FACULTY OF SCIENCE
UNIVERSITY OF COPENHAGEN

Master's Thesis

Tue Haulund - qvr916@alumni.ku.dk

Design and Implementation of a Reversible Object-Oriented Programming Language

Main Supervisor: Torben Ægidius Mogensen
Co-Supervisor: Robert Glück
Submitted: November 8, 2016

Abstract

High-level reversible programming languages are few and far between and in general offer only rudimentary abstractions from the details of the underlying machine. Modern programming languages offer a wide array of language constructs and paradigms to facilitate the design of abstract interfaces, but we currently have a very limited understanding of the applicability of such features for reversible programming languages.

We introduce the first reversible object-oriented programming language, ROOPL, with support for user-defined data types, class inheritance and subtype-polymorphism. The language extends the design of existing reversible imperative languages and it allows for effective implementation on reversible machines.

We provide a formalization of the language semantics, the type system and we demonstrate the computational universality of the language by implementing a reversible Turing machine simulator. ROOPL statements are locally invertible at no extra cost to program size or computational complexity and the language provides direct access to the inverse semantics of each class method.

We describe the techniques required for a garbage-free translation from ROOPL to the reversible assembly language PISA and provide a full implementation of said techniques. Our results indicate that core language features for object-oriented programming carries over to the field of reversible computing in some capacity.

Preface

> “A language that doesn’t affect the way you think about programming, is not worth knowing“
>
> – Alan J. Perlis, *Epigrams on Programming* [39]

The present thesis constitutes a 30 ECTS workload and is submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at the University of Copenhagen (UCPH), Department of Computer Science (DIKU).

The thesis report consists of 110 numbered pages, a title page and a ZIP archive containing source code developed as part of the thesis work. The thesis was submitted for grading on November 8, 2016 and will be subject to an oral defense no later than December 6, 2016.

I would like to express my sincerest appreciation for the invaluable direction and encouragement of my primary academic supervisor, Torben Mogensen. I would also like to thank my co-supervisor Robert Glück, for introducing me to the fascinating field of reversible computing and for his help with the thesis subject. Finally - a heartfelt appreciation is owed to my loving partner Matilde, without whom this thesis would not have been possible.

Copenhagen, Autumn 2016

Tue Haulund

Table of Contents

List of Figures

CHAPTER 1

Introduction

Reversible computing is the study of time-invertible, two-directional models of computation. At any point during a reversible computation, there is at most one previous and one subsequent computational state, both of which are uniquely determined by the current state. The computational process follows a deterministic trajectory of these states in either direction of execution and carefully avoids erasing information such that previous states remain reachable and unique. As a result of this perfect preservation of information, reversible computing offers a possible solution to the heat dissipation problems faced by manufacturers of microprocessors [28].

To realize a fully reversible computing system, we need reversibility at every level of abstraction. Much headway has been made at the circuit and gate level, such as the realization of the reversible *Pendulum* architecture [42] based on the reversible universal Fredkin and Toffoli gates [19]. High-level reversible programming languages are also actively researched, most notably the imperative reversible language *Janus* [30, 49, 46], the procedural reversible language *R* [17, 18] and the functional reversible languages *RFUN* [48] and *Inv* [37]. Recently, translation of these languages to low-level reversible assembly languages has been the subject of some work [2, 25]. A reversible self-interpreter for the reversible imperative language *R-WHILE* was shown in [23].

Throughout this existing body of research, a reversible object-oriented language has yet to be formalized. The present thesis discusses the design of such a language as well as the techniques required to perform a clean (i.e. garbage-free) and correct translation from such a language to a low-level reversible assembly language. As is the case for any programming paradigm, reversible object-oriented programming has its own programming techniques and pitfalls, which we will explore in detail. The language will implement traditional OOP concepts such as encapsulation, subtype polymorphism and dynamic dispatch, albeit in a reversible context.

1.1 Reversible Computing

A great deal of effort is expended on minimizing the power consumption of modern microprocessors, to the point where it is now considered a first-class design constraint. However a theoretical lower limit does exist for our current model of computation. Known since the early 1960's, *Landauer's principle* holds that:

> [...] any logically irreversible manipulation of information, such as the erasure of a bit or the merging of two computation paths, must be accompanied by a corresponding entropy increase in non-information-bearing degrees of freedom of the information-processing apparatus or its environment. [28]

Put simply, Landauer's principle states that the erasure of information in a system is always accompanied by an increase in energy consumption. The exact amount of energy required to erase n bits of information is $n \cdot k_B \cdot T \cdot \ln 2$, where T is the temperature of the circuit in kelvin and k_B is the *Boltzmann constant* (approximately $1.38 \cdot 10^{-23}$ J/K) [7].

This theoretical limit is known as the *von Neumann-Landauer limit* and it places a lower bound on the energy consumption of any computation involving the erasure of information. In a reversible computation, information is never erased, which means reversible computing systems are not subject to the von Neumann-Landauer limit[1].

The naive approach to achieving reversibility is based on the idea of *reversibilization* of a regular irreversible program. As the program is executing, intermediate values are preserved in a program history trace. Known as a *Landauer embedding*, this technique achieves perfect preservation of information [28]. Bennett showed that such an embedding can be created for any irreversible program [6], however the space requirements for this technique grows proportionally to the length of time the program has been running. Given an irreversible program with running time T and space complexity S, a semantically equivalent reversible program with running time $O(T^{1 + \epsilon})$ and space complexity $O(S \ln T)$ can be constructed for some $\epsilon > 0$ [8]. These space requirements make this approach completely impractical for general purposes.

The Landauer embedding is an example of *injectivization* of the function that our program computes. As we cannot accept the generation of this extraneous garbage data, we must limit ourselves to programs that compute functions that are already injective (i.e. one-to-one functions). Reversible programming languages are made up of individually reversible execution steps, each of which must also be injective when viewed as a mapping from one computational state to the next. This one-to-one mapping ensures that the language is both forwards and backwards deterministic, there is always at most one state the computation can transition to, regardless of the direction of execution.

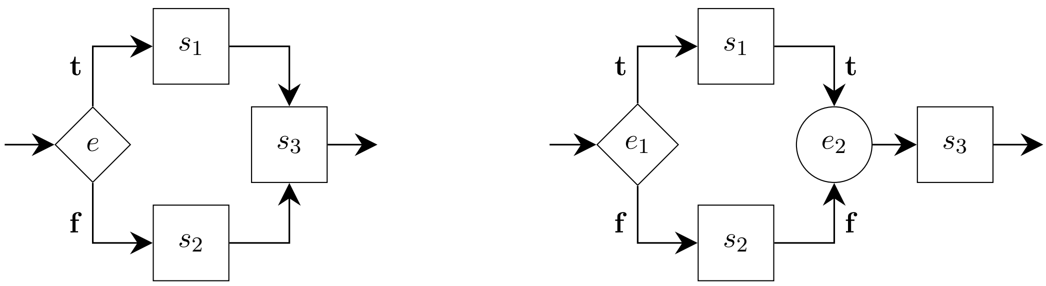

Figure 1.1: Flowcharts of irreversible and reversible variants of a conditional statement followed by some other statement s_3. The reversible variant uses the assertion e_2 to join the two paths of computation reversibly - if the control flow reaches e_2 from the **true**-edge then e_2 must evaluate to **true** and vice versa, otherwise the statement is undefined. [47]

In irreversible programming languages, this mapping can be a many-to-one (non-injective) function since we are then only concerned with forward determinism. The inverse of such a function is a one-to-many relation (sometimes called a *multivalued function*) which means such languages are backwards non-deterministic, as it is impossible to uniquely determine the previous state of computation[2].

Every reversible program has exactly one corresponding inverse program in which every

[1] Aside from its relationship to reversible computing, Landauer's principle also represents a compelling argument that Maxwell's Demon does not violate the second law of thermodynamics [9]

[2] Some languages are both forwards and backwards non-deterministic by design - the logic programming language Prolog is an example of such a language.

execution step is inverted and performed in reverse order of the original program. Since each execution step is locally invertible, as opposed to requiring a full-program analysis, the inversion can be achieved with straightforward recursive descent over the components of the program. Furthermore, given that each single execution step has a single-step inverse, the process of inverting a reversible program bears no additional cost in terms of program size.

Reversible programming languages may provide direct access to the inverse semantics of a code segment, in Janus this is exemplified by the *uncall* statement which invokes the inverse computation of a given procedure [49], while low-level reversible languages typically make use of a direction bit to invoke inverse semantics and reverse execution [41]. This direct access has given rise to some clever programming methodologies. One example is known as the *Lecerf-Bennet reversal*[3] which makes use of *uncomputation* to reversibly purge the variable store of undesired intermediate values after a computation.

For some computations, having direct and inexpensive access to the exact inverse computation can be useful from a software development perspective. For example, implementing a compression algorithm in a reversible language[4] immediately yields the equivalent decompression routine by inversion of the program. Additionally, any effort that has gone into verifying the correctness of the compression algorithm, e.g. testing or perhaps even formal verification techniques such as model checking, can serve as an equally valid testament to the correctness of the inverse program (assuming the process of inversion is itself correct).

Besides the primary motivation of potentially improving the energy efficiency of computers beyond the von Neumann-Landauer limit, the field of reversible computing shows promise in a number of other areas:

Quantum Computing A quantum logic gate represents a transformation which can be applied to an isolated quantum system. For the resulting system to be consistent, the transformation matrix must be *unitary.* Such transformations are inherently reversible, and indeed any reversible boolean function can be converted to a corresponding unitary transformation [12]. As such, the field of quantum computing could stand to benefit from an increased understanding of reversible computing.

Program Debugging Traditional program debugging involves stepping through code line by line, inspecting intermediate results and memory contents accordingly. Recently, vendors have added support for *reverse debugging*, which involves stepping through code in reverse or restoring earlier program states from within a debugging session. This is usually implemented with a continous execution trace but on a reversible computing platform, such functionality is supported as a fundamental property of the system. A reversible extension to the Erlang programming language, for the purpose of supporting reverse debugging was suggested in [38].

Error Recovery In parallel or pipeline-based systems, recovery from an unforeseen error condition often involves undoing recent related changes made to the state of the system. As an example, this is a primary features of most DBMS and it is implemented with special-purpose error recovery logic. On a reversible system however, this can be achieved by simple reverse execution back to the point where the error condition first arose. A reversible DSL for error recovery on robotic assembly lines was presented in [40].

---

[3] Also known as the *local Bennett's method* [49], or the *compute-copy-uncompute paradigm.* It was first proposed by Lecerf [29] and later rediscovered by Bennett [6].

[4] The futility of attempting to implement a *lossy* compression algorithm within a language paradigm that forbids the erasure of information should not be lost on the reader at this point.

Discrete Event Simulation The simulation of systems with asynchronous discrete update events lends itself well to concurrent execution. Suggested in [27], the Dynamic Time Warp (DTW) algorithm is commonly used to synchronize event updates across execution threads. DTW uses update rollbacks to restore the simulation to a synchronized state, in case an event has been committed prematurely. Reversible computation can be used to realize event rollback while avoiding the high overhead of storing execution traces or simulation checkpoints [14].

1.2 Object-Oriented Programming

Like reversible computing, object-oriented programming (OOP) originated in the early 1960's, with the advent of the Simula language [10]. *Unlike* reversible computing, OOP enjoys immense popularity in the software industry, as can be observed by the widespread use of object-oriented languages such as Java and C++. The OOP paradigm attempts to break a problem into many small manageable pieces of related state and behaviour called objects. An object may model an actual object in the problem domain, or it may represent a more abstract grouping of related entities within a program. A distinction is made between a particular kind or type of object, called a *class*, and specific instances of these classes, known simply as objects.

OOP is based on the concept of encapsulation: Only the methods of an object has unrestricted access to the components of that object, thereby protecting the integrity of the internal state and reducing the overall system complexity. Encapsulation is closely related to the principle of information hiding, which holds that compartmentalization of design decisions made in one part of a program can be used to avoid extensive modification of other parts of that program if the design is altered [20, Chapter 1].

A fundamental aspect of OOP is class inheritance, which allows one class to inherit the fields and methods of another class. Most OOP languages also use inheritance to establish an "is-a" relationship between two objects such that one may be substituted for the other by subtype-polymorphism. OOP lends itself well to code-reuse and maintainability of source code, and is often used in combination with imperative or procedural programming paradigms. In general, OOP is a set of techniques for intuitively structuring imperative code - it is a programming methodology rather than a model of computation.

1.3 Motivation

After more than two dozen iterations of Moore's Law [36], the semiconductor industry is fast approaching the von Neumann-Landauer limit. Reversible computing may be a viable solution, but it represents a significant paradigm shift from the currently prevailing irreversible models of computation.

The practicality of reversible computing hinges, inter alia, on the presence of high-level reversible programming languages that can be compiled to low-level reversible assembly code without significant overhead. Ideally, these languages should provide the same tools and features for producing abstract models and interfaces as are available for modern irreversible languages.

Object-oriented programming is immensely popular in the industry but the combination of OOP and reversible computing is entirely uncharted territory. The work presented in this thesis is motivated by the scarcity of high-level reversible programming languages and in particular, by the absence of any reversible object-oriented programming languages.

1.4 Thesis Statement

An effective implementation of a reversible object-oriented programming language is both possible and practical, provided the design of the language observes the limitations required for execution on reversible machines.

1.5 Outline

This thesis consists of 5 chapters, the first of which is this introductory chapter. The remaining 4 chapters are summarised as follows:

Chapter 2 is a brief survey of existing reversible imperative programming languages and instruction sets.

Chapter 3 presents the reversible object-oriented programming language ROOPL, along with a formalization of the language and a discussion of the most significant elements of its design.

Chapter 4 presents the techniques required for a garbage-free and correct compilation from ROOPL source code to PISA instructions.

Chapter 5 contains conclusions and proposals for future work.

The appendix contains the source code listings for the ROOPL compiler, an example ROOPL program and the equivalent translated PISA program.

CHAPTER 2

Reversible Programming Languages

The following chapter contains a survey of reversible instruction sets and reversible imperative programming languages. Given that OOP is an approach for naturally organizing imperative code, it is clear that such languages are of special interest when designing a reversible OOP language. Indeed, the design of our reversible OOP language draws heavily from the design of the languages and instruction sets presented in this section.

2.1 Janus

The reversible programming language *Janus* (named after the two-faced Greco-Roman god of beginnings and endings) was created by Cristopher Lutz and Howard Derby for a class at Caltech in 1982 [30]. It was later rediscovered and formalized in [49] and some modifications were suggested in [46] - the following section deals with this modified version of the language.

Janus Grammar

$prog$	$::=$	$p_{main}\ p^*$	(program)
t	$::=$	$\textbf{int} \mid \textbf{stack}$	(data type)
p_{main}	$::=$	$\textbf{procedure main}\ \texttt{()}\ (\textbf{int}\ x(\texttt{[}\overline{n}\texttt{]})^? \mid \textbf{stack}\ x)^*\ s$	(main procedure)
p	$::=$	$\textbf{procedure}\ q\,\texttt{(}t\ x,\ \ldots,\ t\ x\texttt{)}\ s$	(procedure definition)
s	$::=$	$x\ \odot\texttt{=}\ e \mid x\texttt{[}e\texttt{]}\ \odot\texttt{=}\ e$	(assignment)
	$\mid$	$\textbf{if}\ e\ \textbf{then}\ s\ \textbf{else}\ s\ \textbf{fi}\ e$	(conditional)
	$\mid$	$\textbf{from}\ e\ \textbf{do}\ s\ \textbf{loop}\ s\ \textbf{until}\ e$	(loop)
	$\mid$	$\textbf{push}\,\texttt{(}x,\ x\texttt{)} \mid \textbf{pop}\,\texttt{(}x,\ x\texttt{)}$	(stack modification)
	$\mid$	$\textbf{local}\ t\ x\ \texttt{=}\ e\quad s\quad \textbf{delocal}\ t\ x\ \texttt{=}\ e$	(local variable block)
	$\mid$	$\textbf{call}\ q\,\texttt{(}x,\ \ldots,\ x\texttt{)} \mid \textbf{uncall}\ q\,\texttt{(}x,\ \ldots,\ x\texttt{)}$	(procedure invocation)
	$\mid$	$\textbf{skip} \mid s\ s$	(statement sequence)
e	$::=$	$\overline{n} \mid x \mid x\texttt{[}e\texttt{]} \mid e\ \otimes\ e \mid \textbf{empty}\,\texttt{(}x\texttt{)} \mid \textbf{top}\,\texttt{(}x\texttt{)} \mid \textbf{nil}$	(expression)
$\odot$	$::=$	$\texttt{+} \mid \texttt{-} \mid \texttt{^}$	(operator)
$\otimes$	$::=$	$\odot \mid \texttt{*} \mid \texttt{/} \mid \texttt{\%} \mid \texttt{\&} \mid \texttt{\textbar} \mid \texttt{\&\&} \mid \texttt{\textbar\textbar} \mid \texttt{<} \mid \texttt{>} \mid \texttt{=} \mid \texttt{!=} \mid \texttt{<=} \mid \texttt{>=}$	(operator)

Figure 2.1: EBNF grammar for Janus [46]

Janus is a procedural language with locally-invertible program statements and direct access to inverse semantics. There are 3 data types in Janus: plain integers, fixed-size integer arrays and dynamically-sized integer stacks. Integer variables and integer stacks may be declared locally or statically in the global scope, while integer arrays can only be declared statically.

A Janus program consists of a main procedure followed by any number of secondary procedures. The main procedure acts as the starting point of the program and is preceded by declarations of static variables, which serve as the program output upon termination. Secondary procedures may specify parameters which are passed to the callee by reference. Procedures can not return a value but may use output parameters to achieve similar effects. Procedure bodies are made up of one or more program statements, which may be one of several different forms.

A *conditional statement* in Janus has both a branch condition and an exit assertion, both of which are expressions. The branch condition determines which branch of the conditional is executed, while the exit assertion is used to reversibly join the two paths of computation. If the branch condition evaluates to true, the **then**-branch is executed upon which the exit assertion should also evaluate to true. If the branch condition evaluates to false, the **else**-branch is executed after which the exit assertion should evaluate to false. If the exit assertion does not match the branch condition, the statement is undefined. See Figure 1.1 in Chapter 1 for a flowchart illustrating the mechanics of reversible conditionals.

A *loop statement* has both an entry assertion and an exit condition, both of which are expressions. Initially, the entry assertion must evaluate to true after which the **do**-statement is executed. If the exit condition is then true, the loop terminates, otherwise the **loop**-statement is executed upon which the entry assertion must now evaluate to false. When executed in reverse, the exit condition serves as the entry assertion and vice versa. Figure 2.2 shows a flowchart illustrating the mechanics of reversible loops.

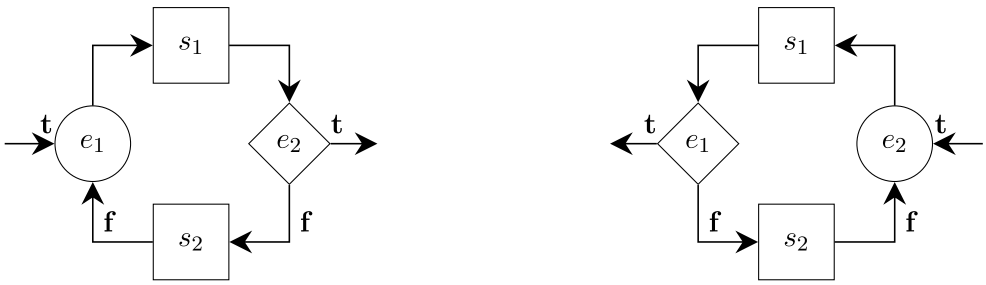

Figure 2.2: Flowcharts of a reversible loop statement, in both directions of execution [47, 49]

The *stack modification statements*, **push** and **pop** are used to manipulate integer stacks in the usual fashion, the only difference being that pushing a variable onto a stack zero-clears the contents of the variable while popping a value into a variable presupposes that the variable is zero-cleared. This means that push and pop are inversions of each other.

A *reversible variable update* in Janus works by updating a variable in the current scope in such a way that the original store remains reachable by subsequent uncomputation. Only updates that are injective in their first argument and have precisely defined inverses are allowed and it is a requirement that the expression being updated with does not in any way depend on the value of the variable being updated (to avoid loss of information). To ensure such an update cannot occur, it is not allowed for the variable identifier on the left side of the update to occur anywhere on the right-hand side. This also mandates a further restriction: no two identifiers may refer to the same location in memory in the same scope (a situation known as aliasing) as this would otherwise be a way to circumvent the aforementioned requirement.

The *local variable block*, denoted by the **local**/**delocal** statement, defines a block scope wherein a new local variable is declared and initialized. After the block statement has executed with the new variable in scope, the variable is cleared by means of an exit expression which must evaluate to the value of the variable (otherwise the statement is undefined as it becomes impossible to reversibly clear the memory occupied by the variable).

The *call* and *uncall* statements are used to invoke procedures in the forwards and backwards direction. Arguments are passed by reference and it is a requirement that the same variable is not passed twice in the same procedure invocation to avoid aliasing of the arguments.

An *expression* in Janus can either be a numeric literal, a variable identifier, an array element, a binary expression or a stack expression. Janus uses 0 to represent the boolean value false, and non-zero to represent true.

```
procedure main()
    int n
    int root
    n += 25
    call root(n, root)

procedure root(int n, int roo) //root := floor (sqrt(n))
    local int bit = 1
        from bit = 1
        do skip
        loop call doublebit(bit)
        until (bit * bit) > n
        from (bit * bit) > n
        do uncall doublebit(bit)
           if ((root + bit) * (root + bit)) <= n
           then root += bit
           fi (root / bit) % 2 != 0
        loop skip
        until bit=1
    delocal int bit = 1
    n -= root * root

procedure doublebit(int bit)
    local int z = bit
    bit += z
    delocal int z = bit / 2
```

Figure 2.3: Example Janus program for computing $\lfloor\sqrt{n}\rfloor$ from [30]

Janus is known to be r-Turing complete as it is able to simulate any reversible Turing machine [46]. An efficient and clean translation from Janus to PISA (See Section 2.4) was presented in [2] and a partial evaluator for Janus was presented in [32]. The reversible control flow constructs used by Janus was explored in detail in [47].

2.2 Unstructured Janus

An unstructured version of Janus was used in [32] as an intermediate language for polyvariant partial evaluation. Specialization of a program written in an imperative programming language is usually accomplished with polyvariant partial evaluation, which is most suitable for programs with unstructured control flow.

A precursor to the unstructured version of Janus was first presented in [47] as a reversible flowchart language. Mogensen suggests a simple transformation from Janus to a modified version of this flowchart language, before the partial evaluation is applied.

The language uses *paired jumps* to organize the unstructured control flow in a reversible manner: Every jump statement must jump to a *from*-statement which uniquely identifies the origin of the jump, thus reversibly joining the control flow. The language also supports conditional jumps which must then target a conditional from-statement, again for the purpose of reversibly joining the two paths of computation.

Unstructured Janus programs are arranged into a series of basic blocks, each consisting of a label, a from statement, a series of reversible assignments and finally a jump. The first block always starts with a *start* statement and the end of the program is marked with a *return* statement. The language is locally invertible, just like its structured counterpart.

The structured reversible program theorem, by Yokoyama et al. in [47] proves that such a language is computationally equivalent to its structured counterpart. Figure 2.4 shows a program for multiplying two odd integers using unstructured Janus.

```
start:
goto f_2

if 0 = prod from f_2 a_2:
if odd(a) goto t1_3 e1_3

t1_3:
prod += b; t += a / 2; a -= t + 1; t -= a
goto t2_3

e1_3:
t += a / 2; a -= t; t -= a
goto e2_3

if !(prod < b) from t2_3 e2_3:
if a = 0 goto f_11 l_2

l_2:
v += b; b += v; v -= b/2
goto a_2

if prod < b + b from f_11 a_11:
v += b / 2; b -= v; v -= b
if odd(b) goto u_11 a_11

u_11:
return
```

Figure 2.4: Unstructured Janus program computing the product of two odd numbers, from [32]

2.3 R

The reversible programming language *R* (not to be confused with the statistical programming language of the same name) is an imperative reversible language developed at MIT in 1997 [17]. The syntax of R is a blend of LISP and C - with programs arranged as nested S-expressions but with support for C-like arrays and pointer arithmetics. R is a compiled language, with the only available compiler targeting the Pendulum reversible instruction set (see Section 2.4).

