6CCS3PRJ Final Year

Adding for-loops and procedure calls to a reversible programming language

Final Project Report

Author: Christian Impollonia

Supervisor: Maribel Fernández

Student ID: 1902896

April 4, 2022

Abstract

The goal of this project is to add for-loops and procedure calls to a pre-existing reversible programming language called RIMP. The main reason to do this is because these two operations are crucial features of most modern programming languages, and they help improve the usability of RIMP. The reasons to implement a reversible language on the other hand, are many: from preventing heat loss that comes with irreversible computation, to improving debugging, these and more will be explored in detail.

This reversibility could be achieved in two ways: either by forcing the user to write in a reversible way and enforcing that notation in compilation, or it can be done by changing the hidden semantics of how the computer processes the code. The latter is what I have chosen.

One of the obstacles encountered was handling recursion within the procedure calls, as it is a complex task to find out the number of recursive calls that have happened at runtime, and reverse them accordingly.

I will finally show a piece of software that fully implements this theory, by taking as input a parse tree of a RIMP program and interpreting it in a completely reversible way, so it is able to go back to the initial state before that program was run.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Christian Impollonia

April 4, 2022

Acknowledgements

I want to thank my supervisor, Professor Maribel Fernández, for the continuous help she provided and for always being available to address any doubts I had throughout this project. I would also like to thank my collegues who worked on the other variants of this project: Mateusz Bednarski, Paisarn Phicheansoontorn and Birwa Shiwani, whose ideas and thoughts during the meetings were of great help. Finally, I am thankful to my family and friends for constantly supporting me in my daily life, without them this work wouldn't be possible.

Contents

Chapter 1

Introduction

Programming languages are generally irreversible. Computation such as 3 + 3 , where the values are user inputs, is something you can't normally recover from. This is because, from the result of 3 + 3, which is 6, and the addition function, it is impossible to know for sure the values that got us to that point. They could be 3 and 3, but they could also be 4 and 2, 5 and 1, and so on. This kind of irreversible computation leads therefore to erasing data, which has been proven to be the cause of computer heat loss by Landauer in 1961. [5] So with software and hardware that are completely reversible, one could theoretically have computers that don't overheat.

Another way to look at it is through functions. We can consider a program as a black box, so we are not interested in what is inside it but only in its result. It is a function f, which can take an input x from the domain X and produce output y from the codomain Y. The output y is obtained of course by calculating f(x).

To be able to go back from an element y to its respective element x, there can be no ambiguity. So if there is 2 or more elements from X that obtain the same element y, there is no way to tell which element y was mapped from. This means that for a function f to be reversible, it needs to be injective. This explains why the previous example breaks reversibility, as the addition of two numbers is not an injective function.

There already are some programming languages that follow all these rules, like Janus [6] and Arrow [9]. Another one is RIMP, a simple reversible programming language written by Professor Maribel Fernández and Dr Ian Mackie, which obtains reversibility by modifying the back-end of how a language is processed in its semantics. [2]

1.1 Aims and Objectives

The requirements of this project are to extend RIMP, by adding two features that it is missing:

- A FOR-LOOP CONSTRUCT , as RIMP only has the possibility of creating while loops, with which you can technically model any loop, but the structure of a for-loop is quite handy to have.
- PROCEDURE CALLS , to avoid code duplication and to make long programs really possible without having to copy-paste entire blocks, procedure calls are essential.

After extending the semantics of RIMP, an interpreter was made available, to evaluate and test abstract syntax trees written with these features.

1.2 Report Structure

This report starts with an overview of the motivations for reversible computing and reversible programming languages in particular, as well as some background from previous work done on the topic. It looks into how that work compares to what I have done, and how it differs from it. Then there is a brief explanation of RIMP and how it works. In the report body, the technical side of the project is covered, specifically the extension of the language, starting from the changes made in the lowest, more theoretical level, up to the higher level, including the design and specification of the software. In the implementation chapter, there is a full description of how the interpreter works and how it is used to test the validity of the new semantics. There is finally an evaluation and a conclusion discussing what was achieved and what else can be done to improve it.

Chapter 2

Background

2.1 Literature review

The work done in this report has importance mostly on the basis of the crucial discovery by Landauer in 1961: that irreversibility places a lower limit on energy consumption [5]. What does this mean exactly?

It means that when we perform an irreversible operation with a computer, it needs to release a minimum amount of energy, which usually translates into heat. And heat management has become an extremely important issue for modern devices.

