UNIVERSITY OF COPENHAGEN

Project Outside Course Scope

Lasse Hay-Schmidt
Department of Computer Science
University of Copenhagen

Constructing an Interpreter for a Reversible Object-Oriented Programming Language

Submitted: January 22, 2021
Revision: 1.0.0

Abstract

Several reversible programming languages have been developed based on different programming paradigms, such as imperative, functional and object-oriented. However, most reversible programming languages are interpetered due to the issue that there does not exist any reversible hardware. Thus, compiled reversible programs have to be run in a virtual machine, which makes it cumbersome.

We aim to provide an option for constructing and running object-oriented reversible programs for the reversible object-oriented programming language (ROOPL). Whenever we talk about ROOPL, we actually talk about the superset implementation ROOPL++, but ROOPL is used for simplicity. Currently it is only possible to compile ROOPL programs to `Pendulum` ISA (`PISA`) assembly code and then run it on a `Pendulum` virtual machine. We introduce a ROOPL interpreter, which removes the issue of compiling and running on a virtual machine. Furthermore, the interpreter introduces functionality such as a web interface and showing runtime error trace stacks.

The interpreter is built using a tree-walker approach and utilizes existing ROOPL architecture by Haulund [5] and Cservenka [3]. Before running the interpreter, scoping and type checking is performed, which avoids potential issues at runtime. Therefore, the interpreted program is as well-typed as its compiled counterpart.

A standalone program inverter for ROOPL is also introduced. One of the main reasons for using a reversible programming language is that one can run a program in forward and backward directions. The program inverter generates an inverted program, which can be used for running programs in the backward direction.

The web interface enables wider accessibility to ROOPL programming, as it will allow users to construct and run programs through a browser. The functionality consists of interpretation, program inversion and compilation. Compilation produces a source file, which can be used with the `PendulumVM`.

Table of Contents

1 Introduction

Reversible computation currenctly serves a niche purpose, where one would want to run a program in the forward and backward direction. One could simply write a single procedure, which could then be used for solving two problems. A classic example is compression and decompression. By writing only a compression procedure, it is possible to run it in reverse, which is equivalent to performing decompression.

This project expands on an existing implementation of a reversible object-oriented programming language (ROOPL) [5] [3]. Whenever ROOPL is mentioned, we actually talk about the superset implementation ROOPL++, however, for simplicity reasons we use the shorter name.

The current ROOPL implementation can compile source code to `Pendulum` ISA (`PISA`) assembly code, which can then be run on a `Pendulum` virtual machine. Thus, writing and running ROOPL programs are not easily accessible as it requires having access to a `PendulumVM`.

Interpreters serve as an alternative to compilation. Using interpreters reduce the time between development and testing as it executes source code directly.

Compilation usually require multiple steps when compiling source code to machine code. After compiling the program can then be run on some designated hardware. However, compilation is also faster as it is better optimized for the hardware.

The distinction between the pros and cons of interpretation and compilation is therefore clear. Whenever a programmer has to develop and test a project, using an interpreter helps eliminate unnecessary waiting. Then after the project has been properly tested and validated, it can be compiled and run on appropriate hardware.

As reversible computation allows a program to be run in the forward and backward direction, one would likely want the backward direction to be easily executed. One such way is to invert the entire program and then simply run it

using an output state as input. Without the inverse program, the programmer would have to manually update the same source code to setup the input and then call or uncall, for the forward and backward direction respectively.

Thus, by generating the inverse program, it is easy to run the forward and backward direction.

When introducing a new system or programming language, one of the challenges is to make it easy and accessible for potential programmer to test. By allowing access through a browser, anyone can connect and try the system out before deciding whether to install the full suite.

1.1 Motivation

The goal of this project is to make using and running ROOPL programs easier by extending the current implementation with additional features. An important aspect of running programs are being able to see runtime information, i.e. exit state or exception trace stack. By showing the exit state, a programmer can then feed the state back into the inverted program to get the initial state. Furthermore, by creating an interpreter the time between writing and testing is shortened compared to using the compiler. This is due to the fact that an interpreter executes code directly. When using a compiler, the programmer would have to compile the program and then run the object code on a `PendulumVM`.

