and finishing state
    state += 1
    Qs += 1
    Qf += 6

    // Start simulation
    call simulate()

method simulate()
    from state = Qs do
        call tapeHead::getSymbol(symbol)  // Fetch current symbol
        call inst()
        uncall tapeHead::getSymbol(symbol)  // Zero-clear symbol
        pc += 1  // Increment pc

        if pc = PC_MAX then
            pc ^= PC_MAX  // Reset pc
        else skip
        fi
        state = q2[pc]-q1[pc]
        symbol += s2[pc]-s1[pc]

        if state = q1[pc] & symbol = SLASH then  // Move rule:
            state += q2[pc]-q1[pc]
            symbol += s2[pc]-s1[pc]
        else skip

        if state = q2[pc] & symbol = s2[pc]
        if state = q1[pc] & s1[pc] = SLASH then  // Move rule:
            state += q2[pc]-q1[pc]
            symbol += s2[pc]-s1[pc]
        else skip

        if s2[pc] = RIGHT then
call moveRight() // Move tape head right
else skip
fi
if s2[pc] = RIGHT then
  uncall moveRight() // Move tape head left
else skip
doi
fi
s2[pc] = LEFT

if s2[pc] = LEFT then
  uncall moveRight() // Move tape head left
else skip
doi
fi
s2[pc] = LEFT
else skip
doi
fi
state = q2[pc] & s1[pc] = SLASH

method moveRight()
local Cell right = nil
local Cell tmp = nil
 uncalled 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 then
    call tmp::getSelf(tapeHead) // Set tape head reference back
    uncopy Cell tmp tapeHead // Clear new tape head
    delete Cell tmp // Delete new tape head
    symbol ^= BLANK // Clear new tape head
    else skip // In reverse:
    fi
  tmp = nil // Allocate new left if current is nil
else
  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

class Program
RTM bni

method main()
// This program contains a RTM implementing
// incrementation of a non-negative n-bit binary number by 1 (modulo 2^n).
// The tape is initialized with | b | 1Â | 1 | 0 | 0 | and after execution, 
// the tape is left with | bÂ | 0 | 0Â | 1 | 1 |
new RTM bni
call bni::init()
RTM.pal

1;; pendulum pal file
2top:  
3l_r_bni:  
4l_Program_vt:  
5l_RTM_vt:  
6l_Cell_vt:  
7l_malloc_top:  
8l_malloc:  
9l_malloc_bot:  
10l_malloc1_top:  
11l_malloc1:  
12l_malloc1_i:  
13cmp_top_12:  
14cmp_bot_13:  
15cmp_bot_13_i:  
16cmp_top_12_i:  
17cmp_top_16:  
18cmp_bot_17:  
19l_o_test:  
20l_o_test_false:  
21l_o_assert_true:  
22l_o_assert_false:  
23XOR $13 $9
24BRA start
25BRA l_o_test:
26BEQ $10 $0
27XOR $10 1
28ADDI $8 -1
29EXCH $19 $17
30XOR $18 $19
31ADDI $1 $1
32XORI $0 1
33EXCH $19 $17
34EXCH $2 $1
35EXCH $11 $1
36XOR $0 1
37ADDI $1 -1
38XOR $17 $4
39EXCH $19 $17
40EXCH $2 $1
41EXCH $11 $1
42SWAPBR $2
43NEG $2
44EXCH $2 $1
45ADDI $1 -1
46XOR $17 $4
47ADD $17 $8
48EXCH $19 $17
49XOR $18 $19
50EXCH $19 $17
51XOR $13 $9
52SUB $13 $7
53XORI $14 1
54BGEZ $13 cmp_bot_12
55BGEZ $13 cmp_bot_12_i
56cmp_bot_13_i:
57cmp_bot_13:
58cmp_bot_12:
59cmp_bot_12_i:
60ADD $13 $7 120
61XORI $14 1
62cmp_bot_12_i:
63cmp_bot_13:
64cmp_bot_13_i:
65cmp_bot_16:
66cmp_bot_17:
67l_o_assert_true:
68BRA l_o_test_false:
69BRA l_o_assert
70BEQ $18 $0
71XOR $18 1
72ADDI $1 $1
73XORI $14 1
74cmp_bot_12_i:
75cmp_bot_13:
76cmp_bot_13_i:
77cmp_bot_16:
78cmp_bot_17:
79XORI $20 1
80Appendix C Example Output 179 of 231
ADD $7 $3
ADDI $7 3
EXCH $8 $7
ADDI $7 -3
SUB $7 $3
EXCH $9 $6
XOR $8 $9
XOR $8 $9
EXCH $9 $6
ADD $7 $3
ADDI $7 3
EXCH $8 $7
ADDI $7 -3
SUB $7 $3
EXCH $9 $6
EXCH $9 $6
XOR $8 $9
XOR $9 $8
EXCH $9 $8
EXCH $9 $6
ADD $7 $3
ADDI $7 3
EXCH $8 $7
ADDI $7 -3
SUB $7 $3
EXCH $9 $6
ADDI $7 3
ADDI $7 3
EXCH $8 $7
ADDI $7 -3
SUB $7 $3
l_getLeft_8_bot:
BRA l_getLeft_8_top
l_getRight_9_top:
BRA l_getRight_9_bot
ADDI $1 1
EXCH $2 $1
EXCH $6 $1
ADDI $1 -1
EXCH $3 $1
ADDI $1 -1
l_getRight_9:
SWAPBR $2
NEG $2
ADDI $1 1
EXCH $3 $1
ADDI $1 1
EXCH $6 $1
EXCH $2 $1
ADDI $1 -1
ADD $7 $3
ADDI $7 4
EXCH $8 $7
ADDI $8 5
EXCH $9 $8
ADDI $8 -5
l_getRight_9_bot:
BRA l_getRight_9_top
l_getSelf_10_top:
BRA l_getSelf_10_bot
ADDI $1 1
EXCH $2 $1
EXCH $6 $1
ADDI $1 -1
EXCH $3 $1
ADDI $1 -1
l_getSelf_10:
SWAPBR $2
NEG $2
ADDI $1 1
EXCH $3 $1
ADDI $1 1
EXCH $6 $1
EXCH $2 $1
ADDI $1 -1
ADD $7 $3
ADDI $7 4
EXCH $8 $7
ADDI $8 5
EXCH $9 $8
ADDI $8 -5
l_getSymbol_11_bot:
BRA l_getSymbol_11_top
ADDI $1 1
EXCH $2 $1
EXCH $6 $1
ADDI $1 -1
EXCH $3 $1
ADDI $1 -1
l_getSymbol_11:
SWAPBR $2
NEG $2
ADDI $1 1
EXCH $3 $1
ADDI $1 1
EXCH $6 $1
EXCH $2 $1
ADDI $1 -1
EXCH $7 $6
ADDI $7 4
EXCH $8 $7
ADDI $8 5
EXCH $9 $8
ADDI $8 -5
l_initLiterals_1_top:
BRA l_initLiterals_1_bot
ADDI $1 1
EXCH $2 $1
EXCH $3 $1
ADDI $1 -1
EXCH $2 $1
ADDI $1 -1
l_initLiterals_1:
SWAPBR $2
NEG $2
ADDI $1 1
EXCH $3 $1
ADDI $1 1
EXCH $6 $1
EXCH $2 $1
ADDI $1 -1
EXCH $7 $6
ADDI $7 4
EXCH $8 $7
ADDI $8 5
EXCH $9 $8
ADDI $8 -5
l_getSelf_10:
SWAPBR $2
NEG $2
ADDI $1 1
EXCH $2 $1
EXCH $6 $1
ADDI $1 -1
EXCH $3 $1
ADDI $1 -1
l_initLiterals_1_bot:
BRA l_initLiterals_1_top
ADDI $1 1
EXCH $2 $1
EXCH $6 $1
ADDI $1 -1
EXCH $3 $1
ADDI $1 -1
l_initLiterals_1:
SWAPBR $2
NEG $2
ADDI $1 1
EXCH $2 $1
EXCH $6 $1
ADDI $1 -1
EXCH $3 $1
ADDI $1 -1
l_initLiterals_1_bot:
SUB $6 3 424
XORI $8 9999 425
ADD $7 8 426
XORI $8 9999 427
ADD $6 3 428
ADDI $6 7 429
EXCH $7 8 430
ADD $6 -7 431
SUB $6 3 432
ADD $6 3 433
ADDI $6 7 434
EXCH $7 $6 435
ADDI $6 -7 436
ADDI $6 -8 437
SUB $6 3 438
ADD $6 3 439
ADDI $6 7 440
EXCH $7 $6 441
ADD $6 3 442
EXCH $7 $6 443
ADDI $6 8 444
SUB $6 3 445
ADDI $6 8 446
ADDI $6 9 447
EXCH $7 8 448
ADD $6 -9 449
SUB $6 3 450
ADDI $6 8 451
XORI $8 9997 452
ADDI $6 1 453
EXCH $7 8 454
ADDI $6 2 455
EXCH $8 $7 456
ADDI $6 -9 457
SUB $6 3 458
ADDI $6 -3 459
SUB $6 3 460
ADDI $6 -1 461
EXCH $7 8 462
ADD $6 -10 463
SUB $6 3 464
ADDI $6 -5 465
EXCH $7 8 466
ADD $6 -10 467
ADDI $6 10 468
EXCH $7 8 469
ADD $6 -10 470
SUB $6 3 471
ADDI $6 3 472
ADDI $6 5 473
EXCH $7 8 474
ADDI $6 -15 475
SUB $6 3 476
ADDI $6 3 477
XORI $8 7 478
ADDI $6 -1 479
ADD $6 3 480
ADDI $6 5 481
EXCH $7 8 482
ADD $6 -15 483
SUB $6 3 484
ADDI $6 1 485
XORI $8 7 486
ADDI $6 1 487
ADDI $6 3 488
EXCH $7 8 489
ADDI $6 1 489
490 | XORI $7$ 8 556 | ADD $6$ 3
491 | XOR $9$ 7 557 | ADDI $6$ 6
492 | EXCH $9$ 8 558 | XORI $7$ 8
493 | ADDI $8$ 1 559 | XOR $9$ 7
494 | XOR $9$ 1 560 | EXCH $9$ 8
495 | EXCH $9$ 8 561 | ADDI $8$ 1
496 | ADDI $8$ -1 562 | XORI $9$ 1
497 | arr_con_27_bot: | EXCH $8$ 563 | EXCH $9$ 8
498 | XORI $7$ 8 564 | ADDI $8$ -1
499 | ADDI $6$ -4 565 | arr_con_29_bot: | EXCH $8$ 56
500 | SUB $6$ 53 566 | XORI $7$ 8
501 | XORI $7$ 8 567 | ADDI $6$ -6
502 | arr_con_28: | ADD $9$ 2 568 | SUB $6$ 3
503 | ADD $9$ 7 569 | ADD $6$ 3
504 | XORI $7$ 8 570 | ADDI $6$ 3
505 | EXCH $8$ 571 | EXCH $8$ 56
506 | ADDI $8$ -1 572 | XOR $7$ 8
507 | EXCH $9$ 573 | EXCH $8$ 56
508 | ADDI $8$ -1 574 | ADDI $7$ 2
509 | EXCH $8$ 575 | ADD $7$ 0
510 | ADDI $8$ -1 576 | ADD $6$ -3
511 | BRA 1ทะ malloc | SUB $6$ 3
512 | ADDI $1$ 1 578 | EXCH $9$ 7
513 | EXCH $8$ 579 | ADD $6$ 3
514 | ADDI $1$ 1 580 | ADDI $6$ 3
515 | EXCH $9$ 581 | SUB $7$ 0
516 | ADDI $1$ 1 582 | ADDI $7$ -2
517 | EXCH $8$ 583 | EXCH $8$ 56
518 | XORI $7$ 8 584 | XOR $7$ 8
519 | SUB $9$ 585 | EXCH $8$ 56
520 | arr_con_28_i: | ADD $9$ -2 586 | ADD $6$ -3
521 | XORI $7$ 8 587 | SUB $6$ 3
522 | ADDI $8$ -1 588 | XORI $10$ 1
523 | ADDI $6$ 5 589 | assArrElem_30: | ADD $9$ 10
524 | XORI $7$ 8 590 | XORI $10$ 1
525 | XOR $9$ 7 591 | ADD $6$ 3
526 | EXCH $9$ 592 | ADD $6$ 3
527 | ADDI $8$ 1 593 | EXCH $8$ 56
528 | XORI $9$ 1 594 | XOR $7$ 8
529 | EXCH $9$ 595 | EXCH $8$ 56
530 | ADDI $8$ -1 596 | ADDI $7$ 2
531 | arr_con_28_bot: | EXCH $8$ 597 | ADD $7$ 0
532 | XORI $7$ 8 598 | ADDI $6$ -3
533 | ADDI $6$ -5 599 | SUB $6$ 3
534 | SUB $6$ 600 | EXCH $9$ 7
535 | XORI $7$ 8 601 | ADD $6$ 3
536 | arr_con_29: | ADDI $9$ 2 602 | ADD $6$ 3
537 | ADD $9$ 7 603 | SUB $7$ 0
538 | XORI $7$ 8 604 | ADDI $7$ -2
539 | EXCH $3$ 5 605 | EXCH $8$ 56
540 | ADDI $1$ -1 606 | XOR $7$ 8
541 | EXCH $9$ 5 607 | EXCH $8$ 56
542 | ADDI $1$ -1 608 | ADDI $6$ -3
543 | EXCH $8$ 5 609 | SUB $6$ 3
544 | ADDI $1$ -1 610 | ADD $6$ 3
545 | BRA 1ทะ malloc | ADDI $6$ 5
546 | ADD $9$ 612 | EXCH $8$ 56
547 | EXCH $8$ 613 | XOR $7$ 8
548 | ADDI $1$ 614 | EXCH $8$ 56
549 | EXCH $9$ 615 | ADDI $7$ 2
550 | ADDI $1$ 616 | ADD $7$ 0
551 | EXCH $3$ 617 | ADDI $6$ -5
552 | XORI $7$ 618 | SUB $6$ 3
553 | SUB $9$ 619 | EXCH $9$ 7
554 | arr_con_29_i: | ADDI $9$ -2 620 | ADD $6$ 3
555 | XORI $7$ 621 | ADDI $6$ 5