It was discovered a little later (from [8])that what causes energy consumption is the bit-erase, or more specifically a deterministic bit-reset: the action of irreversibly resetting a bit with no recorded knowledge of its previous state. Any other operation, like assignment or addition for example, does not dissipate heat, presuming no bits are erased in the process.

For this reason, reversible computing, while it can require some additional work to achieve, is very powerful. Quantum computing for example, is completely reversible, as it is made of unitary operations. This is all discussed in Perumalla's book, [8], which gave a complete overview of reversible computing and all its facets.

The main paper this project is based on is [2], which specifies in detail a reversible programming language called RIMP. This programming language, while Turing complete, can be extended with two key features: for-loops and procedure calls. This is what will be covered in this paper: continuing the work done on RIMP by upgrading the language to include these new functionalities.

2.2 Additional motivations

Aside from the points made above, other reasons to implement reversibility can be: [8]

- PARALLEL COMPUTING , it can be used to improve synchronization as well as concurrency of instructions in programs
- DEBUGGING , by being able to pause and go back and forth within the computation as much as we want, instead of storing a very large trace which is impractical
- FAULT DETECTION , as results can be proven correct by checking if the reverse computation leads back to the initial state
- CRYPTOGRAPHY , as encryption and decryption can often be considered reversible operations

Functions of everyday programs are often forced to be reversible to allow specific features, like browsers that let you go back and go forward, or text-editors that allow us to cancel a mistake we made or "un-cancel" it.

2.3 How to achieve reversibility

There is more than one method to maintain reversibility in a program, two of them are explained in [8] : The one invented by Charles Bennet, which has as a final result the output and the original input, to be able to go back deterministically. The drawback is that it uses more space. This is what I am going to use.

An alternative way would be Owen Maroney's method,which only has the output as a result, as it uses the normal computation going forward, but relaxes the constraints going backwards, by making the computation non-deterministic. This way it saves space, but the input it finds might not be the same as the original input, it could just be any other input that obtains the same output [8].

Of course, it also depends on the definition that is given to reversibility. For some, reversibility could mean being able to go back to the initial state the program started from, ignoring the intermediate steps, while others could have a stricter take on it, by enforcing every single computation step to be reversible. The former is more relevant in our case, but the way RIMP is designed allows either method.

2.4 RIMP Specifics

RIMP is a simple reversible imperative language explained in detail in [2], but I will briefly give its specifics and how it works:

It is a Turing complete language with familiar features like if-conditions and while loops. It has assignments (:=), but also a unique feature which is the opposite of assignment (=:).

The way it achieves reversibility is by having an inversion function, which transforms pieces of code into their reverse. So for example, if we run a for-loop and then the inversion of that for-loop, we will always get back to our initial state.

2.4.1 Semantics

The semantics of a programming language are a way to specify the meaning of the instructions of the language, and it can be done informally (described through wording), or formally through methods like small-step or big-step semantics, or an abstract machine.

These are the big-step semantics of RIMP, written with the information found in [2] :

$$\frac{}{< c, s > \Downarrow_{RIMP} < c, s > \quad if\ c \in Z}(const) \tag{2.1}$$

$$\frac{}{<!l, s > \Downarrow_{RIMP} < n, s > \quad if\ s_1(l) = n}(var) \tag{2.2}$$

$$\frac{< E, s > \Downarrow_{RIMP} < b1, s >}{< \neg E, s > \Downarrow_{RIMP} < b, s > \quad if\ b = not\ b1}(not) \tag{2.3}$$

$$\frac{< E1, s > \Downarrow_{RIMP} < n1, s > \quad < E2, s > \Downarrow_{RIMP} < n2, s >}{< E1\ op\ E2, s > \Downarrow_{RIMP} < n, s > \quad if\ n = n1\ op\ n2}(op) \tag{2.4}$$

$$\frac{}{< skip, s > \Downarrow_{RIMP} < skip, s >}(skip) \tag{2.5}$$

$$\frac{< C1, s > \Downarrow_{RIMP} < skip, s' > \quad < C2, s' > \Downarrow_{RIMP} < skip, s'' >}{< C1; C2, s > \Downarrow_{RIMP} < skip, s'' >}(seq) \tag{2.6}$$