A browser enables users to interact with tools and systems that would otherwise require installation and setup of individual pieces of software. It is therefore beneficial for the usability and accessibility of ROOPL to introduce a web interface.

The web interface allows users to construct and run their programs through their browsers. Thus, removing the complexity of installing and setting up ROOPL. In addition to running ROOPL programs, it is possible to invert programs and compile to PISA code.

1.2 Outline

The report consists of the following three chapters, in addition to the introduction. A summary has been given below.

- **Chapter 2**, introduces design choices and implementation of the ROOPL interpreter. It presents the design behind the interpreter structure used for keeping track of individual object variables.
- **Chapter 3**, presents the standalone program inverter for ROOPL. The chapter introduces program inversion and the inversion rules for ROOPL statements.

- **Chapter 4**, shows the web interface and presents its functionality. The interface acts as a intermediate between ROOPL and an end-user by offering all functionality through a browser.

2 Interpreter

Interpretation serves as an alternative to compilation. Both interpreters and compilers share some of the underlying mechanisms for handling source code. They both parse source code into tokens and generate a parse tree. The difference lies in how the compiler and interpreter handles the final step. A compiler generates machine code that requires to be run on dedicated hardware or a virtual machine. Whereas an interpreter executes the actions directly on the current hardware.

Due to this difference, interpretation eases the program development by allowing the programmer to run their program from source without having to recompile and run the executable after each change. Compilation prove to be costly whenever a program become sufficiently large. Futhermore, the compilation stage may need to run a specific pipeline, which might require help from tools like `Make`.

2.1 Design

The ROOPL interpreter is based on a simple tree-walker approach. It utilizes the existing backend implementation for generating tokens and parse tree, performing class, scope analysis and type checking. The static analysis eliminates a lot of potential runtime issues.

The interpreter traverses the ROOPL program using the just-in-time (JIT) methodology. Nothing is computed ahead-of-time, i.e. when a method is being uncalled, a method inversion has to be performed.

The implementation assumes that the main class is an object where its class variables defines the output state. Current design constraints limit the input state to being setup in the `main` method. As such, `main` is not reversible because the inverted version has a hard dependency on setting up the input state.

2.1.1 Interpreter Structure

For execution of the interpretation step, an internal structure has been defined, which is accessible in any part of the interpreter. The internal structure consists

of an environment and store, an inversion cache, and an object and reference scope. The environment and store will be explained in section 2.1.2.

An inversion cache keeps track of the inverted methods to avoid inverting the same method multiple times, which can be costly. The interpreter searches the cache before inverting a method. If an inversion exists it is returned, otherwise the method is inverted, saved to the cache and then returned. Program inversion is based on rules specified in section 3.1

Since we are dealing with methods that use pass-by-reference and contains no return value, we would like to keep track of the call stack. Thus, we have defined scopes for objects and their references. The object scope contains the order of the called objects, where the top is the current object and the bottom is the main object. In addition, the reference scope contains the references of a corresponding object. A reference can point to a variable in the same object or a reference to a variable in the calling object.

By utilizing those scopes we can trace the call stack and update value references accordingly, i.e. a value is passed through multiple methods before being updated.

```
1  class Foo
2      method addTwo(int y)
3          construct Bar b
4          call b::addOne(y)
5          call b::addOne(y)
6          destruct b
7
8  class Bar
9      method addOne(int z)
10         z += 1
11
12 class Program
13     method main
14         construct Foo f
15         local int x = 0
16         call f::addTwo(x)
17         delocal int x = 2
18         destruct f
```

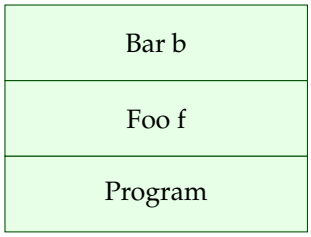

Figure 2.1: Nested method calls with call-by-reference and the object scope after calling `b::addOne(y)`

Figure 2.1 shows an object scope after calling a `Foo` and then a `Bar` object method. The reference of the original `x` in `Program` is passed in each method call. Thus, when executing `Bar::addOne(int z)` the value of `x` is updated.