Appendix C  Example Output 183 of 231
| Line | Instruction | Address | Value |
|------|-------------|---------|-------|
| 622  | SUB $7 50   | 688     | SUB $10 $3 |
| 623  | ADDI $7 -2  | 689     | ADD $6 $3 |
| 624  | EXCH $8 56  | 690     | ADDI $6 $6 |
| 625  | XOR $7 58   | 691     | EXCH $8 $6 |
| 626  | EXCH $8 56  | 692     | XOR $7 58 |
| 627  | ADDI $6 -5  | 693     | EXCH $8 $6 |
| 628  | SUB $6 53   | 694     | ADDI $7 2 |
| 629  | ADD $10 $3  | 695     | ADDI $7 $0 |
| 630  | ADDI $10 50 | 696     | ADDI $6 $6 |
| 631  | EXCH $11 $10| 697     | SUB $6 $3 |
| 632  | ADDI $10 -10| 698     | EXCH $9 $7 |
| 633  | SUB $10 $3  | 699     | ADD $6 $3 |
| 634  | assArrElem_31: | | |
| 635  | ADD $9 $11  | 700     | ADD $6 $6 |
| 636  | ADD $10 $3  | 701     | SUB $7 50 |
| 637  | ADDI $10 50 | 702     | ADDI $7 $2 |
| 638  | EXCH $11 $10| 703     | EXCH $8 $6 |
| 639  | ADD $6 53   | 704     | XOR $7 58 |
| 640  | ADDI $6 5    | 705     | EXCH $8 $6 |
| 641  | EXCH $8 56  | 706     | ADD $6 $3 |
| 642  | XOR $7 58   | 707     | SUB $6 $3 |
| 643  | EXCH $8 56  | 708     | ADDI $6 $4 |
| 644  | EXCH $7 57  | 709     | ADDI $6 $4 |
| 645  | ADD $6 53   | 710     | SUB $6 $3 |
| 646  | ADDI $6 5    | 711     | EXCH $9 $7 |
| 647  | SUB $7 50   | 712     | ADD $6 $3 |
| 648  | ADD $7 -2   | 713     | ADD $6 $2 |
| 649  | SUB $6 53   | 714     | ADD $6 $3 |
| 650  | EXCH $9 57  | 715     | ADD $6 $4 |
| 651  | ADD $6 53   | 716     | SUB $6 $3 |
| 652  | ADDI $6 5    | 717     | EXCH $9 $7 |
| 653  | SUB $7 50   | 718     | ADD $6 $3 |
| 654  | ADDI $7 -2  | 719     | ADD $6 $2 |
| 655  | EXCH $8 56  | 720     | ADDI $6 $2 |
| 656  | XOR $7 58   | 721     | SUB $6 $3 |
| 657  | EXCH $8 56  | 722     | EXCH $8 $6 |
| 658  | ADDI $6 -5  | 723     | XOR $7 58 |
| 659  | SUB $6 53   | 724     | EXCH $8 $6 |
| 660  | ADDI $6 5    | 725     | ADD $6 $3 |
| 661  | ADDI $6 5    | 726     | ADD $6 $3 |
| 662  | EXCH $8 56  | 727     | XORI $10 2|
| 663  | XOR $7 58   | 728     | assArrElem_33: | ADD $9 $10 |
| 664  | EXCH $8 56  | 729     | XORI $10 2|
| 665  | ADDI $7 2    | 730     | ADD $6 $3 |
| 666  | ADD $7 50   | 731     | ADDI $6 $4 |
| 667  | ADDI $6 -6  | 732     | EXCH $8 $6 |
| 668  | SUB $6 53   | 733     | XOR $7 58 |
| 669  | EXCH $9 57  | 734     | EXCH $8 $6 |
| 670  | ADDI $6 5    | 735     | ADDI $7 2 |
| 671  | ADD $6 6    | 736     | ADD $7 50 |
| 672  | SUB $7 50   | 737     | ADDI $6 $4 |
| 673  | ADD $7 -2   | 738     | ADDI $6 $3 |
| 674  | EXCH $8 56  | 739     | EXCH $9 $7 |
| 675  | XOR $7 58   | 740     | ADD $6 $3 |
| 676  | EXCH $8 56  | 741     | ADDI $6 $4 |
| 677  | ADDI $6 -6  | 742     | SUB $7 50 |
| 678  | SUB $6 53   | 743     | ADDI $7 2 |
| 679  | ADD $10 53  | 744     | EXCH $8 $6 |
| 680  | ADDI $10 10 | 745     | XOR $7 58 |
| 681  | EXCH $11 $10| 746     | EXCH $8 $6 |
| 682  | ADD $10 -10 | 747     | ADDI $6 $4 |
| 683  | assArrElem_32: | | |
| 684  | SUB $10 $3  | 748     | SUB $6 $3 |
| 685  | ADDI $9 $11  | 749     | ADD $6 $3 |
| 686  | ADD $10 53  | 750     | ADDI $6 $3 |
| 687  | EXCH $11 $10| 751     | XORI $7 1 |
| 688  | ADDI $10 -10| 752     | EXCH $9 $6 |
| 689  | ADDI $10 -10| 753     | XOR $8 $9 |

Appendix C  Example Output 184 of 231
|      |     |     |     |     |     |
|------|-----|-----|-----|-----|-----|
| 754  | EXCH | $9$ | $6$ | #20 |     |
| 755  | ADDI | $8$ | 2   | #21 |     |
| 756  | ADDI | $8$ | 57  | #22 |     |
| 757  | XORI | $7$ | 1   | #23 |     |
| 758  | ADDI | $6$ | $-3$| #24 | ADD  |
| 759  | SUB  | $6$ | $53$| #25 | SUB  |
| 760  | EXCH | $10$| $8$ | #26 | assArrElem_34: |
| 761  | ADDI | $6$ | 3   | #27 |     |
| 762  | ADDI | $6$ | 3   | #28 |     |
| 763  | XORI | $7$ | 1   | #29 | EXCH |
| 764  | SUB  | $8$ | $57$| #30 | ADDI |
| 765  | ADDI | $8$ | $-2$| #31 | SUB  |
| 766  | EXCH | $9$ | $56$| #32 | ADD  |
| 767  | XOR  | $8$ | $59$| #33 | ADDI |
| 768  | EXCH | $9$ | $56$| #34 | XORI |
| 769  | XORI | $7$ | 1   | #35 | EXCH |
| 770  | ADDI | $6$ | $-3$| #36 | XOR  |
| 771  | SUB  | $6$ | $53$| #37 | EXCH |
| 772  | XORI | $11$| 2   | #38 | ADDI |
| 773  | assArrElem_35: |   |     |     |     |
| 774  | ADD  | $10$| $11$| #39 | ADDI |
| 775  | XORI | $11$| 2   | #40 | XORI |
| 776  | ADDI | $6$ | $23$| #41 | ADDI |
| 777  | ADDI | $6$ | 3   | #42 | SUB  |
| 778  | XORI | $7$ | 1   | #43 | EXCH |
| 779  | EXCH | $9$ | $56$| #44 | ADD  |
| 780  | XOR  | $8$ | $59$| #45 | ADDI |
| 781  | EXCH | $9$ | $56$| #46 | XORI |
| 782  | ADDI | $8$ | 2   | #47 | SUB  |
| 783  | ADD  | $8$ | $7$ | #48 | ADDI |
| 784  | XORI | $7$ | 1   | #49 | EXCH |
| 785  | ADDI | $6$ | $-3$| #50 | XOR  |
| 786  | SUB  | $6$ | $53$| #51 | EXCH |
| 787  | EXCH | $10$| $8$ | #52 | XORI |
| 788  | ADDI | $6$ | $53$| #53 | ADDI |
| 789  | ADDI | $6$ | 3   | #54 | SUB  |
| 790  | SUB  | $8$ | $57$| #55 | ADDI |
| 791  | ADDI | $8$ | $-2$| #56 | XORI |
| 792  | EXCH | $9$ | $56$| #57 | EXCH |
| 793  | XOR  | $8$ | $59$| #58 | XOR  |
| 794  | EXCH | $9$ | $56$| #59 | EXCH |
| 795  | XORI | $7$ | 1   | #60 | ADDI |
| 796  | ADDI | $6$ | $-3$| #61 | ADDI |
| 797  | SUB  | $6$ | $53$| #62 | ADDI |
| 798  | ADDI | $6$ | $53$| #63 | ADDI |
| 799  | ADDI | $6$ | 5   | #64 | SUB  |
| 800  | XORI | $7$ | 1   | #65 | EXCH |
| 801  | EXCH | $9$ | $56$| #66 | ADDI |
| 802  | XOR  | $8$ | $59$| #67 | ADDI |
| 803  | EXCH | $9$ | $56$| #68 | XORI |
| 804  | ADDI | $8$ | 2   | #69 | SUB  |
| 805  | ADDI | $8$ | $57$| #70 | ADDI |
| 806  | XORI | $7$ | 1   | #71 | EXCH |
| 807  | ADDI | $6$ | $-5$| #72 | XOR  |
| 808  | SUB  | $6$ | $53$| #73 | EXCH |
| 809  | EXCH | $10$| $8$ | #74 | XORI |
| 810  | ADDI | $6$ | $53$| #75 | ADDI |
| 811  | ADDI | $6$ | 5   | #76 | SUB  |
| 812  | XORI | $7$ | 1   | #77 | ADDI |
| 813  | SUB  | $8$ | $57$| #78 | ADDI |
| 814  | ADDI | $8$ | $-2$| #79 | EXCH |
| 815  | EXCH | $9$ | $56$| #80 | ADDI |
| 816  | XOR  | $8$ | $59$| #81 | SUB  |
| 817  | EXCH | $9$ | $56$| #82 | assArrElem_36: |
| 818  | XORI | $7$ | 1   | #83 | ADDI |
| 819  | ADDI | $6$ | $-5$| #84 | ADDI |

Appendix C  Example Output 185 of 231
EXCH $12 $11 952 XORI $7 1
ADDI $11 -9 953 SUB $8 $7
SUB $11 $3 954 ADD $8 $2
ADD $6 $3 955 EXCH $9 $6
ADDI $6 6 956 XOR $8 $9
XORI $7 1 957 EXCH $9 $6
EXCH $9 $6 958 XORI $7 1
ADDI $8 $9 959 ADDI $6 $4
EXCH $9 $6 960 SUB $6 $3
ADDI $8 2 961 ADD $6 $3
ADD $8 $7 962 ADDI $6 $3
XORI $7 1 963 XORI $7 2
ADDI $6 -6 964 EXCH $9 $6
SUB $6 $3 965 XOR $8 $9
EXCH $10 $8 966 EXCH $9 $6
ADD $6 $3 967 ADDI $8 2
ADDI $6 6 968 ADD $8 $7
XORI $7 1 969 XORI $7 2
SUB $8 $7 970 ADDI $6 -3
ADDI $8 -2 971 SUB $6 $3
ADD $6 $3 972 ADDI $8 -2
ADD $6 $3 973 EXCH $9 $6
ADD $6 4 974 XOR $8 $9
XORI $7 1 980 EXCH $9 $6
EXCH $9 $6 981 XORI $7 2
XOR $8 $9 982 ADDI $6 -3
EXCH $9 $6 983 SUB $6 $3
ADDI $8 2 984 XORI $11 3
ADD $8 $7 985 assArrElem_38: ADD $10 $11
XORI $7 1 986 XORI $11 3
ADDI $6 -4 987 ADD $6 $3
SUB $6 $3 988 ADDI $6 $3
EXCH $10 $8 989 XORI $7 2
ADDI $6 6 990 EXCH $9 $6
ADDI $6 4 991 XOR $8 $9
XORI $7 1 992 EXCH $9 $6
SUB $8 $7 993 ADDI $8 2
ADDI $8 -2 994 ADD $8 $7
EXCH $9 $6 995 XORI $7 2
XOR $8 $9 996 ADDI $6 -3
EXCH $9 $6 997 SUB $6 $3
XORI $7 1 998 EXCH $10 $8
ADDI $6 -4 999 ADD $6 $3
SUB $6 $3 1000 ADDI $6 $3
XORI $7 1 1001 XORI $7 2
ADDI $10 $11 1002 SUB $8 $7
XORI $11 3 1003 ADDI $8 -2
ADDI $6 $3 1004 EXCH $9 $6
ADDI $6 4 1005 XOR $8 $9
XORI $7 1 1006 EXCH $9 $6
EXCH $9 $6 1007 XORI $7 2
XOR $8 $9 1008 ADDI $6 -3
EXCH $9 $6 1009 SUB $6 $3
ADDI $8 2 1010 ADD $6 $3
ADDI $8 $7 1011 ADDI $6 5
XORI $7 1 1012 XORI $7 2
ADDI $6 -4 1013 EXCH $9 $6
SUBI $6 $3 1014 XOR $8 $9
EXCH $10 $8 1015 EXCH $9 $6
ADDI $6 $3 1016 ADDI $8 2
ADD $6 $3 1017 ADD $8 $7