$$\frac{< E, s > \Downarrow_{RIMP} < n, s >}{< l := E, s > \Downarrow_{RIMP} < skip, s[l \to (n, +(n1, s_2(l)))] >}(:=) \quad where\ n1 = n - s_1(l) \tag{2.7}$$

$$\frac{< E, s' > \Downarrow_{RIMP} < n, s >}{< l =: E, s[l \to (n, +(n1, v))] > \Downarrow_{RIMP} < skip, s' >}(=:) \quad where\ s' = s[l \to (n - n1, v)] \tag{2.8}$$

$$\frac{< E, s > \Downarrow_{RIMP} < true, s > \quad < C1, s > \Downarrow_{RIMP} < skip, s' >}{< if\ E\ then\ C1\ else\ C2, s > \Downarrow_{RIMP} < skip, s' >}(ifT) \tag{2.9}$$

$$\frac{< E, s > \Downarrow_{RIMP} < false, s > \quad < C2, s > \Downarrow_{RIMP} < skip, s' >}{< if\ E\ then\ C1\ else\ C2, s > \Downarrow_{RIMP} < skip, s' >}(ifF) \tag{2.10}$$

$$\frac{< E, s > \Downarrow_{RIMP} < true, s > \quad < C, s > \Downarrow_{RIMP} < skip, s1 > \quad < while\ E\ do\ C, s1 > \Downarrow_{RIMP} < skip, s2 >}{< while\ E\ do\ C, s > \Downarrow_{RIMP} < skip, s2 >}(whileT) \tag{2.11}$$

$$\frac{< E, s > \Downarrow_{RIMP} < false, s >}{< while\ E\ do\ C, s > \Downarrow_{RIMP} < skip, s >}(whileF) \tag{2.12}$$

As the only key difference between the semantics of RIMP and the ones of a generic imperative language like SIMP(see[2] or [1]) are the "assign" (:=) and "un-assign" (=:) operations, it is important to note how they work.

To understand that, an explanation of the store in RIMP is crucial. It is different from how a usual store would work, as it stores, for each variable, not just its value, but an additional value v, which is defined recursively as either 0 or +(n, v), where n is a number. These recursive

values in v are the different runtime values the variable in question has had since its creation. So in practice, the store is represented as a pair (k, v) where k is just the computed value of v, which makes it a bit redundant but convenient to avoid constant computation costs. To look up inside the store we use a function s(l), where l is a variable, and we have two additional functions s_1(l) which obtains k, and s_2(l) which obtains v. This is why the (2.2) axiom needs s_1(l) = n, as s_1(l) is the current runtime value of l.

Having explained this, the way the operation l := E (2.7) works is by updating the store for the variable l, by replacing k (the current runtime value of l) by n (the result of E), and by replacing v (the recursion of the different values of l) with +(n-k, v), so adding a new layer to the recursion, with the new value of l subtracted by the old value of l.

The reverse assignment operation l =: E (2.8)(which is not meant to be used by the user, but is just internal) does the opposite of course: given a variable l which is assigned to a value k equals to n (the result of E), we can update the store for l by replacing k (the current runtime value of l) by k - n1, which is same n1 we calculated before in the assignment operation. It was equal to the new value assigned minus the runtime value of l before assignment. This means that by doing k - n1, where k is the post-assignment value, we will get the pre-assignment value. On the other hand, the value v (the recursion of the different values of l) is updated to just be the previous v, by removing that top layer we added during assignment [2].

2.4.2 Augmentation function

Before the previously mentioned inversion function, the other main function RIMP uses is the source code augmentation function. This function, which we can call *aug*, makes some under-the-hood modifications to the source code that help reverse certain commands later on.

It is important to take a look at the if-statement and the while loop in particular:

For if-statements, what happens is that a series of assignments is added at the beginning, before the command is run. What these assignments do is simply create duplicate variables of all the variables used in the condition E. So if, for example, variable *l1* and *l2* were used in condition E of an if-statement C, the augmented version of C would start with a sequence of assignments: *l1'* := *l1* and *l2'* = *l2*.

l1' and *l2'* are the duplicate variables. Then, the if-statement is run, but with a slight difference: the condition of it is not set to be E, but E', that is the same boolean condition, with the difference that every variable is replaced by its duplicate. The reason this is done is so

any variable used in the boolean condition of an *if* is not used anywhere else in the body of that if-statement. Knowing that, this same unchanged boolean condition can be used as the condition of the reversed if-statement, as we will see later on.