2.1.2 Environment and Store

Before diving into the implementation details, we start of by defininining an environment and store. An environment is a map containing the references between

a variable and a location in the store. A store contains a map between a location and some expresssion. Each location in the store is unique.

In object-oriented programming (OOP) an object usually contains some class variables. Futhermore, it is possible to construct a class multiple times without overwriting the class variables of other objects. Thus, for the implementation each object contains an environment of class variables, if any. By having each object carry its own environment one can easily invoke an object method call without having to find the object's variables in a global environment.

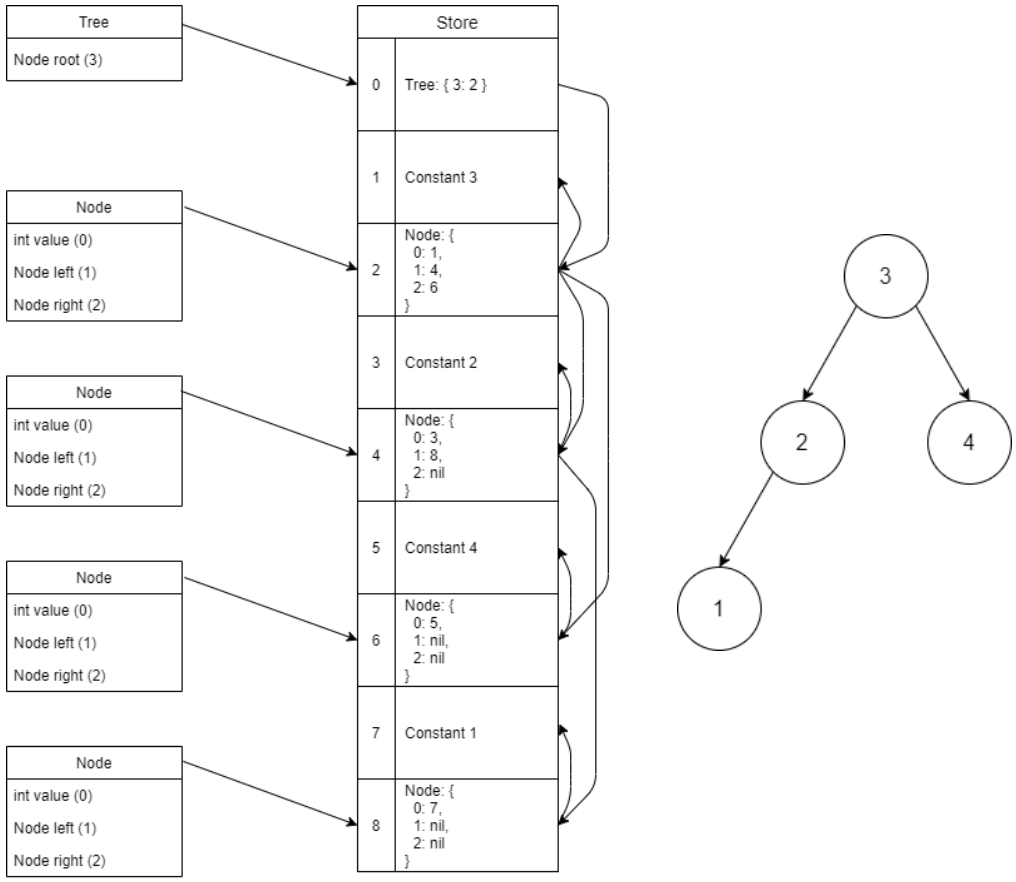

Figure 2.2: Example of object environments and store after constructing a binary search tree, shown to the right, from `test/BinaryTreeArrayUsage.rplpp`

The store expressions can be integers, objects or arrays. An array is implemented as a list of locations in the store. Objects are constructed in the same way. Each object variable also points to a location in the store, which holds a value, i.e. an integer, or another location, i.e. to an array or another object. By using references to a value, one avoids having to update multiple copies of the same value or object whenever a change is made.