Appendix C Example Output 186 of 231
assArrElem_39:
ADD $10 $0 1099
ADD $6 $3 1100
ADDI $6 -5 1101
ADDI $7 2 1102
EXCH $9 $6 1103
XORI $8 $9 1104
EXCH $9 $6 1105
ADDI $8 2 1106
ADD $8 $7 1107
ADDI $7 2 1108
ADD $6 -5 1109
SUB $6 $3 1110
EXCH $10 $8 1111
ADD $6 $3 1112
ADDI $6 5 1113
XORI $7 2 1114
SUBI $8 $7 1115
ADDI $8 -2 1116
EXCH $9 $6 1117
ADDI $9 $6 1118
EXCHI $8 $9 1119
ADDI $7 2 1120
ADDI $6 -5 1121
ADDI $6 $3 1122
ADDI $6 $3 1123
ADDI $6 6 1124
ADDI $7 2 1125
EXCHI $9 $6 1126
ADDI $8 $9 1127
EXCH $9 $6 1128
ADDI $8 2 1129
ADDI $8 $7 1130
assArrElem_41:
ADDI $10 $11 1131
XORI $7 2 1132
ADD $6 -6 1133
SUBI $6 $3 1134
EXCHI $10 $8 1135
ADDI $6 $3 1136
ADDI $6 6 1137
ADDI $7 2 1138
ADDI $8 $7 1139
ADDI $8 -2 1140
ADDI $9 $6 1141
ADDI $9 $6 1142
ADDI $7 2 1143
ADDI $6 -6 1144
ADDI $6 $3 1145
ADDI $11 1 1146
ADDI $11 1 1147
ADDI $6 $3 1148
ADDI $6 $3 1149
EXCH $9 $6 1150
EXCH $9 $6 1348 | SUB $6 $3
ADDI $8 2 1349 | ADD $6 $3
ADDI $8 $7 1350 | ADDI $6 $3
XORI $7 3 1351 | XORI $7 4
ADDI $6 $6 1352 | EXCH $9 $6
SUB $6 $3 1353 | XOR $8 $9
EXCH $10 $8 1354 | EXCH $9 $6
ADD $6 $3 1355 | ADDI $8 $2
ADDI $6 6 1356 | ADD $6 $7
XORI $7 3 1357 | XORI $7 $4
ADDI $8 $7 1358 | ADDI $6 $3
ADDI $8 $2 1359 | SUB $6 $3
EXCH $9 $6 1360 | EXCH $10 $8
XOR $8 $9 1361 | ADD $6 $3
EXCH $9 $6 1362 | ADDI $6 $3
XORI $7 3 1363 | XORI $7 4
SUB $6 $6 1364 | ADDI $6 $3
EXCH $9 $6 1365 | ADDI $8 $2
ADD $6 $3 1366 | EXCH $9 $6
ADDI $6 $4 1367 | XOR $8 $9
EXCH $9 $6 1368 | EXCH $9 $6
EXCH $9 $8 1369 | XORI $7 4
SUB $8 $8 1370 | SUB $6 $3
EXCH $9 $6 1371 | ADD $10 $11
ADDI $8 2 1372 | XORI $11 $3
ADDI $8 $7 1373 | assArrElem_46:
ADDI $7 3 1374 | XORI $11 $3
ADDI $6 $4 1375 | ADD $6 $3
ADDI $6 $3 1376 | ADDI $6 $3
EXCH $9 $8 1377 | XORI $7 4
ADD $6 $4 1378 | EXCH $9 $6
EXCH $9 $8 1379 | XOR $8 $9
ADDI $9 4 1380 | EXCH $9 $6
XORI $7 3 1381 | ADDI $6 $3
EXCH $9 $8 1382 | ADD $8 $7
ADDI $9 $6 1383 | XORI $7 4
ADD $6 $3 1384 | ADD $6 $3
EXCH $9 $6 1385 | SUB $6 $3
EXCH $9 $6 1386 | ADDI $6 $3
ADD $6 $4 1387 | ADDI $6 $3
EXCH $9 $8 1388 | ADD $6 $3
ADD $6 $7 1389 | SUB $8 $7
ADD $10 $11 1390 | XOR $7 4
ADDI $11 2 1391 | ADDI $8 $2
ADDI $6 $3 1392 | EXCH $9 $6
ADDI $6 $4 1393 | XOR $8 $9
ADDI $8 $6 1394 | EXCH $9 $6
EXCH $9 $6 1395 | XORI $7 4
XOR $8 $9 1396 | ADDI $6 $3
EXCH $9 $6 1397 | SUB $6 $3
EXCH $9 $6 1398 | ADD $6 $3
EXCH $9 $6 1399 | ADDI $6 $5
ADDI $6 $7 1399 | XOR $7 4
ADD $8 $8 1400 | SUB $8 $7
ADDI $7 3 1401 | EXCH $9 $6
EXCH $9 $8 1402 | XOR $8 $9
EXCH $10 $8 1403 | EXCH $9 $6
SUB $6 $3 1404 | ADDI $6 $2
ADDI $6 $4 1405 | ADDI $6 $2
EXCH $7 3 1406 | XOR $7 4
ADDI $6 $7 1407 | ADDI $6 $5
ADDI $8 $2 1408 | SUB $6 $3
EXCH $9 $6 1409 | EXCH $10 $8
ADDI $8 $9 1410 | ADD $6 $3
EXCH $9 $6 1411 | ADD $6 $5
ADD $6 $3 1412 | XOR $7 4
ADDI $6 $4 1413 | SUB $8 $7
1546 XORI $7 4 1612 XORI $7 5
1547 ADDI $6 -4 1613 EXCH $9 $6
1548 SUB $6 $3 1614 XOR $8 $9
1549 EXCH $10 $8 1615 EXCH $9 $6
1550 ADD $6 $3 1616 ADDI $8 2
1551 ADDI $6 4 1617 ADD $8 $7
1552 XORI $7 4 1618 XORI $7 5
1553 SUB $8 $7 1619 ADDI $6 -5
1554 ADD $8 -2 1620 SUB $6 $3
1555 EXCH $9 $6 1621 EXCH $10 $8
1556 XOR $8 $9 1622 ADD $6 $3
1557 EXCH $9 $6 1623 ADDI $6 5
1558 XORI $7 4 1624 XORI $7 5
1559 ADDI $6 -4 1625 SUB $8 $7
1560 SUB $6 $3 1626 ADDI $8 -2
1561 ADD $6 $3 1627 EXCH $9 $6
1562 ADDI $6 3 1628 XOR $8 $9
1563 XORI $7 5 1629 EXCH $9 $6
1564 EXCH $9 $6 1630 XORI $7 5
1565 XOR $8 $9 1631 ADDI $6 -5
1566 EXCH $9 $6 1632 SUB $6 $3
1567 ADDI $8 2 1633 ADDI $11 $3
1568 ADD $8 $7 1634 ADDI $11 7
1569 XORI $7 5 1635 EXCH $12 $11
1570 ADDI $6 -3 1636 ADDI $11 -7
1571 SUB $6 $3 1637 SUB $11 $3
1572 assArrElem_51:
1573 EXCH $10 $8 1638 assArrElem_51:
1574 ADD $6 $3 1639 ADD $10 $12
1575 ADDI $6 3 1640 ADDI $11 7
1576 XORI $7 5 1641 EXCH $12 $11
1577 SUB $8 $7 1642 ADDI $11 -7
1578 ADDI $8 -2 1643 SUB $11 $3
1579 EXCH $9 $6 1644 ADD $6 $3
1580 XOR $8 $9 1645 ADDI $6 5
1581 EXCH $9 $6 1646 XORI $7 5
1582 XORI $7 5 1647 EXCH $9 $6
1583 ADDI $6 -3 1648 XOR $8 $9
1584 SUB $6 $3 1649 EXCH $9 $6
1585 XORI $11 4 1650 ADDI $8 2
1586 assArrElem_50:
1587 ADD $10 $11 1651 ADD $8 $7
1588 XORI $11 4 1652 XORI $7 5
1589 ADDI $6 3 1653 ADDI $6 -5
1590 ADDI $6 3 1654 SUB $6 $3
1591 XORI $7 5 1655 EXCH $10 $8
1592 EXCH $9 $6 1656 ADDI $6 $3
1593 XOR $8 $9 1657 ADDI $6 5
1594 EXCH $9 $6 1658 XORI $7 5
1595 ADDI $8 2 1659 SUB $8 $7
1596 ADDI $8 $7 1660 ADDI $8 -2
1597 XORI $7 5 1661 EXCH $9 $6
1598 ADD $6 -3 1662 XOR $8 $9
1599 SUB $6 $3 1663 EXCH $9 $6
1600 assArrElem_50:
1601 EXCH $10 $8 1664 XORI $7 5
1602 ADD $6 $3 1665 ADDI $6 -5
1603 ADDI $6 3 1666 SUB $6 $3
1604 XORI $7 5 1667 ADDI $6 $3
1605 SUB $8 $7 1668 ADDI $6 $3
1606 ADDI $8 -2 1669 XORI $7 5
1607 EXCH $9 $6 1670 EXCH $9 $6
1608 XOR $8 $9 1671 XOR $8 $9
1609 EXCH $9 $6 1672 EXCH $9 $6
1610 XOR $7 5 1673 ADDI $8 2
1611 ASSArrElem_51:
1612 ADD $6 $3 1674 ADD $8 $7
1613 SUB $6 $3 1675 XORI $7 5
1614 ADD $6 $3 1676 ADD $6 $6
1615 ADDI $6 5 1677 SUB $6 $3
1616

Appendix C  Example Output 191 of 231
1942 | ADD $6 63 2008 |
1943 | ADDI $6 4 2009 |
1944 | XORI $7 6 2010 |
1945 | EXCH $9 56 2011 |
1946 | XOR $8 59 2012 |
1947 | EXCH $9 56 2013 |
1948 | ADDI $8 2 2014 |
1949 | ADD $8 57 2015 |
1950 | XORI $7 6 2016 |
1951 | ADDI $6 -4 2017 |
1952 | SUB $6 53 2018 |
1953 | EXCH $10 82 2019 |
1954 | ADDI $6 4 2020 |
1955 | ADDI $6 21 2021 |
1956 | XORI $7 6 2022 |
1957 | SUB $8 57 2023 |
1958 | ADDI $8 -2 2024 |
1959 | EXCH $9 56 2025 |
1960 | XOR $8 59 2026 |
1961 | EXCH $9 56 2027 |
1962 | XORI $7 6 2028 |
1963 | ADDI $6 -4 2029 |
1964 | ADD $6 53 2030 |
1965 | ADDI $6 3 2031 |
1966 | ADDI $6 3 2032 |
1967 | XORI $7 6 2033 |
1968 | EXCH $9 56 2034 |
1969 | XOR $8 59 2035 |
1970 | EXCH $9 56 2036 |
1971 | ADDI $6 4 2037 |
1972 | ADD $8 57 2038 |
1973 | XORI $7 6 2040 |
1974 | ADDI $6 -3 2041 |
1975 | SUB $6 53 |
1976 | EXCH $10 82 2042 |
1977 | ADDI $6 3 2043 |
1978 | ADDI $6 3 2044 |
1979 | XORI $7 6 2045 |
1980 | SUB $8 57 2046 |
1981 | ADDI $8 -2 2047 |
1982 | EXCH $9 56 2048 |
1983 | XOR $8 59 2049 |
1984 | EXCH $9 56 2050 |
1985 | XORI $7 6 2051 |
1986 | ADDI $6 -3 2052 |
1987 | SUB $6 53 |
1988 | XORI $7 6 2054 |
1989 | ADDI $11 5 2056 |
1990 | ADDI $11 5 2057 |
1991 | ADDI $6 3 |
1992 | XORI $7 6 |
1993 | XOR $8 59 |
1994 | EXCH $9 56 |
1995 | XOR $8 59 |
1996 | EXCH $9 56 |
1997 | ADDI $7 2 |
1998 | SUB $8 57 |
1999 | XORI $7 6 |
2000 | AD $6 -3 |
2001 | SUB $6 53 |
2002 | EXCH $10 82 |
2003 | ADDI $6 3 |
2004 | SUB $6 3 |
2005 | XORI $7 6 |
2006 | ADDI $8 57 |
2007 | ADDI $8 -2 |