R Grammar

$$
\begin{array}{rcll}
prog & ::= & s^* & \text{(program)} \\
s & ::= & \textbf{(defmain}\ progname\ s^*\textbf{)} & \text{(main routine)} \\
 & | & \textbf{(defsub}\ subname\ \textbf{(}name^*\textbf{)}\ s^*\textbf{)} & \text{(subroutine)} \\
 & | & \textbf{(defword}\ name\ \overline{n}\textbf{)} & \text{(global variable)} \\
 & | & \textbf{(defarray}\ name\ \overline{n}^*\textbf{)} & \text{(global array)} \\
 & | & \textbf{(call}\ subname\ e^*\textbf{)} & \text{(call subroutine)} \\
 & | & \textbf{(rcall}\ subname\ e^*\textbf{)} & \text{(reverse-call subroutine)} \\
 & | & \textbf{(if}\ e\ \textbf{then}\ s^*\textbf{)} & \text{(conditional)} \\
 & | & \textbf{(for}\ name\ \texttt{=}\ e\ \textbf{to}\ e\ s^*\textbf{)} & \text{(loop)} \\
 & | & \textbf{(let}\ \textbf{(}name\ \texttt{<-}\ e\textbf{)}\ s^*\textbf{)} & \text{(variable binding)} \\
 & | & \textbf{(printword}\ e\textbf{)}\ |\ \textbf{(println)} & \text{(output)} \\
 & | & \textbf{(}loc\ \texttt{++}\textbf{)}\ |\ \textbf{(-}\ loc\textbf{)} & \text{(increment/negate)} \\
 & | & \textbf{(}loc\ \texttt{<->}\ loc\textbf{)}\ |\ \textbf{(}loc\ \odot\ e\textbf{)} & \text{(swap/update)} \\
loc & ::= & name\ |\ \textbf{(}\star\ e\textbf{)}\ |\ \textbf{(}e\ \_\ e\textbf{)} & \text{(location)} \\
e & ::= & loc\ |\ \textbf{(}e\ \otimes\ e\textbf{)}\ |\ \overline{n} & \text{(expression)} \\
\odot & ::= & \texttt{+=}\ |\ \texttt{-=}\ |\ \texttt{\^{}=}\ |\ \texttt{<=<}\ |\ \texttt{>=>} & \text{(update operator)} \\
\otimes & ::= & \texttt{+}\ |\ \texttt{-}\ |\ \texttt{\&}\ |\ \texttt{<<}\ |\ \texttt{>>}\ |\ \texttt{*/} & \text{(expression operator)} \\
 & | & \texttt{=}\ |\ \texttt{<}\ |\ \texttt{>}\ |\ \texttt{<=}\ |\ \texttt{>=}\ |\ \texttt{!=} & \text{(relational operator)}
\end{array}
$$

Figure 2.5: EBNF grammar for R, based on the rules presented in [18, Appdx. C]

Figure 2.5 shows a formal grammar describing the syntax rules of R. An R program consists of any number of statements, but should contain exactly one main routine, defined with the **defmain** statement. The main routine may invoke subroutines which are defined with the **defsub** statement. Also a program may make use of globally scoped variables and arrays, defined with the **defword** and **defarray** statement. These four types of statements may appear anywhere in a program, but only have an actual effect when appearing as top-level statements.

The **call** and **rcall** statements are used to invoke a subroutine in either direction of execution, and correspond to the **call** and **uncall** statements of Janus. Arguments are passed by reference, but only parameters bound to variables or memory references may be modified by the callee. Parameters bound to an expression or a constant should retain their value throughout the body of the subroutine to avoid undefined or irreversible behaviour.

The **if** statement is used for conditional execution. It is a requirement that the value of the conditional expression is the same before and after the conditional statement is executed, otherwise undefined or irreversible behaviour may occur. This limitation guarantees that the condition can be used to determine which branch of computation to follow in either direction of execution. It is equivalent to a Janus conditional with the same expression used as entry condition and exit assertion. A version with an else-branch was also proposed but never implemented in the compiler.

The **for** statement is used for definite iteration. The iteration variable is given an initial value matching the first expression and is then incremented upon each iteration until the termination value is reached. Both expressions must have the same value before and after the loop is executed to guarantee correct behaviour in both directions of execution. The for-loop may also be used for indefinite iteration by modifying the value of the iteration variable in the loop body - which allows the number of iterations to be determined dynamically as the loop proceeds.

A **let** statement creates a new local variable, limited in scope to the statements within the let-block. The local variable is initialized to the value of the let-expression and after the block statements have been executed the value of the let-expression should still match the value of the local variable (although they are not required to have the same value as they did initially). This is a requirement for the program to be able to reversibly zero-clear the local variable before it is reclaimed by the system - it is functionally equivalent to a Janus **local**/**delocal** block where the entry and exit expressions are the same.

The **printword** and **println** statements are used for program output. A **printword** statement will output the value of the given expression, while the **println** statement outputs a single line-break delimiter.

```
(defsub fib (x1 x2 n)
    (if (n = 0) then
        (x1 += 1)
        (x2 += 1) )

    (if (n != 0) then
        (n -= 1)
        (call fib x1 x2 n)
        (x1 += x2)
        (x1 <-> x2)
        (n += 1) ) ) ; Restore value of n for conditional

(defword x1 0)
(defword x2 0)
(defword n 4)
(defmain fibprog (call fib x1 x2 n))
```

Figure 2.6: Example R program for computing the n^{th} Fibonacci pair, adapted from example program in [46]

Memory modification in R is done by the increment, negate, swap and update statements. These statements operate on *memory locations* which may be represented either by variable identifiers, by expressions referring to memory addresses or by expressions referring to specific elements of an array (with an underscore representing array indexing). The update statements are subject to the same restrictions as in Janus, namely that the value of the expressions being updated with must not at the same time depend on the memory location being updated. This is necessary to ensure that the update does not erase information. The **<=<** and **>=>** operators represent *arithmetic* left and right rotations.

Expressions in R can be either memory locations, numeric literals or binary operations. The supported operators are numerical addition, subtraction and bitwise conjunction (**+**, **-**, **&**), logical left and right shifts (**<<**, **>>**), relational operators[5] (**=**, **<**, **<=**, **!=**, **>**, **>=**) and fractional product (***/**), which is the product of a signed integer and a fixed-precision fraction between −1 and 1.

[5] As described in [18, Appdx. C], the R compiler only supports the use of relational operators in conditional expressions but this can be considered a limitation of the implementation, not of the language.

2.4 PISA

The Pendulum microprocessor and the Pendulum ISA (PISA) is a logically reversible computer architecture created at MIT by Carlin James Vieri [42, 43, 18, 44]. The Pendulum architecture resembles a mix of PDP-8 and RISC and it was the first reversible programmable processor and instruction set.

PISA is a MIPS-like assembly language that has gone through several incarnations. The version presented in this section is known as the *PISA Assembly Language* (PAL) and it is compatible with the Pendulum virtual machine, PendVM [16].

PISA Grammar

$$
\begin{array}{rcll}
prog & ::= & ((l\ \texttt{:})^{?}\ i)^{+} & \text{(program)} \\
i & ::= & \texttt{ADD}\ r\ r \mid \texttt{ADDI}\ r\ c \mid \texttt{ANDX}\ r\ r\ r \mid \texttt{ANDIX}\ r\ r\ c & \text{(instruction)} \\
 & \mid & \texttt{NORX}\ r\ r\ r \mid \texttt{NEG}\ r \mid \texttt{ORX}\ r\ r\ r \mid \texttt{ORIX}\ r\ r\ r \mid \texttt{RL}\ r\ c & \\
 & \mid & \texttt{RLV}\ r\ r \mid \texttt{RR}\ r\ c \mid \texttt{RRV}\ r\ r \mid \texttt{SLLX}\ r\ r\ c \mid \texttt{SLLVX}\ r\ r\ r & \\
 & \mid & \texttt{SRAX}\ r\ r\ c \mid \texttt{SRAVX}\ r\ r\ r \mid \texttt{SRLX}\ r\ r\ c \mid \texttt{SRLVX}\ r\ r\ r & \\
 & \mid & \texttt{SUB}\ r\ r \mid \texttt{XOR}\ r\ r \mid \texttt{XORI}\ r\ c \mid \texttt{BEQ}\ r\ r\ l \mid \texttt{BGEZ}\ r\ l & \\
 & \mid & \texttt{BGTZ}\ r\ l \mid \texttt{BLEZ}\ r\ l \mid \texttt{BLTZ}\ r\ l \mid \texttt{BNE}\ r\ r\ l \mid \texttt{BRA}\ l & \\
 & \mid & \texttt{EXCH}\ r\ r \mid \texttt{SWAPBR}\ r \mid \texttt{RBRA}\ l \mid \texttt{START} \mid \texttt{FINISH} & \\
 & \mid & \texttt{DATA}\ c & \\
c & ::= & \cdots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \cdots & \text{(immediate)}
\end{array}
$$

Syntax Domains

$$
\begin{array}{rr}
prog \in \text{Programs} & i \in \text{Instructions} \\
r \in \text{Registers} & l \in \text{Labels}
\end{array}
$$

Figure 2.7: Syntax domains and EBNF grammar for PISA

In a conventional processor, the rules governing control flow are quite simple: After each instruction, add 1 to the program counter. In case of a jump, set the program counter to the address of the label being jumped to. In a reversible processor like Pendulum, these rules are much more involved since simply overwriting the contents of the program counter would constitute a loss of information which break reversibility.

The Pendulum processor uses three special-purpose registers for control flow logic:

1. The *program counter* (PC) for storing the address of the current instruction
2. The *branch register* (BR) for storing jump offsets
3. The *direction bit* (DIR) for keeping track of the execution direction

After each instruction, if the branch register is zero, we simply add the direction bit to the program counter. The direction bit is either 1 or -1 depending on the direction of execution so this corresponds to regular stepwise execution in either direction.

If the branch register is not zero, the product of the branch register and the direction bit is added to the program counter. When a PISA program is assembled to machine code, the target labels of each of the jump instructions are replaced with *relative offsets*. When a jump instruction is then executed, the relative offset is placed in the branch register and when the PC is updated, control flow jumps to the target label. Using *paired branches*, the PISA programmer can clear the branch register after a jump by always jumping only to jump instruction that points back to the original jump. This has the effect of adding the negation of the relative offset to the branch register, thereby zero-clearing it.

Aside from the usual conditional jump instructions (Branch-if-equal, branch-if-zero et cetera), PISA also contains the unconditional jump instruction **BRA** and the unconditional reverse-jump instruction **RBRA** which also flips the direction bit and can therefore be used to implement uncall or reverse-call functionality. When the direction bit is -1, the instructions are inverted so that addition becomes subtraction, left-rotation becomes right-rotation and so on. See Figure 2.8 for a table illustrating how PISA instructions are inverted when the execution direction is flipped.

i	i^{-1}
ADD r_1 r_2	**SUB** r_1 r_2
SUB r_1 r_2	**ADD** r_1 r_2
ADDI r c	**ADDI** r $-c$
RL r c	**RR** r c
RR r c	**RL** r c
RLV r_1 r_2	**RRV** r_1 r_2
RRV r_1 r_2	**RLV** r_1 r_2

Figure 2.8: Inversion rules for PISA instructions, all other instructions are self-inverse

PISA also has the **SWAPBR** instruction which affords direct control over the contents of the branch register (but crucially, not the PC directly) and makes it possible to implement dynamic jumps such as switch/case structures or function pointers. **SWAPBR** can also be used to allow incoming jumps from more than one location.

The special instructions **START** and **FINISH** are used to mark the beginning and end of a PISA program while the memory exchange instruction **EXCH** provides simultaneous reversible memory-read and memory-write functionality. The **DATA** instruction stores an immediate value in the corresponding memory cell and can be used to mark the static storage space of a program.

The remaining instructions are similar to those of other RISC processors and implement various register update functionality (bitwise-AND, bitwise-XOR and so on) albeit in a reversible manner. For example, bitwise-AND is performed with the **ANDX** instruction which XORs the resulting value into a third register to ensure reversibility.

Figure 2.9 shows an example PISA program. The design of the Pendulum control flow logic is based in part on work by Cezzar [15] and Hall [24]. A complete formalization of the PISA language and the Pendulum machine was given in [4] and a translation from Janus to PISA was presented in [2]. PISA is also the target language of the R compiler [17, 18] and in this thesis we use PISA as the target language for the translation presented in Chapter 4.

```
;; Call Fall:
10: ADDI $4 1000   ;h += 1000
11: ADDI $5 5      ;tend += 5
12: BRA 16         ;call

;; Uncall Fall:
18: ADDI $6 40     ;v += 40
19: ADDI $7 4
20: ADDI $5 4
21: RBRA 7         ;uncall

;; Subroutine Fall:
27: BRA 9
28: SWAPBR $8      ;br <=> rtn
29: NEG $8         ;rtn=-rtn
30: BGTZ $7 5      ;t > 0?
31: ADDI $6 10     ;v += 10
32: ADDI $4 5      ;h += 5
33: SUB $4 $6      ;h -= v
34: ADDI $7 1      ;t += 1
35: BNE $7 $5 -5   ;t != tend?
36: BRA -9
```

Figure 2.9: Example PISA program for simulating free-falling objects, from [4]

2.5 BobISA

The reversible computer architecture Bob and its instruction set BobISA were created at the University of Copenhagen by Thomsen et al. [41, 13]. Bob is a *Harvard architecture* which is characterized by having separate storage for instructions and data[6].

BobISA was designed to be sufficiently expressive to serve as the target for high-level compilers while still being relatively straightforward to implement in hardware. BobISA consists of 17 instructions and is known to be *r-Turing complete* [41].

BobISA Grammar

$$
\begin{array}{rcll}
\mathit{prog} & ::= & i^{+} & \text{(program)} \\
i & ::= & \texttt{ADD}\ r\ r \mid \texttt{SUB}\ r\ r \mid \texttt{ADD1}\ r \mid \texttt{SUB1}\ r & \text{(instruction)} \\
 & \mid & \texttt{NEG}\ r \mid \texttt{XOR}\ r\ r \mid \texttt{XORI}\ r\ c \mid \texttt{MUL2}\ r & \\
 & \mid & \texttt{DIV2}\ r \mid \texttt{BGEZ}\ r\ o \mid \texttt{BLZ}\ r\ o \mid \texttt{BEVN}\ r\ o & \\
 & \mid & \texttt{BODD}\ r\ o \mid \texttt{BRA}\ o \mid \texttt{SWBR}\ r \mid \texttt{RSWB}\ r & \\
 & \mid & \texttt{EXCH}\ r\ r & \\
c & ::= & \cdots \mid \texttt{-1} \mid \texttt{0} \mid \texttt{1} \mid \cdots & \text{(immediate)} \\
o & ::= & \texttt{-128} \mid \cdots \mid \texttt{0} \mid \cdots \mid \texttt{127} & \text{(offset)}
\end{array}
$$

Figure 2.10: EBNF grammar for BobISA

[6] As opposed to a *von Neumann architecture* which does not distinguish between program instructions and data.

The control flow logic of Bob is identical to that of PISA, with a few caveats:

– There are only 8-bits to store jump offsets, so a plain jump cannot be of more than 127 lines.

– The **SWBR** instruction which is similar to the **SWAPBR** instruction of PISA, can be used for jump offsets longer than 127.

– BobISA also has the **RSWB** instruction which flips the direction bit in addition to swapping out the branch register.

While the jump targets in the BobISA grammar in Figure 2.10 are represented in terms of offsets, a construction similar to that of PISA could be used, where jumps are specified with instruction labels that are then converted to offsets during program assembly.

The remaining instructions are self-explanatory and most of them have PISA equivalents, with the exception of **MUL2** and **DIV2**. These instructions operate on 4-bit two's-complement numbers and will either double or halve the value of a given register. To avoid overflow and division of odd numbers, these instructions are only well-defined for a subset of the representable values as illustrated in Figure 2.11. Input values outside of this subset are mapped to output in such a way that reversibility is preserved. Figure 2.11 also shows the inversion rules for those BobISA instructions that are not self-inverse. Like PISA, the inverse semantics of each instruction is used when the processor is running in reverse.

x	MUL2(x)
-4	-8
-3	-6
-2	-4
-1	-2
0	0
1	2
2	4
3	6

x	DIV2(x)
-8	-4
-6	-3
-4	-2
-2	-1
0	0
2	1
4	2
6	3

i	i^{-1}
ADD r_1 r_2	**SUB** r_1 r_2
SUB r_1 r_2	**ADD** r_1 r_2
ADD1 r	**SUB1** r
SUB1 r	**ADD1** r
MUL2 r	**DIV2** r
DIV2 r	**MUL2** r

Figure 2.11: Tables showing well-defined inputs and outputs for **MUL2** and **DIV2** instructions as well as the inversion rules for BobISA instructions

A complete low-level design with schematics and HDL programs was developed for the Bob architecture. Only 473 reversible gates are required to construct a Bob processor, totalling only 6328 transistors [41]. A translation from the reversible functional language RFUN to BobISA was presented in [25].

CHAPTER 3

The ROOPL Language

The *Reversible Object-Oriented Programming Language* (ROOPL) is, to our knowledge, the first reversible programming language with built-in support for object-oriented programming and user-defined types. ROOPL is statically typed and supports inheritance, encapsulation and subtype-polymorphism via dynamic dispatch. ROOPL is purely reversible, in the sense that no computation history is required for backwards execution. Rather, each component of a ROOPL program is locally invertible at no extra cost to program size. The basic components of the language, such as control flow structures and variable updates draw heavy inspiration from the reversible imperative language Janus [49, 46], however the overall structure of a ROOPL program differs vastly from that of a Janus program.

```
class Program
    int result
    int n

    method main()
        n ^= 4

        construct Fib f
            //Compute-copy-uncompute
            call f::fib(n)
            call f::get(result)
            uncall f::fib(n)
        destruct f

class Fib
    int x1
    int x2

    method fib(int n)
        if n = 0 then
            x1 ^= 1
            x2 ^= 1
        else
            n -= 1
            call fib(n)
            x1 += x2
            x1 <=> x2
        fi x1 = x2

    method get(int out)
        out ^= x2
```

Figure 3.1: Example ROOPL program computing the n^{th} Fibonacci pair, adapted from example program in [46]

3.1 Syntax

A ROOPL program consists of one or more class definitions, each of which may contain any number of member variables and one or more methods. Each program should contain exactly one class with a nullary method named *main* which acts as the program entry point. This class will be instantiated when the program starts, and the fields of this object will act as the output of the program in much the same way that the variable store acts as the output of a Janus program.

ROOPL Grammar

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

Syntax Domains

$prog \in$ Programs	$s \in$ Statements	$n \in$ Constants
$cl \in$ Classes	$e \in$ Expressions	$x \in$ VarIDs
$t \in$ Types	$\odot \in$ ModOps	$q \in$ MethodIDs
$m \in$ Methods	$\otimes \in$ Operators	$c \in$ ClassIDs

Figure 3.2: Syntax domains and EBNF grammar for ROOPL

A *class definition* consists of the keyword **class** followed by the class name. If the class is a subclass of another, it is specified with the keyword **inherits** followed by the name of the base class. Next, any number of class fields are declared, each of which may be either integers or references to other objects (these are the only types in ROOPL). Finally, each class definition contains at least one method which is defined with the keyword **method** followed by the method name, a comma-separated list of parameters and the method body. A class must have at least one method, as method calls are the only mechanism of interfacing with an object.

A *reversible assignment* in ROOPL uses the same C-like syntax as a reversible assignment in Janus. A variable can be updated either through addition (**+=**), subtraction (**-=**) or bitwise XOR (**^=**). It is only possible to reversibly update the value of some variable x by some expression e in this manner, if the value of e does not depend, in any way, on the value of x. We can enforce this limitation by explicitly disallowing any occurrences of the identifier x in the expression e, but this is only sufficient if we can also guarantee that no other identifiers refer to the same location in memory as x (See Section 3.2).

A *variable swap* denoted by the token **<=>** swaps the value of two integer variables or two object references. This was supported in Janus as syntactic sugar for the statement sequence:

$$x_1 \;\texttt{\^{}=}\; x_2 \quad x_2 \;\texttt{\^{}=}\; x_1 \quad x_1 \;\texttt{\^{}=}\; x_2$$

which achieves the same effect as x_1 **<=>** x_2, given that x_1 and x_2 are both integers [49]. In ROOPL, we might wish to swap two object references, for which the XOR operation is undefined, so the swap statement has been made explicit in the language.

Loops and conditional statements are syntactically (and semantically) identical to Janus loops and Janus conditionals. The use of assertions at control flow join points ensure that we can execute these statements in reverse, in a deterministic manner.

An *object block* denotes the instantiation and lifetime of a ROOPL object. The statement consist of the keyword **construct** followed by a class name and a variable identifier. Then follows the block statement s within which the newly created object will be accessible, and finally the keyword **destruct** followed by the object identifier signifies the end of the object block.

A *method invocation* may refer either to a local method or to a method in another object - both variants can be both called and uncalled. An *expression* may be either a constant, a variable, the special value **nil** or a binary expression.

3.2 Argument Aliasing

To avoid situations where multiple identifiers refer to the same memory location within the same scope, known as *aliasing*, we must place some restrictions on method invocations. One source of aliasing occurs when the same identifier is passed to more than one parameter of a method:

```
method foo(int a)
    call bar(a, a)

method bar(int x, int y)
    x -= y //Irreversible update!
```

Such situations are easily avoided by prohibiting method calls with the same identifier passed to more than one parameter, which is the same approach used in Janus. Another, similar source of aliasing is when a field of an object is passed to a parameter of a method of that same object:

```
class Object
    int a

    method main()
        a += 5
        call foo(a)

    method foo(int b)
        a -= b //Irreversible update!
```

In this case we can disallow object fields as arguments to local methods, and since the object field is already in scope in the callee, there is little point in also passing it as an argument. ROOPL uses two separate statements to distinguish between local and non-local method invocations, so it is a simple matter of prohibiting object fields as arguments to local call statements.

Finally, we must make sure that non-local method invocations are indeed non-local, which might not be the case if an object has obtained a reference to itself. We can avoid such a situation by disallowing non-local method calls to some object x which also passes x as an argument.

3.3 Parameter Passing Schemes

The most common parameter passing modes and their implications for reversible languages were briefly discussed in [46] while a more in-depth investigation was performed in [35]. The common call-by-value scheme is generally not suitable for reversible languages since the values accumulated in the function parameters after a function has executed, must be disposed of somehow when the function returns, which would result in a loss of information. It is also difficult to reconstruct multiple arguments given only a single return value, which is the main reason that Janus uses the call-by-reference strategy. With this approach, a function can simply store results in the parameter variables and sidestep traditional single return values altogether. The values in the parameters are handed back to the caller instead of being erased.

Another approach, which is likely simpler to implement in practice, is call-by-value-result presented in [35]. Call-by-value-result involves swapping the function arguments into local variables in the called procedure, and copying them back after the body has been executed. This approach hinges upon the callee not being able to alter the argument variables other than through the local copies, which can only occur if more than one identifier, referring to the same argument, is in scope.

Call-by-reference and call-by-value-result are semantically equivalent parameter passing schemes in the absence of aliasing [35], and therefore either scheme can be used. The operational semantics of ROOPL (Section 3.8) uses call-by-reference.

3.4 Object Model

ROOPL is a class-based programming language, it is based on the notion of *classes* that serve as blueprints for specific objects or class instances. Alternatively, a language may allow objects to serve as blueprints for other objects - this is known as prototype-based programming. Prototype-based programming is dominated by dynamically-typed[7], interpreted languages (examples include JavaScript and Lua). While there is no immediate reason to believe that dynamic typing is not a feasible strategy for a reversible programming language, it is as of yet an unexplored notion.

Some OOP languages have very intricate object models - Java includes support for access modifiers, static methods and fields, final classes (that may not be subclassed), final methods (that cannot be overridden in a subclass) and both implementation inheritance and interface inheritance. C++ supports friend classes, virtual and non-virtual methods, abstract methods, private inheritance and multiple inheritance.

These features facilitate the creation of very rich models and interfaces but they are less interesting from our perspective: implementation on a reversible machine. The rules imposed

[7]For an example of a statically typed language with a prototype-based object model, see Omega [11].

by these features on the classes of a program are generally enforced at compile-time - wholly independently from the target architecture and its limitations (with the exception of dynamic dispatch which has to be handled at runtime).

The object model of ROOPL is therefore very simple compared to these languages - introducing access modifiers or static methods to ROOPL is possible but would be a meaningless venture as the implementation of such features would be identical for an irreversible language. The ROOPL object model is based on the following key points:

- All class fields are protected, they may be accessed only from within class methods and subclass methods
- All class methods are public, they may be accessed from other objects
- All class methods are virtual and may be overridden in a subclass (but only by a method with the same type signature, there is no support for method overloading)
- A class may inherit only from a single base class (*single inheritance*)
- Any method that takes an object reference of some type τ also works when passed a reference of type τ' if τ' is a subclass of τ (*subtype polymorphism*)
- Local method calls are statically dispatched (*closed recursion*), only method calls to other objects are dynamically dispatched

Note that the single inheritance object model of ROOPL still allows for inheritance hierarchies of arbitrary depth (known as *multi-level* inheritance).

3.5 Object Instantiation

In irreversible OOP languages, object instantiation is typically accomplished in two or three general steps:

1. A suitable amount of memory is reserved for the object
2. All fields are initialized to some neutral value
3. The class constructor is executed, establishing the class invariants of the object

When the program (or the garbage collector) deallocates the object, the memory is (typically) simply marked as unused. Any leftover values from the internal state of the object will be irreversibly overwritten if/when another object is initialized in the same part of memory later on. In a reversible language we cannot clear leftover values in memory like this as that would constitute a loss of information.

Instead we require unused memory to already be zero-cleared at the time of object creation, so the fields of each new object have a known initial value. The only way to achieve this reversibly is to uncompute all the state accumulated inside an object before it is deallocated, returning all fields to the value zero. This cannot be done automatically so this responsibility lies with the program itself.

```
class Object
    int data

    method add5()
        data += 5

    method get(int out)
        out ^= data

class Program
    int result

    method main()
        construct Object obj      //Allocate object
            call obj::add5()      //Perform computation
            call obj::get(result) //Fetch result
            uncall obj::add5()    //Uncompute internal state
        destruct obj              //Reversibly deallocate object
```

Figure 3.3: Simple example program illustrating the mechanics of an object block

A ROOPL object exists only within a **construct**/**destruct** block. Consider the statement:

$$\textbf{construct}\ c\ x \quad s \quad \textbf{destruct}\ x$$

the mechanics of such a statement are as follows:

1. Memory for an object of class c is allocated. All fields are automatically zero-initialized by virtue of residing in already zero-cleared memory.
2. The block statement s is executed, with the name x representing a reference to the newly allocated object.
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.