Figure 2.2 shows a `Tree` and a couple of `Node` obejcts. Each object points to a location in the store, i.e. `Tree` points to location 0 and its variable, `root`, points to location 2, a `Node` object. The `Node` object has its own variables, `value`, `left` and `right`, which points to locations 1, 4 and 6, respectively.

An object variable is based on the its scoped variable, which is the number next to the variable name. By using store locations we can rely on the scoped variables as we can have multiple definitions of variable 0 (`value`) without experiencing overshadowing.

2.1.3 Object Structure

Since we are dealing with an object-oriented language, the natural choice for representing an object would be by using an established standard within the OOP community. Two of the most common choices are JSON and XML. XML serves more purposes than just data-interchanging, i.e. as a markup language, but it is also more verbose, complex and not very human-readable [7]. Thus, the choice landed on JSON as it consisted of the necessary features, i.e. the ability to naturally represent arrays, while being easy to read.

JSON (Javascript Object Notation) [4] is data-interchanging language. Due to it being lightweight, language independent and human-readable it is a popular choice for communicating between servers and clients.

In addition to having a human-readable data representation of the state, we also enable a potential future interaction with other mainstream OOP languages.

We can extract the state of irreversible languages, such as Java or C#, which can then be reversed using a similar ROOPL implementation by using the irreversible output state as input. Figure 2.3 shows an example of the object structure of the input and output for computing Fibonacci numbers.

However, ROOPL does not currently support any loading of input state. The object representation is strictly used for presenting the current state in runtime errors or as output.

Fibonacci.rpplp

```
1  class Program
2      int n
3      int x1
4      int x2
5
6      method main()
7          n += 4
8          call fib()
9
10     method fib()
11         if n = 0 then
12             x1 += 1
13             x2 += 1
14         else
15             n -= 1
16             call fib()
17             x1 += x2
18             x1 <=> x2
19         fi x1 = x2
```

Input state

```
{
    "n": 4,
    "x1": 0,
    "x2": 0
}
```

Output state

```
{
    "n": 0,
    "x1": 5,
    "x2": 8
}
```

Figure 2.3: Input/output JSON representation of a Fibonacci ROOPL program

2.2 Exception Handling

Runtime exceptions are an important part of constructing a program. It is often hard to consider all possible states a program can have. Thus, by showing where the code threw an exception with a relevant error message and related stack trace, a programmer more easily debug their code. On the other hand, if an exception is imprecise or lacks information on the actual issue, it is useless and potentially misleading.

In the current ROOPL setup it is not possible to get detailed runtime error messages when executing a compiled program on a `PendulumVM`. The usability of constructing ROOPL program is therefore not very good. Programmers new to the language would be discouraged as it would be impossible to guess what caused some runtime exception. The interpreter exception handling aims to increase the usability of ROOPL by showing information that might be useful when debugging.

```
1  class Foo
2      int z
3
4      method set(int y)
5          z ^= y
6
7      method get(int y)
8          y ^= z
9
10 class Program
11     int u
12
13     method main()
14         local Foo f = nil
15         local int x = 5
16         call f::set(x)
17         call f::get(u)
18         delocal int x = 5
19         delocal Foo f = nil
```

```
Exception thrown: Calling uninitialized object
  In "call f::set(x)"
  where { "u": 0, "f": nil, "x": 5 }

  In "local int x = 5 ... delocal int x = 5"
  where { "u": 0, "f": nil }

  In "local Foo f = nil ... delocal Foo f = nil"
  where { "u": 0 }
```

Figure 2.4: Program calling method of uninitialized object

A simple approach has been used when constructing the exception layout. An exception consists of a cause and a stack trace. The cause is the exception message thrown. The stack trace contains the statements called and the store at the given time.

Figure 2.4 shows an exception thrown for when trying to call a method for an uninitialized object. The exception tells the programmer that an uninitialized object was called but not which one or where. The stack trace shows the line that threw the exception and the store at the given time. One may note that `f` is `nil`, which corresponds to the thrown exception. The trace then narrows down in which statement the exception occured, if the program consisted of multiple `call f::set(x)` statements.