Appendix C Example Output 194 of 231
1974  EXCH $9 $6  2140  ADD $6 $3
1975  XOR $8 $9  2141  ADDI $6 $4
1976  EXCH $9 $6  2142  XORI $7 $7
1977  ADDI $8 2  2143  SUB $8 $7
1978  ADD $8 $7  2144  ADDI $8 $2
1979  XORI $7 7  2145  EXCH $9 $6
1980  ADDI $6 $6  2146  XOR $8 $9
1981  SUB $6 $3  2147  EXCH $9 $6
1982  EXCH $8 $8  2148  XORI $7 $7
1983  ADD $6 $3  2149  ADDI $6 $4
1984  ADDI $6 $6  2150  SUB $6 $3
1985  XORI $7 $7  2151  XORI $11 $6
1986  SUB $8 $7  2152  assArrElem_61:  ADD $10 $11
1987  ADDI $8 $2  2153  XORI $11 $6
1988  EXCH $9 $6  2154  ADD $6 $3
1989  XOR $8 $9  2155  ADDI $6 $4
1990  EXCH $9 $6  2156  XORI $7 $7
1991  XORI $7 $7  2157  EXCH $9 $6
1992  ADDI $6 $6  2158  XOR $8 $9
1993  SUB $6 $3  2159  EXCH $9 $6
1994  ADD $11 $3  2160  ADDI $8 $2
1995  ADDI $11 $10  2161  ADD $8 $7
1996  EXCH $12 $11  2162  XORI $7 $7
1997  ADDI $11 $10  2163  ADDI $6 $4
1998  SUB $11 $3  2164  SUB $6 $3
1999  assArrElem_60:  ADD $10 $12  2165  EXCH $10 $8
2000  ADDI $11 $3  2166  ADD $6 $3
2001  ADD $11 $10  2167  ADDI $6 $4
2002  EXCH $12 $11  2168  XORI $7 $7
2003  ADDI $11 $10  2169  SUB $8 $7
2004  SUB $11 $3  2170  ADDI $8 $2
2005  ADD $6 $3  2171  EXCH $9 $6
2006  ADD $6 $6  2172  XOR $8 $9
2007  XORI $7 $7  2173  EXCH $9 $6
2008  EXCH $9 $6  2174  XORI $7 $7
2009  XORI $7 $7  2175  ADDI $6 $4
2010  EXCH $9 $6  2176  SUB $6 $3
2011  ADDI $8 $2  2177  l_initRules_2_bot:  BRA
2012  ADDI $8 $7  2178  l_initRules_2_top:  BRA
2013  XORI $7 $7  2179  l_initTape_3_top:  BRA
2014  ADDI $6 $6  2180  l_initTape_3_bot:  BRA
2015  SUB $6 $3  2181  _l_initRules_2_bot:  BRA
2016  EXCH $10 $8  2182  _l_initRules_2_top:
2017  ADDI $6 $3  2183  l_initTape_3:
2018  ADDI $6 $3  2184  swapBR $2
2019  XORI $7 $7  2185  NEG $2
2020  SUB $8 $7  2186  ADDI $1 $1
2021  ADDI $8 $2  2187  EXCH $9 $6
2022  EXCH $9 $6  2188  EXCH $3 $1
2023  XORI $7 $7  2189  EXCH $9 $6
2024  XORI $7 $7  2190  EXCH $3 $1
2025  XORI $7 $7  2191  EXCH $9 $6
2026  XORI $7 $7  2192  XOR $8 $0
2027  XORI $7 $7  2193  localBlock_149:  XOR $6 $1
2028  XORI $7 $7  2194  localBlock_148:  XOR $7 $1
2029  XORI $7 $7  2195  localBlock_147:  XOR $8 $1
2030  XORI $7 $7  2196  localBlock_148:  XOR $7 $1
2031  XORI $7 $7  2197  localBlock_149:  XOR $8 $1
2032  XORI $7 $7  2198  localBlock_147:  XOR $9 $0
2033  XORI $7 $7  2199  localBlock_148:  XOR $9 $1
2034  XORI $7 $7  2200  localBlock_149:  XOR $1 $1
2035  XORI $7 $7  2201  localBlock_148:  XOR $0 $1
2036  XORI $7 $7  2202  localBlock_147:  XOR $0 $1
2037  XORI $7 $7  2203  localBlock_149:  XOR $1 $1
localBlock_145:
XOR $10 $1 2271
XOR $11 $0 2272
EXCH $11 $1 2273 obj_con_63_i:
ADDI $12 -8
ADDI $1 -1 2274
EXCH $3 $1 2275
EXCH $1 -1 2276
ADDI $10 $1 2277
EXCH $1 -1 2278
EXCH $9 $1 2279
ADDI $1 -1 2280
EXCH $8 $1 2281
ADDI $1 -1 2282
EXCH $7 $1 2283
ADDI $1 -1 2284
EXCH $6 $1 2285
ADDI $1 -1 2286
XORI $12 10
ADDI $12 8 2287
EXCH $12 $1 2288
ADDI $11 1
ADDI $1 -1 2289
EXCH $11 $1 2290
EXCH $12 $1 2291
EXCH $12 -8 2292
ADDI $12 -8 2297
ADDI $12 $1 2293
EXCH $11 $1 2294
ADDI $12 $1 2295
EXCH $12 $1 2296
ADDI $12 $1 2297
ADDI $12 $1 2298
EXCH $6 $1 2299
ADDI $1 1 2300
EXCH $7 $1 2301
ADDI $1 1 2302
EXCH $8 $1 2303
ADDI $1 1 2304
EXCH $9 $1 2305 obj_con_64:
ADDI $12 8
ADDI $1 1 2306
EXCH $10 $1 2307
ADDI $1 -1 2308
EXCH $3 $1 2309
ADDI $1 1 2310
XORI $12 10 2311
EXCH $12 $11 2312
ADDI $11 1 2313
XORI $12 11 2314
EXCH $12 $11 2315
obj_con_64_i:
ADDI $11 -1 2316
ADDI $12 $6 2317
ADDI $12 $6 2318
ADDI $1 -1 2319
ADDI $12 $10 2320
ADDI $12 $10 2321
ADDI $1 -1 2322
ADDI $9 $1 2323
ADDI $8 $1 2324
ADDI $8 $1 2325
ADDI $7 $1 2326
ADDI $12 $1 2327
ADDI $12 $1 2328
ADDI $1 -1 2329
ADDI $1 -1 2330
EXCH $10 $1 2331
EXCH $11 $1 2332
ADDI $11 -1 2333 obj_con_64_bot:
ADDI $11 -1
ADDI $1 -1 2334
ADDI $1 1 2335
EXCH $3 $1
obj_con_66_i:
2336 ADDI $1 -1 2402 ADDI $1 1
2337 EXCH $10 $1 2403 EXCH $7 $1
2338 ADDI $1 -1 2404 ADDI $1 1
2339 EXCH $9 $1 2405 EXCH $8 $1
2340 ADDI $1 -1 2406 ADDI $1 1
2341 EXCH $8 $1 2407 EXCH $9 $1
2342 ADDI $1 -1 2408 ADDI $1 1
2343 EXCH $7 $1 2409 EXCH $10 $1
2344 ADDI $1 -1 2410 ADDI $1 1
2345 EXCH $6 $1 2411 EXCH $3 $1
2346 ADDI $1 -1 2412 XORI $12 10
2347 obj_con_65:
2348 ADDI $12 8 2413 EXCH $12 $11
2349 EXCH $12 $1 2414 ADDI $11 1
2345 ADDI $1 -1 2415 XORI $12 1
2350 EXCH $11 $1 2416 EXCH $12 $11
2351 ADDI $1 -1 2417 obj_con_66_bot:
2352 BRA l_malloc 2418 EXCH $11 $10
2353 ADDI $1 1 2419 ADDI $11 $3
2354 EXCH $11 $1 2420 ADDI $11 14
2355 ADDI $1 1 2421 XORI $12 1
2356 EXCH $12 $1 2422 ADDI $11 -14
2357 obj_con_65_i:
2358 ADDI $12 -8 2423 SUB $11 3
2359 EXCH $6 $1 2424 ADDI $13 3
2360 ADDI $1 1 2425 ADDI $13 10
2361 EXCH $7 $1 2426 EXCH $14 $13
2362 ADDI $1 1 2427 ADDI $13 -10
2363 EXCH $8 $1 2428 SUB $13 3
2364 ADDI $1 1 2429 ADDI $12 $14
2365 EXCH $9 $1 2430 ADDI $13 3
2366 ADDI $1 1 2431 ADDI $13 10
2367 EXCH $10 $1 2432 EXCH $14 $13
2368 ADDI $1 1 2433 ADDI $13 -10
2369 EXCH $3 $1 2434 SUB $13 3
2370 XORI $12 10 2435 ADDI $11 3
2371 EXCH $12 $11 2436 EXCH $12 $11
2372 ADDI $11 1 2437 ADDI $11 -14
2373 XORI $12 1 2438 ADDI $11 3
2374 ADDI $12 $1 2439 SUB $11 3
2375 obj_con_66_bot:
2376 ADDI $11 -1 2440 XORI $12 11
2377 EXCH $11 $9 2441 loadMetAdd_67:
2378 EXCH $3 $1 2442 EXCH $13 $12
2379 ADDI $1 -1 2443 ADDI $13 3
2380 ADDI $10 $1 2444 EXCH $14 $13
2381 ADDI $1 -1 2445 XORI $15 $14
2382 EXCH $9 $1 2446 ADDI $14 $13
2383 ADDI $1 -1 2447 ADDI $13 -3
2384 EXCH $8 $1 2448 EXCH $13 $12
2385 ADDI $1 -1 2449 EXCH $11 $6
2386 EXCH $0 $1 2450 ADDI $16 $3
2387 EXCH $7 $1 2451 ADDI $16 $14
2388 ADDI $1 -1 2452 EXCH $3 $1
2389 EXCH $6 $1 2453 ADDI $1 -1
2390 ADDI $1 -1 2454 EXCH $10 $1
2391 obj_con_66:
2392 ADDI $12 8 2455 ADDI $1 -1
2393 EXCH $12 $1 2456 EXCH $9 $1
2394 ADDI $1 -1 2457 ADDI $1 -1
2395 EXCH $11 $1 2458 EXCH $8 $1
2396 ADDI $1 -1 2459 ADDI $1 -1
2397 BRA l_malloc 2460 EXCH $7 $1
2398 ADDI $1 1 2461 ADDI $1 -1
2399 ADDI $11 $1 2462 EXCH $6 $1
2400 ADDI $1 1 2463 ADDI $1 -1
2401 obj_con_66_i:
2402 ADDI $12 -8 2465 ADDI $1 -1
2403 ADDI $1 1 2466 EXCH $12 $1
2404 EXCH $6 $1 2467 ADDI $1 -1

Appendix C Example Outup 197 of 231
2468 l_rjmp_top_69: ADDI $15 -2468 2534
2469 l_rjmp_68: ADDI $15 -2468 2534
2470 l_rjmp_bot_70: ADDI $15 -2468 2534
2471 NEG $15 2537
2472 BRA l_rjmp_top_68 $15
2473 ADDI $15 2468 2539
2474 ADDI $1 1 2540
2475 EXCH $12 $1 2541
2476 ADDI $1 1 2542
2477 EXCH $16 $1 2543
2478 ADDI $1 1 2544 l_rjmp_top_73: BRA l_rjmp_bot_74
2479 EXCH $6 $1 2545 l_rjmp_72: SWAPBR $15
2480 ADDI $1 1 2546
2481 EXCH $7 $1 2547 l_rjmp_bot_74: BRA l_rjmp_top_73
2482 ADDI $1 1 2548
2483 EXCH $8 $1 2549
2484 ADDI $1 1 2550
2485 EXCH $9 $1 2551
2486 ADDI $1 1 2552
2487 EXCH $10 $1 2553
2488 ADDI $1 1 2554
2489 EXCH $3 $1 2555
2490 ADDI $16 -14 2556
2491 SUB $16 $3 2557
2492 EXCH $11 $6 2558
2493 EXCH $13 $12 2559
2494 ADDI $13 3 2560
2495 EXCH $14 $13 2561
2496 XOR $15 $14 2562
2497 EXCH $14 $13 2563
2498 ADDI $13 -3 2564
2499 loadMetAdd_67_i: EXCH $13 $12 2565
2500 XOR $12 $11 2566
2501 EXCH $11 $6 2567
2502 ADD $11 $3 2568
2503 ADDI $11 14 2569
2504 EXCH $12 $11 2570
2505 ADDI $11 14 2571
2506 ADDI $11 14 2572
2507 XORI $13 1 2573
2508 ADD $12 $13 2574 loadMetAdd_71_i: EXCH $13 $12
2509 XORI $13 1 2575
2510 ADD $11 $3 2576
2511 ADDI $11 14 2577
2512 EXCH $12 $11 2578
2513 ADDI $11 -14 2579
2514 SUB $11 $3 2580
2515 EXCH $11 $7 2581
2516 XOR $12 $11 2582
2517 loadMetAdd_71: EXCH $13 $12 2583
2518 ADDI $13 3 2584
2519 EXCH $14 $13 2585
2520 XOR $15 $14 2586
2521 EXCH $14 $13 2587
2522 ADDI $13 -3 2588
2523 EXCH $13 $12 2589
2524 EXCH $11 $7 2590
2525 ADDI $16 $3 2591
2526 ADD $16 $14 2592 loadMetAdd_75: EXCH $13 $12
2527 EXCH $3 $1 2593
2528 ADDI $1 -1 2594
2529 EXCH $10 $1 2595
2530 ADDI $1 -1 2596
2531 EXCH $9 $1 2597
2532 ADDI $1 -1 2598
2533 EXCH $8 $1 2599

Appendix C Example Output
2600 | ADD $16 $3 2666 | XOR $12 $11
2601 | ADDI $16 14 2667 | loadMetAdd_79: EXCH $13 $12
2602 | EXCH $3 $1 2668 | ADDI $13 $3
2603 | ADD $1 -1 2669 | EXCH $14 $13
2604 | EXCH $10 $1 2670 | XOR $15 $14
2605 | ADDI $1 -1 2671 | EXCH $14 $13
2606 | EXCH $9 $1 2672 | ADDI $13 -3
2607 | ADDI $1 -1 2673 | EXCH $13 $12
2608 | EXCH $8 $1 2674 | EXCH $11 $10
2609 | ADD $1 -1 2675 | ADD $16 $3
2610 | EXCH $7 $1 2676 | ADDI $16 $14
2611 | ADDI $1 -1 2677 | EXCH $3 $1
2612 | EXCH $6 $1 2678 | ADDI $1 -1
2613 | ADD $1 -1 2679 | EXCH $10 $1
2614 | EXCH $16 $1 2680 | ADDI $1 -1
2615 | ADD $1 -1 2681 | EXCH $9 $1
2616 | EXCH $12 $1 2682 | ADDI $1 -1
2617 | ADDI $1 -1 2683 | EXCH $8 $1
2618 | ADDI $15 -2618 2684 | ADDI $1 -1
2619 | l_rjmp_top_77: RBRA l_rjmp_bot_78
2620 | l_jmp_76: SWAPBR $15 2686 | ADDI $1 -1
2621 | NEG $15 2687 | EXCH $6 $1
2622 | l_rjmp_bot_78: BRA l_rjmp_top_77
2623 | ADDI $15 2618 2689 | EXCH $16 $1
2624 | ADDI $1 1 2690 | ADDI $1 -1
2625 | EXCH $12 $1 2691 | EXCH $12 $1
2626 | ADDI $1 1 2692 | ADDI $1 -1
2627 | EXCH $16 $1 2693 | ADDI $15 -2693
2628 | ADDI $1 1 2694 | l_rjmp_top_81: RBRA l_rjmp_bot_82
2629 | EXCH $6 $1 2695 | l_jmp_80: SWAPBR $15
2630 | ADDI $1 1 2696 | NEG $15
2631 | EXCH $7 $1 2697 | l_rjmp_bot_82: BRA l_rjmp_top_81
2632 | ADDI $1 1 2698 | ADDI $15 2693
2633 | EXCH $8 $1 2699 | ADDI $1
2634 | ADDI $1 1 2700 | EXCH $12 $1
2635 | EXCH $9 $1 2701 | ADDI $1 1
2636 | ADDI $1 1 2702 | EXCH $16 $1
2637 | EXCH $10 $1 2703 | ADDI $1 1
2638 | ADDI $1 1 2704 | EXCH $6 $1
2639 | EXCH $3 $1 2705 | ADDI $1 1
2640 | ADDI $16 -14 2706 | EXCH $7 $1
2641 | SUB $16 $3 2707 | ADDI $1 1
2642 | EXCH $11 $8 2708 | EXCH $8 $1
2643 | EXCH $13 $12 2709 | ADDI $1 1
2644 | ADDI $13 3 2710 | EXCH $9 $1
2645 | EXCH $14 $13 2711 | ADDI $1 1
2646 | XOR $15 $14 2712 | EXCH $10 $1
2647 | EXCH $14 $13 2713 | ADDI $1 1
2648 | ADDI $13 -3 2714 | EXCH $3 $1
2649 | loadMetAdd_75_i: EXCH $13 $12 2715 | ADDI $16 -14
2650 | XOR $12 $11 2716 | SUB $16 $3
2651 | EXCH $11 $8 2717 | EXCH $11 $10
2652 | ADD $11 $3 2718 | EXCH $13 $12
2653 | ADDI $11 14 2719 | ADDI $13 3
2654 | EXCH $12 $11 2720 | EXCH $14 $13
2655 | ADDI $11 -14 2721 | XOR $15 $14
2656 | SUB $11 $3 2722 | EXCH $14 $13
2657 | XORI $13 1 2723 | ADDI $13 -3
2658 | ADD $12 $13 2724 | loadMetAdd_79_i: EXCH $13 $12
2659 | XORI $13 1 2725 | XOR $12 $11
2660 | ADDI $11 $3 2726 | EXCH $11 $10
2661 | ADDI $11 14 2727 | ADD $11 $3
2662 | EXCH $12 $11 2728 | ADDI $11 2
2663 | ADDI $11 -14 2729 | EXCH $12 $11
2664 | SUB $11 $3 2730 | ADDI $11 -2
2665 | EXCH $11 $10 2731 | SUB $11 $3