As for the while-loop, a *counter* is added, which does not appear anywhere in the semantics. This counter is initialized before the while is started, and increased by 1 for every iteration. A different counter is used for each while loop: so while-loop 1 will have counter_1, while-loop 2 counter_2 and so on. So the generic counter will just be written as counter_i. Of course the *counter* variable becomes a protected name, so it can't be used by programmers.

A last modification that is done is the creation of a lookup table T, which stores the condition E of each while loop in the program. The reason all this is done will be more clear when we look at the reversibility of the while loop.

2.4.3 Inversion function

As said before, the way RIMP manages reversibility is by having a function, called *rev*, which takes a piece of code P and computes the reverse of P.

These are the results rev gives for each expression or operation we have in RIMP: [2]

$$rev(E) = E \tag{2.13}$$

$$rev(x := E) = x =: E \tag{2.14}$$

$$rev(x =: E) = x := E \tag{2.15}$$

$$rev(skip) = skip \tag{2.16}$$

$$rev(C1;\ C2) = rev(C2);\ rev(C1) \tag{2.17}$$

$$rev(if\ E\ then\ C1\ else\ C2) = if\ E\ then\ rev(C1)\ else\ rev(C2) \tag{2.18}$$

$$rev(while\ E\ do\ C) = while\ !counter_i > 0\ do\ rev(C)\ if\ counter_i \notin E \tag{2.19}$$

$$rev(while\ !counter_i > 0\ do\ C) = while\ E\ do\ rev(C)\ if\ T(i) = E \tag{2.20}$$

$$\tag{2.21}$$

This function, like the augmentation function above, will be modified in this paper to include cases for for-loops and procedure calls as well.

As can be seen, the if-statement (2.18) is very easy to reverse, taking into account the previous

augmentation. As it lets us assume unmodifiable variables in the condition E, we just need to use that same condition for an *if* that has reversed bodies.

The *counter* seen previously is used in the condition for the reversed while loop, which decreases it by 1 each time and stops when the counter is 0, as it was when initialized. So what the counter does is just memorize the amount of times the loop was run, so that the reverse of the loop can be run the same exact amount of times.

The reverse of the while loop simply uses this counter as a new condition, with the rev function making sure that it will only reverse if the variable "counter" is not present in the condition E. If the "counter" is present, it will reverse the inverse of the loop to the standard while-loop.

The variable T at the end of 2.20 is the lookup table created in the *aug* function. To obtain back the original condition E from the "counter" condition which doesn't tell us much about the previous one, the only way is to have that lookup table. T(i) simply searches the table for the condition E for that while-loop i.

2.4.4 How RIMP achieves reversibility

As thoroughly explained in [2], the way RIMP achieves reversibility can be by having an abstract machine (a more detailed and low level way of specifying semantics) with two key elements: (there is five in total, but I will focus on explaining the two stacks that are important for reversibility)

- A CONTROL STACK C which contains all the instructions in the program, and pops each instruction out as it is executed
- A BACK STACK B which is initially empty, where instructions are gradually added as they leave c, after being reversed by the function rev.

This way, when the program will finish executing, we will have an empty c, and a b containing the entire reversed program. By simply swapping the two stacks, we can now run the same program in reverse. This can also easily be done half way through the program, with both stacks not empty.

The way RIMP reversibility will be achieved and implemented in this paper will slightly differ, as there is no use of an abstract machine or back-stack, but the big-step semantics are used for evaluation instead. This will be explained in detail in the Software Design section as well as the Implementation section.

Chapter 3

Language Design

3.1 Notation and syntax

The first step to extend the language RIMP with for-loops and procedure calls is to specify what notation we will use for these operations, as it can vary between different programming languages.

3.1.1 For-loops notation

First of all I will specify the notation for the for-loops that will be added, and what they do. They work by having four operations:

- THE INITIAL ASSIGNMENT that is run before the for-loop starts, and only once, to initialize the counter. It is enforced that the command in this section is a variable assignment for any variable v. This is done to facilitate reversibility.
- THE BOOLEAN CONDITION that is checked before each loop iteration. If it evaluates to true the body of the loop is run once, then the condition is checked again. There could be the same variable v from the initial assignment inside the condition, which is the classic format, but it could also just be any boolean expression, even just *true* (which would cause an infinite loop)
- THE TAIL COMMAND that is run at the end of each loop iteration as soon as it finished running the body. It can be a counter increment for example, but it can even be skip.
- THE LOOP BODY which contains the main code we want to loop around. If the loop condition is never satisfied, it will run 0 times.