3 Program Inverter

Running a program in the forward direction to generate an output state from some input is the general execution path for any programming language. However, with reversible programming languages, such as ROOPL, one can run a program in the backward direction to generate an original input state from some output. This feature is one of the main points for using the reversible programming paradigm.

Thus, having the possibility of inverting a program enables running in the backward direction. It provides the user with the inverted program, which means that one only has to setup the output state and then run the program to generate the original input. Inverting a ROOPL program depends on some inversion rules, which is presented in section 3.1.

One should note that the program inverter and interpreter does not share the same underlying inverting flow. It is therefore entirely possible that there is/can be variation in the inverted program in each of the two implementations. However, each of the inversion flows have been tested and compared and no variations have been found as of yet.

3.1 Program Inversion

The purpose of a program inverter is to transform a program `P` into its inverse $\texttt{P}^{-1}$. Inversion of a ROOPL program is performed by recursive descent over its structure and inverting its statements.

For inverting ROOPL program statements, we use the ROOPL inversion rules presented by Haulund [5] with the extensions by Cservenka [3]. The original inversion rules contained errors in the `if`, `from` and `local` statements. The `if` and `from` statements did not swap the position of the entry, $\texttt{e}_1$, and exit, $\texttt{e}_2$, expressions. Furthermore, the `local` statement now uses numbered expressions, $\texttt{e}_1$ and $\texttt{e}_2$ as the entry and exit expressions are different and they change place after inversion. Figure 3.1 has corrected the inversion of these statements.

$\mathcal{I}[\![\textbf{skip}]\!] = \textbf{skip}$
$\mathcal{I}[\![x \texttt{ += } e]\!] = x \texttt{ -= } e$
$\mathcal{I}[\![x \texttt{ ^= } e]\!] = x \texttt{ ^= } e$
$\mathcal{I}[\![x[e_1] \texttt{ += } e_2]\!] = x[e_1] \texttt{ -= } e_2$
$\mathcal{I}[\![x[e_1] \texttt{ ^= } e_2]\!] = x[e_1] \texttt{ ^= } e_2$
$\mathcal{I}[\![\textbf{new } c\ x]\!] = \textbf{delete } c\ x$
$\mathcal{I}[\![\textbf{delete } c\ x]\!] = \textbf{new } c\ x$
$\mathcal{I}[\![\textbf{call } q(\dots)]\!] = \textbf{uncall } q(\dots)$
$\mathcal{I}[\![\textbf{uncall } q(\dots)]\!] = \textbf{call } q(\dots)$
$\mathcal{I}[\![\textbf{if } e_1 \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi } e_2]\!]$
$\mathcal{I}[\![\textbf{from } e_1 \textbf{ do } s_1 \textbf{ loop } s_2 \textbf{ until } e_2]\!]$
$\mathcal{I}[\![\textbf{construct } c\ x \quad s \quad \textbf{destruct } x]\!]$
$\mathcal{I}[\![\textbf{local } t\ x = e_1 \quad s \quad \textbf{delocal } t\ x = e_2]\!]$

$\mathcal{I}[\![s_1\ s_2]\!] = \mathcal{I}[\![s_2]\!]\ \mathcal{I}[\![s_1]\!]$
$\mathcal{I}[\![x \texttt{ -= } e]\!] = x \texttt{ += } e$
$\mathcal{I}[\![x \texttt{ <=> } e]\!] = x \texttt{ <=> } e$
$\mathcal{I}[\![x[e_1] \texttt{ -= } e_2]\!] = x[e_1] \texttt{ += } e_2$
$\mathcal{I}[\![x[e_1] \texttt{ <=> } e_2]\!] = x[e_1] \texttt{ <=> } e_2$
$\mathcal{I}[\![\textbf{copy } c\ x\ x']\!] = \textbf{uncopy } c\ x\ x'$
$\mathcal{I}[\![\textbf{uncopy } c\ x\ x']\!] = \textbf{copy } c\ x\ x'$
$\mathcal{I}[\![\textbf{call } x :: q(\dots)]\!] = \textbf{uncall } x :: q(\dots)$
$\mathcal{I}[\![\textbf{uncall } x :: q(\dots)]\!] = \textbf{call } x :: q(\dots)$
$= \textbf{if } e_2 \textbf{ then } \mathcal{I}[\![s_1]\!] \textbf{ else } \mathcal{I}[\![s_2]\!] \textbf{ fi } e_1$
$= \textbf{from } e_2 \textbf{ do } \mathcal{I}[\![s_1]\!] \textbf{ loop } \mathcal{I}[\![s_2]\!] \textbf{ until } e_1$
$= \textbf{construct } c\ x \quad \mathcal{I}[\![s]\!] \quad \textbf{destruct } x$
$= \textbf{local } t\ x = e_2 \quad \mathcal{I}[\![s]\!] \quad \textbf{delocal } t\ x = e_1$