If the fields of the object are not zero-cleared after the block statement, it becomes impossible for the system to reversibly reclaim the memory occupied by the object. It is up to the program to maintain this invariant.

3.6 Inheritance Semantics

Before we can define the type system and formal semantics of the language, we need a precise definition of the object model as described in Section 3.4 and Section 3.5. Given the *dynamic type* of some object, we wish to determine the class fields and class methods of the object such that inherited fields and methods are included, unless overridden by the derived class.

$$\mathrm{gen}(\overbrace{cl_1,\ \ldots,\ cl_n}^{p}) \quad = \quad \overbrace{\left[\ \alpha(cl_1)\ \mapsto\ \beta(cl_1),\ \ldots,\ \alpha(cl_n)\ \mapsto\ \beta(cl_n)\ \right]}^{\Gamma}$$

$$\alpha\big(\ \mathbf{class}\ c\ \cdots\ \big)\ =\ c \qquad\qquad \beta\big(\ cl\ \big)\ =\ \big(\ \mathrm{fields}(cl),\ \mathrm{methods}(cl)\ \big)$$

Figure 3.4: Definition of function *gen*, for constructing the class map of a given program

To this end, we define the *class map* Γ of a program p as a finite map from class identifiers (type names) to tuples of the method and field declarations of that class. The application of a class map Γ to some class identifier cl is denoted $\Gamma(cl)$. Figure 3.4 shows the definition of function *gen*, which is used to construct the class map of a program.

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

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

$$A\ \uplus\ B\ \stackrel{\mathbf{def}}{=}\ A\ \cup\ \Big\{\ m\ \in\ B\ \Big|\ \nexists\, m'\ \big(\ \zeta(m')\ =\ \zeta(m)\ \wedge\ m'\ \in\ A\ \big)\ \Big\}$$

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

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

Figure 3.5: Definition of functions for modelling class inheritance

Figure 3.5 shows the definition of the functions *fields* and *methods* which determines the class fields and class methods for a given class. The set operation $\uplus$ implements method overriding by dropping methods from the base class if a method with the same name exists in the derived class.

3.7 Type System

The type system of ROOPL is specified by the syntax-directed typing rules shown in the following sections. There are three main type judgments covering expressions, statements and whole ROOPL programs. The inference rules are presented in the style of Winskell [45] and are arranged in such a way that a complete type derivation can only be constructed for well-typed programs. The next section establishes the notation and presents auxiliary definitions.

3.7.1 Preliminaries

The set of types in ROOPL is given by the grammar:

$$\tau \ ::= \ \textbf{int} \ \big| \ c \ \in \ \text{ClassIDs}$$

A *type environment* Π is a finite map from variable identifiers to types. The application of a type environment Π to some identifier x is denoted by $\Pi(x)$. Update $\Pi' = \Pi[x \mapsto \tau]$ defines a type environment Π' s.t. $\Pi'(x) = \tau$ and $\Pi'(y) = \Pi(y)$ if $y \neq x$. The empty type environment is written $[\,]$. The function $vars : Expressions \rightarrow VarIDs$, is given by the following recursive definition:

$$\begin{aligned}
\text{vars}(\overline{n}) &= \varnothing \\
\text{vars}(\textbf{nil}) &= \varnothing \\
\text{vars}(x) &= \{\ x\ \} \\
\text{vars}(e_1 \otimes e_2) &= \text{vars}(e_1) \cup \text{vars}(e_2)
\end{aligned}$$

To facilitate support for subtype polymorphism, we also define a binary subtype relation $c_1 \prec: c_2$ for classes:

1. $c_1 \prec: c_2$ if c_1 inherits from c_2
2. $c \prec: c$ (*reflexivity*)
3. $c_1 \prec: c_3$ if $c_1 \prec: c_2$ and $c_2 \prec: c_3$ (*transitivity*)

3.7.2 Expressions

The type judgment:

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

defines the type of expressions. We say that under environment Π, expression e has type τ.

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

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

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

Figure 3.6: Typing rules for ROOPL expressions

The type rules T-Con, T-Var and T-Nil defines the types of simple expressions. Numeric literals are always of type **int**, the type of some variable x depends on its type in the type environment Π and the **nil**-literal can have any non-integer type. All binary operations are defined for integers, while the equality and inequality comparisons are also defined for object references.

3.7.3 Statements

The type judgment:

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

defines the well-typed statements. We say that under type environment Π within class c, the statement s is well-typed with class map Γ.

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

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

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

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

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

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

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

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

Figure 3.7: Typing rules for ROOPL statements

The type rule T-AssVar defines well-typed variable assignments as only those where both sides of the assignment are of type **int** and the assignee identifier x does not occur in the expression e. Rules T-If and T-Loop define the set of well-typed conditionals and loop statements - the entry and exit conditions must be integers, while the branch and loop statements should be well-typed themselves. An object block is well-typed if the block statement is, with the new object x bound in the type environment. The skip statement is always well-typed while a statement sequence is well-typed provided each of its constituent statements are as well. A variable swap statement is well-typed only if both of its operands have the same type.

A local method invocation is well-typed, in accordance with type rule T-CALL, only if:

- The number of arguments matches the arity of the method
- No class fields are passed as arguments to the method (See Section 3.2)
- There are no duplicate arguments (See Section 3.2)
- Each argument is a subtype of the type of the equivalent formal parameter

The type rule T-CALLO establishes similar conditions for foreign method invocations, for which there is no restriction on class fields being used as arguments. There is however, the condition that the callee object x_0 is not also passed as an argument. The type rules T-UC and T-UCO describe the conditions for uncalling methods and they are both defined in terms of their inverse counterparts.

3.7.4 Programs

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

$$\frac{\Gamma(c) = \left(\overbrace{\{\langle t_1,\ f_1\rangle,\ \ldots,\ \langle t_i,\ f_i\rangle\}}^{fields},\ \overbrace{\{m_1,\ \ldots,\ m_n\}}^{methods}\right) \quad \Pi = [f_1 \mapsto t_1,\ \ldots,\ f_i \mapsto t_i] \qquad \langle \Pi,\ c\rangle \vdash^{\Gamma}_{meth} m_1 \quad \cdots \quad \langle \Pi,\ c\rangle \vdash^{\Gamma}_{meth} m_n}{\vdash^{\Gamma}_{class} c}\ \text{T-C}\small{\text{LASS}}$$

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

Figure 3.8: Typing rules for ROOPL methods, classes and programs

The type rules T-PROG, T-CLASS and T-METHOD defines the set of well-typed programs, classes and methods respectively.

A class is well-typed iff each of its methods are well-typed with all class fields bound to their respective types in the type environment. A method is well-typed iff its body is well-typed with all parameters bound to their respective types in the type environment. A ROOPL program is well-typed iff all of its classes are well-typed and there exists a nullary method named **main**. See Figure 3.5 for the definition of function *methods*.

3.8 Language Semantics

The operational semantics of ROOPL are specified by the syntax-directed inference rules shown in the following sections. There are three main judgments: the evaluation of ROOPL expressions, the execution of ROOPL statements and the execution of ROOPL programs. The next section establishes the notation and presents some auxiliary definitions.

3.8.1 Preliminaries

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

The application of an environment γ to some variable identifier x is denoted by $\gamma(x)$. Update $\gamma' = \gamma[x \mapsto l]$ defines an environment γ' such that $\gamma'(x) = l$ and $\gamma'(y) = \gamma(y)$ if $y \neq x$. The empty environment is written [ ]. The same notation is used for stores.

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

Figure 3.9: Semantic values

3.8.2 Expressions

The judgment:

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

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

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

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

Figure 3.10: Semantic inference rules for evaluation of ROOPL expressions

There are no side effects on the store when evaluating a ROOPL expression. Like in Janus, the logic value *true* is represented by any non-zero integer, while *false* is represented by zero.

For the sake of simplicity, **nil** evaluates to 0, which can never be the value of a non-nil reference, thereby ensuring that the equality and inequality operators behave as expected.

$$\llbracket\texttt{+}\rrbracket(v_1,\ v_2) \ = \ v_1 \ + \ v_2 \qquad \llbracket\texttt{\%}\rrbracket(v_1,\ v_2) \ = \ v_1 \ \mathit{mod}\ v_2$$

$$\llbracket\texttt{-}\rrbracket(v_1,\ v_2) \ = \ v_1 \ - \ v_2 \qquad \llbracket\texttt{\&}\rrbracket(v_1,\ v_2) \ = \ v_1 \ \mathit{and}\ v_2$$

$$\llbracket\texttt{*}\rrbracket(v_1,\ v_2) \ = \ v_1 \ \times \ v_2 \qquad \llbracket\texttt{|}\rrbracket(v_1,\ v_2) \ = \ v_1 \ \mathit{or}\ v_2$$

$$\llbracket\texttt{/}\rrbracket(v_1,\ v_2) \ = \ \frac{v_1}{v_2} \qquad \llbracket\texttt{\^{}}\rrbracket(v_1,\ v_2) \ = \ v_1 \ \mathit{xor}\ v_2$$

$$\llbracket\texttt{\&\&}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 0 & \text{if } v_1 \ = \ 0 \ \lor \ v_2 \ = \ 0 \\ 1 & \text{otherwise} \end{cases} \qquad \llbracket\texttt{<=}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 1 & \text{if } v_1 \ \leq \ v_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket\texttt{||}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 0 & \text{if } v_1 \ = \ v_2 \ = \ 0 \\ 1 & \text{otherwise} \end{cases} \qquad \llbracket\texttt{>=}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 1 & \text{if } v_1 \ \geq \ v_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket\texttt{<}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 1 & \text{if } v_1 \ < \ v_2 \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket\texttt{=}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 1 & \text{if } v_1 \ = \ v_2 \\ 0 & \text{otherwise} \end{cases}$$

$$\llbracket\texttt{>}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 1 & \text{if } v_1 \ > \ v_2 \\ 0 & \text{otherwise} \end{cases} \qquad \llbracket\texttt{!=}\rrbracket(v_1,\ v_2) \ = \ \begin{cases} 1 & \text{if } v_1 \ \neq \ v_2 \\ 0 & \text{otherwise} \end{cases}$$

Figure 3.11: Definition of the functions $\llbracket\otimes\rrbracket$, where $\otimes$ represents any of the binary expression operators

The inference rules Con, Var and Nil defines the meaning of expressions containing simple values or variables, while BinOp defines the meaning of expressions containing any of the arithmetic operators {**+**, **-**, *****, **/**, **%**}, bitwise operators {**&**, **|**, **^**}, logical operators {**&&**, **||**} or relational operators {**<**, **>**, **=**, **!=**, **<=**, **>=**}, all of which are defined in Figure 3.11.

3.8.3 Statements

The judgment:

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

defines the meaning of statements. We say that under environment γ and object l, statement s with class map Γ reversibly transforms store μ to store μ'. The location l is simply the location in the store μ of the *current object*. It is equivalent to the value of the *this* or *self* keywords of other OOP languages but cannot be referred to explicitly in ROOPL. Figure 3.12a on page 34 and Figure 3.12b on page 35 shows the operational semantics of ROOPL statements.

Rule Skip defines the meaning of the skip statement which has no effect on the store μ. Rule Seq defines the meaning of statement sequences and rule AssVar defines reversible assignments.

The rules LoopMain, LoopBase and LoopRec defines the meaning of loops. If assertion e_1 holds, the loop is entered by rule LoopMain. Then the loop iterates by rule LoopRec until e_2 does not hold, terminating the loop by rule LoopBase. Since conditionals and loops in ROOPL are comparable to those in Janus, these rules are similar to those presented in [46].

The semantics of conditional statements are given by rules IfTrue and IfFalse. If the entry condition evaluates to *true* (non-zero), then the **then**-branch is executed and the exit assertion

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

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

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

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

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

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

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

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

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

Figure 3.12a: Semantic inference rules for execution of ROOPL statements

should also evaluate to *true*. If the entry condition evaluates to *false*, the **else**-branch is executed and the exit assertion should evaluate to *false*.

Rule Call defines the meaning of invoking a method local to the current object. The method q in the current class c should have exactly n formal parameters $y_1,\ \ldots,\ y_n$, matching the n arguments $x_1,\ \ldots,\ x_n$. The resulting store μ' is the store obtained from executing the method body s in the object environment γ' with the arguments bound to the formal parameters.

Rule Uncall essentially reverses the direction of execution by requiring the input store of a

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

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

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

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

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

Figure 3.12b: Semantic inference rules for execution of ROOPL statements (*cont.*)

call statement to serve as the output store of the inverse **uncall** statement. A similar technique was used in [49, 46].

Rule CallObj governs invocation of methods not local to the current object. The resulting store μ' is the store obtained from executing the method body s in the environment γ' of the object x_0, with the arguments bound to the formal parameters. The inverse rule UncallObj is defined using the same approach used for rule Uncall.

Even if x_0 has been upcast to a base class (as allowed by the type system, see Section 3.7) earlier in the program, the class name c refers to the *dynamic* type of x_0. As a result, the method lookup will correctly yield the appropriate method from the derived class - in accordance with the concept of subtype-polymorphism (the actual mechanism used to achieve dynamic dispatch, virtual lookup tables, are considered an implementation detail at this point). Method dispatch in ROOPL depends only on the name of the method and the type of the callee object, not on the number of arguments nor their individual types (*single dispatch*).

Rule ObjBlock defines the meaning of a **construct**/**destruct** block and the semantics of object construction and destruction. The **construct**/**destruct** blocks of ROOPL are similar to the **local**/**delocal** blocks of Janus. In both cases, it is the program itself that is responsible for reversibly returning the memory to a state where it can be reclaimed by the system and in the presence of recursion, there is no upper bound on the size the store can grow to. Like in Janus, if x is already in scope when a block scope is entered, that variable is shadowed by the new object x within the statement block (*static lexical scoping*).

The new memory locations l', r and $a_1, \ldots, a_n$ should be unused in the store μ and they should all represent distinct memory locations. The identifiers $f_1, \ldots, f_n$ representing the fields of the new object are bound to the unused memory locations $a_1, \ldots, a_n$ in the new object environment γ'. Next, we let μ' be the updated store containing:

- The location l mapped to the object tuple $\langle c, \gamma' \rangle$
- The object reference r mapped to the location l
- The n new object fields mapped to 0

The result store μ'' (restricted to the domain of μ) is the store obtained from executing the block statement s in store μ' under environment γ mapping x to the object reference r, provided all object fields are zero-cleared in μ'' afterwards (otherwise the statement is undefined).

3.8.4 Programs

The judgment:

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

defines the meaning of ROOPL programs. Whichever class in p contains the main method is instantiated and the main method body is executed. The result is a partial function σ mapping identifiers to values, corresponding to the class fields of the main class.

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

Figure 3.13: Semantic inference rule for execution of ROOPL programs

Rule Main defines the meaning of a ROOPL program. The fields $f_1, \ldots, f_i$ of the class c containing the main method are bound in a new environment γ to the first i memory addresses (excluding address 0 which is reserved for **nil**). The first i memory addresses are then initialized to 0 in a new environment μ as well as the address $i+1$ which maps to the new instance of the main object. The modified store μ' is obtained from executing the body s of the main method. The composite function $(\mu' \ \circ \ \gamma)$, which maps each class field to its final value, serves as the output of executing p.

3.9 Program Inversion

A common formulation of the Church-Turing thesis states that a function f is computable iff there exists some Turing Machine that computes it. By extension, if some program p, written in a Turing-equivalent programming language[8], computes a function f then f is computable.

Program inversion is the process of determining an inverse program of p, computing the function f^{-1}. Given a computable function $f : X \to Y$, we wish to find a program computing the function $f' : Y \to X$ such that:

$$f(x) = y \quad \Leftrightarrow \quad f'(y) = x$$

Since f is computable, we can compute $f'(y)$ by simulating f on all inputs $x \in X$ until the result is y. This is a variation of McCarthy's generate-and-test technique [31], which implies that we can always find the inverse program if f is computable. Unfortunately, this is a completely impractical approach to program inversion. McCarthy himself described his approach in the following terms:

> [...] this procedure is extremely inefficient. It corresponds to looking for a proof of a conjecture by checking in some order all possible English essays. [31]

Recently, more practical methods for automatic program inversion of irreversible programs have superseded the generate-and-test algorithm [22]. In the context of reversible programming languages, program inversion is both simple and efficient. Reversible languages like Janus and ROOPL support *local inversion* of program statements - no contextual information or whole-program analysis is needed [21]. This is a property of reversible languages that follows from the nature of their design and the constraints they impose on the programmer. The statement inverter $\mathcal{I}$ in Figure 3.14 maps ROOPL statements to their inverse counterparts.

$$\mathcal{I}\,[\![\textbf{skip}]\!] = \textbf{skip} \qquad \mathcal{I}\,[\![s_1\ s_2]\!] = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]$$

$$\mathcal{I}\,[\![x \texttt{ += } e]\!] = x \texttt{ -= } e \qquad \mathcal{I}\,[\![x \texttt{ -= } e]\!] = x \texttt{ += } e$$

$$\mathcal{I}\,[\![x \texttt{ ^= } e]\!] = x \texttt{ ^= } e \qquad \mathcal{I}\,[\![x_1 \texttt{ <=> } x_2]\!] = x_1 \texttt{ <=> } x_2$$

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

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

$$\mathcal{I}\,[\![\textbf{if}\ e_1\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi}\ e_2]\!] = \textbf{if}\ e_1\ \textbf{then}\ \mathcal{I}[\![s_1]\!]\ \textbf{else}\ \mathcal{I}[\![s_2]\!]\ \textbf{fi}\ e_2$$

$$\mathcal{I}\,[\![\textbf{from}\ e_1\ \textbf{do}\ s_1\ \textbf{loop}\ s_2\ \textbf{until}\ e_2]\!] = \textbf{from}\ e_1\ \textbf{do}\ \mathcal{I}[\![s_1]\!]\ \textbf{loop}\ \mathcal{I}[\![s_2]\!]\ \textbf{until}\ e_2$$

$$\mathcal{I}\,[\![\textbf{construct}\ c\ x\quad s\quad \textbf{destruct}\ x]\!] = \textbf{construct}\ c\ x\quad \mathcal{I}[\![s]\!]\quad \textbf{destruct}\ x$$

Figure 3.14: Statement inverter for ROOPL statements

In ROOPL, statement inversion does not change the size of statements and as a consequence, a ROOPL program is exactly the same size as its own inverse. Furthermore, provided that every statement has the same computational complexity as its inverse, it follows that ROOPL programs have the same computational complexity as their inverted counterparts.

[8] or indeed any algorithm specified in a Turing-equivalent model of computation

$$\mathcal{I}_c \left[\!\!\left[\begin{array}{l} \textbf{class } c \ \cdots \\ \qquad \textbf{method } q_1 \ \texttt{(...)} \ s_1 \\ \qquad\quad \vdots \\ \qquad \textbf{method } q_n \ \texttt{(...)} \ s_n \end{array} \right]\!\!\right] = \begin{array}{l} \textbf{class } c \ \cdots \\ \qquad \textbf{method } q_1 \ \texttt{(...)} \ \mathcal{I}'[\![s_1]\!] \\ \qquad\quad \vdots \\ \qquad \textbf{method } q_n \ \texttt{(...)} \ \mathcal{I}'[\![s_n]\!] \end{array}$$

$$\mathcal{I}_{prog} \ [\![cl_1 \ \cdots \ cl_n]\!] \ = \ \mathcal{I}_c[\![cl_1]\!] \ \cdots \ \mathcal{I}_c[\![cl_n]\!]$$

Figure 3.15: Program and class inverters for ROOPL

Whole-program inversion is accomplished by straightforward recursive descent over the components and statements of the program. Figure 3.15 shows the definition of the ROOPL program inverter $\mathcal{I}_{prog}$, which inverts each method in each class to produce the inverse program. The program inverter $\mathcal{I}_{prog}$ is an *involution*, so inverting a program twice will yield the original program.

$$\begin{array}{llll} \mathcal{I}' \ [\![\textbf{call } q\texttt{(...)}]\!] \ = \ \textbf{call } q\texttt{(...)} & & \mathcal{I}' \ [\![\textbf{call } x\texttt{::}q\texttt{(...)}]\!] \ = \ \textbf{call } x\texttt{::}q\texttt{(...)} \\ \mathcal{I}' \ [\![\textbf{uncall } q\texttt{(...)}]\!] \ = \ \textbf{uncall } q\texttt{(...)} & & \mathcal{I}' \ [\![\textbf{uncall } x\texttt{::}q\texttt{(...)}]\!] \ = \ \textbf{uncall } x\texttt{::}q\texttt{(...)} \end{array}$$

$$\mathcal{I}' \ [\![s]\!] \ = \ \mathcal{I}[\![s]\!]$$

Figure 3.16: Modified statement inverter for ROOPL statements

Because calling a method is equivalent to uncalling the same method inverted, if we change call-statements into uncall-statements and vice-versa, the inversion of the method body is cancelled out.

To fix this issue, we use a modified version of the statement inverter for the whole-program inversion, that does not invert calls and uncalls. Figure 3.16 shows the modified statement inverter $\mathcal{I}'$.

3.9.1 Invertibility of Statements

Theorem 3.1 shows that $\mathcal{I}$ is in fact a statement inverter. If executing statement s in store μ yields μ', then executing statement $\mathcal{I}[\![s]\!]$ in store μ' should yield μ.

Theorem 3.1. *(Invertibility of statements)*

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

Proof. The proof is by structural induction on the semantic derivation of $\mathcal{S}$ but is omitted. It suffices to show that $\mathcal{S}$ implies $\mathcal{S}'$ - since this can also serve as proof that $\mathcal{S}'$ implies $\mathcal{S}$ because $\mathcal{I}$ is an involution.