To avoid ambiguity in the distinction between the 4 operations, the notation was designed to have a keyword between each element of the for-loop. So the loop will start with ”for”, then have a ”cond” between the initial assignment and the boolean condition, then have an ”step” before the tail command, and then a ”do” between that and the loop body. The loop body, exactly like the while loops, can be *optionally* wrapped by round brackets, useful in case the body of the *for* is a sequence. In case the brackets are not written, the parser will assume only the first command in the sequence is part of the body. To give an example of a valid RIMP for-loop:

$$for \;\; i := 0 \;\; cond \;\; !i < 10 \;\; step \;\; i :=!i + 1 \;\; do \;\; (n :=!n + i) \tag{3.1}$$

Or, in a more abstract way, it is defined generically as:

$$for \;\; C1 \;\; cond \;\; E \;\; step \;\; C2 \;\; do \;\; (C3) \tag{3.2}$$

where we know C is a command (of which C1 is strictly an assignment) and E is an expression, a boolean in this case.

3.1.2 Procedure calls notation

The same needs to be done for procedure calls, which are slightly simpler in structure.
They are built of just two sections: the first one, ”call” is a new keyword that is added, like ”for” in the for-loop. While the second is the name of the function that is being called. So an example of a procedure call in RIMP can be:

$$call \;\; foo \tag{3.3}$$

Or in a more generic definition:

$$call \;\; N \tag{3.4}$$

Where N is any function name, that has been previously specified.
Of course this is not enough, as it can be seen that in RIMP there is no way to actually give a name to a function. So there needs to be notation to also associate a function, or more generally a command C, to a name we can call. To do this, the following structure is established: a function definition starts by the keyword ”def”, then is followed by the name we give to the function, then the word ”as”, and ending with the command (or sequence of commands) C, wrapped by *optional* round brackets to avoid ambiguity when sequences are present, just like

the for-loops. Below is an example:

$$def \quad foo \quad as \quad (i := 0) \tag{3.5}$$

So now, if we want to set the variable i to 0, we don't need to write it again, but we can just write *call foo*. This is how it is defined in a generic way:

$$def \quad N \quad as \quad (C) \tag{3.6}$$

Where we know C is a generic command or list of commands from RIMP, while N is just any string excluding the protected words.

3.2 Extending the semantics of RIMP

The next step is to give "meaning" to these two features. This is crucial so that these instruction get recognized and have specific effects. So the big-step semantics of RIMP was extended with some new entries.

3.2.1 Big-step semantics of for-loops

To include for-loops in RIMP, a rule was added to the big-step semantics:

$$\frac{< C1;\ while\ E\ do\ (C3; C2), s > \Downarrow_{RIMP} < skip, s1 >}{< for\ C1\ cond\ E\ end\ C2\ do\ C3, s > \Downarrow_{RIMP} < skip, s1 >} (for) \tag{3.7}$$

In concept, a for-loop can be quite easily translated into a while-loop. The only real difference between the two is that the for-loop has an initial assignment which is not repeated, but that can easily be considered the start of a sequence of commands.
So if we look at the rule (3.7), we start by evaluating the initial assignment, C1, which is run regardless of the loop body being run or not. After that, in sequence, the for-loop corresponds to a while loop with condition E that has as a body the main for-loop body C3 and then, appended at the end, the tail command C2, which can be considered part of the body of the while loop. The important thing is that C2 is run at the end, or it won't follow the exact meaning of the *for*.

3.2.2 Big-step semantics of procedure calls

To include procedure calls in RIMP, four sections were added to the big-step semantics: one for function definition, one for function call and two un-call and un-def rules, that are not supposed to be used by the user, but they should only invert what the call and def do, similar to the un-assign operation.

$$\frac{}{< def\ N\ as\ C, s > \Downarrow_{RIMP} < skip, s >}(def)\ With\ (C, rev(C))\ added\ to\ table\ D \tag{3.8}$$

$$\frac{< C, s > \Downarrow_{RIMP} < skip, s1 >}{< call\ N, s > \Downarrow_{RIMP} < skip, s1 >}(call)\ \ Where\ C = D(N)_1 \tag{3.9}$$

$$\frac{}{< undef\ N, s > \Downarrow_{RIMP} < skip, s >}(undef)\ With\ (C, rev(C))\ removed\ from\ table\ D \tag{3.10}$$

$$\frac{< C, s > \Downarrow_{RIMP} < skip, s1 >}{< uncall\ N, s > \Downarrow_{RIMP} < skip, s1 >}(uncall)\ \ Where\ C = D(N)_2 \tag{3.11}$$

For the procedure definition (3.8), nothing seems to happen in theory within the program, with an empty top of the rule. That is because in practice, defining a function doesn't do anything, there are no steps of computation involved, so it is an axiom.