Figure 3.1: ROOPL statement inverter, fixed from [5] and [3]

3.2 Design

As with interpretation, the program inverter utilizes the existing parsing and scope/type checking flow. This helps to ensure that the inverted program is a well-typed ROOPL program. However, this also means that any comments will not be presented in the inverted program as they are removed during parsing.

The inverter inverts the entire program which includes the `main` method. As discussed in section 2, due to current design constraints the `main` method is not reversible. Thus, the program inversion will result in an invalid `main` method, which needs to be updated to use the output state.

The decision is based on the fact that program inversion should be applied to the entire program. In addition, whether or not `main` had been inverted, `main` would still not be a valid method in the inverted program due to its dependency on setting up the input state.

4 Web Interface

The web interface design and functionality are based on earlier work by Budde and Nielsen [8] [2] on an interpreter for JANUS. JANUS is a reversible imperative language. The interface will therefore be familiar for users that has already used the JANUS web interface making the ROOPL interface easy to use.

The aim for the interface was to provide easy access to writing and testing ROOPL programs. Thus, the compilation functionality was preserved. As such, it is possible to run, invert or compile a program. Since the compiled programs are very long, the compilation option downloads the compiled PISA code, which can then be used with the `PendulumVM`.

Figure 4.1: ROOPL web interface

The interface contains some example programs that are designed to provide an understanding of the ROOPL syntax or its functionality. The simple example programs for performing arithmetic, using loops or conditionals and method calling. Whereas the more advanced example programs provide an overview

of the functionality and what tasks ROOPL can perform. Figure 4.1 shows the interface, the list of example programs and result of computing the fourth Fibonacci number.

One could also want to invert a program. Figure 4.2 shows the result of inverting a simple conditional ROOPL program.