Appendix C  Example Output 199 of 231
2732 | EXCH $13 $6 2798 | ADDI $1 1
2733 | swap_83: XOR $12 $13 2799 | EXCH $13 $1
2734 | XOR $13 $12 2800 | ADDI $1 1
2735 | XOR $12 $13 2801 | EXCH $6 $1
2736 | EXCH $13 $6 2802 | ADDI $1 1
2737 | ADD $11 $3 2803 | EXCH $7 $1
2738 | ADD $11 2 2804 | ADDI $1 1
2739 | EXCH $12 $11 2805 | EXCH $8 $1
2740 | ADDI $11 -2 2806 | ADDI $1 1
2741 | SUB $11 $3 2807 | EXCH $9 $1
2742 | EXCH $11 $6 2808 | ADDI $1 1
2743 | ADD $12 $3 2809 | EXCH $10 $1
2744 | ADDI $12 2 2810 | ADDI $1 1
2745 | EXCH $13 $12 2811 | EXCH $3 $1
2746 | ADDI $12 -2 2812 | ADD $11 $3
2747 | SUB $12 $3 2813 | ADDI $11 2
2748 | copy_84: XOR $11 $13 2814 | EXCH $12 $11
2749 | ADDI $13 4 2815 | ADDI $1 -2
2750 | EXCH $14 $13 2816 | SUB $11 $3
2751 | ADDI $14 1 2817 | EXCH $14 $13
2752 | EXCH $14 $13 2818 | ADDI $14 2
2753 | ADD $12 $1 2819 | EXCH $15 $14
2754 | ADD $12 $1 2820 | XOR $16 $15
2755 | ADDI $12 2 2821 | EXCH $15 $14
2756 | EXCH $13 $12 2822 | ADDI $14 -2
2757 | ADDI $12 -2 2823 | loadMetAdd_85_i: EXCH $14 $13
2758 | SUB $12 $3 2824 | EXCH $13 $12
2759 | EXCH $11 $6 2825 | ADDI $13 $3
2760 | ADD $11 $3 2826 | ADDI $11 2
2761 | ADDI $11 2 2827 | EXCH $12 $11
2762 | EXCH $12 $11 2828 | ADDI $11 -2
2763 | ADDI $11 -2 2829 | SUB $11 $3
2764 | SUB $11 $3 2830 | EXCH $11 $6
2765 | XOR $13 $12 2831 | EXCH $12 $7
2766 | loadMetAdd_85: EXCH $14 $13 2832 | copy_89: XOR $11 $12
2767 | ADDI $14 2 2833 | ADDI $12 1
2768 | EXCH $15 $14 2834 | EXCH $13 $12
2769 | XOR $16 $15 2835 | ADDI $13 1
2770 | EXCH $15 $14 2836 | ADDI $13 $12
2771 | ADD $14 -2 2837 | ADDI $12 -1
2772 | EXCH $14 $13 2838 | EXCH $12 $7
2773 | ADD $11 $3 2839 | EXCH $11 $6
2774 | ADDI $11 2 2840 | EXCH $11 $7
2775 | EXCH $12 $11 2841 | EXCH $12 $11
2776 | ADD $11 -2 2842 | loadMetAdd_90: EXCH $13 $12
2777 | SUB $11 $3 2843 | ADDI $13 $2
2778 | EXCH $3 $1 2844 | EXCH $14 $13
2779 | ADDI $1 -1 2845 | XOR $15 $14
2780 | EXCH $10 $1 2846 | EXCH $14 $13
2781 | ADDI $1 -1 2847 | ADDI $13 -2
2782 | EXCH $9 $1 2848 | EXCH $13 $12
2783 | ADDI $1 -1 2849 | EXCH $11 $7
2784 | EXCH $8 $1 2850 | EXCH $3 $1
2785 | ADD $1 -1 2851 | ADDI $1 -1
2786 | EXCH $7 $1 2852 | EXCH $10 $1
2787 | ADDI $1 -1 2853 | ADDI $1 -1
2788 | EXCH $6 $1 2854 | EXCH $9 $1
2789 | ADDI $1 -1 2855 | ADDI $1 -1
2790 | EXCH $13 $1 2856 | EXCH $8 $1
2791 | ADDI $1 -1 2857 | ADDI $1 -1
2792 | ADDI $16 -2792 2858 | EXCH $7 $1
2793 | l_rjmp_top_87: RBRA l_rjmp_bot_88 #59 | ADDI $1 -1
2794 | l_rjmp_86: SWAPBR $16 2860 | EXCH $6 $1
2795 | NEG $16 2861 | ADDI $1 -1
2796 | l_rjmp_bot_88: BRA l_rjmp_top_88 #62 | EXCH $12 $1
2797 | ADDI $16 2792 2863 | ADDI $1 -1

Appendix C  Example Output  200 of 231
2864 | l_rjmp_top_92: | ADDI $15 -2864 | l_jmp_96: | SWAPBR $15
2865 | l_rjmp_91: | SWAPBR $15 2932 | l_rjmp_bot_98: | BRA l_rjmp_top_97
2866 | NEG $15 2933 | ADDI $15 2928
2867 | l_rjmp_bot_93: | BRA l_rjmp_top_92 2934 | ADDI $1 1
2868 | l_jmp_91: | ADDI $15 2864 | EXCH $12 $1
2869 | l_rjmp_top_97: | ADDI $1 1 2936 | ADDI $1 1
2870 | EXCH $12 $1 2937 | EXCH $6 $1
2871 | ADDI $1 1 2938 | ADDI $1 1
2872 | EXCH $6 $1 2939 | EXCH $7 $1
2873 | ADDI $1 1 2940 | ADDI $1 1
2874 | EXCH $7 $1 2941 | EXCH $8 $1
2875 | ADDI $1 1 2942 | ADDI $1 1
2876 | EXCH $8 $1 2943 | EXCH $9 $1
2877 | ADDI $1 1 2944 | ADDI $1 1
2878 | EXCH $9 $1 2945 | EXCH $10 $1
2879 | ADDI $1 1 2946 | ADDI $1 1
2880 | EXCH $10 $1 2947 | EXCH $3 $1
2881 | ADDI $1 1 2948 | EXCH $11 $8
2882 | EXCH $3 $1 2949 | EXCH $13 $12
2883 | EXCH $11 $7 2950 | ADDI $13 2
2884 | EXCH $13 $12 2951 | EXCH $14 $13
2885 | ADDI $13 2 2952 | XOR $15 $14
2886 | EXCH $14 $13 2953 | EXCH $14 $13
2887 | XOR $15 $14 2954 | ADDI $13 2
2888 | EXCH $14 $13 2955 | ADDI $13 -2
2889 | ADDI $13 -2 2956 | XOR $12 $11
2890 | l_loadMetAdd_90_i: | EXCH $13 $12 2957 | XOR $11 $8
2891 | XOR $12 $11 2958 | EXCH $11 $6
2892 | EXCH $11 $7 2959 | EXCH $12 $9
2893 | EXCH $11 $6 2960 | copy_99:
2894 | EXCH $12 $8 2961 | ADDI $12 1
2895 | copy_94: | XOR $12 $12 2962 | EXCH $13 $12
2896 | ADDI $12 1 2963 | ADDI $13 1
2897 | EXCH $13 $12 2964 | EXCH $13 $12
2898 | ADDI $13 1 2965 | ADDI $12 1
2899 | EXCH $13 $12 2966 | EXCH $12 $9
2900 | ADDI $12 1 2967 | EXCH $11 $6
2901 | EXCH $13 $12 2968 | EXCH $11 $9
2902 | EXCH $11 $6 2969 | XOR $12 $11
2903 | EXCH $11 $8 2970 | loadMetAdd_100:
2904 | XOR $12 $11 2971 | EXCH $13 $12
2905 | LOAD MetAdd_95: | ADDI $13 2 | EXCH $14 $13
2906 | EXCH $13 $12 2972 | XOR $15 $14
2907 | ADDI $13 2 2973 | EXCH $14 $13
2908 | EXCH $14 $13 2974 | ADDI $13 2
2909 | XOR $15 $14 2975 | EXCH $13 $12
2910 | EXCH $14 $13 2976 | EXCH $11 $9
2911 | ADDI $13 2 2977 | EXCH $13 $12
2912 | EXCH $13 $12 2978 | EXCH $3 $1
2913 | EXCH $11 $8 2979 | ADDI $1 1
2914 | EXCH $3 $1 2980 | EXCH $10 $1
2915 | ADDI $1 -1 2981 | ADDI $1 -1
2916 | EXCH $10 $1 2982 | EXCH $9 $1
2917 | ADDI $1 -1 2983 | ADDI $1 -1
2918 | EXCH $9 $1 2984 | EXCH $8 $1
2919 | ADDI $1 -1 2985 | ADDI $1 -1
2920 | EXCH $8 $1 2986 | EXCH $7 $1
2921 | ADDI $1 -1 2987 | ADDI $1 -1
2922 | EXCH $7 $1 2988 | EXCH $6 $1
2923 | ADDI $1 -1 2989 | ADDI $1 -1
2924 | EXCH $6 $1 2990 | EXCH $12 $1
2925 | ADDI $1 -1 2991 | ADDI $1 -1
2926 | EXCH $12 $1 2992 | ADDI $15 -2992
2927 | ADDI $1 -1 2993 | l_rjmp_top_102: | RBRA l_rjmp_bot_103
2928 | l_rjmp_bot_97: | ADDI $15 -2928 2994 | l_jmp_101: | SWAPBR $15
2929 | RBRA l_rjmp_bot_995: | NEG $15

Appendix C Example Output 201 of 231
Appendix C  Example Output 203 of 231
3524  EXCH  $14  $13  3590  ADDI  $1  1
3525  ADD  $11  3  3591  EXCH  $9  $1
3526  ADDI  $11  2  3592  XOR  $9  $0
3527  EXCH  $12  $11  3593  localBlock_147_i:  XOR  $8  $1
3528  ADDI  $11  $2  3594  ADDI  $1  1
3529  SUB  $11  3  3595  EXCH  $8  $1
3530  EXCH  $7  $1  3596  XOR  $8  $0
3531  ADDI  $1  $1  3597  localBlock_148_i:  XOR  $7  $1
3532  EXCH  $10  $11  3598  ADDI  $1  1
3533  ADDI  $1  $1  3599  EXCH  $7  $1
3534  EXCH  $9  $1  3600  XOR  $7  $0
3535  ADDI  $1  $1  3601  localBlock_149_i:  XOR  $6  $1
3536  EXCH  $8  $1  3602  l_initTape_3_bot:  BRA
3537  ADDI  $1  $1  3603  l_initTape_3_top:
3538  EXCH  $6  $1  3604  l_init_4_top:  BRA  l_init_4_bot
3539  ADDI  $1  $1  3605  l_init_4:
3540  EXCH  $7  $1  3606  EXCH  $1  $1
3541  EXCH  $13  $1  3607  ADDI  $1  $1
3542  ADDI  $1  $1  3608  l_init_4:  SWAPBR  $2
3543  ADDI  $16  $3544  3609  NEG  $2
3544  l_rjmp_top_143:  RRBA  l_rjmp_bot_144
3545  l_rjmp_bot_144:
3546  RA
3547  l_jmp_142:
3548  l_rjmp_bot_144:
3549  l_initLiterals_1
3550  l_initRules_2
3551  l_initTape_3
3552  l_initTape_3_bot:
3553  l_init_4_bot
3554  l_init_4_bot
3555  l_init_4_bot
3556  l_init_4_bot
3557  l_init_4_bot
3558  l_init_4_bot
3559  l_init_4_bot
3560  l_init_4_bot
3561  l_init_4_bot
3562  l_init_4_bot
3563  l_init_4_bot
3564  l_init_4_bot
3565  l_init_4_bot
3566  l_init_4_bot
3567  l_init_4_bot
3568  l_init_4_bot
3569  l_init_4_bot
3570  l_init_4_bot
3571  l_init_4_bot
3572  l_init_4_bot
3573  l_init_4_bot
3574  l_init_4_bot
3575  l_init_4_bot
3576  l_init_4_bot
3577  l_init_4_bot
3578  l_init_4_bot
3579  l_init_4_bot
3580  l_init_4_bot
3581  l_init_4_bot
3582  l_init_4_bot
3583  l_init_4_bot
3584  l_init_4_bot
3585  l_init_4_bot
3586  l_init_4_bot
3587  l_init_4_bot
3588  l_init_4_bot
3589  l_init_4_bot
Appendix C  Example Output 207 of 231
Appendix C  Example Ouput 209 of 231