What happens in the background is very important though. We need a way to memorize the name of the procedure and associate it to a command. This is done at compile-time by using the table D, which takes a pair, or tuple, of commands as entries. Each entry, like a Map, is assigned to a string, which is in this case the name of the function, which acts as a key. When a function is defined in RIMP, using function name N and function body C, a new entry in the table D is added, which is, formally:

$$D + (N \mapsto (C, rev(C))) \tag{3.12}$$

So associated with the string N, there will be a pair, which contains the command C as the first part, and the reverse of that command C as the second part.

Why the reverse is added will be more clear later on.

The *undef* (3.10) semantics is also an axiom, so it doesn't change anything within the program. What happens externally is that it just reverts the table D to the previous state by deletion. This will be explained in more detail in the inversion function extension.

As for the call semantics (3.9), what happens is that the command C associated with the name N is evaluated. So it is obtained from the table D, which will obviously contain an entry for N, as a function definition must have happened before the function call. This, like for most programming languages, is assumed, as it is checked before runtime. Without a previous function definition, the function call will cause an error.
To obtain the command C from the table D, it is sufficient to search D for key N, and then just obtain the first, or leftmost, part of the tuple.
More formally:

$$D(N)_1 \tag{3.13}$$

Where the small 1 represents the first part of the pair.

The rule at 3.11 to un-call procedures is necessary for the inversion function to work, and it will be explained there in more detail.
As can be seen though, it is identical to rule 3.9, with the only difference that instead of obtaining the first member of the pair inside the map, it obtains the second member, the *rev(C)*.

3.3 Updating the augmentation and inversion functions

After allowing RIMP to support these two new features, the hard task is to make sure they are reversible. To do this the augmentation and inversion functions *rev* must be extended with entries for for-loops and procedure calls, as the latter will need to have a way to transform both into their reverse program.

3.3.1 Adding cases for for-loops

For-loops can be reversed in a very similar way to while loops, as they have a very close internal structure. We know the for-loop, except for the initial assignment which only happens once, uses a while loop the whole way through. On a similar note the while loop, to be reversible, needs to have a counter that is initialized once and then increased at the end of every iteration, which can be easily done in the for-loop with a similar modification.

Augmenting the for-loop

Following a close procedure to the while-loop previously explained (as well as in [2]), we need to update the augmentation function so that it can translate the for-loop as it is parsed, to a version with a counter. We can use the same protected variable name "counter" for for-loops as well as while-loops. So, taking a generic for-loop like 3.2, it can be parsed as the sequence:

$$counter := 0;\ for\ C1\ cond\ E\ step\ (C2;\ counter :=!counter + 1)\ do\ C3 \tag{3.14}$$

What happens is simply that the counter initializing statement is added before the for-loop happens. Immediately after that, the initial assignment of the for-loop is run. While the counter incrementing statement is added ad the end of the tail command C2, knowing that section is always run after each loop body. The way for-loops are built actually makes this modification arguably more elegant than it is in the while-loops.

Writing the inverse of the for-loop

Now that we've established how the for-loop is modified in the back-end, the output of the function *rev* can be written, counting as input our generic for-loop from 3.2:

$$rev(for\ C1\ cond\ E\ step\ C2\ do\ C3)\ =\ for\ skip\ cond\ !counter_i > 0\ step\ rev(C3)\ do\ rev(C2)\ ;rev(C1)\ if counter_i \notin E \tag{3.15}$$

As can be seen above, in the reversed for-loop, the initial command C1 is just skip. This is because it acts as a first operation which is not repeated more than once, so it would need to reverse a "one-time" end operation, which does not exist in for-loops, or at least in the way they were designed here. The loop condition, like for the reversed while-loops, is simply to check that the iteration counter is above 0.

The loop body and tail command are swapped, as the reverse of C3 becomes the tail command while the reverse of C2 becomes the loop body. This is because, the way the semantics were written, the tail command is run just after the loop body, so the reverse of the program must have each reversed body run *after* the reversed tail command. Swapping the two was by far the easiest way to do this.