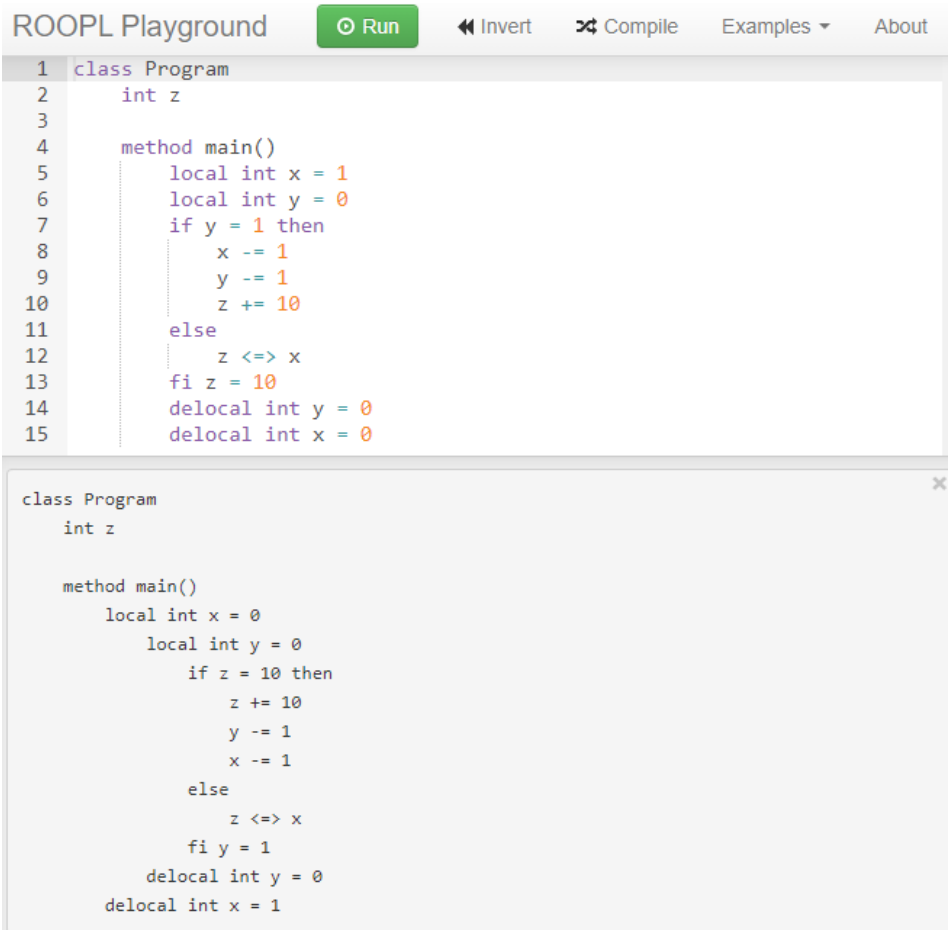

Figure 4.2: The result of inverting a simple conditional program

The web interface consist of a simple HTML page, some Javascript functions for communicating with the backend and a PHP backend for executing the ROOPL functionality. The PHP code is used for fetching examples, and running the interpreter, inverter and compiler. The Javascript sets up the input method, i.e. running or compiling, and shows the execution result in an output pane or initiate a download, if `compile` is invoked. If the execution throws an exception, the exception is shown in a red output pane. An exception thrown during the static analyzing, or at runtime, as described in section 2.2 are shown in a red output pane. An example of an exception, based on the program shown

in section 2.2, in the web interface is shown in figure 4.3.

Figure 4.3: ROOPL web interface showing a thrown exception in the output pane

5 The ROOPL System

One of the main goals of this project was to increase usability and accessibility of ROOPL. Thus, for promoting reproducability and ease of use by utilizing tools such as containerization and MAKE. The developed system can be found in [6].

MAKE allows defininining a set of targets that executes a series of commands, which eliminates the need for new users to remember specifics.

DOCKER enables programmers to construct and interact with containers. A container allows for running an application in an isolated instance on a host that contains only the necessary dependencies [1]. It is therefore possible to reproduce the same setup on multiple machines without experiencing any derivations.

DOCKER is used for containerization of the web interface and ROOPL system. The project defines `Dockerfile`, which is a text document that specifies the commands needed for constructing a container image. It is then possible for any programmer to build the image and run the ROOPL web interface without having to install anything but DOCKER.

`build`	Builds the ROOPL source code.
`install`	Installs the ROOPL executable on the machine. If executable has not been built then the `build` target will be run.
`docker-build`	Builds the container image of the web interface.
`docker-run`	Runs the web interface image and exposes the interface on port 8080. If the image has not been built then the `docker-build` target will be run.
`rplpp-test`	Tests ROOPL by executing the programs in the `test/` folder. Each test case shows the output state. Expected output states can be found in folder `test/result/`.

Figure 5.1: `Makefile` targets

An included `Makefile` specifies targets for building, installing and testing

ROOPL, see figure 5.1, which checks whether any changes to the source code still produce runnable code. Furthermore, targets for building the container image and running the image has been included to allow for quickly setting up or testing the ROOPL language.

By utilizing these tools in the ROOPL system, one can easily get a working copy of ROOPL running as intended by the designers. It also enables programmers to make changes to the underlying ROOPL definitions and test their changes in an isolated environment to assert whether the language is broken. A list of the tool versions are shown in figure 5.2.

DOCKER	20.10.2
HASKELL STACK	lts-16.21
MAKE	4.2.1

Figure 5.2: Version of tools used

6 | Conclusion

Programming today is dominated by irreverisble programming languages, such as Python, C++ and C#. The reversible programming paradim is not widely used, most likely due to its design which might be seen as constraints. Its primary functionality is to be able to run a program in forward and backward direction. However, this also introduces additional complexity because one would have to consider a program in both directions, i.e. what would an entry and exit state look like. In addition it can be hard to find the correct hardware that can run reversible programs.