3.9.2 Type-Safe Statement Inversion

When given a well-typed statement, the statement inverter $\mathcal{I}$ should always produce a well-typed (inverse) statement. This is an important property of the language as it prevents situations where some method can be *called* successfully, but *uncalling* the same method produces an error or undefined behaviour. The following theorem expresses this property:

Theorem 3.2. *(Inversion of well-typed statements)*

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

Proof. By structural induction on $\mathcal{T}$:

$$\textbf{Case } \mathcal{T} \ = \ \frac{\overbrace{x \ \notin \ \text{vars}(e)}^{\mathcal{C}_1} \qquad \overbrace{\Pi \ \vdash_{expr} \ e \ : \ \textbf{int}}^{\mathcal{E}} \qquad \overbrace{\Pi(x) \ = \ \textbf{int}}^{\mathcal{C}_2}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ x \ \odot\texttt{=}\ e} \ \textsc{AssVar}$$

In this case, $\mathcal{I}[\![x \ \odot\texttt{=}\ e]\!] \ = \ x \ \odot'\texttt{=}\ e$ for some $\odot'$, so $\mathcal{T}'$ will also be a derivation of rule AssVar. Therefore we can just reuse the expression derivation $\mathcal{E}$ and the conditions $\mathcal{C}_1$ and $\mathcal{C}_2$ to construct $\mathcal{T}'$:

$$\mathcal{T}' \ = \ \frac{\overbrace{x \ \notin \ \text{vars}(e)}^{\mathcal{C}_1} \qquad \overbrace{\Pi \ \vdash_{expr} \ e \ : \ \textbf{int}}^{\mathcal{E}} \qquad \overbrace{\Pi(x) \ = \ \textbf{int}}^{\mathcal{C}_2}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ x \ \odot'\texttt{=}\ e}$$

$$\textbf{Case } \mathcal{T} \ = \ \frac{\Pi(x_1) \ = \ \Pi(x_2)}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ x_1 \ \texttt{<=>} \ x_2} \ \textsc{T-SwpVar}$$

Since $\mathcal{I}[\![x_1 \ \texttt{<=>} \ x_2]\!] \ = \ x_1 \ \texttt{<=>} \ x_2$, we can just use the derivation of $\mathcal{T}$ for $\mathcal{T}'$:

$$\mathcal{T}' \ = \ \frac{\Pi(x_1) \ = \ \Pi(x_2)}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ x_1 \ \texttt{<=>} \ x_2}$$

$$\textbf{Case } \mathcal{T} \ = \ \frac{\overbrace{\Pi \ \vdash_{expr} \ e_1 \ : \ \textbf{int}}^{\mathcal{E}_1} \quad \overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ s_1}^{\mathcal{S}_1} \quad \overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ s_2}^{\mathcal{S}_2} \quad \overbrace{\Pi \ \vdash_{expr} \ e_2 \ : \ \textbf{int}}^{\mathcal{E}_2}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ \textbf{if } e_1 \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi } e_2} \ \textsc{T-If}$$

We have: $\quad \mathcal{I}\ [\![\textbf{if } e_1 \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi } e_2]\!] \ = \ \textbf{if } e_1 \textbf{ then } \mathcal{I}[\![s_1]\!] \textbf{ else } \mathcal{I}[\![s_2]\!] \textbf{ fi } e_2$

By the induction hypothesis on $\mathcal{S}_1$ we get: $\quad \mathcal{S}_1' \ = \ \langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ \mathcal{I}[\![s_1]\!]$

By the induction hypothesis on $\mathcal{S}_2$ we get: $\quad \mathcal{S}_2' \ = \ \langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ \mathcal{I}[\![s_2]\!]$

Using $\mathcal{E}_1$, $\mathcal{S}_1'$, $\mathcal{S}_2'$ and $\mathcal{E}_2$ we can construct $\mathcal{T}'$:

$$\mathcal{T}' \ = \ \frac{\overbrace{\Pi \ \vdash_{expr} \ e_1 \ : \ \textbf{int}}^{\mathcal{E}_1} \quad \overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ \mathcal{I}[\![s_1]\!]}^{\mathcal{S}_1'} \quad \overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ \mathcal{I}[\![s_2]\!]}^{\mathcal{S}_2'} \quad \overbrace{\Pi \ \vdash_{expr} \ e_2 \ : \ \textbf{int}}^{\mathcal{E}_2}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt} \ \textbf{if } e_1 \textbf{ then } \mathcal{I}[\![s_1]\!] \textbf{ else } \mathcal{I}[\![s_2]\!] \textbf{ fi } e_2}$$

$$\textbf{Case } \mathcal{T} \;=\; \dfrac{\overbrace{\Pi \vdash_{expr} e_1 : \textbf{int}}^{\mathcal{E}_1} \quad \overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} s_1}^{\mathcal{S}_1} \quad \overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} s_2}^{\mathcal{S}_2} \quad \overbrace{\Pi \vdash_{expr} e_2 : \textbf{int}}^{\mathcal{E}_2}}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \textbf{from } e_1 \textbf{ do } s_1 \textbf{ loop } s_2 \textbf{ until } e_2} \ \text{T-Loop}$$

We have: $\mathcal{I}\, [\![\textbf{from } e_1 \textbf{ do } s_1 \textbf{ loop } s_2 \textbf{ until } e_2]\!] \;=\; \textbf{from } e_1 \textbf{ do } \mathcal{I}[\![s_1]\!] \textbf{ loop } \mathcal{I}[\![s_2]\!] \textbf{ until } e_2$

By the induction hypothesis on $\mathcal{S}_1$ we get: $\mathcal{S}_1' \;=\; \langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_1]\!]$

By the induction hypothesis on $\mathcal{S}_2$ we get: $\mathcal{S}_2' \;=\; \langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_2]\!]$

Using $\mathcal{E}_1$, $\mathcal{S}_1'$, $\mathcal{S}_2'$ and $\mathcal{E}_2$ we can construct $\mathcal{T}'$:

$$\mathcal{T}' \;=\; \dfrac{\overbrace{\Pi \vdash_{expr} e_1 : \textbf{int}}^{\mathcal{E}_1} \quad \overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_1]\!]}^{\mathcal{S}_1'} \quad \overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_2]\!]}^{\mathcal{S}_2'} \quad \overbrace{\Pi \vdash_{expr} e_2 : \textbf{int}}^{\mathcal{E}_2}}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \textbf{from } e_1 \textbf{ do } \mathcal{I}[\![s_1]\!] \textbf{ loop } \mathcal{I}[\![s_2]\!] \textbf{ until } e_2}$$

$$\textbf{Case } \mathcal{T} \;=\; \dfrac{\overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} s_1}^{\mathcal{S}_1} \quad \overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} s_2}^{\mathcal{S}_2}}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} s_1\ s_2} \ \text{T-Seq}$$

We have: $\mathcal{I}\, [\![s_1\ s_2]\!] \;=\; \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]$

By the induction hypothesis on $\mathcal{S}_1$ we get: $\mathcal{S}_1' \;=\; \langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_1]\!]$

By the induction hypothesis on $\mathcal{S}_2$ we get: $\mathcal{S}_2' \;=\; \langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_2]\!]$

Using $\mathcal{S}_1'$ and $\mathcal{S}_2'$ we can construct $\mathcal{T}'$:

$$\mathcal{T}' \;=\; \dfrac{\overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_2]\!]}^{\mathcal{S}_2'} \quad \overbrace{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_1]\!]}^{\mathcal{S}_1'}}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]}$$

$$\textbf{Case } \mathcal{T} \;=\; \dfrac{}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \textbf{skip}} \ \text{T-Skip}$$

Since $\mathcal{I}\, [\![\textbf{skip}]\!] \;=\; \textbf{skip}$, and T-Skip is axiomatic, we can choose $\mathcal{T}$ as:

$$\mathcal{T}' \;=\; \dfrac{}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \textbf{skip}}$$

$$\textbf{Case } \mathcal{T} \;=\; \dfrac{\overbrace{\langle \Pi[x \mapsto c'],\ c \rangle \vdash^{\Gamma}_{stmt} s}^{\mathcal{S}}}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \textbf{construct } c'\ x \quad s \quad \textbf{destruct } x} \ \text{T-ObjBlock}$$

We have: $\mathcal{I}\, [\![\textbf{construct } c\ x \quad s \quad \textbf{destruct } x]\!] \;=\; \textbf{construct } c\ x \quad \mathcal{I}[\![s]\!] \quad \textbf{destruct } x$

By the induction hypothesis on $\mathcal{S}$ we get: $\mathcal{S}' \;=\; \langle \Pi[x \mapsto c'],\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s]\!]$

Which we can use to construct $\mathcal{T}'$:

$$\mathcal{T}' \;=\; \dfrac{\overbrace{\langle \Pi[x \mapsto c'],\ c \rangle \vdash^{\Gamma}_{stmt} \mathcal{I}[\![s]\!]}^{\mathcal{S}'}}{\langle \Pi,\ c \rangle \vdash^{\Gamma}_{stmt} \textbf{construct } c'\ x \quad \mathcal{I}[\![s]\!] \quad \textbf{destruct } x}$$

Case $\mathcal{T} \;=\; \dfrac{\cdots}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{call}\ q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}$ T-Call

We have: $\quad \mathcal{I}\,[\![\textbf{call}\ q\,\texttt{(}\ldots\texttt{)}]\!] \;=\; \textbf{uncall}\ q\,\texttt{(}\ldots\texttt{)}$

Which means $\mathcal{T}'$ must be of the form:

$$\mathcal{T}' \;=\; \dfrac{\overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{call}\ q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}^{\mathcal{S}}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{uncall}\ q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}$$

Where we can simply use the derivation of $\mathcal{T}$ in place of $\mathcal{S}$.

Case $\mathcal{T} \;=\; \dfrac{\cdots}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{call}\ x_0\texttt{::}q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}$ T-CallO

We have: $\quad \mathcal{I}\,[\![\textbf{call}\ x\texttt{::}q\,\texttt{(}\ldots\texttt{)}]\!] \;=\; \textbf{uncall}\ x\texttt{::}q\,\texttt{(}\ldots\texttt{)}$

Which means $\mathcal{T}'$ must be of the form:

$$\mathcal{T}' \;=\; \dfrac{\overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{call}\ x_0\texttt{::}q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}^{\mathcal{S}}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{uncall}\ x_0\texttt{::}q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}$$

Where we can simply use the derivation of $\mathcal{T}$ in place of $\mathcal{S}$.

Case $\mathcal{T} \;=\; \dfrac{\overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{call}\ q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}^{\mathcal{S}}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{uncall}\ q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}$ T-UC

We have: $\quad \mathcal{I}\,[\![\textbf{uncall}\ q\,\texttt{(}\ldots\texttt{)}]\!] \;=\; \textbf{call}\ q\,\texttt{(}\ldots\texttt{)}$

Which means we can just use the derivation $\mathcal{S}$ as $\mathcal{T}'$.

Case $\mathcal{T} \;=\; \dfrac{\overbrace{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{call}\ x_0\texttt{::}q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}^{\mathcal{S}}}{\langle \Pi,\ c\rangle \ \vdash^{\Gamma}_{stmt}\ \textbf{uncall}\ x_0\texttt{::}q\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)}}$ T-UCO

We have: $\quad \mathcal{I}\,[\![\textbf{uncall}\ x\texttt{::}q\,\texttt{(}\ldots\texttt{)}]\!] \;=\; \textbf{call}\ x\texttt{::}q\,\texttt{(}\ldots\texttt{)}$

Which means we can just use the derivation $\mathcal{S}$ as $\mathcal{T}'$.

□

Using Theorem 3.2, we can show that well-typedness is also preserved over inversion of methods. By type rule T-Method (See Figure 3.8, page 31), we see that a method is well-typed iff its body is well-typed.

The class inverter $\mathcal{I}_c$ (See Figure 3.15) defines the inverse of a method q with body s, as the same method with the body $\mathcal{I}[\![s]\!]$. By Theorem 3.2, we know that if s is well-typed, then so is $\mathcal{I}[\![s]\!]$ - by extension, if q is well-typed then so is the inverse of q.

By the definition of the class inverter and the program inverter, it is clear that this result also extends to inversion of classes and inversion of programs.

3.10 Language Extensions

The language extensions introduced in this section are not part of the core language, but are used in the ROOPL programs we present in subsequent sections and chapters.

3.10.1 Local Variables

Due to the restriction prohibiting member variables being passed to methods of the same object, it is sometimes necessary to create proxy objects or needlessly complicated structures to achieve relatively simple tasks. The restriction only serves to avoid aliasing situations, so we can make the life of a ROOPL programmer easier by adding the **local**/**delocal** blocks from Janus to ROOPL:

$$\textbf{local int}\ x\ =\ e_1 \quad s \quad \textbf{delocal}\ x\ =\ e_2$$

Unlike in Janus, only integers can be allocated this way. If x is already in scope at the time this statement occurs, the new x shadows the definition of the existing x, just as is the case for object blocks. The semantics of this statement were already covered in [46] and do not differ in any noticeable way in ROOPL.

3.10.2 Class Constructors and Deconstructors

In OOP, a *class invariant* is a constraint placed on the internal state of an object. Consider a *Date* class representing a specific day of the year, with member variables denoting the day of the month and the month of the year as integers. An obvious invariant for this class is that the day of the month should always be between 1 and 31 inclusively and the month should always be between 1 and 12 inclusively. Class invariants are an instance of *contract programming*[9] that is especially relevant for OOP, where we wish to hide the internal constraints of a class behind the public interface.

In ROOPL, all newly created objects are always zero-initialized, which is directly at odds with the notion of class invariants. In our example, this means that all *Date* objects start out representing day 0 of month 0 which is outside of our established invariant and inconsistent with the rules of the system we are modelling. If we, instead, allow the programmer to specify how an object should be initialized, we can make sure that class invariants are enforced throughout an objects' lifetime.

$$\begin{array}{lcl} & & \textbf{construct}\ c\ x \\ \textbf{construct}\ c\ x\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)} & & \textbf{call}\ x\texttt{::}\mathrm{constructor}\,\texttt{(}x_1,\ \ldots,\ x_n\texttt{)} \\ s & \stackrel{\textbf{def}}{=\!=} & s \\ \textbf{destruct}\ x\,\texttt{(}z_1,\ \ldots,\ z_n\texttt{)} & & \textbf{uncall}\ x\texttt{::}\mathrm{constructor}\,\texttt{(}z_1,\ \ldots,\ z_n\texttt{)} \\ & & \textbf{destruct}\ x \end{array}$$

Figure 3.17: Class constructor/deconstructor extension

[9]Popularized by languages such as *Eiffel* and *D*, which both include support for automatically verifying class invariants at runtime.

Figure 3.17 shows a new form of the **construct**/**destruct** statement, which automatically invokes the special method *constructor* when a new object is created, establishing the class invariants of the object. After the block statement is executed, the constructor is then automatically uncalled (we call this the *deconstructor* call) before the object is then finally deallocated. The purpose of the deconstructor is to uncompute the state accumulated within the object by the constructor (and possibly by other method invocations within s).

Ideally the compiler should be able to enforce that the default constructor (which zero-initializes the object) is only ever invoked when the class in question does not specify its own constructor. The proposed implementation only shows how to implement class constructors/deconstructors in terms of the core language.

```
class Date
    int day      //Invariant:   1 <= day <= 31
    int month    //Invariant:   1 <= month <= 12

    method constructor(int d, int m)
        if d <= 0 then
            day += 1
        else
            day += d % 31
        fi d <= 0

        if m <= 0 then
            month += 1
        else
            month += m % 12
        fi m <= 0

    method nextDay()
        if day = 31 then
            day -= 30
            call nextMonth()
        else
            day += 1
        fi day = 1

    method nextMonth()
        if month = 12 then
            month -= 11
        else
            month += 1
        fi month = 1

    method getDay(int out)
        out ^= day

    method getMonth(int out)
        out ^= month
```

Figure 3.18: ROOPL class representing a calendar date

Note that there is no requirement that the constructor and deconstructor are given the same arguments. The only requirements are that the class invariants are established after the constructor call and that the internal state of the object is zero-cleared after the deconstructor call. Figure 3.18 shows how an implementation of a simplified *Date* class might look in ROOPL, with accessors and constructor/deconstructor method included.

3.10.3 Expression Arguments

Like in both Janus and R, we permit expressions to be used as arguments to a method provided the method does not directly alter the value of the parameter in any way. If the value of the expression parameter is altered by the callee, the meaning of the call is undefined.

$$\textbf{call } q\,\texttt{(}\ \ldots,\ e,\ \ldots\ \texttt{)} \quad \stackrel{\textbf{def}}{=\!=} \quad \begin{array}{l} \textbf{local int } x' \ = e \\ \textbf{call } q\,\texttt{(}\ \ldots,\ x',\ \ldots\ \texttt{)} \\ \textbf{delocal } x' \ = e \end{array}$$

Figure 3.19: Language extension for expressions as method arguments

3.10.4 Method Reversal

Because arguments are passed by reference, a method invocation can bring about changes to many or all of the argument variables in the caller. On top of this, ROOPL methods are impure and can result in alterations being made to the internal state of one or more objects.

$$\textbf{reversal } q\,\texttt{(}x_1,\ x_2\texttt{)}\ s \quad \stackrel{\textbf{def}}{=\!=} \quad \begin{array}{l} \textbf{call } q\,\texttt{(}x_1,\ x_2\texttt{)} \\ s \\ \textbf{uncall } q\,\texttt{(}x_1,\ x_2\texttt{)} \end{array}$$

Figure 3.20: Language extension for single-statement method reversals

A common pattern for reversibly dealing with side effects and extra data is to sandwich the statement block handling the result between a call and an uncall of the method in question. This allows the programmer to copy the result or utilize it in some computation without worrying about the subsequent clean up. Figure 3.20 shows a language extension that conveniently reduces this pattern to a single statement.

3.10.5 Short Form Control Flow

For the sake of convenience, we introduce short forms for conditionals and loops.

$$\begin{array}{rcl} \textbf{if } e_1 \textbf{ then } s \textbf{ fi } e_2 & \stackrel{\textbf{def}}{=\!=} & \textbf{if } e_1 \textbf{ then } s \textbf{ else skip fi } e_2 \\ \textbf{from } e_1 \textbf{ do } s \textbf{ until } e_2 & \stackrel{\textbf{def}}{=\!=} & \textbf{from } e_1 \textbf{ do } s \textbf{ loop skip until } e_2 \\ \textbf{from } e_1 \textbf{ loop } s \textbf{ until } e_2 & \stackrel{\textbf{def}}{=\!=} & \textbf{from } e_1 \textbf{ do skip loop } s \textbf{ until } e_2 \end{array}$$

Figure 3.21: Syntactic sugar for short form conditionals and loops

3.11 Language Idioms

Like in conventional programming languages, specific program patterns are used, in ROOPL, to express recurring tasks or constructs that are not built-in features of the language. Such programming idioms are discussed in the following sections.

3.11.1 Zero-Cleared Copying

Care must be taken when copying and clearing values in a reversible language. Copying the value of one variable to another can only be done reversibly if the destination variable is zero-cleared, otherwise the value of the destination variable must be overwritten, resulting in a loss of information. Likewise, clearing the value of some variable is only possible if the same value is stored elsewhere at the same point in time, also to prevent loss of information. In ROOPL, both copying and clearing can be achieved with an XOR-assignment:

$$x \mathbin{\texttt{\textasciicircum=}} y$$

If $x = y$ before the above statement, then x is zero-cleared. If $x = 0$ before the assignment, then the value of y is copied into x. This technique was first described in [49].

3.11.2 Mutators and Accessors

```
class Object
    int data

    method get(int out)
        out ^= data

    method swap(int in)
        data <=> in

    method sub(int val)
        data -= val

    method add(int val)
        data += val

    method xor(int val)
        data ^= val
```

Figure 3.22: Basic mutator and accessor methods in ROOPL

In accordance with the principle of encapsulation, the member variables of a ROOPL object are not directly accessible from outside the methods of that object. To facilitate access, we can implement special accessor and mutator methods (colloquially known as *getters* and *setters*).

The semantics of accessors and mutators are slightly different in a reversible language. In conventional OOP languages, a mutator will simply assign a new value to the member variable, overwriting the existing value. In ROOPL we are limited to *reversible mutators*, exemplified by the methods *swap*, *sub*, *add* and *xor* in Figure 3.22.

The swap mutator works mostly like a conventional mutator, but rather than irreversibly overwriting the existing value, it places that value in the parameter, leaving the caller responsible for uncomputing or clearing it.

Since ROOPL does not support return values, we must supply the accessor method *get* with an output parameter. Provided the argument variable is zero-cleared before invocation, the value of the member variable is copied into the argument and thereby made accessible to the caller, outside of the object.

3.11.3 Abstract Methods

An *abstract method* is a method with only a method signature but no method body. If a class contains an abstract method, it cannot be instantiated. Instead a subclass can override the abstract method and provide a method body, in which case the subclass can be instantiated. Abstract methods are used as a way to define interfaces - the base class contains a number of abstract methods that all subclasses must implement.

```
//Shape interface
class Shape
    method resize(int scale)
        skip //Abstract method

    method translate(int x, int y)
        skip //Abstract method

    method draw()
        skip //Abstract method

    method getArea(int out)
        skip //Abstract method
```

Figure 3.23: Example of an interface in ROOPL

ROOPL does not have any special facilities for supporting abstract methods (See Section 3.4) but we can simulate abstract methods and class interfaces by using the **skip** statement as a method body for the abstract methods of an interface. Figure 3.23 shows an example of a class interface defined in this manner.

3.11.4 Call-Uncall

A core tenet of modern software development is the DRY-principle [26], short for Don't Repeat Yourself. It holds that duplication in logic should be eliminated via abstraction, which usually entails using methods and procedures to facilitate code reuse in a program[10].

In a reversible language like ROOPL, however, every statement has two distinct meanings depending on the direction of execution and therefore twice as many possible applications for the programmer to consider. As such, the potential for code reuse in ROOPL programs is considerable - many common programming tasks have an equally common inverse (the canonical examples are the *push* and *pop* operations of a stack), but in ROOPL such inversions are free in terms of programming effort and code size.

Another idiomatic use of the uncall mechanism is the compute-copy-uncompute technique, which reversibly uncomputes intermediate values left over after a computation, retaining only the desired results.

[10] In fact the DRY-principle also holds that duplication in process and testing should be eliminated by automation. In the absence of DRY, a software project is said to become WET (Write Everything Twice), which is generally considered a very error-prone approach to software development.

3.11.5 Linked Lists

While Janus included built-in support for arrays [49] and stacks [46], ROOPL does not support any data structures or collections as language primitives[11]. Using recursion and recursively defined data types, we can define a linked list in ROOPL even without built-in support for arrays or other types of collections.

```
class Node //Represents a single node in the list
    int data
    Node next //Reference to next node in the list

    //Constructor method
    method constructor(int d, Node n)
        data ^= d
        next <=> n

    //Accessor & mutator methods
    method add(int out)
        out += data

    method sub(int out)
        out -= data

    method xor(int out)
        out ^= data

    method swap(int out)
        out <=> data

    method swapNext(Node out)
        out <=> next

    method length(int out) //Finds the length of the list
        out += 1
        if next != nil then
            call next::length(out)
        fi next != nil

    method insert(int n, Node new) //Inserts a (single) new node in the list
        if n = 0 then
            next <=> new
        else
            if n = 1 then
                next <=> new
            fi n = 1

            if next != nil then
                n -= 1
                call next::insert(n, new)
                n += 1
            fi next != nil
        fi n = 0
```

Figure 3.24a: Example of recursively defined linked lists in ROOPL

Figure 3.24a shows the definition of a *Node* class which contains a single integer and a reference to the next node in the list, which is always **nil** for the last node in a list. The node

[11] There is no inherent reason such language constructs could not be added to ROOPL, and they would likely improve the expressiveness of the language. However, they are not especially noteworthy nor interesting from an OOP perspective and were therefore not included.

provides a constructor and a variety of accessors to both the data and the next node.

The *Node* class also implements a method *length* for recursively computing the length of the list. The method *insert* is used to insert a single node into the list at a given index, or alternatively, extracting a node from the list when uncalled.

```
class Iterator //Iterator interface
    int result

    //Abstract method
    method run(Node head, Node next)
        skip

    //Accessor
    method get(int out)
        out <=> result

class ListBuilder
    int n //The length of the list to build
    Iterator it //The iterator instance to run
    Node empty //Helper node

    //Constructor method
    method constructor(int len, Iterator i)
        n += len
        it <=> i

    method build(Node head)
        if n = 0 then
            if head != nil
                //List is done, run the iterator
                call it::run(head, empty)
            fi head != nil
        else
            //Not yet done, construct next node
            construct Node next(n, head)
                n -= 1
                call build(next)
                n += 1
            destruct next(n, head)
        fi n = 0
```

Figure 3.24b: Example of recursively defined linked lists in ROOPL (*cont.*)

The *ListBuilder* class defined in Figure 3.24b is used to recursively construct lists of arbitrary length from back to front. As a *Node* is constructed, it is passed its own (1-based) index in the list and a reference to the next node in the list. When the list has been built, an iterator is invoked on the head of the list (working front-to-back). When the iterator finally returns, the list is deconstructed.

The class *Sum* in Figure 3.24c on page 49, is an example of a class that implements the *Iterator* interface. It iterates over the nodes in a list, summing up the value of their contents. The class *Program* illustrates how to use *ListBuilder* and *Sum* to build a linked-list and iterate over it. By using the *Iterator* interface we make the list builder more generic - it doesn't care what kind of operation we want to perform on the list, it only cares that the iterator object it is given conforms to the interface that it knows about.

The list is created by recursively entering a **construct**/**destruct** block. When the desired length is reached, the recursion halts, the iterator is invoked and then the list is deconstructed simply by unwinding the call stack, one call (and one corresponding list node) at a time.

```
class Sum inherits Iterator
    int sum

    method run(Node head, Node next)
        call head::add(sum)
        call head::swapNext(next)
        if next = nil then
            result += sum //Finished
        else
            call run(next, head) //More work to do
        fi next = nil
        uncall head::swapNext(next) //Return list to original state
        uncall head::add(sum)

class Program
    int result //Final result
    Node empty //Helper node

    method main()
        local int n = 5 //List length
        construct Sum it //Construct iterator
            construct ListBuilder lb(n, it) //Construct list builder
                call lb::build(empty) //Build & iterate
            destruct lb(n, it)
            call it::get(result) //Fetch result
        destruct it
        delocal n = 5
```

Figure 3.24c: Example of recursively defined linked lists in ROOPL (*cont.*)

This style of programming is similar to continuation-passing style (CPS) - the iterator acts as a continuation that the builder can pass the list on to after it has been constructed. There is no way for the builder to return the list back to the initial caller, as that would involve unwinding the call stack and thus deconstructing the list in the process. The main difference between this approach and CPS is that CPS is usually accomplished by passing the continuation directly as a function, but since ROOPL does not support higher-order functions we are limited to using objects.

3.12 Computational Strength

A programming language is said to be *computationally universal* or *Turing complete* if it is capable of simulating any single-taped Turing Machine, which in turn means it is capable of computing any of the computable functions. Reversible programming languages like Janus and ROOPL are not Turing complete since they are only capable of computing exactly those computable functions that are also injective.

Yokoyama et al. suggests simulation of the *reversible* Turing machines as the computational benchmark for reversible programming languages [46]. A reversible Turing machine (RTM) is any Turing machine computing an injective function [6, 47]. If a reversible programming language is able to cleanly simulate any RTM, then we say that it is *reversibly universal* or *r-Turing complete*.

The original versions of Janus [30, 49] were not r-Turing complete since they only supported static fixed-size storage. The latest version of the language adds support for dynamic storage and was proven to be r-Turing complete by construction of an RTM interpreter [46]. In the following sections, we present techniques for constructing a similar RTM interpreter using ROOPL. The intepreter serves as a proof that ROOPL is also reversibly universal.

3.12.1 RTM Representation

We use the same Turing machine formalism as used in [46], with state transitions represented by quadruples:

Definition 3.1. *(Quadruple Turing Machine)*

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

Q is the finite, non-empty set of states

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

$b \in \Gamma$ is the blank symbol

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

$q_s \in Q$ is the starting state

$q_f \in Q$ is the final state

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

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

Definition 3.2. *(Reversible Turing Machine)*

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

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

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

In ROOPL we can represent the set of states $\{q_1,\ \ldots,\ q_n\}$ and the tape alphabet Γ as integers. The shift rule symbol / and the direction symbols L and R are then represented by the integer variables **`SLASH`**, **`LEFT`** and **`RIGHT`** respectively.