| Line | Instruction | Address |
|------|-------------|---------|
| 3899 | ADDI $10 15 | 3952     |
| 3900 | EXCH $11 $10 | 3953   |
| 3901 | ADDI $10 -15 | 3954   |
| 3902 | SUB $10 $3 | 3955   |
| 3903 | ADD $8 $3 | 3956   |
| 3904 | ADDI $8 16 | 3957   |
| 3905 | EXCH $9 $8 | 3958   |
| 3906 | ADDI $8 -16 | 3959   |
| 3907 | SUB $8 $53 | 3960   |
| 3908 | XORI $7 1 | 3961   |
| 3909 | assert_true_165: | 3962   |
| 3910 | BRA assert_167 | 3963   |
| 3911 | test_false_166: | 3964   |
| 3912 | assert_true_167: | 3965   |
| 3913 | ADD $8 $3 | 3966   |
| 3914 | ADDI $8 16 | 3967   |
| 3915 | EXCH $9 $8 | 3968   |
| 3916 | ADDI $8 -16 | 3969   |
| 3917 | SUB $8 $53 | 3970   |
| 3918 | BNE $9 50 | 3971   |
| 3919 | cmp_top_172: | 3972   |
| 3920 | cmp_bot_173 | 3973   |
| 3921 | f_top_174: | 3974   |
| 3922 | f_bot_175: | 3975   |
| 3923 | xor $7 $11 | 3976   |
| 3924 | BEQ $10 $0 | 3977   |
| 3925 | ORI $11 1 | 3978   |
| 3926 | BEQ $10 $0 | 3979   |
| 3927 | cmp_bot_173: | 3980   |
| 3928 | cmp_top_172_i: | 3981   |
| 3929 | cmp_bot_173_i: | 3982   |
| 3930 | ADD $8 $3 | 3983   |
| 3931 | ADDI $8 16 | 3984   |
| 3932 | EXCH $9 $8 | 3985   |
| 3933 | ADDI $8 -16 | 3986   |
| 3934 | SUB $8 $53 | 3987   |
| 3935 | ADDI $8 13 | 3988   |
| 3936 | SUB $7 $53 | 3989   |
| 3937 | EXCH $8 $7 | 3990   |
| 3938 | ADD $7 -11 | 3991   |
| 3939 | ADDI $7 13 | 3992   |
| 3940 | SUB $7 $53 | 3993   |
| 3941 | EXCH $8 $7 | 3994   |
| 3942 | ADD $7 -11 | 3995   |
| 3943 | ADDI $7 13 | 3996   |
| 3944 | SUB $7 $53 | 3997   |
| 3945 | cmp_top_176: | 3998   |
| 3946 | cmp_bot_177 | 3999   |
| 3947 | f_top_178: | 4000   |
| 3948 | f_bot_179: | 4001   |
| 3949 | xor $12 1 | 4002   |
| 3950 | BEQ $11 $0 | 4003   |
| 3951 | xor $6 $12 | 4004   |
Appendix C  Example Output 210 of 231
Appendix C  Example Output 211 of 231
4264 | ADDI $15 - 3 4330 | ADD $9 $3
4265 | SUB $15 $3 4331 | ADDI $9 4
4266 | XOR $21 $14 4332 | ADD $10 $3
4267 | SUB $21 $20 4333 | ADDI $10 16
4268 | ADD $8 $21 4334 | EXCH $11 $10
4269 | ADD $21 $20 4335 | ADDI $10 16
4270 | XOR $21 $14 4336 | SUB $10 $3
4271 | ADD $15 $3 4337 | SUB $12 $11
4272 | ADDI $15 3 4338 | ADDI $12 2
4273 | ADDI $16 $3 4339 | EXCH $13 9
4274 | ADDI $16 16 4440 | XOR $12 13
4275 | EXCH $17 16 4441 | EXCH $13 9
4276 | ADDI $16 16 4442 | ADD $10 $3
4277 | SUB $16 $3 4443 | ADDI $10 16
4278 | EXCH $19 15 4444 | EXCH $11 10
4279 | XOR $18 19 4445 | ADDI $10 16
4280 | EXCH $19 15 4446 | SUB $10 $3
4281 | ADDI $18 2 4447 | ADDI $9 4
4282 | ADD $18 17 4448 | SUB $9 $3
4283 | ADD $16 $3 4449 | ADD $7 $3
4284 | ADDI $16 16 4450 | ADDI $7 11
4285 | EXCH $17 16 4451 | EXCH $8 $7
4286 | ADDI $16 16 4452 | ADDI $7 11
4287 | SUB $16 $3 4453 | SUB $7 $3
4288 | ADDI $15 3 4454 | ADD $7 3
4289 | SUB $15 $3 4455 | ADDI $7 14
4290 | EXCH $20 18 4456 | EXCH $8 $7
4291 | ADDI $15 3 4457 | ADDI $7 14
4292 | ADD $16 $3 4458 | SUB $7 $3
4293 | ADDI $16 16 4459 | ADD $9 $3
4294 | ADDI $16 16 4460 | ADD $9 6
4295 | EXCH $17 16 4461 | ADD $10 $3
4296 | ADDI $16 16 4462 | ADD $10 16
4297 | SUB $16 $3 4463 | EXCH $11 10
4298 | SUB $18 17 4464 | ADDI $10 16
4299 | ADDI $18 2 4465 | SUB $10 $3
4300 | EXCH $19 15 4466 | EXCH $13 9
4301 | XOR $18 19 4467 | XOR $12 $13
4302 | EXCH $19 15 4468 | EXCH $13 9
4303 | ADDI $16 3 4469 | ADD $12 2
4304 | ADDI $16 16 4470 | ADD $12 $11
4305 | EXCH $17 16 4471 | ADDI $10 $3
4306 | ADDI $16 16 4472 | ADDI $10 16
4307 | SUB $16 $3 4473 | EXCH $11 10
4308 | ADDI $15 3 4474 | ADDI $10 16
4309 | SUB $15 $3 4475 | SUB $10 $3
4310 | ADD $9 $3 4476 | ADDI $9 6
4311 | ADDI $9 4 4477 | SUB $9 $3
4312 | ADD $10 $3 4478 | EXCH $14 12
4313 | ADDI $10 16 4479 | ADD $9 $3
4314 | ADD $11 $10 4480 | ADD $9 6
4315 | ADDI $10 16 4481 | ADD $10 $3
4316 | SUB $10 $3 4482 | ADDI $10 16
4317 | EXCH $13 $9 4483 | EXCH $11 10
4318 | XOR $12 $13 4484 | ADDI $10 16
4319 | EXCH $13 $9 4485 | SUB $10 $3
4320 | ADDI $12 2 4486 | SUB $12 $11
4321 | ADDI $12 11 4487 | ADD $12 2
4322 | ADD $10 $3 4488 | EXCH $13 9
4323 | ADDI $10 16 4489 | XOR $12 $13
4324 | EXCH $11 $10 4490 | EXCH $13 9
4325 | ADDI $10 16 4491 | ADD $10 $3
4326 | SUB $10 $3 4492 | ADDI $10 16
4327 | ADDI $9 - 4 4493 | EXCH $11 10
4328 | SUB $9 $3 4494 | ADDI $10 16
4329 | EXCH $14 12 4495 | SUB $10 $3

Appendix C Example Output 212 of 231
ADDI $9 -6 4462
ADD $15 3 4464
ADDI $15 5 4465
ADD $16 3 4466
EXCH $17 16 4467
ADDI $16 16 4468
SUB $16 3 4469
ADD $16 3 4470
EXCH $19 15 4471
EXCH $18 19 4472
EXCH $19 15 4473
ADDI $18 2 4474
ADDI $18 17 4475
EXCH $16 16 4476
ADDI $16 3 4477
EXCH $17 16 4478
ADDI $16 -16 4479
SUB $16 3 4480
ADDI $15 -5 4481
EXCH $15 3 4482
ADDI $16 -16 4483
EXCH $20 18 4484
ADD $15 3 4485
EXCH $17 16 4486
ADDI $17 16 4487
EXCH $16 -16 4488
SUB $16 3 4489
EXCH $19 15 4490
ADD $18 17 4491
ADDI $18 -2 4492
EXCH $19 15 4493
ADD $19 18 4494
EXCH $19 15 4495
ADD $10 16 4496
EXCH $16 16 4497
ADDI $18 -6 4498
EXCH $16 -16 4499
SUB $16 3 4500
ADD $15 5 4501
ADD $15 3 4502
EXCH $21 14 4503
ADDI $21 20 4504
ADD $8 21 4505
EXCH $21 20 4506
ADDI $21 20 4507
EXCH $20 18 4508
ADDI $9 6 4509
ADD $16 3 4510
XOR $12 13 4511
ADD $17 16 4512
EXCH $16 -16 4513
SUB $16 3 4514
EXCH $19 15 4515
SUB $18 19 4516
EXCH $19 15 4517
ADDI $18 2 4518
ADD $18 -6 4519
ADD $16 3 4520
ADDI $16 16 4521
EXCH $17 16 4522
ADDI $16 -16 4523
ADD $16 3 4524
ADDI $15 -5 4525
SUB $15 3 4526
EXCH $20 18 4527
BRA assert_true_181
BRA test_false_182
BRA test_180

Appendix C  Example Output 213 of 231
assert_183:  BNE $6 $0 4591  ADDI $21 $2
assert_true_181  ADD $7 $3 4592  ADDI $21 $20
ADDI $7 $1 4593  ADD $19 $3
EXCH $8 $7 4594  ADDI $19 $16
ADDI $7 $7 4595  EXCH $20 $19
EXCH $7 $3 4596  ADDI $19 $16
ADDI $9 $3 4597  SUB $19 $3
ADDI $9 $4 4598  ADD $18 $6
ADDI $10 $3 4599  SUB $18 $3
ADDI $10 $16 4600  EXCH $23 $21
EXCH $11 $10 4601  ADD $18 $3
ADDI $10 $3 4602  ADD $18 $6
SUB $10 $3 4603  ADD $19 $3
EXCH $13 $9 4604  ADDI $19 $16
EXCH $12 $13 4605  EXCH $20 $19
XOR $13 $9 4606  ADDI $19 $16
EXCH $13 $9 4607  SUB $19 $3
ADDI $12 $11 4608  SUB $21 $20
ADD $12 $11 4609  ADDI $21 $2
ADD $10 $3 4610  EXCH $22 $18
ADDI $10 $16 4611  EXCH $22 $18
EXCH $11 $10 4612  EXCH $22 $18
ADDI $10 $16 4613  ADD $19 $3
SUB $10 $3 4614  ADDI $19 $16
ADDI $9 $4 4615  EXCH $20 $19
SUB $9 $3 4616  ADDI $19 $16
EXCH $14 $12 4617  SUB $19 $3
ADDI $9 $3 4618  ADDI $18 $6
ADDI $9 $4 4619  SUB $18 $3
ADDI $10 $3 4620  cmp_top_192:  BNE $17 $23
ADDI $10 $16 4621  cmp_bot_193:  XORI $24 $1
EXCH $11 $10 4622  cmp_top_192  BEQ $25 $0
SUB $10 $3 4623  cmp_bot_193  BEQ $25 $0
SUB $12 $11 4624  cmp_top_192  BEQ $25 $0
ADDI $12 $2 4625  cmp_bot_193  BEQ $25 $0
EXCH $13 $9 4626  cmp_top_194  BEQ $25 $0
XOR $12 $13 4627  cmp_bot_194  BEQ $25 $0
EXCH $13 $9 4628  cmp_top_194  BEQ $25 $0
ADD $10 $3 4629  cmp_bot_194  BEQ $25 $0
ADDI $10 $16 4630  cmp_top_194  BEQ $25 $0
SUB $10 $3 4631  cmp_bot_194  BEQ $25 $0
ADDI $9 $4 4632  cmp_bot_194  BEQ $25 $0
SUB $9 $3 4633  cmp_bot_194  BEQ $25 $0
BNE $8 $14 4634  cmp_bot_193:  BNE $17 $23
BNE $15 $1 4635  cmp_bot_193:  cmp_top_192_i  BNE $17 $23
cmp_top_190:  XORI $24 $1
cmp_bot_191:  cmp_top_190  XORI $24 $1
cmp_bot_191:  cmp_top_190  cmp_top_190  cmp_bot_191
ADD $16 $3 4636  ADDI $18 $3
ADD $16 $14 4637  ADD $18 $6
EXCH $17 $16 4638  ADDI $19 $3
ADD $16 $14 4639  ADDI $19 $16
SUB $16 $3 4640  EXCH $20 $19
ADD $18 $3 4641  ADDI $20 $19
ADDI $19 $3 4642  ADDI $21 $2
EXCH $20 $19 4643  SUB $21 $20
ADDI $19 $16 4644  ADD $19 $3
SUB $18 $3 4645  ADD $19 $3
EXCH $22 $18 4646  ADD $21 $20
XOR $21 $22 4647  ADD $19 $3
EXCH $22 $18 4648  ADDI $19 $16