I set out to increase usability and accessibility of a reversible object-oriented programming language (ROOPL). One of the biggest hurdles is that running a compiled ROOPL program did not provide any runtime exceptions, due to the design of the `Pendulum` virtual machine. Furthermore, the output state was not shown in the execution result. The programmer would have to know the output state if it were to be used, thus, rendering the program execution obsolete.

As a step to increase the usability, an interpreter for ROOPL was constructed. An interpreter allows a programmer to immediately run their code. This enables one to avoid issues like having to wait for compilation or potentially executing on another machine. The interpreter uses an internal structure that handles tracking of an object call stack and variable references. Every object keeps track of its own environment. thus avoiding problems such as accessing variables which should not be accessible by the object or used in a call argument.

In addition to creating an interpreter, one would likely want to be able to invert their program. The purpose of inverting a program is to be able to run it in the backward direction using an output state as input. As such, a program inverter was implemented as well. The inverter obeyed a set of inversion rules set by Haulund [5] and Cservenka [3]. The program inverter inverts the entire program, including the `main` method, which actually renders the output program invalid. One has to update the inverted `main` with the output state and remove references to an input state to make the program valid.

For accessibility, a web interface was created. This enables anyone with a browser to construct and run ROOPL programs. The interface is based on ear-

lier work by Budde and Nielsen [2]. The ROOPL web interface supports every functionality of the command-line version, i.e. interpreting, program inversion and compilation.

6.1 Future Work

This project has introduced some interesting features and changes to ROOPL. One of the more interesting ideas is to be able to specify an initial state for a program. Currently it is only possible to generate a state programmatically in the `main` method. However, this introduces the issue that a ROOPL program is never truly reversible because the inverted program is invalid.

A potential benefit for providing an initial state is that one can freely run the program in the forward or backward direction. In addition, it is likely easier to construct more advanced programs by being able to specify a complex state using JSON rather than programmatically in ROOPL.

While creating the interpreter, we found out that ROOPL does not support arrays as method arguments due to some issue in the type checker. Thus, by fixing the type checker, as well as potentially updating the downstream parts, i.e. interpreter and compiler, one can use arrays in method parameters.

Currently the interpreter and program inverter does not share the same inversion code. Thus, it would be beneficial to construct a shared inversion module to avoid potential mismatch issues when using each of the inversion flows.

References

[1] What is a container? https://www.docker.com/resources/what-container.

[2] Budde, M., and Nielsen, C. S. Jana: An interpreter for Janus, the reversible programming language. https://github.com/mbudde/jana.

[3] Cservenka, M. H., Glück, R., Haulund, T., and Mogensen, T. Æ. Data structures and dynamic memory management in reversible languages. In *Reversible Computation* (2018), J. Kari and I. Ulidowski, Eds., vol. 11106 of *Lecture Notes in Computer Science*, Springer International Publishing, pp. 269–285.

[4] ECMA International. ECMA 404: The JSON data interchange format, 2017. http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf.

[5] Haulund, T., Mogensen, T. Æ., and Glück, R. Implementing reversible object-oriented language features on reversible machines. In *Reversible Computation* (2017), I. Phillips and H. Rahaman, Eds., Springer International Publishing, pp. 66–73.

[6] Hay-Schmidt, L. ROOPLPP: The ROOPL++ reversible programming language. https://github.com/haysch/ROOPLPP.

[7] Parr, T. Soapbox: Humans should not have to grok XML, Aug 2001. https://www.ibm.com/developerworks/xml/library/x-sbxml/x-sbxml-pdf.pdf.

[8] Yokoyama, T., and Glück, R. A reversible programming language and its invertible self-interpreter. In *Proceedings of the 2007 ACM SIGPLAN Workshop on Partial Evaluation and Semantics-based Program Manipulation, 2007, Nice, France, January 15-16, 2007* (2007), G. Ramalingam and E. Visser, Eds., ACM, pp. 144–153.