With this representation, we can model a transition rule as an object containing four integers **`q1`**, **`s1`**, **`s2`** and **`q2`** where **`s1`** equals **`SLASH`** for shift rules. A linked list of such transition rules serves as the full transition table δ. Using the techniques described in Section 3.11.5 we can look up the appropriate transition rule at each step of the simulation, with an index variable that rolls around to 0 whenever it exceeds the length of the transition table.

Since states are numbers in our simulation, we can use a single integer variable which is updated as the simulation runs, to keep track of the current state of the RTM. After each iteration of the RTM simulation - the current state is compared to the final state **`Qf`**, if they are the same the simulation stops.

3.12.2 Tape Representation

The tape of an RTM has to be able to grow unboundedly in both directions[12]. With the tape alphabet being represented by integers, we can use a simple object containing just an integer to model a tape cell. The full tape is represented by a linked list of such cells.

The position of the tape head of the RTM determines which tape cell is currently being inspected or modified. In our simulation we can use an integer variable to store the position of the tape head as an index into the list of tape cells. Initially, the tape should contain just the input and the tape head should be at index 0. After each simulated step of the RTM we:

1. Calculate the current length of the tape.
2. If the position of the tape head is less than zero: The tape head has moved off the left end of the tape. We allocate a new cell, prepend it to the list and zero-clear the tape head position.
3. If the position of the tape head exceeds the current length of the tape: The tape head has moved off the right end of the tape. We allocate a new cell and append it to the tape list.

Our model of the tape can now also grow unboundedly in both directions.

3.12.3 RTM Simulation

Figure 3.25 shows the method *inst* which executes a single instruction given a reference to the head of the tape, the position of the tape head, the current state of the RTM and four integers representing the transition rule to be executed.

```
method inst(Cell tape, int pos, int state, int q1, int s1, int s2, int q2)
    local int symbol = 0
    call tape::lookup(pos, symbol) //Fetch current symbol

    if state = q1 && s1 = symbol then //SYMBOL RULE
        state += q2 - q1 //Update state to q2
        symbol += s2 - s1 //Update symbol to s2
        call tape::add(pos, s2 - s1) //Update tape cell to s2
    fi state = q2 && s2 = symbol

    uncall tape::lookup(pos, symbol) //Zero-clear symbol
    delocal symbol = 0

    if state = q1 && s1 = SLASH then //SHIFT RULE
        state += q2 - q1 //Update state to q2

        if s2 = RIGHT then
            pos += 1 //Move tape head right
        fi s2 = RIGHT

        if s2 = LEFT then
            pos -= 1 //Move tape head left
        fi s2 = LEFT
    fi state = q2 && s1 = SLASH
```

Figure 3.25: Method for executing a single TM transition

[12] The term *linear bounded automaton* is used to denote TM-like automatons with an upper bound on the size of the tape.

Figure 3.26 shows the recursively defined *simulate* method which is the main method responsible for running the RTM simulation. It extends the tape in either direction when necessary, fetches the transition quadruple, updates the program counter and copies the result when the RTM halts.

```
method simulate(Cell tape, int pos, int state, int pc)
    local int len = 0
    call tape::length(len) //Calculate length of tape

    if pos > len then //Append new tape cell
        construct Cell new(BLANK, empty)
        call tape::insert(pos, len)
        call simulate(tape, pos, state, pc) //Continue simulation
        uncall tape::insert(pos, len)
        destruct new(BLANK, empty)
    else
        if pos < 0 then //Prepend new tape cell
            construct Cell new(BLANK, tape)
            tape <=> new
            pos += 1
            call simulate(tape, pos, state, pc) //Continue simulation
            pos -= 1
            tape <=> new
            destruct new(BLANK, tape)
        else
            local int q1 = 0, s1 = 0, s2 = 0, q2 = 0
            call incPc(pc, PC_MAX) //Increment pc
            call RTM::get(pc, q1, s1, s2, q2) //Fetch transition quadruple

            call inst(tape, pos, state, q1, s1, s2, q2)

            if state = Qf then //If RTM simulation is finished
                call tape::get(result) //Copy result of simulation
            else
                call simulate(tape, pos, state, pc) //Continue simulation
            fi state = Qf

            uncall inst(tape, pos, state, q1, s1, s2, q2)

            uncall RTM::get(pc, q1, s1, s2, q2) //Clear transition quadruple
            uncall incPc(pc, PC_MAX) //Decrement pc
            delocal q1 = 0, s1 = 0, s2 = 0, q2 = 0
        fi pos < 0
    fi pos > len

    uncall tape::length(len) //Clear length of tape
    delocal len = 0
```

Figure 3.26: Main RTM simulation method

Unlike the RTM simulator created with Janus, which uses a pair of stack primitives to represent the RTM tape, the ROOPL RTM simulator cannot finish with the TM tape as the program output. Whenever a tape cell is created, the simulator invokes the next operation recursively - but when the TM halts, the call stack of the simulation must unwind before the main method and the program can finally terminate, which results in the tape cells being deallocated one by one. The program must even ensure that the tape cells are zero-cleared before they are deallocated which can only be done reversibly by uncomputing the simulation. When the TM halts, the entire simulation therefore runs again in reverse to return the tape cells to their original state as the simulator proceeds down the call stack.

CHAPTER 4

Compilation

This chapter presents the code generation schemes used to translate ROOPL source code to *PISA Assembly Language* (PAL). The translated programs are semantically equivalent to the source programs and generate no additional garbage data. Due to the syntactic and semantic similarities between Janus and ROOPL, some of the techniques presented here are similar to those presented in [2] which describes the translation from Janus to PAL.

4.1 Preliminaries

See Section 2.4 in Chapter 2 for a brief description of the PISA instruction set that we target in this chapter. A more in-depth presentation of PISA and the Pendulum architecture can be found in [42]. For presentation purposes, we will make use of the three pseudoinstructions defined in Figure 4.1.

$$
\begin{aligned}
\texttt{\textbf{SUBI}}\ \ r\ \ i &\ \stackrel{\textbf{def}}{=\!=}\ \texttt{\textbf{ADDI}}\ \ r\ \ -i \\
\texttt{\textbf{PUSH}}\ \ r &\ \stackrel{\textbf{def}}{=\!=}\ \Big[\texttt{\textbf{EXCH}}\ \ r\ \ r_{sp}\ ,\ \ \texttt{\textbf{ADDI}}\ \ r_{sp}\ \ 1\Big] \\
\texttt{\textbf{POP}}\ \ r &\ \stackrel{\textbf{def}}{=\!=}\ \Big[\texttt{\textbf{SUBI}}\ \ r_{sp}\ \ 1\ ,\ \ \texttt{\textbf{EXCH}}\ \ r\ \ r_{sp}\Big]
\end{aligned}
$$

Figure 4.1: Definition of pseudoinstructions **`SUBI`**, **`PUSH`** and **`POP`**

Our translation uses *virtual function tables* and *object layout prefixing* to implement subtype polymorphism. Every class method of the source program is translated to a series of PISA instructions. The translated methods accept an extra hidden parameter for the object pointer, which points to the object that the method is associated with and is used to access the instance variables of that object.

4.2 Memory Layout

We use a series of labelled load-time **`DATA`** instructions at the beginning of each translated program to initialize a portion of memory with virtual function tables and other static data that the translated program needs. We refer to this portion of program memory as *static storage* because it is statically sized and initialized.

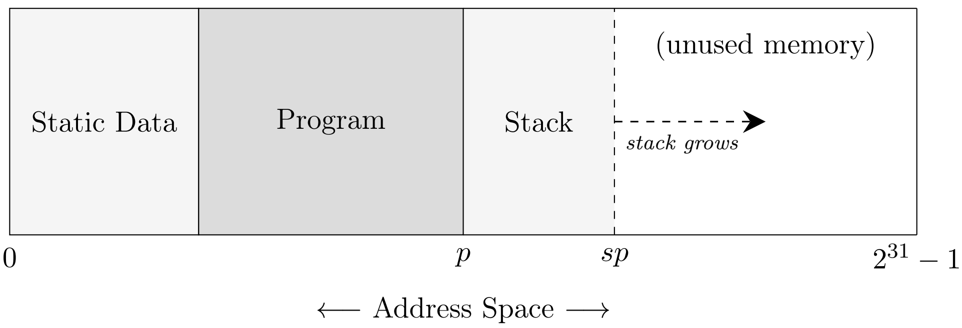

Figure 4.2: Memory layout of a ROOPL program

Figure 4.2 shows the full layout of a ROOPL program in memory:

1. The static storage segment begins at address 0 and contains static data initialised with **DATA** instructions.
2. The program segment is placed just after the static storage segment and contains the actual program instructions which consists mainly of translated class methods.
3. The program stack is placed after the program segment at address p. The stack is a LIFO structure which grows and shrinks as the program executes.

The program stack is used to store activation records, objects and local variables. The stack is accessed with the stack pointer sp and initially $sp = p$.

4.3 Dynamic Dispatch

Dynamic dispatch is a mechanism for selecting which implementation of a method to invoke, based on the type of the associated object at run time.

```
class Shape
    int x
    int y

    method getArea(int out)
    method resize(int scale)
    method translate(int x, int y)
    method draw()

class Rectangle inherits Shape
    int a
    int b

    method getArea(int out)

class Circle inherits Shape
    int radius

    method getArea(int out)
    method getRadius(int out)
```

l_Shape_vt :	**DATA**	90	; Shape::getArea
	DATA	106	; Shape::resize
	DATA	124	; Shape::translate
	DATA	140	; Shape::draw
l_Rectangle_vt :	**DATA**	74	; Rectangle::getArea
	DATA	106	; Shape::resize
	DATA	124	; Shape::translate
	DATA	140	; Shape::draw
l_Circle_vt :	**DATA**	26	; Circle::getArea
	DATA	106	; Shape::resize
	DATA	124	; Shape::translate
	DATA	140	; Shape::draw
	DATA	42	; Circle::getRadius

Figure 4.3: Virtual function table layout for a simple class hierarchy with overridden methods

Since ROOPL allows an object of type τ to be passed to a method expecting an object of type τ' if $\tau \prec: \tau'$, any method calls invoked on the object must be dispatched to the correct implementation in case τ overrides a method in τ'. This can only be done at run time since it is impossible to determine the actual type of an object at compile time.

There are several ways to implement dynamic dispatch but the most common implementation uses virtual function tables (*vtables*) to determine which implementation to dispatch to. Every class in a translated ROOPL program has a vtable which is used to map method names to the memory addresses of the method implementation for that class. Figure 4.3 shows how vtables in ROOPL are arranged for a simple class hierarchy:

- The *Shape* class has no base class and therefore the vtable entries all point to the original (non-overriden) method implementations.
- The *Rectangle* class inherits from *Shape* and overrides the *getArea* method but does not override any other methods. Correspondingly, the vtable points to the overriding implementation of *getArea* but points to the original implementations for the other methods *resize*, *translate* and *draw*.
- The *Circle* class is similar to *Rectangle* but also adds a method *getRadius* which is added to the vtable after the entries for the methods inherited from *Shape*.

When a method is invoked on an object, the vtable is inspected at some statically determined offset. In our example, offset 0 is used for invocations of method *getArea*, offset 1 is used for method *resize*, offset 2 for *translate* and offset 3 for *draw*.

Placing the vtable entry for *getRadius* after the entries for the inherited methods ensures that the inherited methods are placed at the same offsets in the vtable for all subclasses of *Shape*. Therefore if a method is invoked on an object of type *Shape*, the same offset is used to look up the address in the vtable regardless of the actual, dynamic type of the callee object. This technique is known as *prefixing* and it greatly simplifies the translation of polymorphic behaviour. We also utilize prefixing in the memory layout of ROOPL objects for similar benefits.

4.4 Object Layout

Each ROOPL object consists of a pointer to the class vtable followed by a number of memory cells corresponding to the number of instance variables.

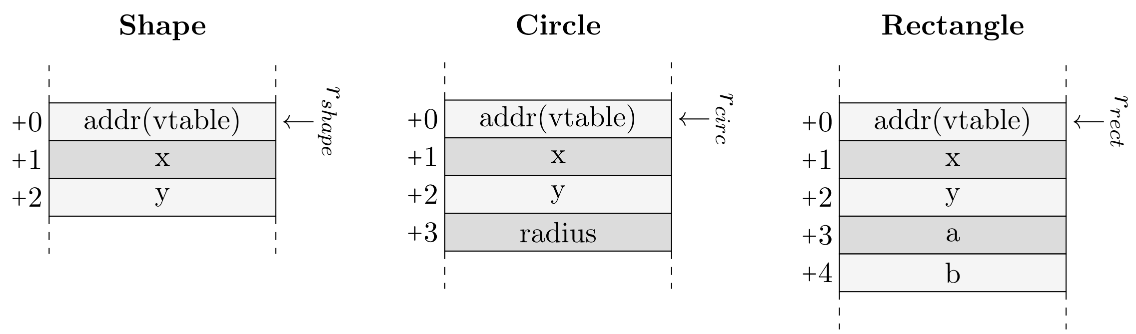

Figure 4.4: Illustration of prefixing in the memory layout of 3 ROOPL objects

Figure 4.4 illustrates the layout of 3 objects based on the class hierarchy from Figure 4.3. When a statement or expression refers to an instance variable, the variable offset is added to the hidden object pointer which is then dereferenced (using **EXCH**) to fetch the value of the instance variable. Again we utilize prefixing to ensure the variable offsets are identical across subclasses of the same type.

Because the class vtable pointer is always stored at offset 0, a vtable lookup is accomplished simply by dereferencing the pointer to the callee object, adding the method offset and then dereferencing the resulting address which yields the memory address of the method implementation.

4.5 Program Structure

The overall structure of a translated ROOPL program is illustrated in Figure 4.5. After the static storage segment follows a series of translated class methods in turn followed by a section of code which acts as the starting point of the program.

	$\cdots\cdots$			; Static data declarations
	$\cdots\cdots$			; Code for program class methods
start :	**START**			; Program starting point
	ADDI	r_{sp}	p	; Initialize stack pointer
	XOR	r_m	r_{sp}	; Store address of main object in r_m
	XORI	r_v	$label_{vt}$	; Store address of vtable in r_v
	EXCH	r_v	r_{sp}	; Push address of vtable onto stack
	ADDI	r_{sp}	$size_m$	; Allocate space for main object
	PUSH	r_m		; Push '*this*' onto stack
	BRA	$label_m$		; Call main procedure
	POP	r_m		; Pop '*this*' from stack
	SUBI	r_{sp}	$size_m$	; Deallocate space of main object
	EXCH	r_v	r_{sp}	; Pop vtable address into r_v
	XORI	r_v	$label_{vt}$	; Clear r_v
	XOR	r_m	r_{sp}	; Clear r_m
	SUBI	r_{sp}	p	; Clear stack pointer
finish :	**FINISH**			; Program exit point

Figure 4.5: Overall layout of a translated ROOPL program

This section is responsible for initializing the stack pointer, allocating an instance of the object containing the main method, calling the main method, deallocating the main object and finally clearing the stack pointer:

The stack pointer is initialized simply by adding the base address of the stack to whichever register r_{sp} should contain the stack pointer. The base address of the stack varies with the size of the translated program but is always known at compile-time - in Figure 4.5 the base address of the stack is simply denoted p. After the stack is in place, we allocate an instance of the main object on the stack by pushing the address of the vtable (denoted $label_{vt}$) onto the stack and adding the size of the object to the stack pointer (denoted $size_m$). We then push the address of this object onto the stack and unconditionally branch to the main method at $label_m$. The address of the main object is popped off the stack by the callee and serves as the object pointer.

After the main method returns, we pop the address of the main object from the stack, deallocate the object and clear the stack pointer. This is done by inverting the steps taken to initialize the stack and the object. After the program terminates, the values of the main object member variables will be left in memory where the stack used to be. This is clearly not an ideal location for the program output to reside, we address this concern in Section 4.14.

4.6 Class Methods

The calling convention described in [2] is a generalized version of the PISA calling convention presented in [18], modified to support recursion. The ROOPL translation uses a similar approach with added support for method parameters (including the hidden object pointer) with pass-by-reference semantics.

(1)	q_{top} :	**BRA**	m_{bot}	
(2)		**POP**	r_{ro}	; Load return offset
(3)		**PUSH**	r_{x_2}	; Restore argument x_2
(4)		**PUSH**	r_{x_1}	; Restore argument x_1
(5)		**PUSH**	r_{this}	; Restor this-pointer
(6)	$label_q$:	**SWAPBR**	r_{ro}	; Method entry and exit point
(7)		**NEG**	r_{ro}	; Negate return offset
(8)		**POP**	r_{this}	; Load this-pointer
(9)		**POP**	r_{x_1}	; Load argument x_1
(10)		**POP**	r_{x_2}	; Load argument x_2
(11)		**PUSH**	r_{ro}	; Store return offset
(12)		······		; Code for method body q_{body}
(13)	q_{bot} :	**BRA**	m_{top}	

Figure 4.6: PISA translation of a ROOPL method

Figure 4.6 shows the PISA translation of a ROOPL method taking two parameters x_1 and x_2, with method body q_{body}. The caller transfers control to instruction **(6)** after which the object-pointer and method arguments are popped off the stack, the return offset is stored and the body is executed. The method prologue works identically for both directions of execution and it works with local method calls (which are simple static branch instructions) and with method calls invoked on other objects (which are dynamically dispatched). This avoids the need for multiple translations of the same method to support reverse execution, which would greatly increase the size of the translated programs.

The **SWAPBR** instruction is used here to facilitate incoming jumps from more than one location, which would otherwise be impossible to achieve with PISA's paired-branch instructions. The return offset is swapped into register r_{ro}, negated (since the return offset is simply the negation of the incoming jump offset) and is then stored on the stack. When the method body finishes, the return offset is swapped back into the branch register, thereby returning the flow of execution to the caller. The arguments and offsets that are accumulated on the program stack during a (possibly nested or recursive) method invocation are cleared as the stack unwinds and the method returns. When the main method call eventually returns, just before the program terminates, the stack will have been returned to its initial, empty state.

4.7 Method Invocations

In ROOPL, method invocations on the current object are always statically dispatched. This behaviour is known as *closed recursion*. The effect of this is that local method invocations in a base class, will always dispatch to the method within that class, even if it has been overridden in a derived class. Using dynamic dispatch semantics for local method invocations (*open recursion*) leads to increased program size, increased execution time and it makes program behaviour harder to reason about[13].

Figure 4.7 shows the translation of local method invocations. The arguments are pushed on the stack in reverse order, followed by the object pointer. The jump itself is performed with an unconditional branch instruction to a statically determined label. After the method returns, the object pointer and the arguments are popped off the stack.

call q **(**x_1, x_2**)**				**uncall** q **(**x_1, x_2**)**			
(1)	**PUSH**	r_{x_2}	; Push x_2 onto stack	(1)	**PUSH**	r_{x_2}	; Push x_2 onto stack
(2)	**PUSH**	r_{x_1}	; Push x_1 onto stack	(2)	**PUSH**	r_{x_1}	; Push x_1 onto stack
(3)	**PUSH**	r_t	; Push '*this*' onto stack	(3)	**PUSH**	r_t	; Push '*this*' onto stack
(4)	**BRA**	$label_q$	; Jump to method	(4)	**RBRA**	$label_q$	; Reverse jump to method
(5)	**POP**	r_t	; Pop '*this*' from stack	(5)	**POP**	r_t	; Pop '*this*' from stack
(6)	**POP**	r_{x_1}	; Pop x_1 from stack	(6)	**POP**	r_{x_1}	; Pop x_1 from stack
(7)	**POP**	r_{x_2}	; Pop x_2 from stack	(7)	**POP**	r_{x_2}	; Pop x_2 from stack

Figure 4.7: PISA translation of local method invocations

Uncalling a method is accomplished with the reverse branch instruction which flips the direction of execution after jumping to the method. Note that since we are using pass-by-reference semantics, we are in fact passing memory addresses as arguments to the method, which in turn points to the locations of the values of x_1 and x_2. The callee is responsible for dereferencing the arguments when they are used in the method body, using the **EXCH** instruction.

Translation of non-local method calls always uses dynamic dispatch, which is slightly more involved than just jumping to a statically determined instruction label. The steps for dynamically dispatching to a method associated with a different object are:

1. Look up the address of the method in the object vtable and create a local copy
2. Calculate the relative jump offset from the method invocation to the method prologue
3. Push the arguments on the stack along with the new object pointer
4. Perform the jump
5. Pop the arguments from the stack
6. Undo the jump offset calculation, to reobtain the absolute address of the method
7. Look up the address of the method in the class vtable again, to clear the local copy

[13] Open recursion also breaks encapsulation and has been identified as the root cause of the *fragile base class problem* [1]

Figure 4.8 shows the translation of a dynamic method call. The first step is to dereference the callee-object to obtain the address of the class vtable. We then look up the address of the method by adding the vtable offset ($offset_q$) to the vtable address.

Note how this lookup involves *swapping* the address stored in the vtable in static memory with the value of a register. This means the vtable is in fact altered and we need to return it to its original state before we perform the jump, since the callee might need to lookup the same method address later on. We can restore the vtable with a Lecerf-reversal by creating a copy of the method address in a register, and then undoing the lookup thereby swapping the original method address back into the vtable.

call x**::**q**(**x_1, x_2**)**

(1)		**EXCH**	r_v	r_x	; Get address of vtable
(2)		**ADDI**	r_v	$offset_q$	; Lookup q in vtable
(3)		**EXCH**	r_t	r_v	; Get address of q
(4)		**XOR**	r_{tgt}	r_t	; Copy address of q
(5)		**EXCH**	r_t	r_v	; Place address back in vtable
(6)		**SUBI**	r_v	$offset_q$	; Restore vtable pointer
(7)		**EXCH**	r_v	r_x	; Restore object pointer
(8)		**PUSH**	r_{x_2}		; Push x_2 onto stack
(9)		**PUSH**	r_{x_1}		; Push x_1 onto stack
(10)		**PUSH**	r_x		; Push new '*this*' onto stack
(11)		**SUBI**	r_{tgt}	$label_{jmp}$	; Calculate jump offset
(12)	$label_{jmp}$ **:**	**SWAPBR**	r_{tgt}		; Jump to method
(13)		**NEG**	r_{tgt}		; Restore r_{tgt} to original value
(14)		**ADDI**	r_{tgt}	$label_{jmp}$	; Restore absolute jump value
(15)		**POP**	r_x		; Pop new '*this*' from stack
(16)		**POP**	r_{x_1}		; Pop x_1 from stack
(17)		**POP**	r_{x_2}		; Pop x_2 from stack
(18)		**EXCH**	r_v	r_x	; Get address of vtable
(19)		**ADDI**	r_v	$offset_q$	; Lookup q in vtable
(20)		**EXCH**	r_t	r_v	; Get address of q
(21)		**XOR**	r_{tgt}	r_t	; Clear address of q
(22)		**EXCH**	r_t	r_v	; Place address back in vtable
(23)		**SUBI**	r_v	$offset_q$	; Restore vtable pointer
(24)		**EXCH**	r_v	r_x	; Restore object pointer

Figure 4.8: PISA translation of a non-local method invocation

Since the usual branch instructions (**BRA**, **RBRA**, et cetera) can only jump to static instruction labels, we must use the **SWAPBR** instruction to swap the jump offset into the branch register. Because the vtable only stores absolute method addresses, we have to calculate the jump offset manually for each method call. We can accomplish this by subtracting the memory address of the **SWAPBR** instruction from the method address.

After the method returns, we negate the jump offset (to cancel out the negation done by the callee in the method prologue) and add the address of the **SWAPBR** instruction to the jump offset to obtain the original absolute value of the method. To avoid leaving this method address in a register or on the stack as garbage data, we repeat the vtable lookup to clear the local method address copy. In total, the vtable is consulted 4 times per method invocation.

uncall x**::**q**(**x_1, x_2**)**

(11)		**SUBI**	r_{tgt}	$label_{jmp}$	; Calculate jump offset
(12)	top_{jmp}	**RBRA**	bot_{jmp}		; Flip direction
(13)	$label_{jmp}$ **:**	**SWAPBR**	r_{tgt}		; Jump to method
(14)		**NEG**	r_{tgt}		; Restore r_{tgt} to original value
(15)	bot_{jmp}	**BRA**	top_{jmp}		; Paired branch
(16)		**ADDI**	r_{tgt}	$label_{jmp}$	; Restore absolute jump value

Figure 4.9: PISA translation of a non-local reverse method invocation

Uncalling a non-local method is analogous to calling a non-local method, with the added caveat that the direction of execution should be reversed before the jump occurs. Unlike BobISA (which has the **RSWB** instruction, see Section 2.5 in Chapter 2), PISA does not have a single instruction which swaps the branch register and flips the direction bit simultaneously. Figure 4.9 shows how this is instead accomplished with an **RBRA**/**BRA** pair. The vtable lookup and cleanup is identical to the approach used in Figure 4.8.