Appendix C  Example Ouput  214 of 231
4649 | EXCH $20 $19 | 4713 | EXCH $13 $9 |
4650 | ADDI $19 -16 | 4714 | ADD $10 3 |
4651 | SUB $19 $3 | 4715 | ADDI $10 16 |
4652 | ADDI $18 -6 | 4716 | EXCH $11 10 |
4653 | SUB $18 $3 | 4717 | ADDI $10 -16 |
4654 | EXCH $23 $21 | 4718 | SUB $10 3 |
4655 | ADD $18 $3 | 4719 | ADDI $9 -4 |
4656 | ADDI $18 6 | 4720 | SUB $9 3 |
4657 | ADD $19 $3 | 4721 | ADD $7 3 |
4658 | ADD $19 16 | 4722 | ADDI $7 11 |
4659 | EXCH $20 $19 | 4723 | EXCH $8 7 |
4660 | ADDI $19 -16 | 4724 | ADDI $7 -11 |
4661 | SUB $19 $3 | 4725 | SUB $7 3 |
4662 | SUB $21 $20 | 4726 | ADD $7 3 |
4663 | ADDI $21 -2 | 4727 | ADDI $7 11 |
4664 | EXCH $22 $18 | 4728 | EXCH $8 7 |
4665 | XOR $21 $22 | 4729 | ADDI $7 -11 |
4666 | EXCH $22 $18 | 4730 | SUB $7 3 |
4667 | ADD $19 $3 | 4731 | ADDI $9 3 |
4668 | ADDI $19 16 | 4732 | ADDI $10 3 |
4669 | EXCH $20 $19 | 4733 | ADDI $10 16 |
4670 | ADD $19 -16 | 4734 | ADD $11 10 |
4671 | SUB $19 $3 | 4735 | SUB $10 16 |
4672 | ADDI $18 -6 | 4736 | ADD $10 -16 |
4673 | SUB $18 $3 | 4737 | SUB $10 3 |
4674 | ADD $16 $3 | 4738 | EXCH $10 9 |
4675 | ADDI $16 14 | 4739 | XOR $12 13 |
4676 | EXCH $17 $16 | 4740 | EXCH $13 9 |
4677 | ADD $16 -14 | 4741 | ADD $12 2 |
4678 | SUB $16 $3 | 4742 | ADD $12 11 |
4679 | cmp_bot_191_i: | 4743 | ADD $10 3 |
4680 | cmp_top_190_i | 4744 | ADDI $10 16 |
4681 | cmp_top_190_i: | 4745 | ADDI $11 10 |
4682 | cmp_bot_191_i | 4746 | ADDI $10 -16 |
4683 | ADDI $15 1 | 4747 | SUB $10 3 |
4684 | XOR $16 $14 | 4748 | ADD $9 3 |
4685 | cmp_top_190_i: | 4749 | SUB $9 3 |
4686 | EXP $15 1 | 4750 | ADD $11 12 |
4687 | cmp_bot_191_i | 4751 | ADDI $9 3 |
4688 | ADDI $16 14 | 4752 | ADD $9 3 |
4689 | SUB $16 $3 | 4753 | ADD $10 3 |
4690 | cmp_top_190_i: | 4754 | ADD $10 16 |
4691 | cmp_bot_191_i | 4755 | EXCH $11 10 |
4692 | cmp_top_190_i: | 4756 | EXCH $13 9 |
4693 | EXP $16 $16 | 4757 | EXP $10 3 |
4694 | EXP $16 $16 | 4758 | EXP $10 16 |
4695 | EXP $16 $16 | 4759 | EXP $11 10 |
4696 | EXP $16 $16 | 4760 | EXP $13 9 |
4697 | EXP $16 $16 | 4761 | EXP $13 9 |
4698 | EXP $16 $16 | 4762 | EXP $13 9 |
4699 | EXP $16 $16 | 4763 | EXP $10 3 |
4700 | EXP $16 $16 | 4764 | EXP $10 16 |
4701 | EXP $16 $16 | 4765 | EXP $11 10 |
4702 | EXP $16 $16 | 4766 | EXP $10 16 |
4703 | EXP $16 $16 | 4767 | EXP $10 3 |
4704 | EXP $16 $16 | 4768 | EXPI $9 3 |
4705 | EXP $16 $16 | 4769 | EXPI $9 3 |
4706 | cmp_top_200: | 4770 | EXP $8 14 |
4707 | cmp_bot_201 | 4771 | EXP $15 1 |
4708 | cmp_top_201: | 4772 | EXPI $9 3 |
4709 | cmp_bot_200 | 4773 | EXP $16 3 |
4710 | cmp_top_200: | 4774 | EXPI $16 5 |
4711 | cmp_bot_201 | 4775 | EXP $17 3 |
4712 | cmp_top_200: | 4776 | EXP $17 16 |
| Line Numbers | Code | Comments |
|-------------|------|----------|
| 4777-4884 | ADDI $17 -16 | 4836 | SUB $22 $3 |
| 4778-4885 | ADDI $17 -3 | 4837 | ADD $16 -3 |
| 4779-4886 | EXCH $20 $16 | 4838 | ADDI $16 $5 |
| 4780-4887 | XOR $19 $20 | 4839 | ADD $17 $3 |
| 4781-4888 | EXCH $20 $16 | 4840 | ADDI $17 $16 |
| 4782-4889 | ADDI $19 2 | 4841 | EXCH $18 $17 |
| 4783-4890 | ADD $19 $18 | 4842 | ADDI $17 -16 |
| 4784-4891 | ADD $17 $3 | 4843 | SUB $17 $3 |
| 4785-4892 | ADDI $17 16 | 4844 | EXCH $20 $16 |
| 4786-4893 | EXCH $18 $17 | 4845 | XOR $19 $20 |
| 4787-4894 | ADDI $17 -16 | 4846 | EXCH $20 $16 |
| 4788-4895 | SUB $17 $3 | 4847 | ADDI $19 2 |
| 4789-4896 | ADD $16 -5 | 4848 | ADD $19 $18 |
| 4790-4897 | SUB $16 $3 | 4849 | ADDI $17 $3 |
| 4791-4898 | EXCH $21 $19 | 4850 | ADDI $17 $16 |
| 4792-4899 | ADD $16 $3 | 4851 | EXCH $18 $17 |
| 4793-4900 | ADDI $16 5 | 4852 | ADDI $17 -16 |
| 4794-4901 | ADD $17 $3 | 4853 | SUB $17 $3 |
| 4795-4902 | ADDI $17 16 | 4854 | ADDI $16 -5 |
| 4796-4903 | EXCH $18 $17 | 4855 | SUB $16 $3 |
| 4797-4904 | ADDI $17 -16 | 4856 | EXCH $21 $19 |
| 4798-4905 | SUB $17 $3 | 4857 | ADD $16 $3 |
| 4799-4906 | SUB $19 $18 | 4858 | ADDI $16 $5 |
| 4800-4907 | ADDI $19 -2 | 4859 | ADDI $17 $3 |
| 4801-4908 | EXCH $20 $16 | 4860 | ADDI $17 $16 |
| 4802-4909 | XOR $19 $20 | 4861 | EXCH $20 $17 |
| 4803-4910 | EXCH $20 $16 | 4862 | ADDI $17 -16 |
| 4804-4911 | ADD $17 $3 | 4863 | SUB $17 $3 |
| 4805-4912 | ADD $17 16 | 4864 | SUB $19 $18 |
| 4806-4913 | EXCH $18 $17 | 4865 | ADDI $19 -2 |
| 4807-4914 | ADD $17 -16 | 4866 | EXCH $20 $16 |
| 4808-4915 | EXCH $18 $17 | 4867 | EXCH $20 $16 |
| 4809-4916 | ADD $16 -5 | 4868 | EXCH $20 $16 |
| 4810-4917 | SUB $16 $3 | 4869 | ADD $17 $3 |
| 4811-4918 | ADD $22 $3 | 4870 | ADDI $17 $3 |
| 4812-4919 | ADDI $22 7 | 4871 | EXCH $18 $17 |
| 4813-4920 | EXCH $23 $22 | 4872 | ADDI $17 -16 |
| 4814-4921 | ADDI $22 -7 | 4873 | SUB $17 $3 |
| 4815-4922 | SUB $22 $3 | 4874 | ADDI $16 -5 |
| 4816-4923 | BNE $21 $23 | 4875 | SUB $16 $3 |
| 4817-4924 | cmp_top_202: | 4876 | cmp_bot_201_i: | 4877 |
| 4818 | cmp_bot_203 | 4878 | cmp_top_200_i: | 4879 |
| 4819-4930 | cmp_bot_203: | 4879 | cmp_top_200_i: | 4880 |
| 4820 | cmp_top_202 | 4880 | cmp_bot_201_i: | 4881 |
| 4821-4931 | f_bot_204: | 4882 | f_bot_201_i: | 4883 |
| 4822-4932 | f_bot_205: | 4884 | cmp_bot_202_i: | 4885 |
| 4823-4933 | f_top_204 | 4886 | cmp_bot_200_i: | 4887 |
| 4824-4934 | f_bot_205_i: | 4888 | cmp_top_201_i: | 4889 |
| 4825-4935 | f_top_204_i: | 4890 | cmp_top_200_i: | 4891 |
| 4826-4936 | f_bot_205_i: | 4892 | cmp_top_202_i: | 4893 |
| 4827-4937 | ANDX $25 $15 | 4894 | cmp_top_201_i: | 4895 |
| 4828-4938 | BEQ $25 $0 | 4896 | cmp_bot_203_i: | 4897 |
| 4829-4939 | XORI $26 $1 | 4898 | cmp_bot_202_i: | 4899 |
| 4830-4940 | BEQ $25 $0 | 4900 | cmp_bot_203_i: | 4901 |
| 4831-4941 | ANDX $25 $15 | 4902 | cmp_bot_202_i: | 4903 |
| 4832-4942 | BEQ $25 $0 | 4904 | cmp_bot_203_i: | 4905 |
| 4833-4943 | XORI $24 $1 | 4906 | cmp_bot_201_i: | 4907 |
| 4834-4944 | BEQ $25 $0 | 4908 | cmp_bot_202_i: | 4909 |

Appendix C  Example Output 216 of 231
ADD $9 $3 4964 | EXCH $11 $10
ADDI $9 $3 4965 | ADDI $10 $16 4966
ADD $10 $3 4966 | SUB $16 $3 4967
ADDI $10 4967 | ADDI $9 $4
EXCH $11 $10 4968 | SUB $9 $3
ADDI $10 $16 4969 | ADD $15 $3
SUB $10 $3 4970 | ADDI $15 $3
ADD $12 $11 4971 | ADD $16 $3
ADDI $12 $2 4972 | ADD $16 $16
EXCH $13 $9 4973 | EXCH $17 $16
XOR $12 $113 4974 | ADDI $16 $16
EXCH $13 $9 4975 | SUB $16 $3
ADD $10 $3 4976 | EXCH $19 $15
ADDI $10 $16 4977 | XOR $18 $19
EXCH $11 $10 4978 | EXCH $19 $15
ADDI $10 $16 4979 | ADDI $18 $2
SUB $10 $3 4980 | ADD $18 $17
ADDI $9 $3 4981 | ADD $16 $3
SUB $9 $3 4982 | ADDI $16 $16
ADD $7 $3 4983 | EXCH $17 $16
ADD $7 $11 4984 | ADDI $16 $16
EXCH $8 $7 4985 | SUB $16 $3
ADD $7 $11 4986 | ADDI $15 $3
SUB $7 $3 4987 | SUB $15 $3
BEQ $6 $50 4988 | EXCH $20 $18
XORI $6 $1 4989 | ADD $15 $3
ADD $7 $3 4990 | ADDI $15 $3
ADDI $7 $11 4991 | ADD $16 $3
EXCH $8 $7 4992 | ADDI $16 $16
ADDI $7 $11 4993 | EXCH $17 $16
EXCH $8 $7 4994 | ADDI $16 $16
SUB $7 $3 4995 | SUB $16 $3
ADD $7 $3 4996 | SUB $18 $17
ADD $9 $3 4997 | ADDI $18 $2
SUB $10 $3 4998 | ADDI $15 $3
ADDI $10 $16 4999 | EXCH $19 $15
EXCH $11 $10 5000 | XOR $18 $19
ADDI $10 $16 5001 | ADD $16 $3
SUB $10 $3 5002 | ADD $16 $16
EXCH $13 $9 5003 | EXCH $17 $16
SUB $10 $3 5004 | ADD $16 $16
ADDI $10 $16 5005 | SUB $16 $3
ADDI $10 $16 5006 | ADDI $15 $3
ADDI $10 $11 5007 | SUB $15 $3
ADD $10 $3 5008 | XOR $21 $14
ADDI $10 $16 5009 | SUB $21 $20
EXCH $11 $10 5010 | ADD $8 $21
ADDI $10 $16 5011 | ADD $21 $20
SUB $10 $3 5012 | XOR $21 $14
ADDI $9 $4 5013 | ADD $15 $3
SUB $9 $3 5014 | ADDI $15 $3
EXCH $14 $12 5015 | ADD $16 $3
ADDI $9 $3 5016 | ADDI $16 $16
ADDI $9 $4 5017 | EXCH $17 $16
ADDI $10 $3 5018 | ADDI $16 $16
ADDI $10 $16 5019 | SUB $16 $3
EXCH $11 $10 5020 | EXCH $19 $15
ADDI $10 $16 5021 | XOR $18 $19
EXCH $11 $10 5022 | EXCH $19 $15
SUB $10 $3 5023 | ADD $18 $2
SUB $12 $11 5024 | ADD $18 $17
EXCH $13 $9 5025 | ADD $16 $3
EXCH $12 $11 5026 | ADDI $16 $16
EXCH $13 $9 5027 | EXCH $17 $16
ADD $10 $3 5028 | ADDI $16 $16
ADDI $10 $16 5029 | SUB $16 $3
ADDI $15 -3 5096
ADD $8 $3
ADDI $53 6
ADDI $53 3 $6
ADD $9 $3
ADDI $53 9 $16
ADDI $53 5100
EXCH $10 $9
ADDI $53 5101
ADDI $9 $16
ADDI $53 5102
SUB $9 $3
EXCH $53 5103
EXCH $12 $8
ADDI $53 5104
XOR $11 $12
SUB $53 5105
EXCH $12 $8
ADDI $53 5106
ADD $11 2
ADD $10 -2 5107
ADDI $11 $10 5108
ADDI $9 $3
ADD $10 -19 5109
ADDI $9 $16
ADDI $19 $15 5110
EXCH $10 $9
ADDI $16 $3 5111
ADDI $9 $16
ADDI $16 $16 5112
SUB $9 $3
EXCH $17 $16 5113
ADDI $13 $11
ADDI $16 -16 5114
SUB $8 $3
ADD $16 $15 5115
EXCH $53 $11
ADD $15 -3 5116
SUB $9 $3
ADD $8 $3
ADD $15 $3 5117
EXCH $11 $10 5122
ADDI $9 $3
ADD $10 -16 5123
SUB $11 $10 5124
ADDI $11 $10 5125
EXCH $12 $8
XOR $12 $13 5126
XOR $11 $12
EXCH $13 $9 5127
EXCH $12 $8
ADD $12 2 5128
ADD $9 $3
ADD $12 $11 5129
ADDI $9 $16
ADD $10 $3 5130
ADDI $10 $9 $9
ADDI $10 $16 5131
ADDI $9 $16
EXCH $11 $10 5132
SUB $9 $3
ADDI $10 -16 5133
ADDI $8 $6
ADDI $10 $3 5134
SUB $8 $3
ADD $9 -4 5135
ADDI $14 $3
ADD $9 $3 5136
SUB $14 $9
EXCH $14 $12 5137
EXCH $12 $8
ADD $14 $3 5138
ADDI $14 $9
ADD $9 $3 5139
SUB $14 $3
ADD $10 $3 5140
ADDI $13 $15
cmp_top_210:
cmp_bot_211
ADDI $10 $3 5141
BNE $13 $15
ADD $10 $16 5142
cmp_bot_211
cmp_top_210
SUB $10 $3 5143
BEQ $16 $0
SUB $12 $11 5144
f_bot_212:
f_bot_213
f_bot_213
EXCH $13 $9 5145
f_bot_213:
BEQ $16 $0
EXCH $13 $9 5146
f_top_212
XOR $17 $1
XOR $16 $1
ADD $10 $3 5147
SUB $7 $17
ADDI $10 $16 5148
BEQ $16 $0
f_bot_213:
f_bot_213
ADDI $11 $10 5149
XOR $17 $1
ADD $10 $16 5150
f_top_212:
f_top_212
ADDI $9 -4 5151
XOR $16 $1
SUB $9 $3 5152
cmp_bot_211:
cmp_top_210
ADDI $7 $3 5153
cmp_top_210_i
ADDI $7 $3 5154
cmp_bot_211_i
ADD $14 $3
SUB $7 $3 5155
BNE $13 $15
Appendix C  Example Output 218 of 231
ADDI $14 9 5218
EXCH $15 $14 5219
ADD $14 $9 5220
SUB $14 $3 5221
ADD $8 $3 5222
ADD $8 6 5223
ADD $9 $3 5224
ADD $9 $9 5225
EXCH $10 $9 5226
ADD $9 $9 5227
ADD $9 $3 5228
EXCH $12 $8 5229
XOR $11 $12 5230
SUB $11 $2 5232
ADD $11 $10 5233
ADD $9 $3 5234
SUB $9 $3 5235
ADD $9 $9 5236
ADDI $9 $10 5237
SUB $9 $3 5238
ADD $9 $16 5239
EXCH $10 $9 5240
ADD $9 $9 5242
ADD $8 $6 5243
ADD $9 $3 5244
ADD $9 $16 5245
EXCH $10 $9 5246
ADDI $9 $9 5247
ADDI $9 $3 5248
ADDI $11 $10 5249
EXCH $12 $8 5250
XORI $11 $12 5252
cmp_top_214:
  cmp_bot_215
EXCH $12 $8 5253
ADD $9 $3 5254
cmp_bot_215:
  XORI $9 $9 5255
EXCH $10 $9 5256
ADD $9 $9 5257
SUB $9 $3 5258
ADD $9 $9 5259
SUB $9 $3 5260
ADD $9 $9 5261
SUB $9 $3 5262
ADD $9 $9 5263
ADD $8 $6 5264
ADDI $8 $3 5265
ADDI $8 6 5266
SUB $8 $3 5267
ADD $8 6 5268
ADD $8 $3 5269
ADD $8 6 5270
SUB $8 $3 5271
ADD $8 6 5272
ADD $9 $9 5273
EXCH $12 $8 5274
XORI $11 $12 5275
EXCH $12 $8 5275