4.8 Object Blocks

Since the stack is maintained over (but not during) execution of a statement, we can store ROOPL objects on the program stack. The execution of an object block begins with allocation of a new object on the top of the stack. Then the block statement is executed, after which the object will again be on the top of the stack, ready for deallocation.

construct $c\ x \quad s$ **destruct** x

(1)	**XOR**	r_x	r_{sp}	; Store address of new object x in r_x
(2)	**XORI**	r_v	$label_{vt}$	; Store address of vtable in r_v
(3)	**EXCH**	r_v	r_{sp}	; Push address of vtable onto stack
(4)	**ADDI**	r_{sp}	$size_c$	; Allocate space for new object
(5)	······			; Code for statement s
(6)	**SUBI**	r_{sp}	$size_c$	; Deallocate space occupied by zero-cleared object
(7)	**EXCH**	r_v	r_{sp}	; Pop vtable address into r_v
(8)	**XORI**	r_v	$label_{vt}$	; Clear r_v
(9)	**XOR**	r_x	r_{sp}	; Clear r_x

Figure 4.10: PISA translation of an object block

Figure 4.10 illustrates how this is accomplished in practice. The immediate $label_{vt}$ is the address of the vtable for class c and $size_c$ is the size of the class. The size of a class is the number of instance variables plus 1, for accomodating the vtable pointer. Within the block statement s, the register r_x contains the address of the new object x.

4.9 Local Blocks

Figure 4.11 shows the translation of a local integer block. Local blocks are not part of the core language (See Section 3.10.1 in Chapter 3), but are included as a language extension, borrowed from Janus.

$$\textbf{local int } x \ = \ e_1 \quad s \quad \textbf{delocal } x \ = \ e_2$$

(1)	······			; Code for $r_e \leftarrow [\![e_1]\!]$
(2)	`XOR`	r_x	r_{sp}	; Store address of new integer x in r_x
(3)	`XOR`	r_t	r_e	; Copy value of e_1 into r_t
(4)	`PUSH`	r_t		; Push value of e_1 onto stack
(5)	······			; Inverse of **(1)**
(6)	······			; Code for statement s
(7)	······			; Code for $r_e \leftarrow [\![e_2]\!]$
(8)	`POP`	r_t		; Pop value of x into r_t
(9)	`XOR`	r_t	r_e	; Clear value of r_t with r_e
(10)	`XOR`	r_x	r_{sp}	; Clear reference to x
(11)	······			; Inverse of **(7)**

Figure 4.11: PISA translation of a local block

Again, the translation can take advantage of the fact that the program stack is preserved over statement execution. This means we can place the local integers on the stack and pop them off after the block statement has been executed. Local integers are initialized with some expression e_1 and zero-cleared with another expression e_2. Evaluation of an irreversible expression in a reversible assembly language is bound to generate some amount of garbage data so we use a Lecerf-reversal to uncompute this garbage data after initializing the local variable with e_1, and again after clearing the local variable with e_2.

4.10 Control Flow

At the level of assembly language, control flow statements are usually realized via direct alteration of the program counter, which is clearly not an option for a translation targeting a reversible instruction set such as PISA. Another complication arises in the evaluation of the expressions acting as entry and exit conditions, since ROOPL expressions are irreversible.

Axelsen suggests a simple approach for arranging the translation of Janus CFOs in such a way that the garbage data produced by evaluation of the entry and exit expressions can be uncomputed without significant code duplication [2]. Since Janus (and ROOPL) uses the value 0 for the boolean value *false* and non-zero for the boolean value *true*, we can safely reduce the result of evaluating the entry and exit expressions to either 0 or 1 while still preserving the semantics of the source program.

if e_1 **then** s_1 **else** s_2 **fi** e_2

(1)		$\cdots\cdots$		; Code for $r_e \leftarrow [\![e_1]\!]_c$
(2)		**XOR**	$r_t\ r_e$	; Copy value of e_1 into r_t
(3)		$\cdots\cdots$		; Inverse of **(1)**
(4)	$test$:	**BEQ**	$r_t\ r_0\ test_{false}$	; Jump if $e_1 = 0$
(5)		**XORI**	r_t 1	; Clear r_t
(6)		$\cdots\cdots$		; Code for statement s_1
(7)		**XORI**	r_t 1	; Set $r_t = 1$
(8)	$assert_{true}$:	**BRA**	$assert$	; Jump
(9)	$test_{false}$:	**BRA**	$test$	; Receive jump
(10)		$\cdots\cdots$		; Code for statement s_2
(11)	$assert$:	**BNE**	$r_t\ r_0\ assert_{true}$	; Receive jump
(12)		$\cdots\cdots$		; Code for $r_e \leftarrow [\![e_2]\!]_c$
(13)		**XOR**	$r_t\ r_e$	; Clear r_t
(14)		$\cdots\cdots$		; Inverse of **(12)**

from e_1 **do** s_1 **loop** s_2 **until** e_2

(1)		**XORI**	r_t 1	; Set $r_t = 1$
(2)	$entry$:	**BEQ**	$r_t\ r_0\ assert$	; Receive jump
(3)		$\cdots\cdots$		; Code for $r_e \leftarrow [\![e_1]\!]_c$
(4)		**XOR**	$r_t\ r_e$	; Clear r_t
(5)		$\cdots\cdots$		; Inverse of **(3)**
(6)		$\cdots\cdots$		; Code for statement s_1
(7)		$\cdots\cdots$		; Code for $r_e \leftarrow [\![e_2]\!]_c$
(8)		**XOR**	$r_t\ r_e$	; Copy value of e_2 into r_t
(9)		$\cdots\cdots$		; Inverse of **(7)**
(10)	$test$:	**BNE**	$r_t\ r_0\ exit$	; Exit if $e_2 = 1$
(11)		$\cdots\cdots$		; Code for statement s_2
(12)	$assert$:	**BRA**	$entry$	; Jump to top
(13)		**XORI**	r_t 1	; Clear r_t

Figure 4.12: PISA translation of conditonals (left) and loops (right), from [2]

This allows us to perform the uncomputation of the expression evaluation (which clears extraneous garbage data) *before* the branch is executed, while still being able to subsequently clear the register holding the result of the evaluation. Conditional statements and loops in ROOPL are essentially identical to those in Janus and this approach is therefore perfectly suitable for our ROOPL to PISA translation. Figure 4.12 shows the translation of both conditional statements and loops.

4.11 Reversible Updates

Figure 4.13 shows the translation of reversible variable updates and variable swapping. Since PISA does not have a built-in register swap instruction, we use the classic XOR-swap to exchange the contents of the two registers reversibly.

x_1 **<=>** x_2

(1)	**XOR**	r_{x_1}	r_{x_2}
(2)	**XOR**	r_{x_2}	r_{x_1}
(3)	**XOR**	r_{x_1}	r_{x_2}

$x \odot$**=** e

(1)	$\cdots\cdots$			; Code for $r_e \leftarrow [\![e]\!]$
(2)	$[\![\odot]\!]_i$	r_x	r_e	; Assign e to x
(3)	$\cdots\cdots$			; Inverse of **(1)**

Figure 4.13: PISA translation of variable updates and variable swapping

Variable updates are accomplished with one of three instructions as well as an expression evaluation which is reversed after the update, in order to clear any accumulated garbage data. The update instruction in **(2)** is given by the function $[\![\odot]\!]_i : \mathit{ModOps} \rightarrow \mathit{Instructions}$:

$$[\![\texttt{+}]\!]_i = \texttt{ADD} \qquad [\![\texttt{-}]\!]_i = \texttt{SUB} \qquad [\![\texttt{\^{}}]\!]_i = \texttt{XOR}$$

See Section 3.1 in Chapter 3 and Section 2.4 in Chapter 2 for the ROOPL and PISA syntax domains.

4.12 Expression Evaluation

When implementing evaluation of irreversible expressions in a reversible language, we have to accept the generation of some garbage data. Since ROOPL expressions are irreversible, every evaluation of an expression must be accompanied by a subsequent *unevaluation* in order to clear any accumulated garbage data in registers and memory. This technique keeps the translation clean at the statement-level.

Code generation for evaluation of expressions is done by recursive descent over the structure of the expression tree. Numerical constants, variables and **`nil`**-nodes represent the base cases while binary expressions represent the recursive cases. A few of the binary operators supported in ROOPL (such as addition and bitwise exclusive-or) have single-instruction equivalents in PISA, but most operators are translated to more than one PISA instruction.

We consider the issue of register allocation for expression evaluation to be outside the scope of our translation. See [2, Section 4.5] for an examination of reversible register allocation in PISA. A novel approach for reducing register pressure, by leveraging reversible computations to recompute registers instead of spilling them to memory, is presented in [5].

4.13 Error Handling

Aside from being syntactically correct and well-typed, a ROOPL program is required to meet a number of conditions that cannot, in general, be determined at compile time:

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

It is entirely up to the programmer to make sure these conditions are met by the program. If either of these conditions are not met, the program will silently continue with erroneous execution. To avoid such a situation, we can insert run time error checks that terminates the program or jumps to some error handler in case of programmer error.

Figure 4.14 shows the translation of a local integer block with added dynamic error checks. In case the value of the local integer v_i does not match the value of the **`delocal`**-expression v_e, the register r_t will contain the non-zero value $v_i \oplus v_e$ at instruction **`(13)`**. If this is the case, we jump to an error routine at $label_{error}$.

The error check at **`(1)`** serves the same purpose as its counterpart, when the flow of execution is reversed, but has no effect otherwise since r_t is empty before the statement is executed.

$$\textbf{local int } x \ = \ e_1 \quad s \quad \textbf{delocal } x \ = \ e_2$$

(1)	**BNE**	r_t	r_0	$label_{error}$	; Dynamic error check
(2)	······				; Code for $r_e \leftarrow [\![e_1]\!]$
(3)	**XOR**	r_x	r_{sp}		; Store address of new integer x in r_x
(4)	**XOR**	r_t	r_e		; Copy value of e_1 into r_t
(5)	**PUSH**	r_t			; Push value of e_1 onto stack
(6)	······				; Inverse of **(1)**
(7)	······				; Code for statement s
(8)	······				; Code for $r_e \leftarrow [\![e_2]\!]$
(9)	**POP**	r_t			; Pop value of x into r_t
(10)	**XOR**	r_t	r_e		; Clear value of r_t with r_e
(11)	**XOR**	r_x	r_{sp}		; Clear reference to x
(12)	······				; Inverse of **(7)**
(13)	**BNE**	r_t	r_0	$label_{error}$	; Dynamic error check

Figure 4.14: PISA translation of a local block, with run time error checking

Dynamic error checks for conditionals, loops and object blocks can be implemented using a similar technique.

4.14 Implementation

We implemented a ROOPL compiler (ROOPLC), utilizing the techniques presented in the preceding sections. The compiler serves as a proof-of-concept and does not perform any optimization of the target programs whatsoever. ROOPLC is written in Haskell (GHC, version 7.10.3) and the output was tested using the PendVM Pendulum simulator [16].

Appendix A contains the source code listings for the ROOPL compiler and Appendix B contains an example ROOPL program and the corresponding translated PISA program. The source code for the ROOPL compiler, additional test programs and the C source code for the PendVM simulator are also included in the enclosed ZIP archive.

The ROOPL compiler follows the PISA conventions that register r_0 is preserved as 0, r_1 contains the stack pointer and r_2 stores return offsets for method invocations. Additionally, the compiler will always use r_3 to store the object pointer. The remaining 28 general purpose registers are used for variables, parameters and intermediate expression evaluation results.

In ROOPL, the class fields of the main class act as the program output. The program prelude, as described in Section 4.5, leaves the value of these variables on the program stack after the program terminates. For the sake of convenience, the compiler instead copies these values from the program stack to static memory before termination. The compiler is structured as 6 separate compilation phases:

1. Parsing The parsing phase transforms the input program from textual representation to an abstract syntax tree. The parser was implemented using the monadic parser combinators from the `Text.Parsec` library. See Section 3.1 for details on the ROOPL syntax.

2. Class Analysis The class analysis phase verifies a number of properties of the classes in the program: Inheritance cycle detection, duplicate method names, duplicate field names and unknown base classes. The class analysis phase also computes the size of each class and constructs tables mapping class names to methods, instance variables et cetera.

3. Scope Analysis The scope analysis phase maps every occurrence of every identifier to a unique variable or method declaration. The scope analysis phase is also responsible for constructing the class virtual tables and the symbol table.

4. Type Checking The type checker uses the symbol table and the abstract syntax tree to verify that the program satisfies the ROOPL type system, as described in Section 3.7.

5. Code Generation The code generation phase translates the abstract syntax tree to a series of PISA instructions in accordance with the code generation schemes presented in this chapter. Rudimentary register allocation is also handled during code generation.

6. Macro Expansion The macro expansion phase is responsible for expanding macros left in the translated PISA program after code generation and for final processing of the output.

The size blowup from ROOPL to PISA is by a factor of 10 to 15 in terms of LOC. The nature of the target programs suggest that basic peephole optimization could reduce program size drastically.

CHAPTER 5

Conclusion

We described and formalized the reversible object-oriented programming language ROOPL and we discussed the considerations that went into its design. The language extends the design of existing imperative reversible languages in the literature and represents the first effort towards introducing OOP methodology to the field of reversible computing.

The combination of reversible computing and object-oriented programming is entirely uncharted territory and we identified the most interesting or novel points of intersection between the two disciplines, such as reversible class mutators and the proposed constructor/deconstructor extension.

Since ROOPL is the first imperative reversible language with non-trivial user-defined data types, we presented a complete static type system for the language and proved that well-typedness is preserved over statement inversion. We also demonstrated the computational strength of the language by implementing a reversible Turing machine simulator.

Finally, we established the techniques required for a clean translation from ROOPL to the reversible low-level machine language PISA and we demonstrated the feasibility of supporting core OOP features such as class inheritance and subtype polymorphism in a reversible programming language, by means of object layout prefixing and virtual function tables. We created a proof-of-concept compiler which fully implements our translation techniques.

If reversible computing is to contend with conventional computing models, we need reversibility at every level of abstraction. To this end, much has been accomplished at the circuit, gate and machine levels but aside from the work on reversible functional programming, there is little on offer in terms of high level languages and abstractions. The work presented in this thesis is a step in the direction of reconciling the abstraction techniques of conventional programming languages with the reversible programming paradigm. With ROOPL we have demonstrated that reversible object-oriented programming languages are both possible and practical.

5.1 Future Work

In order to move away from the syntactically coupled allocation and deallocation mechanics used in ROOPL, more work is needed on the topics of reversible memory heaps and reversible dynamic memory management. Some work has already been done on these topics with regards to reversible functional languages [3, 33, 34].

ROOPL offers only the minimal toolset necessary for object-oriented programming. Advanced OOP features such as mixins, traits and generic classes could also prove to be useful in a reversible programming language and the implementation of such features could be the subject of further work.

Compilation of reversible languages is still in its infancy and the existing body of work focuses exclusively on correctness and avoiding garbage data. The practicality of reversible languages depends in part on compilation techniques that are not only correct but also *performant*, both in terms of execution time and program size. In particular, optimization techniques that utilize the bidirecitonal nature of reversible programs to reduce code size shows promise and there is need for general and well-performing solutions to the reversible register allocation problem.

References

[1] Aldrich, J. “Selective Open Recursion: Modular Reasoning about Components and Inheritance”. In: *FSE 2004 Workshop on Specification and Verification of Component-Based Systems*. 2004.

[2] Axelsen, H. B. “Clean Translation of an Imperative Reversible Programming Language”. In: *Proceedings of the 20th International Conference on Compiler Construction: Part of the Joint European Conferences on Theory and Practice of Software*. Springer-Verlag, 2011, pp. 144–163.

[3] Axelsen, H. B. and Glück, R. “Reversible Representation and Manipulation of Constructor Terms in the Heap”. In: *Proceedings of the 5th International Conference on Reversible Computation*. Springer-Verlag, 2013, pp. 96–109.

[4] 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 Diekert, V., Volkov, M. V., and Voronkov, A. Springer Berlin Heidelberg, 2007, pp. 56–69.

[5] Bahi, M. and Eisenbeis, C. “Rematerialization-based Register Allocation Through Reverse Computing”. In: *Proceedings of the 8th ACM International Conference on Computing Frontiers*. ACM, 2011, 24:1–24:2.

[6] Bennet, C. H. “Logical Reversibility of Computation”. In: *IBM Journal of Research and Development* 17.6 (1973), pp. 525–532.

[7] Bennet, C. H. “The Thermodynamics of Computation - a Review”. In: *International Journal of Theoretical Physics* 21.12 (1982), pp. 905–940.

[8] Bennet, C. H. “Time/Space Trade-offs for Reversible Computation”. In: *SIAM Journal on Computing* 18.4 (1989), pp. 766–776.

[9] Bennet, C. H. “Notes on Landauer’s Principle, Reversible Computation, and Maxwell’s Demon”. In: *Studies in History and Philosophy of Modern Physics* 34 (2003), pp. 501–510.

[10] Birtwistle, G. M. et al. *SIMULA BEGIN*. Philadelphia, PA: AUERBACH Publishers, 1973.

[11] Blaschek, G. “Type-Safe Object-Oriented Programming with Prototypes - The Concepts of Omega”. In: *Structured Programming* 12 (1991), pp. 217–225.

[12] Bocharov, A. and Svore, K. M. “From Reversible Logic Gates to Universal Quantum Bases”. In: *European Association of Theoretical Computer Science*. Vol. 110. 2013, pp. 79–85.

[13] Carøe, M. K. “Design of Reversible Computing Systems”. PhD Thesis. University of Copenhagen, DIKU, 2012.

[14] Carothers, C. D., Perumalla, K. S., and Fujimoto, R. M. “Efficient Optimistic Parallel Simulations Using Reverse Computation”. In: *ACM Transactions on Modeling and Computer Simulations* 9.3 (1999), pp. 224–253.

[15] Cezzar, R. “Design of a Processor Architecture Capable of Forward and Reverse Execution”. In: vol. 2. 1991, pp. 885–890.

[16] Clark, C. R. *Improving the Reversible Programming Language R and its Supporting Tools*. Senior Project. 2001. URL: http://www.cise.ufl.edu/research/revcomp/users/cclark/pendvm-fall2001/.

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

[18] Frank, M. P. “Reversibility for Efficient Computing”. PhD Thesis. Massachusetts Institute of Technology, 1999.

[19] Fredkin, E. and Toffoli, T. “Conservative Logic”. In: *International Journal of Theoretical Physics* 21.3 (1982), pp. 219–253.

[20] Gamma, E. et al. *Design Patterns: Elements of Reusable Object-oriented Software*. Boston, MA: Addison-Wesley Longman Publishing, 1995.

[21] Glück, R. and Kawabe, M. “A Program Inverter for a Functional Language with Equality and Constructors”. In: *Programming Languages and Systems: First Asian Symposium, APLAS 2003, Beijing, China, November 27-29, 2003. Proceedings*. Ed. by Ohori, A. Springer Berlin Heidelberg, 2003, pp. 246–264.

[22] Glück, R. and Kawabe, M. “Derivation of Deterministic Inverse Programs Based on LR Parsing”. In: *Functional and Logic Programming: 7th International Symposium, FLOPS 2004, Nara, Japan, April 7-9, 2004. Proceedings*. Ed. by Kameyama, Y. and Stuckey, P. J. Springer Berlin Heidelberg, 2004, pp. 291–306.

[23] Glück, R. and Yokoyama, T. “A Linear-Time Self-Interpreter of a Reversible Imperative Language”. In: vol. 33. 3. Japan Society for Software Science and Technology, 2016, pp. 108–128.

[24] Hall, J. “A Reversible Instruction Set Architecture and Algorithms”. In: *Physics and Computation, 1994. PhysComp ’94, Proceedings., Workshop on* (1994), pp. 128–134.

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

[26] Hunt, A. and Thomas, D. *The Pragmatic Programmer: From Journeyman to Master*. Boston, MA: Addison-Wesley Longman Publishing, 1999.

[27] Jefferson, D. and Sowizral, H. A. “Fast Concurrent Simulation Using the Time Warp Mechanism: Part I, Local Control”. In: (1982).

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

[29] Lecerf, Y. “Machines de Turing réversibles. Insolubilité récursive en $n \in N$ de l’équation $u = \Theta^n u$, où Θ est un "isomorphisme de codes"”. In: *Comptes Rendus Hebdomadaires des Séances de L’académie des Sciences* 257 (1963), pp. 2597–2600.

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

[31] McCarthy, J. “The Inversion of Functions Defined by Turing Machines”. In: *Automata Studies, Annals of Mathematical Studies*. Ed. by Shannon, C. E. and McCarthy, J. 34. Princeton University Press, 1956, pp. 177–181.

[32] Mogensen, T. Æ. "Partial Evaluation of the Reversible Language Janus". In: *Proceedings of the 20th ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation*. ACM, 2011, pp. 23–32.

[33] 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 Yamashita, S. and Minato, S.-i. Springer International Publishing, 2014, pp. 82–94.

[34] 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 Krivine, J. and Stefani, J.-B. Springer International Publishing, 2015, pp. 79–94.

[35] Mogensen, T. Æ. "An Investigation of Scoping and Parameter-Passing in Janus-Like Languages". University of Copenhagen, DIKU. 2016.

[36] Moore, G. E. "Cramming More Components onto Integrated Circuits". In: *Electronics* 38.8 (1965), pp. 114–117.

[37] Mu, S.-C., Hu, Z., and Takeichi, M. "An Injective Language for Reversible Computation". In: *Mathematics of Program Construction: 7th International Conference, MPC 2004, Stirling, Scotland, UK, July 12-14, 2004. Proceedings*. Ed. by Kozen, D. Springer Berlin Heidelberg, 2004, pp. 289–313.

[38] Nishida, N., Palacios, A., and Vidal, G. "Towards Reversible Computation in Erlang". In: *LOPSTR'16: 26th International Symposium on Logic-based Program Synthesis and Transformation*. 2016.

[39] Perlis, A. J. "Epigrams on Programming". In: *SIGPLAN Notices* 17.9 (1982), pp. 7–13.

[40] Schultz, U. P. et al. "Towards a Domain-Specific Language for Reversible Assembly Sequences". In: *Reversible Computation: 7th International Conference, RC 2015, Grenoble, France, July 16-17, 2015, Proceedings*. Ed. by Krivine, J. and Stefani, J.-B. Springer International Publishing, 2015, pp. 111–126.

[41] 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 Vos, A. D. and Wille, R. Springer Berlin Heidelberg, 2011, pp. 30–42.

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

[43] Vieri, C. J. "Reversible Computer Engineering and Architecture". PhD Thesis. Massachusetts Institute of Technology, 1999.

[44] Vieri, C. J. et al. "A Fully Reversible Asymptotically Zero Energy Microprocessor". In: *Proceedings of the ISCA Workshop* (1998).

[45] Winskel, G. *The Formal Semantics of Programming Languages: An Introduction*. Cambridge, MA: MIT Press, 1993.

[46] Yokoyama, T., Axelsen, H. B., and Glück, R. "Principles of a Reversible Programming Language". In: *Proceedings of the 5th Conference on Computing Frontiers*. ACM, 2008, pp. 43–54.

[47] Yokoyama, T., Axelsen, H. B., and Glück, R. "Reversible Flowchart Languages and the Structured Reversible Program Theorem". In: *Automata, Languages and Programming: 35th International Colloquium, ICALP 2008, Reykjavik, Iceland, July 7-11, 2008, Proceedings, Part II*. Ed. by Aceto, L. et al. Springer Berlin Heidelberg, 2008, pp. 258–270.

[48] Yokoyama, T., Axelsen, H. B., and Glück, R. "Towards a Reversible Functional Language". In: *RC 11: Proceedings of the Third International Conference on Reversible Computation*. Ed. by Vos, A. D. and Wille, R. Springer Berlin Heidelberg, 2011, pp. 14–29.

[49] Yokoyama, T. and Glück, R. "A Reversible Programming Language and its Invertible Self-Interpreter". In: *PEPM 2007: Proceedings of the Workshop on Partial Evaluation and Program Manipulation*. ACM, 2007, pp. 144–153.

APPENDIX A

ROOPLC Source Code

A.1 AST.hs

```
module AST where

{-- AST Primitives --}

type TypeName = String
type MethodName = String

data DataType = IntegerType
              | ObjectType TypeName
              | NilType
    deriving (Show)

instance Eq DataType where
    IntegerType     == IntegerType     = True
    NilType         == NilType         = True
    NilType         == (ObjectType _)  = True
    (ObjectType _)  == NilType         = True
    (ObjectType t1) == (ObjectType t2) = t1 == t2
    _               == _               = False

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
```

```haskell
                | Nil
                | Binary BinOp (GExpr v) (GExpr v)
    deriving (Show, Eq)

--Statements
data GStmt m v = Assign v ModOp (GExpr v)
               | Swap v 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 v (GExpr v) [GStmt m v] (GExpr v)
               | LocalCall m [v]
               | LocalUncall m [v]
               | ObjectCall v MethodName [v]
               | ObjectUncall v MethodName [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, base class, fields, methods
data GCDecl m v = GCDecl TypeName (Maybe TypeName) [GDecl v] [GMDecl m v]
    deriving (Show, Eq)

--Program
data 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)]
```

A.2 PISA.hs

```haskell
{-# LANGUAGE FlexibleInstances, TypeSynonymInstances #-}

module PISA where

import Data.List (intercalate)
import Control.Arrow

import AST (TypeName, MethodName)

type Label = String

data 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
              | RL Register i
              | RLV Register Register
              | RR Register i
              | RRV Register Register
              | SLLX Register Register i
              | SLLVX Register Register Register
              | SRAX Register Register i
              | SRAVX Register Register Register
              | SRLX Register Register i
              | 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
              | BNE Register Register Label
              | BRA Label
              | EXCH Register Register
              | SWAPBR Register
              | RBRA Label
              | START
              | FINISH
              | DATA i
              | SUBI Register i --Pseudo
    deriving (Eq)

data GProg i = GProg [(Maybe Label, GInstr i)]

{-- Macro PISA Definitions --}

data Macro = Immediate Integer
           | AddressMacro Label
           | SizeMacro TypeName
           | OffsetMacro TypeName MethodName
           | ProgramSize
    deriving (Show, Eq)
```

```haskell

type MInstruction = GInstr Macro
type MProgram = GProg Macro

invertInstructions :: [(Maybe Label, MInstruction)] -> [(Maybe Label, MInstruction)]
invertInstructions = reverse . map (second invertInstruction . first (fmap (++ "_i")))
    where invertInstruction (ADD r1 r2) = SUB r1 r2
          invertInstruction (SUB r1 r2) = ADD r1 r2
          invertInstruction (ADDI r i) = SUBI r i
          invertInstruction (SUBI r i) = ADDI r i
          invertInstruction (RL r i) = RR r i
          invertInstruction (RLV r1 r2) = RRV r1 r2
          invertInstruction (RR r i) = RL r i
          invertInstruction (RRV r1 r2) = RLV r1 r2
          invertInstruction (BEQ r1 r2 l) = BEQ r1 r2 $ l ++ "_i"
          invertInstruction (BGEZ r l) = BGEZ r $ l ++ "_i"
          invertInstruction (BGTZ r l) = BGTZ r $ l ++ "_i"
          invertInstruction (BLEZ r l) = BLEZ r $ l ++ "_i"
          invertInstruction (BLTZ r l) = BLTZ r $ l ++ "_i"
          invertInstruction (BNE r1 r2 l) = BNE r1 r2 $ l ++ "_i"
          invertInstruction (BRA l) = BRA $ l ++ "_i"
          invertInstruction (RBRA l) = RBRA $ l ++ "_i"
          invertInstruction inst = inst

{-- Output PISA Definitions --}

type Instruction = GInstr Integer
type Program = GProg Integer

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

instance Show Instruction where
    show (ADD r1 r2) = unwords ["ADD   ", show r1, show r2]
    show (ADDI r i) = unwords ["ADDI  ", show r, show i]
    show (ANDX r1 r2 r3) = unwords ["ANDX  ", show r1, show r2, show r3]
    show (ANDIX r1 r2 i) = unwords ["ANDIX ", 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 (ORX r1 r2 r3) = unwords ["ORX   ", show r1, show r2, show r3]
    show (ORIX r1 r2 i) = unwords ["ORIX  ", show r1, show r2, show i]
    show (RL r i) = unwords ["RL    ", show r, show i]
    show (RLV r1 r2) = unwords ["RLV   ", show r1, show r2]
    show (RR r i) = unwords ["RR    ", show r, show i]
    show (RRV r1 r2) = unwords ["RRV   ", show r1, show r2]
    show (SLLX r1 r2 i) = unwords ["SLLX  ", show r1, show r2, show i]
    show (SLLVX r1 r2 r3) = unwords ["SLLVX ", show r1, show r2, show r3]
    show (SRAX r1 r2 i) = unwords ["SRAX  ", show r1, show r2, show i]
    show (SRAVX r1 r2 r3) = unwords ["SRAVX ", show r1, show r2, show r3]
    show (SRLX r1 r2 i) = unwords ["SRLX  ", show r1, show r2, show i]
    show (SRLVX r1 r2 r3) = unwords ["SRLVX ", show r1, show r2, show r3]
    show (SUB r1 r2) = unwords ["SUB   ", show r1, show r2]
    show (XOR r1 r2) = unwords ["XOR   ", show r1, show r2]
    show (XORI r i) = unwords ["XORI  ", show r, show i]
    show (BEQ r1 r2 l) = unwords ["BEQ   ", show r1, show r2, l]
    show (BGEZ r l) = unwords ["BGEZ  ", show r, l]
    show (BGTZ r l) = unwords ["BGTZ  ", show r, l]
    show (BLEZ r l) = unwords ["BLEZ  ", show r, l]
    show (BLTZ r l) = unwords ["BLTZ  ", show r, l]
    show (BNE r1 r2 l) = unwords ["BNE   ", show r1, show r2, l]
    show (BRA l) = unwords ["BRA   ", l]
    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 file p = writeFile file $ showProgram p
```

A.3 Parser.hs

```haskell
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 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"]

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

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 }

```

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

{-- Parser Primitives --}

identifier :: Parser String
identifier = Token.identifier tokenParser

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

colon :: Parser String
colon = Token.colon tokenParser

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

typeName :: Parser TypeName
typeName = identifier

methodName :: Parser MethodName
methodName = identifier

{-- Expression Parsers --}

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

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

nil :: Parser Expression
nil = Nil <$ reserved "nil"

expression :: Parser Expression
expression = buildExpressionParser opTable $ constant <|> 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

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

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"
    <*> block
    <* reserved "until"
    <*> expression

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

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

objectCall :: Parser Statement
objectCall =
    reserved "call"
    >> ObjectCall
    <$> identifier
    <* colon
    <* colon
    <*> methodName
    <*> parens (commaSep identifier)

objectUncall :: Parser Statement
objectUncall =
    reserved "uncall"
    >> ObjectUncall
    <$> identifier
    <* colon
    <* colon
    <*> methodName
    <*> parens (commaSep identifier)

localBlock :: Parser Statement
localBlock =
    reserved "local"
    >> reserved "int"
    >> LocalBlock
    <$> identifier
    <* symbol "="
    <*> expression
    <*> block
    <* reserved "delocal"
```

```
    <* identifier
    <* symbol "="
    <*> expression

objectBlock :: Parser Statement
objectBlock =
    reserved "construct"
    >> ObjectBlock
    <$> typeName
    <*> identifier
    <*> block
    <* reserved "destruct"
    <* identifier

skip :: Parser Statement
skip = Skip <$ reserved "skip"

statement :: Parser Statement
statement = try assign
        <|> swap
        <|> conditional
        <|> loop
        <|> try localCall
        <|> try localUncall
        <|> objectCall
        <|> objectUncall
        <|> localBlock
        <|> objectBlock
        <|> skip

block :: Parser [Statement]
block = many1 statement

{-- Top Level Parsers --}

dataType :: Parser DataType
dataType = IntegerType <$ reserved "int" <|> ObjectType <$> typeName

variableDeclaration :: Parser VariableDeclaration
variableDeclaration = GDecl <$> dataType <*> identifier

methodDeclaration :: Parser MethodDeclaration
methodDeclaration =
    reserved "method"
    >> GMDecl
    <$> methodName
    <*> parens (commaSep variableDeclaration)
    <*> block

classDeclaration :: Parser ClassDeclaration
classDeclaration =
    reserved "class"
    >> GCDecl
    <$> typeName
    <*> optionMaybe (reserved "inherits" >> typeName)
    <*> many variableDeclaration
    <*> many1 methodDeclaration

program :: Parser Program
program = spaces >> GProg <$> many1 classDeclaration <* eof

parseString :: String -> Except String Program
parseString s = ExceptT (Identity $ first show $ parse program "" s)
```

A.4 ClassAnalyzer.hs

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving, FlexibleContexts #-}

module ClassAnalyzer (classAnalysis, CAState(..)) where

import Data.Maybe
import Data.List

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

import AST

type Size = Integer

data CAState =
    CAState {
        classes :: [(TypeName, ClassDeclaration)],
        subClasses :: [(TypeName, [TypeName])],
        superClasses :: [(TypeName, [TypeName])],
        classSize :: [(TypeName, Size)],
        classMethods :: [(TypeName, [MethodDeclaration])],
        mainClass :: Maybe TypeName
    } deriving (Show, Eq)

newtype ClassAnalyzer a = ClassAnalyzer { runCA :: StateT CAState (Except String) a }
    deriving (Functor, Applicative, Monad, MonadState CAState, MonadError String)

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

getBaseClass :: TypeName -> ClassAnalyzer (Maybe TypeName)
getBaseClass n = getClass n >>= getBase
    where getBase (GCDecl _ b _ _) = return b

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)

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)

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)

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)
```

```haskell

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)

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 }

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

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

setSubClasses :: ClassDeclaration -> ClassAnalyzer ()
setSubClasses (GCDecl n b _ _) = modify (\s -> s { subClasses = (n, []) : subClasses s }) >> addSubClass n b

addSubClass :: TypeName -> Maybe TypeName -> ClassAnalyzer ()
addSubClass _ Nothing = return ()
addSubClass n (Just b) = gets subClasses >>= \sc ->
    case lookup b sc of
        Nothing -> modify $ \s -> s { subClasses = (b, [n]) : sc }
        (Just sc') -> modify $ \s -> s { subClasses = (b, n : sc') : delete (b, sc') sc }

setSuperClasses :: ClassDeclaration -> ClassAnalyzer ()
setSuperClasses (GCDecl n _ _ _) = gets subClasses >>= \sc ->
    modify $ \s -> s { superClasses = (n, map fst $ filter (\(_, sub) -> n `elem` sub) sc) : superClasses s }

getClassSize :: ClassDeclaration -> ClassAnalyzer Size
getClassSize (GCDecl _ Nothing fs _) = return $ 1 + genericLength fs
getClassSize (GCDecl _ (Just b) fs _) = getClass b >>= getClassSize >>= \sz -> return $ sz + genericLength fs

setClassSize :: ClassDeclaration -> ClassAnalyzer ()
setClassSize c@(GCDecl n _ _ _) = getClassSize c >>= \sz ->
    modify $ \s -> s { classSize = (n, sz) : classSize s }

resolveClassMethods :: ClassDeclaration -> ClassAnalyzer [MethodDeclaration]
resolveClassMethods (GCDecl _ Nothing _ ms) = return ms
resolveClassMethods (GCDecl n (Just b) _ ms) = getClass b >>= resolveClassMethods >>= combine
    where checkSignature (GMDecl m ps _, GMDecl m' ps' _) = when (m == m' && ps /= ps') (throwError $ "Method " ++ m ++ " in class " ++ n ++ " has invalid method signature")
          compareName (GMDecl m _ _) (GMDecl m' _ _) = m == m'
          combine ms' = mapM_ checkSignature ((,) <$> ms <*> ms') >> return (unionBy compareName ms ms')

setClassMethods :: ClassDeclaration -> ClassAnalyzer ()
```

```haskell
setClassMethods c@(GCDecl n _ _ _) = resolveClassMethods c >>= \cm ->
    modify $ \s -> s { classMethods = (n, cm) : classMethods s }

caProgram :: Program -> ClassAnalyzer Program
caProgram (GProg p) =
    do mapM_ setClasses p
       mapM_ setSubClasses p
       mapM_ setSuperClasses p
       mapM_ setClassSize p
       mapM_ setClassMethods p
       mapM_ checkDuplicateClasses p
       mapM_ checkDuplicateFields p
       mapM_ checkDuplicateMethods p
       mapM_ checkBaseClass p
       mapM_ checkCyclicInheritance p
       mapM_ setMainClass p
       mc <- gets mainClass
       when (isNothing mc) (throwError "No main method defined")
       return $ GProg rootClasses
    where rootClasses = filter noBase p
          noBase (GCDecl _ Nothing _ _) = True
          noBase _ = False

classAnalysis :: Program -> Except String (Program, CAState)
classAnalysis p = runStateT (runCA $ caProgram p) initialState
```

A.5 ScopeAnalyzer.hs

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

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

import Data.Maybe
import Data.List

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

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
        (s:_) -> 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

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

saExpression :: Expression -> ScopeAnalyzer SExpression
```

```
saExpression (Constant v) = pure $ Constant v
saExpression (Variable n) = Variable <$> saLookup n
saExpression Nil = pure Nil
saExpression (Binary binop e1 e2) =
    Binary binop
    <$> saExpression e1
    <*> saExpression e2

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

        (Swap n1 n2) ->
            Swap
            <$> saLookup n1
            <*> saLookup n2

        (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

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

        (LocalBlock n e1 stmt e2) ->
            do e1' <- saExpression e1
               enterScope
               n' <- saInsert (LocalVariable IntegerType n) n
               stmt' <- mapM saStatement stmt
               leaveScope
               e2' <- saExpression e2
               return $ LocalBlock n' e1' stmt' e2'

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

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

        (ObjectCall o m args) ->
            when (args /= nub args || o `elem` args) (throwError $ "Irreversible invocation of method " ++ m)
```

```
            >> ObjectCall
            <$> saLookup o
            <*> pure m
            <*> mapM saLookup args

        (ObjectUncall o m args) ->
            when (args /= nub args || o `elem` args) (throwError $ "Irreversible invocation of method " ++ m)
            >> ObjectUncall
            <$> saLookup o
            <*> pure m
            <*> mapM saLookup args

        Skip -> pure Skip

    where var (Variable n) = [n]
          var (Binary _ e1 e2) = var e1 ++ var e2
          var _ = []

          isCF ClassField{} = True
          isCF _ = False

          rlookup = flip lookup

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

setMainMethod :: SIdentifier -> ScopeAnalyzer ()
setMainMethod i = modify $ \s -> s { mainMethod = i }

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

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

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

prefixVtable :: [(SIdentifier, MethodName)] -> (SIdentifier, MethodName) -> [(SIdentifier, MethodName)]
prefixVtable [] m' = [m']
```

```haskell
prefixVtable (m:ms) m' = if comp m m' then m':ms else m : prefixVtable ms m'
    where comp (_, n) (_, n') = n == n'

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
           offset' = genericLength fs + offset
       modify $ \s -> s { virtualTables = (c, m3) : virtualTables s }
       sc <- getSubClasses c
       ms' <- concat <$> mapM (saClass offset' m3) sc
       ms'' <- mapM saMethod $ zip (repeat c) ms
       leaveScope
       return $ ms' ++ ms''
    where insertClassField (o, GDecl tp n) = saInsert (ClassField tp n c o) n
          insertMethod (GMDecl n ps _) = saInsert (Method (map getType ps) n) n >>= getMethodName
          getType (GDecl tp _) = tp

saProgram :: Program -> ScopeAnalyzer SProgram
saProgram (GProg cs) = concat <$> mapM (saClass 1 []) cs

scopeAnalysis :: (Program, CAState) -> Except String (SProgram, SAState)
scopeAnalysis (p, s) = runStateT (runSA $ saProgram p) $ initialState s
```

A.6 TypeChecker.hs

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module TypeChecker (typeCheck) where

import Data.List

import Control.Monad.Reader
import Control.Monad.Except

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

checkCall :: [SIdentifier] -> [DataType] -> TypeChecker ()
checkCall args ps = when (la /= lp) (throwError err) >> mapM getType args >>= \as -> mapM_ checkArgument (zip as ps)
    where la = length args
          lp = length ps
          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 (ta, tp) = expectType tp ta

tcExpression :: SExpression -> TypeChecker DataType
tcExpression (Constant _) = pure IntegerType
tcExpression (Variable n) = getType n
```

```
tcExpression Nil = pure NilType
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

        (Swap n1 n2) ->
            do t1 <- getType n1
               t2 <- getType n2
               expectType t1 t2

        (Conditional e1 s1 s2 e2) ->
            tcExpression e1
            >>= expectType IntegerType
            >> mapM_ tcStatement s1
            >> mapM_ tcStatement s2
            >> 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 n e1 stmt e2) ->
            getType n
            >>= expectType IntegerType
            >> tcExpression e1
            >>= expectType IntegerType
            >> mapM_ tcStatement stmt
            >> tcExpression e2
            >>= expectType IntegerType

        (LocalCall m args) ->
            getParameterTypes m
            >>= checkCall args

        (LocalUncall m args) ->
            getParameterTypes m
            >>= checkCall args

        (ObjectCall o m args) ->
            do t <- getType o
```

```haskell
               case t of
                   (ObjectType tn) -> getDynamicParameterTypes tn m >>= checkCall args
                   _ -> throwError $ "Non-object type " ++ show t ++ " does not support method invocation"

        (ObjectUncall o m args) ->
            do t <- getType o
               case t of
                   (ObjectType tn) -> getDynamicParameterTypes tn m >>= checkCall args
                   _ -> throwError $ "Non-object type " ++ show t ++ " does not support method invocation"

        Skip -> pure ()

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
```

A.7 CodeGenerator.hs

```haskell
{-# LANGUAGE GeneralizedNewtypeDeriving #-}

module CodeGenerator (generatePISA) where

import Data.List

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

import AST
import PISA
import ClassAnalyzer
import ScopeAnalyzer

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

data CGState =
    CGState {
        labelIndex :: SIdentifier,
        registerIndex :: Integer,
        labelTable :: [(SIdentifier, Label)],
        registerStack :: [(SIdentifier, Register)],
        saState :: SAState
    } deriving (Show, Eq)

newtype CodeGenerator a = CodeGenerator { runCG :: StateT CGState (Except String) a }
    deriving (Functor, Applicative, Monad, MonadState CGState, MonadError String)

initialState :: SAState -> CGState
initialState s = CGState { labelIndex = 0, registerIndex = 4, labelTable = [], registerStack = [], saState = s }

registerZero :: Register
registerZero = Reg 0

registerSP :: Register
registerSP = Reg 1

registerRO :: Register
registerRO = Reg 2

registerThis :: Register
registerThis = Reg 3

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

popRegister :: CodeGenerator ()
popRegister = modify $ \s -> s { registerIndex = (-1) + registerIndex s, registerStack = drop 1 $ registerStack s }

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

popTempRegister :: CodeGenerator ()
```

```haskell
popTempRegister = modify $ \s -> s { registerIndex = (-1) + registerIndex s }

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

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

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 }

getMethodLabel :: SIdentifier -> CodeGenerator Label
getMethodLabel m = gets labelTable >>= \lt ->
    case lookup m lt of
        (Just l) -> return l
        Nothing -> insertMethodLabel m >> getMethodLabel m

getUniqueLabel :: Label -> CodeGenerator Label
getUniqueLabel l =
    do i <- gets labelIndex
       modify $ \s -> s { labelIndex = 1 + i }
       return $ l ++ "_" ++ show i

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

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)], popTempRegister >> ua)

cgBinOp :: BinOp -> Register -> Register -> CodeGenerator (Register, [(Maybe Label, MInstruction)], CodeGenerator ())
cgBinOp Add r1 r2 = tempRegister >>= \rt -> return (rt, [(Nothing, XOR rt r1), (Nothing, ADD rt r2)], popTempRegister)
cgBinOp Sub r1 r2 = tempRegister >>= \rt -> return (rt, [(Nothing, XOR rt r1), (Nothing, SUB rt r2)], popTempRegister)
cgBinOp Xor r1 r2 = tempRegister >>= \rt -> return (rt, [(Nothing, XOR rt r1), (Nothing, XOR rt r2)], popTempRegister)
cgBinOp BitAnd r1 r2 = tempRegister >>= \rt -> 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"

cgExpression :: SExpression -> CodeGenerator (Register, [(Maybe Label, MInstruction)],
  ↪  CodeGenerator ())
cgExpression (Constant 0) = return (registerZero, [], return ())
cgExpression (Constant n) = tempRegister >>= \rt -> return (rt, [(Nothing, XORI rt $
  ↪  Immediate n)], popTempRegister)
cgExpression (Variable i) = loadVariableValue i
cgExpression Nil = return (registerZero, [], return ())
cgExpression (Binary op e1 e2) =
```

```haskell
    do (r1, l1, u1) <- cgExpression e1
       (r2, l2, u2) <- cgExpression e2
       (ro, lo, uo) <- cgBinOp op r1 r2
       return (ro, l1 ++ l2 ++ lo, uo >> u2 >> u1)

cgBinaryExpression :: SExpression -> CodeGenerator (Register, [(Maybe Label, MInstruction)], CodeGenerator ())
cgBinaryExpression e =
    do (re, le, ue) <- cgExpression e
       rt <- tempRegister
       l_top <- getUniqueLabel "f_top"
       l_bot <- getUniqueLabel "f_bot"
       let flatten = [(Just l_top, BEQ re registerZero l_bot),
                      (Nothing, XORI rt $ Immediate 1),
                      (Just l_bot, BEQ re registerZero l_top)]
       return (rt, le ++ flatten, popTempRegister >> ue)

cgAssign :: SIdentifier -> ModOp -> SExpression -> CodeGenerator [(Maybe Label, MInstruction)]
cgAssign n modop e =
    do (rt, lt, ut) <- loadVariableValue n
       (re, le, ue) <- cgExpression e
       ue >> ut
       return $ lt ++ le ++ [(Nothing, cgModOp modop rt re)] ++ invertInstructions (lt ++ le)
    where cgModOp ModAdd = ADD
          cgModOp ModSub = SUB
          cgModOp ModXor = XOR

loadForSwap :: SIdentifier -> CodeGenerator (Register, [(Maybe Label, MInstruction)], CodeGenerator ())
loadForSwap n = gets (symbolTable . saState) >>= \st ->
    case lookup n st of
        (Just ClassField {}) -> loadVariableValue n
        (Just (LocalVariable IntegerType _)) -> loadVariableValue n
        (Just (LocalVariable (ObjectType _) _)) -> loadVariableAddress n
        (Just (MethodParameter IntegerType _)) -> loadVariableValue n
        (Just (MethodParameter (ObjectType _) _)) -> loadVariableAddress n
        _ -> throwError $ "ICE: Invalid variable index " ++ show n

cgSwap :: SIdentifier -> SIdentifier -> CodeGenerator [(Maybe Label, MInstruction)]
cgSwap n1 n2 = if n1 == n2 then return [] else
    do (r1, l1, u1) <- loadForSwap n1
       (r2, l2, u2) <- loadForSwap n2
       u2 >> u1
       let swap = [(Nothing, XOR r1 r2), (Nothing, XOR r2 r1), (Nothing, XOR r1 r2)]
       return $ l1 ++ l2 ++ swap ++ invertInstructions (l1 ++ l2)

cgConditional :: SExpression -> [SStatement] -> [SStatement] -> SExpression -> CodeGenerator [(Maybe Label, MInstruction)]
cgConditional e1 s1 s2 e2 =
    do l_test <- getUniqueLabel "test"
       l_assert_t <- getUniqueLabel "assert_true"
       l_test_f <- getUniqueLabel "test_false"
       l_assert <- getUniqueLabel "assert"
       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 $ le1 ++ [(Nothing, XOR rt re1)] ++ invertInstructions le1 ++
                [(Just l_test, BEQ rt registerZero l_test_f), (Nothing, XORI rt $ Immediate 1)] ++
                s1' ++ [(Nothing, XORI rt $ Immediate 1), (Just l_assert_t, BRA l_assert), (Just l_test_f, BRA l_test)] ++
```

```haskell
                s2' ++ [(Just l_assert, BNE rt registerZero l_assert_t)] ++
                le2 ++ [(Nothing, XOR rt re2)] ++ invertInstructions le2

cgLoop :: SExpression -> [SStatement] -> [SStatement] -> SExpression -> CodeGenerator [(Maybe
    Label, MInstruction)]
cgLoop e1 s1 s2 e2 =
    do l_entry <- getUniqueLabel "entry"
       l_test <- getUniqueLabel "test"
       l_assert <- getUniqueLabel "assert"
       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 ++
                s1' ++ le2 ++ [(Nothing, XOR rt re2)] ++ invertInstructions le2 ++
                [(Just l_test, BNE rt registerZero l_exit)] ++ s2' ++
                [(Just l_assert, BRA l_entry), (Just l_exit, BRA l_test), (Nothing, XORI rt $
    Immediate 1)]

cgObjectBlock :: TypeName -> SIdentifier -> [SStatement] -> CodeGenerator [(Maybe Label,
    MInstruction)]
cgObjectBlock tp n stmt =
    do rn <- pushRegister n
       rv <- tempRegister
       popTempRegister --rv
       stmt' <- concat <$> mapM cgStatement stmt
       popRegister --rn
       let create = [(Nothing, XOR rn registerSP),
                     (Nothing, XORI rv $ AddressMacro $ "l_" ++ tp ++ "_vt"),
                     (Nothing, EXCH rv registerSP),
                     (Nothing, ADDI registerSP $ SizeMacro tp)]
       return $ create ++ stmt' ++ invertInstructions create

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
       let create re rt = [(Nothing, XOR rn registerSP),
                           (Nothing, XOR rt re),
                           (Nothing, EXCH rt registerSP),
                           (Nothing, ADDI registerSP $ Immediate 1)]
           load = le1 ++ create re1 rt1 ++ invertInstructions le1
           clear = le2 ++ invertInstructions (create re2 rt2) ++ invertInstructions le2
       return $ load ++ stmt' ++ clear

cgCall :: [SIdentifier] -> [(Maybe Label, MInstruction)] -> Register -> CodeGenerator [(Maybe
    Label, MInstruction)]
cgCall args jump this =
    do (ra, la, ua) <- unzip3 <$> mapM loadVariableAddress 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, ADDI registerSP $ Immediate 1)]

cgLocalCall :: SIdentifier -> [SIdentifier] -> CodeGenerator [(Maybe Label, MInstruction)]
cgLocalCall m args = getMethodLabel m >>= \l_m -> cgCall args [(Nothing, BRA l_m)] registerThis

cgLocalUncall :: SIdentifier -> [SIdentifier] -> CodeGenerator [(Maybe Label, MInstruction)]
cgLocalUncall m args = getMethodLabel m >>= \l_m -> cgCall args [(Nothing, RBRA l_m)] registerThis

getType :: SIdentifier -> CodeGenerator TypeName
getType i = gets (symbolTable . saState) >>= \st ->
    case lookup i st of
        (Just (LocalVariable (ObjectType tp) _)) -> return tp
        (Just (ClassField (ObjectType tp) _ _ _)) -> return tp
        (Just (MethodParameter (ObjectType tp) _)) -> return tp
        _ -> throwError $ "ICE: Invalid object variable index " ++ show i

loadMethodAddress :: (SIdentifier, Register) -> MethodName -> CodeGenerator (Register, [(Maybe Label, MInstruction)])
loadMethodAddress (o, ro) m =
    do rv <- tempRegister
       rt <- tempRegister
       rtgt <- tempRegister
       popTempRegister >> popTempRegister >> popTempRegister
       offsetMacro <- OffsetMacro <$> getType o <*> pure m
       let load = [(Nothing, EXCH rv ro),
                   (Nothing, ADDI rv offsetMacro),
                   (Nothing, EXCH rt rv),
                   (Nothing, XOR rtgt rt),
                   (Nothing, EXCH rt rv),
                   (Nothing, SUBI rv offsetMacro),
                   (Nothing, EXCH rv ro)]
       return (rtgt, load)

loadForCall :: SIdentifier -> CodeGenerator (Register, [(Maybe Label, MInstruction)], CodeGenerator ())
loadForCall n = gets (symbolTable . saState) >>= \st ->
    case lookup n st of
        (Just ClassField {}) -> loadVariableValue n
        (Just _) -> loadVariableAddress n
        _ -> throwError $ "ICE: Invalid variable index " ++ show n

cgObjectCall :: SIdentifier -> MethodName -> [SIdentifier] -> CodeGenerator [(Maybe Label, MInstruction)]
cgObjectCall o m args =
    do (ro, lo, uo) <- loadForCall o
       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 >> uo
       let load = lo ++ [(Nothing, XOR rt ro)] ++ loadAddress ++ invertInstructions lo
       return $ load ++ call ++ invertInstructions load

cgObjectUncall :: SIdentifier -> MethodName -> [SIdentifier] -> CodeGenerator [(Maybe Label, MInstruction)]
cgObjectUncall o m args =
    do (ro, lo, uo) <- loadForCall o
```

```haskell
       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),
                 (Just l_rjmp_bot, BRA l_rjmp_top),
                 (Nothing, ADDI rtgt $ AddressMacro l_jmp)]
       call <- cgCall args jp rt
       popTempRegister >> uo
       let load = lo ++ [(Nothing, XOR rt ro)] ++ loadAddress ++ invertInstructions lo
       return $ load ++ call ++ invertInstructions load

cgStatement :: SStatement -> CodeGenerator [(Maybe Label, MInstruction)]
cgStatement (Assign n modop e) = cgAssign n modop e
cgStatement (Swap n1 n2) = cgSwap n1 n2
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
cgStatement (LocalUncall m args) = cgLocalUncall m args
cgStatement (ObjectCall o m args) = cgObjectCall o m args
cgStatement (ObjectUncall o m args) = cgObjectUncall o m args
cgStatement Skip = return []

cgMethod :: (TypeName, SMethodDeclaration) -> CodeGenerator [(Maybe Label, MInstruction)]
cgMethod (_, GMDecl m ps body) =
    do l <- getMethodLabel m
       rs <- addParameters
       body' <- concat <$> mapM cgStatement body
       clearParameters
       let lt = l ++ "_top"
           lb = l ++ "_bot"
           mp = [(Just lt, BRA lb),
                 (Nothing, SUBI registerSP $ Immediate 1),
                 (Nothing, EXCH registerRO registerSP)]
                 ++ concatMap pushParameter rs ++
                [(Nothing, EXCH registerThis registerSP),
                 (Nothing, ADDI registerSP $ Immediate 1),
                 (Just l, SWAPBR registerRO),
                 (Nothing, NEG registerRO),
                 (Nothing, SUBI registerSP $ Immediate 1),
                 (Nothing, EXCH registerThis registerSP)]
                 ++ invertInstructions (concatMap pushParameter rs) ++
                [(Nothing, EXCH registerRO registerSP),
                 (Nothing, ADDI registerSP $ Immediate 1)]
       return $ mp ++ body' ++ [(Just lb, BRA lt)]
    where addParameters = mapM (pushRegister . (\(GDecl _ p) -> p)) ps
          clearParameters = replicateM_ (length ps) popRegister
          pushParameter r = [(Nothing, EXCH r registerSP), (Nothing, ADDI registerSP $ Immediate 1)]

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

getMainLabel :: CodeGenerator Label
getMainLabel = gets (mainMethod . saState) >>= getMethodLabel

getMainClass :: CodeGenerator TypeName
```

```haskell
getMainClass = gets (mainClass . caState . saState) >>= \mc ->
    case mc of
        (Just tp) -> return tp
        Nothing -> throwError "ICE: No main method defined"

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

cgOutput :: TypeName -> CodeGenerator ([(Maybe Label, MInstruction)], [(Maybe Label, MInstruction)])
cgOutput tp =
    do mfs <- getFields tp
       co <- concat <$> mapM cgCopyOutput (zip [1..] $ reverse 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 = [SUBI registerSP $ Immediate o,
                             EXCH rt registerSP,
                             XORI ra $ AddressMacro $ "l_r_" ++ n,
                             EXCH rt ra,
                             XORI ra $ AddressMacro $ "l_r_" ++ n,
                             ADDI registerSP $ Immediate o]
                 return $ zip (repeat Nothing) copy

cgProgram :: SProgram -> CodeGenerator PISA.MProgram
cgProgram p =
    do vt <- cgVirtualTables
       rv <- tempRegister
       popTempRegister
       ms <- concat <$> mapM 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 registerSP ProgramSize),
                 (Nothing, XOR registerThis registerSP),
                 (Nothing, XORI rv $ AddressMacro mvt),
                 (Nothing, EXCH rv registerSP),
                 (Nothing, ADDI registerSP $ SizeMacro mtp),
                 (Nothing, EXCH registerThis registerSP),
                 (Nothing, ADDI registerSP $ Immediate 1),
                 (Nothing, BRA l_main),
                 (Nothing, SUBI registerSP $ Immediate 1),
                 (Nothing, EXCH registerThis registerSP)]
                 ++ co ++
                [(Nothing, SUBI registerSP $ SizeMacro mtp),
                 (Nothing, EXCH rv registerSP),
                 (Nothing, XORI rv $ AddressMacro mvt),
                 (Nothing, XOR registerThis registerSP),
                 (Nothing, SUBI registerSP ProgramSize),
                 (Just "finish", FINISH)]
       return $ PISA.GProg $ [(Just "top", BRA "start")] ++ out ++ vt ++ ms ++ mn

generatePISA :: (SProgram, SAState) -> Except String (PISA.MProgram, SAState)
generatePISA (p, s) = second saState <$> runStateT (runCG $ cgProgram p) (initialState s)
```