APPENDIX C Example Output 219 of 231
assert_true_197:  
BRA assert_199  
5586  
test_false_198:  
BRA test_196  
5587  
assert_199:  
assert_true_197  
5588  
ADD $7 $3  
5590  
ADD $7 $11  
5591  
EXCH $8 $7  
5592  
ADD $7 $11  
5593  
SUB $7 $3  
5594  
ADD $9 $3  
5595  
ADD $9 $4  
5596  
ADD $10 $3  
5597  
ADD $10 $16  
5598  
EXCH $11 $10  
5599  
ADD $10 $16  
5600  
SUB $10 $3  
5601  
ADD $10 $3  
5602  
EXCH $13 $9  
5603  
XOR $12 $13  
5604  
EXCH $13 $9  
5605  
ADD $12 $2  
5606  
ADD $12 $11  
5607  
ADD $10 $16  
5608  
EXCH $11 $10  
5609  
ADD $10 $16  
5610  
SUB $10 $3  
5611  
ADD $9 $4  
5612  
SUB $9 $3  
5613  
EXCH $14 $12  
5614  
ADD $9 $3  
5615  
ADD $9 $4  
5616  
ADD $10 $3  
5617  
cmp_top_232:  
cmp_bot_233  
5618  
ADD $10 $16  
5619  
EXCH $11 $10  
5620  
SUB $10 $3  
5621  
ADD $12 $2  
5622  
EXCH $13 $9  
5623  
XOR $12 $13  
5624  
EXCH $13 $9  
5625  
ADD $10 $3  
5626  
EXCH $11 $10  
5627  
ADD $10 $16  
5628  
SUB $10 $3  
5629  
ADD $9 $4  
5630  
cmp_top_230:  
cmp_bot_231  
5631  
cmp_bot_231:  
cmp_top_230  
5632  
ADD $16 $3  
5633  
ADD $17 $3  
5634  
EXCH $18 $17  
5635  
ADD $18 $17  
5636  
SUB $17 $3  
5637  
EXCH $20 $16  
5638  
XOR $19 $20  
5639  
EXCH $20 $16  
5640  
ADD $19 $2  
5641  
ADD $19 $18  
5642  
Appendix C Example Output 222 of 231
SUB $17 $3 5707  ADDI $12 -2
EXCH $20 $16 5708  EXCH $13 $9
XOR $19 $20 5709  XOR $12 $13
EXCH $20 $16 5710  EXCH $13 $9
ADDI $19 2 5711  ADD $10 $3
ADD $19 $18 5712  ADD $10 16
ADDI $17 3 5713  EXCH $11 $10
ADDI $17 16 5714  ADDI $10 -16
EXCH $18 $17 5715  SUB $10 $3
ADDI $17 -16 5716  ADDI $9 -4
SUB $17 3 5717  SUB $9 $3
ADDI $16 -5 5718  ADD $7 $3
SUB $16 3 5719  ADDI $7 11
EXCH $21 $19 5720  EXCH $8 $7
ADDI $16 3 5721  ADDI $7 -11
ADDI $16 5 5722  SUB $7 $3
ADDI $17 3 5723  l_inst_6_bot:  BRA l_inst_6_top
ADDI $17 16 5724  l_moveRight_7_bot:  BRA
EXCH $18 $17 5725  ADDI $1 1
ADD $17 -16 5726  ADDI $1 1
ADDI $18 $18 5727  EXCH $3 $1
ADD $19 -2 5728  EXCH $2 $1
EXCH $20 $16 5729  l_moveRight_7:  SWAPBR $2
OR $19 $20 5730  NEG $2
EXCH $20 $16 5731  ADDI $1 1
ADDI $17 $3 5732  EXCH $3 $1
ADDI $17 16 5733  EXCH $2 $1
EXCH $18 $17 5734  ADDI $1 -1
ADDI $17 -16 5735  localBlock_302:  XOR $6 $1
SUB $17 3 5736  XOR $7 $0
ADDI $16 -5 5737  EXCH $7 $1
SUB $16 3 5738  ADDI $1 -1
BNE $8 $14 5739  localBlock_301:  XOR $7 $1
cmp_bot_231_i:  cmp_top_230_i
XORI $15 1 5740  XOR $8 $0
BNE $8 $14 5741  EXCH $8 $1
ADD $9 9 5742  ADDI $1 -1
ADD $9 4 5743  ADDI $3 $3
ADD $10 $3 5744  ADDI $8 -2
ADDI $10 16 5745  EXCH $9 $8
EXCH $11 $10 5746  EXCH $8 $2
ADDI $10 16 5747  SUB $8 $3
EXCH $11 $10 5748  XOR $10 $9
ADD $10 -16 5749  loadMetAdd_236:  EXCH $11 $10
SUB $10 $3 5750  ADDI $11 3
EXCH $13 $9 5751  ADD $11 $3
xor $12 $13 5752  EXCH $12 $11
EXCH $13 $9 5753  EXCH $12 $11
ADDI $12 12 5754  ADDI $11 -3
ADDI $12 11 5755  ADDI $11 $10
ADD $10 $3 5756  ADD $8 $3
ADDI $10 16 5757  ADDI $8 2
EXCH $11 $10 5758  EXCH $9 $8
ADD $10 -16 5759  ADD $8 -2
SUB $10 $3 5760  SUB $8 $3
ADD $9 -4 5761  ADDI $14 $3
SUB $9 $3 5762  ADDI $14 14
EXCH $14 $12 5763  EXCH $3 $1
ADD $9 3 5764  ADDI $1 -1
ADDI $9 4 5765  EXCH $7 $1
ADD $10 $3 5766  ADDI $1 -1
ADDI $10 16 5767  EXCH $6 $1
EXCH $11 $10 5768  ADD $1 $1
ADDI $10 -16 5769  ADDI $14 $1
SUB $10 $3 5770  ADDI $1 -1
SUB $12 $11 5771  EXCH $10 $1
loadMetAdd_289_i:
ADDI $12 -1 6463    ADDI $1 -1
EXCH $12 $11 6464    EXCH $7 $1
ADD $9 $3 6465      ADDI $1 -1
ADDI $9 2 6466      EXCH $6 $1
EXCH $10 $9 6467    ADDI $1 -1
ADDI $9 -2 6468     EXCH $15 $1
SUB $9 $3 6469      ADDI $1 -1
EXCH $3 $1 6470     EXCH $11 $1
ADDI $1 -1 6471     ADDI $1 -1
EXCH $7 $1 6472     ADDI $14 -6471
ADDI $1 -1 6473 l_jmp_292: SWAPBR $14
EXCH $6 $1 6474     NEG $14
ADDI $1 -1 6475     ADD $14 6471
EXCH $11 $1 6476    ADDI $1 1
ADDI $1 -1 6477     EXCH $11 $1
ADDI $14 -6411 6478 ADDI $1 1
l_jmp_290: SWAPBR $14 6479 EXCH $15 $1
NEG $14 6480      ADDI $1 1
ADDI $14 6411 6481 EXCH $6 $1
ADDI $1 $1 6482    ADDI $1 1
EXCH $11 $1 6483    EXCH $7 $1
ADDI $1 1 6484     ADDI $1 1
EXCH $6 $1 6485     EXCH $3 $1
ADDI $1 1 6486     ADDI $15 -14
EXCH $7 $1 6487     SUB $15 $3
ADDI $1 1 6488     ADD $9 $3
EXCH $3 $1 6489     ADDI $9 2
ADDI $9 $3 6490     EXCH $10 $9
ADDI $9 2 6491      ADDI $9 -2
EXCH $10 $9 6492     SUB $9 $3
ADDI $9 -2 6493     EXCH $12 $11
SUB $9 $3 6494      ADDI $12 3
EXCH $12 $11 6495    EXCH $13 $12
ADDI $12 1 6496     XOR $14 $13
EXCH $13 $12 6497    EXCH $13 $12
XOR $14 $13 6498     ADD $12 -3
EXCH $13 $12 6499 loadMetAdd_291_i: EXCH $12 $11
ADDI $12 -1 6500    XOR $11 $10
EXCH $12 $11 6501    ADD $9 $3
XOR $11 $10       6502 ADDI $9 2
SUB $9 $3          6503 EXCH $10 $9
ADDI $9 2          6504 ADDI $9 -2
EXCH $10 $9        6505 SUB $9 $3
ADDI $9 -2         6506 assert_245: BNE $8 $0
SUB $9 $3          6507 assert_true_243
ADDI $9 $3         6508 cmp_top_293: BNE $9 $0
EXCH $10 $9        6509 cmp_bot_294
ADDI $9 -2         6510    cmp_bot_293
SUB $9 $3          6511 cmp_top_293
ADDI $9 2          6512 f_bot_296
EXCH $10 $9        6513 f_top_295: BEQ $10 $0
ADDI $9 -2         6514    f_bot_296
SUB $9 $3          6515 f_top_295:BEQ $10 $0
ADDI $9 2          6516 f_bot_296:BEQ $10 $0
EXCH $10 $9        6517 f_top_295:BEQ $10 $0
ADDI $9 -2         6518 cmp_top_293_i:BNE $9 $0
SUB $9 $3          6519 cmp_top_293_i:BNE $9 $0
ADDI $15 $3        6520 cmp_top_293_i:BNE $9 $0
cmp_bot_294_i

6521  EXCH $9 $6 6586  XOR $7 $0
6522  ADDI $8 $3 6587  localBlock_302_i: XOR $6 $1
6523  ADDI $8 2 6588  l_moveRight_7_bot: BRA
6524  EXCH $9 $8 6589  l_moveRight_7_top
6525  ADDI $8 -2 6590  ADDI $1 1
6526  SUB $8 $3 6591  EXCH $2 $1
6527  XOR $10 $9 6592  EXCH $3 $1
6528  EXCH $11 $10 6593  ADDI $1 -1
loadMetAdd_297:
6529  ADDI $11 1 6594  l_main_0:
6530  EXCH $12 $11 6595  SWAPBR $2
6531  XOR $13 $12 6596  ADDI $1 1
6532  EXCH $12 $11 6597  EXCH $3 $1
6533  ADDI $11 -1 6598  EXCH $2 $1
6534  EXCH $11 $10 6599  ADDI $1 -1
6535  ADD $8 $3 6600  EXCH $3 $1
6536  ADDI $8 2 6601  ADDI $1 -1
6537  EXCH $9 $8 6602  obj_con_303: ADDI $8 $2
6538  ADDI $8 -2 6603  EXCH $8 $1
6539  SUB $8 $3 6604  ADDI $1 -1
6540  EXCH $3 $1 6605  EXCH $7 $1
6541  ADDI $1 -1 6606  ADDI $1 -1
6542  EXCH $7 $1 6607  BRA l_malloc
6543  ADDI $1 -1 6608  ADDI $1 1
6544  EXCH $6 $1 6609  EXCH $7 $1
6545  ADDI $1 -1 6610  ADDI $1 1
6546  EXCH $10 $1 6611  EXCH $8 $1
6547  ADDI $1 -1 6612 obj_con_303_i: ADDI $8 -32
6548  ADDI $13 -6548 6613  ADDI $1 1
6549  l_rjmp_top_299: 6614  ADDI $6 $3
6550  l_rjmp_298: 6615  ADD $6 $3
6551  l_rjmp_bot_299: 6616  ADDI $6 2
6552  BRA l_rjmp_top_28087: 6617  ADDI $11 1
6553  ADDI $13 6658 6618  ADDI $1 2
6554  EXCH $10 $1 6619  EXCH $8 $7
6555  ADDI $1 1 6620  ADDI $1 1
6556  EXCH $11 $10 6621  EXCH $8 $1
6557  ADDI $1 1 6622  obj_con_303_bot: ADDI $7 -1
6558  EXCH $6 $1 6623  ADDI $6 -2
6559  EXCH $7 $1 6624  ADDI $6 -2
6560  ADDI $1 1 6625  SUB $6 $3
6561  EXCH $3 $1 6626  ADD $6 $3
6562  ADD $8 $3 6627  ADDI $6 2
6563  ADDI $8 2 6628  EXCH $7 $6
6564  EXCH $9 $8 6629  ADDI $6 -2
6565  ADDI $8 -2 6630  SUB $6 $3
6566  SUB $8 $3 6631  XOR $8 $7
6567  EXCH $11 $10 6632  loadMetAdd_304: EXCH $9 $8
6568  ADDI $11 1 6633  ADDI $9 3
6569  EXCH $12 $11 6634  EXCH $10 $9
6570  XOR $13 $12 6635  XOR $11 $10
6571  EXCH $12 $11 6636  EXCH $10 $9
6572  ADDI $11 -1 6637  ADDI $9 -3
loadMetAdd_297_i:
6573  EXCH $11 $10 6638  loadMetAdd_304:
6574  XOR $10 $9 6639  EXCH $9 $8
6575  ADD $8 $3 6640  ADD $6 $3
6576  ADD $8 2 6641  ADDI $6 2
6577  EXCH $9 $8 6642  EXCH $7 $6
6578  ADDI $8 -2 6643  ADDI $6 -2
6579  SUB $8 $3 6644  SUB $6 $3
6580  ADD $1 1 6645  ADDI $3 $1
6581  EXCH $8 $1 6646  ADDI $1 -1
6582  XOR $8 $0 6647  EXCH $8 $0
6583  localBlock_301_i:
6584  EXCH $11 $10 6648  ADDI $11 -6647
6585  EXCH $7 $1 6649  l_jmp_305: SWAPBR $11
6586  EXCH $7 $1 6650  NEG $11

Appendix C Example Output 230 of 231
ADDI $11 6647 6684
ADDI $1 1 6685
EXCH $8 $1 6686 XOR $1 $5 6654
ADDI $1 1 6687 ADDI $1 2048 6655
EXCH $3 $1 6688 ADDI $1 -4 6656
ADD $6 $3 6689 XOR $3 $1 6657
ADDI $6 2 6690 XORI $6 2 6658
EXCH $7 $6 6691 EXCH $6 $3 6659
ADDI $6 -2 6692 ADDI $1 -1 6660
EXCH $7 $6 6693 EXCH $3 $1 6661
EXCH $9 58 6694 ADDI $1 -1 6662
ADDI $9 3 6695 BRA l_main_0 6663
EXCH $10 $9 6696 ADDI $1 1 6664
EXCH $10 $9 6696 ADDI $3 $1 6665
EXCH $10 $9 6696 ADDI $3 1 6666
EXCH $10 $9 6696 ADDI $3 1 6667
loadMetAdd_304_i:
EXCH $9 58 6700 EXCH $6 $3 6668
XOR $8 $7 6701 XORI $7 1 6669
ADD $6 $3 6702 EXCH $6 $7 6670
ADD $6 2 6703 XORI $7 1 6671
EXCH $7 56 6704 ADDI $3 -1 6672
ADDI $6 -2 6705 ADDI $3 -1 6673
SUB $6 $3 6706 ADDI $1 1 6674
finish:
FINISH

Appendix C Example Output 231 of 231

start:
BRA l_main_0_top 6707 EXCH $6 $3 6675
BRA top 6708 XORI $6 2 6676
START 6709 XOR $3 $1 6677
ADDI $4 6715 6710 ADDI $1 4 6678
XOR $5 04 6711 ADDI $1 -2048 6679
ADDI $5 10 6712 XOR $1 $5 6680
XOR $7 55 6713 ADDI $5 -10 6681
ADDI $4 10 6714 XOR $5 $4 6682
ADDI $4 -1 6715 ADDI $4 -6715 6683
EXCH $7 04 6716 finish:
FINISH