A.8 MacroExpander.hs

```haskell
{-# 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 ScopeAnalyzer
import ClassAnalyzer

type Size = Integer
type Address = Integer
type Offset = Integer

data MEState =
    MEState {
        addressTable :: [(Label, Address)],
        sizeTable :: [(TypeName, Size)],
        offsetTable :: [(TypeName, [(MethodName, Offset)])],
        programSize :: Size
    } deriving (Show, Eq)

newtype MacroExpander a = MacroExpander { runME :: ReaderT MEState (Except String) a }
    deriving (Functor, Applicative, Monad, MonadReader MEState, MonadError String)

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
          getName _ = error "ICE: Invalid method index"

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 }
    where toPair (a, (Just l, _)) = Just (l, 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

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 ->
```

```haskell
    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

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

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 (SRAVX r1 r2 r3) = return $ SRAVX 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

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

expandMacros :: (MProgram, SAState) -> Except String Program
expandMacros (p, s) = runReaderT (runME $ meProgram p) $ initialState p s
```

A.9 ROOPLC.hs

```haskell
import Control.Monad.Except
import System.IO

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

type Error = String

main :: IO ()
main =
    do input <- getContents
       either (hPutStrLn stderr) (putStr . showProgram) (compileProgram input)

compileProgram :: String -> Either Error Program
compileProgram s =
    runExcept $
    parseString s
    >>= classAnalysis
    >>= scopeAnalysis
    >>= typeCheck
    >>= generatePISA
    >>= expandMacros
```

APPENDIX B

Example Output

B.1 LinkedList.rpl

```
class Program
    int result
    int n
    Node foo

    method BuildList(Node head)
        if n = 0 then
            call head::sum(result)
        else
            construct Node next
                call next::constructor(n, head)
                n -= 1
                call BuildList(next)
                n += 1
                uncall next::constructor(n, head)
            destruct next
        fi n = 0

    method main()
        n += 7
        construct Node tail
        foo <=> tail
        call BuildList(tail)
        foo <=> tail
        destruct tail

class Node
    int data
    Node next

    method constructor(int d, Node n)
        next <=> n
        data ^= d

    method sum(int s)
        s += data
        if next = nil then
            skip
        else
            call next::sum(s)
        fi next = nil
```

B.2 LinkedList.pal

```
;; pendulum pal file
top:                    BRA    start
l_r_result:             DATA   0
l_r_n:                  DATA   0
l_r_foo:                DATA   0
l_Node_vt:              DATA   302
                        DATA   338
l_Program_vt:           DATA   15
                        DATA   242
l_BuildList_2_top:      BRA    l_BuildList_2_bot
                        ADDI   $1 -1
                        EXCH   $2 $1
                        EXCH   $4 $1
                        ADDI   $1 1
                        EXCH   $3 $1
                        ADDI   $1 1
l_BuildList_2:          SWAPBR $2
                        NEG    $2
                        ADDI   $1 -1
                        EXCH   $3 $1
                        ADDI   $1 -1
                        EXCH   $4 $1
                        EXCH   $2 $1
                        ADDI   $1 1
                        ADD    $6 $3
                        ADDI   $6 2
                        EXCH   $7 $6
cmp_top_8:              BNE    $7 $0 cmp_bot_9
                        XORI   $8 1
cmp_bot_9:              BNE    $7 $0 cmp_top_8
f_top_10:               BEQ    $8 $0 f_bot_11
                        XORI   $9 1
f_bot_11:               BEQ    $8 $0 f_top_10
                        XOR    $5 $9
f_bot_11_i:             BEQ    $8 $0 f_top_10_i
                        XORI   $9 1
f_top_10_i:             BEQ    $8 $0 f_bot_11_i
cmp_bot_9_i:            BNE    $7 $0 cmp_top_8_i
                        XORI   $8 1
cmp_top_8_i:            BNE    $7 $0 cmp_bot_9_i
                        EXCH   $7 $6
                        ADDI   $6 -2
                        SUB    $6 $3
test_4:                 BEQ    $5 $0 test_false_6
                        XORI   $5 1
                        XOR    $6 $4
                        EXCH   $7 $6
                        ADDI   $7 1
                        EXCH   $8 $7
                        XOR    $9 $8
                        EXCH   $8 $7
                        ADDI   $7 -1
                        EXCH   $7 $6
                        ADD    $7 $3
                        ADDI   $7 1
                        EXCH   $3 $1
                        ADDI   $1 1
                        EXCH   $4 $1
                        ADDI   $1 1
                        EXCH   $7 $1
                        ADDI   $1 1
                        EXCH   $6 $1
                        ADDI   $1 1
```

```
                          ADDI   $9 -63
l_jmp_12:                 SWAPBR $9
                          NEG    $9
                          ADDI   $9 63
                          ADDI   $1 -1
                          EXCH   $6 $1
                          ADDI   $1 -1
                          EXCH   $7 $1
                          ADDI   $1 -1
                          EXCH   $4 $1
                          ADDI   $1 -1
                          EXCH   $3 $1
                          ADDI   $7 -1
                          SUB    $7 $3
                          EXCH   $7 $6
                          ADDI   $7 1
                          EXCH   $8 $7
                          XOR    $9 $8
                          EXCH   $8 $7
                          ADDI   $7 -1
                          EXCH   $7 $6
                          XOR    $6 $4
                          XORI   $5 1
assert_true_5:            BRA    assert_7
test_false_6:             BRA    test_4
                          XOR    $6 $1
                          XORI   $7 4
                          EXCH   $7 $1
                          ADDI   $1 3
                          XOR    $7 $6
                          EXCH   $8 $7
                          ADDI   $8 0
                          EXCH   $9 $8
                          XOR    $10 $9
                          EXCH   $9 $8
                          ADDI   $8 0
                          EXCH   $8 $7
                          ADD    $8 $3
                          ADDI   $8 2
                          EXCH   $3 $1
                          ADDI   $1 1
                          EXCH   $6 $1
                          ADDI   $1 1
                          EXCH   $8 $1
                          ADDI   $1 1
                          EXCH   $4 $1
                          ADDI   $1 1
                          EXCH   $7 $1
                          ADDI   $1 1
                          ADDI   $10 -112
l_jmp_13:                 SWAPBR $10
                          NEG    $10
                          ADDI   $10 112
                          ADDI   $1 -1
                          EXCH   $7 $1
                          ADDI   $1 -1
                          EXCH   $4 $1
                          ADDI   $1 -1
                          EXCH   $8 $1
                          ADDI   $1 -1
                          EXCH   $6 $1
                          ADDI   $1 -1
                          EXCH   $3 $1
                          ADDI   $8 -2
                          SUB    $8 $3
                          EXCH   $8 $7
```

```
                         ADDI   $8 0
                         EXCH   $9 $8
                         XOR    $10 $9
                         EXCH   $9 $8
                         ADDI   $8 0
                         EXCH   $8 $7
                         XOR    $7 $6
                         ADD    $7 $3
                         ADDI   $7 2
                         EXCH   $8 $7
                         XORI   $9 1
                         SUB    $8 $9
                         XORI   $9 1
                         EXCH   $8 $7
                         ADDI   $7 -2
                         SUB    $7 $3
                         EXCH   $4 $1
                         ADDI   $1 1
                         EXCH   $6 $1
                         ADDI   $1 1
                         EXCH   $3 $1
                         ADDI   $1 1
                         BRA    l_BuildList_2
                         ADDI   $1 -1
                         EXCH   $3 $1
                         ADDI   $1 -1
                         EXCH   $6 $1
                         ADDI   $1 -1
                         EXCH   $4 $1
                         ADD    $7 $3
                         ADDI   $7 2
                         EXCH   $8 $7
                         XORI   $9 1
                         ADD    $8 $9
                         XORI   $9 1
                         EXCH   $8 $7
                         ADDI   $7 -2
                         SUB    $7 $3
                         XOR    $7 $6
                         EXCH   $8 $7
                         ADDI   $8 0
                         EXCH   $9 $8
                         XOR    $10 $9
                         EXCH   $9 $8
                         ADDI   $8 0
                         EXCH   $8 $7
                         ADD    $8 $3
                         ADDI   $8 2
                         EXCH   $3 $1
                         ADDI   $1 1
                         EXCH   $6 $1
                         ADDI   $1 1
                         EXCH   $8 $1
                         ADDI   $1 1
                         EXCH   $4 $1
                         ADDI   $1 1
                         EXCH   $7 $1
                         ADDI   $1 1
                         ADDI   $10 -188
l_rjmp_top_15:           RBRA   l_rjmp_bot_16
l_jmp_14:                SWAPBR $10
                         NEG    $10
l_rjmp_bot_16:           BRA    l_rjmp_top_15
                         ADDI   $10 188
                         ADDI   $1 -1
                         EXCH   $7 $1
```

```
                          ADDI   $1 -1
                          EXCH   $4 $1
                          ADDI   $1 -1
                          EXCH   $8 $1
                          ADDI   $1 -1
                          EXCH   $6 $1
                          ADDI   $1 -1
                          EXCH   $3 $1
                          ADDI   $8 -2
                          SUB    $8 $3
                          EXCH   $8 $7
                          ADDI   $8 0
                          EXCH   $9 $8
                          XOR    $10 $9
                          EXCH   $9 $8
                          ADDI   $8 0
                          EXCH   $8 $7
                          XOR    $7 $6
                          ADDI   $1 -3
                          EXCH   $7 $1
                          XORI   $7 4
                          XOR    $6 $1
assert_7:                 BNE    $5 $0 assert_true_5
                          ADD    $6 $3
                          ADDI   $6 2
                          EXCH   $7 $6
cmp_top_17:               BNE    $7 $0 cmp_bot_18
                          XORI   $8 1
cmp_bot_18:               BNE    $7 $0 cmp_top_17
f_top_19:                 BEQ    $8 $0 f_bot_20
                          XORI   $9 1
f_bot_20:                 BEQ    $8 $0 f_top_19
                          XOR    $5 $9
f_bot_20_i:               BEQ    $8 $0 f_top_19_i
                          XORI   $9 1
f_top_19_i:               BEQ    $8 $0 f_bot_20_i
cmp_bot_18_i:             BNE    $7 $0 cmp_top_17_i
                          XORI   $8 1
cmp_top_17_i:             BNE    $7 $0 cmp_bot_18_i
                          EXCH   $7 $6
                          ADDI   $6 -2
                          SUB    $6 $3
l_BuildList_2_bot:        BRA    l_BuildList_2_top
l_main_3_top:             BRA    l_main_3_bot
                          ADDI   $1 -1
                          EXCH   $2 $1
                          EXCH   $3 $1
                          ADDI   $1 1
l_main_3:                 SWAPBR $2
                          NEG    $2
                          ADDI   $1 -1
                          EXCH   $3 $1
                          EXCH   $2 $1
                          ADDI   $1 1
                          ADD    $4 $3
                          ADDI   $4 2
                          EXCH   $5 $4
                          XORI   $6 7
                          ADD    $5 $6
                          XORI   $6 7
                          EXCH   $5 $4
                          ADDI   $4 -2
                          SUB    $4 $3
                          XOR    $4 $1
                          XORI   $5 4
                          EXCH   $5 $1
```

```
                        ADDI   $1 3
                        ADD    $5 $3
                        ADDI   $5 3
                        EXCH   $6 $5
                        XOR    $6 $4
                        XOR    $4 $6
                        XOR    $6 $4
                        EXCH   $6 $5
                        ADDI   $5 -3
                        SUB    $5 $3
                        EXCH   $4 $1
                        ADDI   $1 1
                        EXCH   $3 $1
                        ADDI   $1 1
                        BRA    l_BuildList_2
                        ADDI   $1 -1
                        EXCH   $3 $1
                        ADDI   $1 -1
                        EXCH   $4 $1
                        ADD    $5 $3
                        ADDI   $5 3
                        EXCH   $6 $5
                        XOR    $6 $4
                        XOR    $4 $6
                        XOR    $6 $4
                        EXCH   $6 $5
                        ADDI   $5 -3
                        SUB    $5 $3
                        ADDI   $1 -3
                        EXCH   $5 $1
                        XORI   $5 4
                        XOR    $4 $1
l_main_3_bot:           BRA    l_main_3_top
l_constructor_0_top:    BRA    l_constructor_0_bot
                        ADDI   $1 -1
                        EXCH   $2 $1
                        EXCH   $4 $1
                        ADDI   $1 1
                        EXCH   $5 $1
                        ADDI   $1 1
                        EXCH   $3 $1
                        ADDI   $1 1
l_constructor_0:        SWAPBR $2
                        NEG    $2
                        ADDI   $1 -1
                        EXCH   $3 $1
                        ADDI   $1 -1
                        EXCH   $5 $1
                        ADDI   $1 -1
                        EXCH   $4 $1
                        EXCH   $2 $1
                        ADDI   $1 1
                        ADD    $6 $3
                        ADDI   $6 2
                        EXCH   $7 $6
                        XOR    $7 $5
                        XOR    $5 $7
                        XOR    $7 $5
                        EXCH   $7 $6
                        ADDI   $6 -2
                        SUB    $6 $3
                        ADD    $6 $3
                        ADDI   $6 1
                        EXCH   $7 $6
                        EXCH   $8 $4
                        XOR    $7 $8
```

```
                        EXCH   $8 $4
                        EXCH   $7 $6
                        ADDI   $6 -1
                        SUB    $6 $3
l_constructor_0_bot:    BRA    l_constructor_0_top
l_sum_1_top:            BRA    l_sum_1_bot
                        ADDI   $1 -1
                        EXCH   $2 $1
                        EXCH   $4 $1
                        ADDI   $1 1
                        EXCH   $3 $1
                        ADDI   $1 1
l_sum_1:                SWAPBR $2
                        NEG    $2
                        ADDI   $1 -1
                        EXCH   $3 $1
                        ADDI   $1 -1
                        EXCH   $4 $1
                        EXCH   $2 $1
                        ADDI   $1 1
                        EXCH   $5 $4
                        ADD    $6 $3
                        ADDI   $6 1
                        EXCH   $7 $6
                        ADD    $5 $7
                        EXCH   $7 $6
                        ADDI   $6 -1
                        SUB    $6 $3
                        EXCH   $5 $4
                        ADD    $6 $3
                        ADDI   $6 2
                        EXCH   $7 $6
cmp_top_25:             BNE    $7 $0 cmp_bot_26
                        XORI   $8 1
cmp_bot_26:             BNE    $7 $0 cmp_top_25
f_top_27:               BEQ    $8 $0 f_bot_28
                        XORI   $9 1
f_bot_28:               BEQ    $8 $0 f_top_27
                        XOR    $5 $9
f_bot_28_i:             BEQ    $8 $0 f_top_27_i
                        XORI   $9 1
f_top_27_i:             BEQ    $8 $0 f_bot_28_i
cmp_bot_26_i:           BNE    $7 $0 cmp_top_25_i
                        XORI   $8 1
cmp_top_25_i:           BNE    $7 $0 cmp_bot_26_i
                        EXCH   $7 $6
                        ADDI   $6 -2
                        SUB    $6 $3
test_21:                BEQ    $5 $0 test_false_23
                        XORI   $5 1
                        XORI   $5 1
assert_true_22:         BRA    assert_24
test_false_23:          BRA    test_21
                        ADD    $6 $3
                        ADDI   $6 2
                        EXCH   $7 $6
                        XOR    $8 $7
                        EXCH   $9 $8
                        ADDI   $9 1
                        EXCH   $10 $9
                        XOR    $11 $10
                        EXCH   $10 $9
                        ADDI   $9 -1
                        EXCH   $9 $8
                        EXCH   $7 $6
                        ADDI   $6 -2
```

```
                         SUB    $6 $3
                         EXCH   $3 $1
                         ADDI   $1 1
                         EXCH   $4 $1
                         ADDI   $1 1
                         EXCH   $8 $1
                         ADDI   $1 1
                         ADDI   $11 -400
l_jmp_29:                SWAPBR $11
                         NEG    $11
                         ADDI   $11 400
                         ADDI   $1 -1
                         EXCH   $8 $1
                         ADDI   $1 -1
                         EXCH   $4 $1
                         ADDI   $1 -1
                         EXCH   $3 $1
                         ADD    $6 $3
                         ADDI   $6 2
                         EXCH   $7 $6
                         EXCH   $9 $8
                         ADDI   $9 1
                         EXCH   $10 $9
                         XOR    $11 $10
                         EXCH   $10 $9
                         ADDI   $9 -1
                         EXCH   $9 $8
                         XOR    $8 $7
                         EXCH   $7 $6
                         ADDI   $6 -2
                         SUB    $6 $3
assert_24:               BNE    $5 $0 assert_true_22
                         ADD    $6 $3
                         ADDI   $6 2
                         EXCH   $7 $6
cmp_top_30:              BNE    $7 $0 cmp_bot_31
                         XORI   $8 1
cmp_bot_31:              BNE    $7 $0 cmp_top_30
f_top_32:                BEQ    $8 $0 f_bot_33
                         XORI   $9 1
f_bot_33:                BEQ    $8 $0 f_top_32
                         XOR    $5 $9
f_bot_33_i:              BEQ    $8 $0 f_top_32_i
                         XORI   $9 1
f_top_32_i:              BEQ    $8 $0 f_bot_33_i
cmp_bot_31_i:            BNE    $7 $0 cmp_top_30_i
                         XORI   $8 1
cmp_top_30_i:            BNE    $7 $0 cmp_bot_31_i
                         EXCH   $7 $6
                         ADDI   $6 -2
                         SUB    $6 $3
l_sum_1_bot:             BRA    l_sum_1_top
start:                   BRA    top
                         START
                         ADDI   $1 480
                         XOR    $3 $1
                         XORI   $4 6
                         EXCH   $4 $1
                         ADDI   $1 4
                         EXCH   $3 $1
                         ADDI   $1 1
                         BRA    l_main_3
                         ADDI   $1 -1
                         EXCH   $3 $1
                         ADDI   $1 -1
                         EXCH   $4 $1
```

```
                         XORI   $5 3
                         EXCH   $4 $5
                         XORI   $5 3
                         ADDI   $1 1
                         ADDI   $1 -2
                         EXCH   $4 $1
                         XORI   $5 2
                         EXCH   $4 $5
                         XORI   $5 2
                         ADDI   $1 2
                         ADDI   $1 -3
                         EXCH   $4 $1
                         XORI   $5 1
                         EXCH   $4 $5
                         XORI   $5 1
                         ADDI   $1 3
                         ADDI   $1 -4
                         EXCH   $4 $1
                         XORI   $4 6
                         XOR    $3 $1
                         ADDI   $1 -480
finish:                  FINISH
```