As the final step, of course the initial command needs to be reversed as well, and that is simply done by adding the reverse of C1 outside of the for-loop, to be run as soon as the loop is finished.

Obviously, like for the while-loop, this only works if "counter" is not used in the loop condition.

Below is the opposite of all this, so what happens when we have the counter as the loop condition, so basically what the output of the *rev* function is when the input is the for-loop reverse we described above:

$$rev(for\ skip\ cond\ !counter_i > 0\ step\ C1\ do\ C2\ ;C3) \ = \ for\ rev(C3)\ cond\ E\ step\ rev(C2)\ do\ rev(C1) \ \ if\ T(i) = E \tag{3.16}$$

Exactly like for the while-loops, a lookup table T (which can be shared between for and while loops) is necessary to store the condition of each for-loop i, so it is possible to retrieve it later.

3.3.2 Adding cases for procedure calls

Reversing procedure calls is a much more complex problem, especially when we factor in recursion. Because when we want to reverse and "un-call" a function it is not a simple task to know how many times that function has been recursively called. To attempt to tackle this problem, some inspiration was drawn from [4], though the process that Hoey, Ulidowski and Yuen use is very different as they deal with program concurrency, while it is implied for RIMP that programs do not run in parallel.

Writing the inverse of the def command

As explained above, the *def* command is reversed to a command that will just remove the entry from the table D, to make sure that reversibility is maintained even outside of our program state, "under the hood". The command that does this is *undef* N, with the semantics given previously in 3.10. Even if there is no need to keep the command C present in *def* as it is not relevant on deletion, it is crucial to keep it in the *undef* command to maintain reversibility in the other direction as well, as without knowing C, it would be impossible to go from *undef* to *def*. The entry in the inversion function, therefore, is simply:

$$rev(def\ N\ as\ C) \qquad = \qquad undef\ N\ as\ C \tag{3.17}$$

and in the other direction:

$$rev(undef\ N\ as\ C) \qquad = \qquad def\ N\ as\ C \tag{3.18}$$

Writing the inverse of the call command

In a similar way to *def*, the way the semantics was written for the *call* command (3.9) makes it quite basic to write the entry of the inversion function:

$$rev(call\ N) \qquad = \qquad uncall\ N \tag{3.19}$$

Which is the same in the other direction:

$$rev(uncall\ N) \qquad = \qquad call\ N \tag{3.20}$$

To understand this better it is useful to look at the semantics of call(3.9) and uncall(3.11). If we consider the pair of commands C and rev(C) inside the map D, we know that call N, in practice, is a substitute for writing C.
So to obtain the result of *call N* we just obtain the first member of the pair associated with N. So the reverse of that, which is *uncall N* will need to obtain the reverse of C, which is the second member of the pair. So the only difference between *call* and *uncall* in their semantics is what member of the pair in table D they access. Call accesses $D(N)_1$, the leftmost member, while uncall accesses $D(N)_2$, the rightmost member.

Program Examples

To give a better idea of the syntax of RIMP and the new features, here are a few examples of common programs, written in RIMP:

- Fibonacci sequence of 10 [3](also one of the test cases later): f1 := 0; f2 := 1; f3 := 0; for i := 3 cond !i < 11 step i := !i + 1 do (f3 := !f1 + !f2; f1 := !f2; f2 := !f3)
- Factorial of 7, defined and then called: def fact as (f := 1; n := 8; for i := 1 cond !i < !n step i := !i + 1 do f := !f * !i); call fact
- Power of 15, defined and then called: n := 15; def pow as n := !n * !n; call pow

Chapter 4

Software Specification and Design

4.1 Specification

As stated previously, the aim of this project is to extend the semantics of RIMP to make it support for-loops and procedure calls.
To show how it was done, an interpreter for the RIMP language was written, which is able to evaluate for-loops and procedure calls as well as every previous RIMP feature.
This interpreter takes an abstract syntax tree as input, and returns the values in the store after its execution. After that, it reverses the program completely and outputs the reversed tree. The final outputs are the values in the store after the execution of the input tree, the printed reversed tree, and the values in the store after the execution of the reverse tree, which should be the same as the initial values. As it is a small interpreter, it has a short specification, which I will list below in the form of User Stories.

4.1.1 User Stories

- As a user, I want to be able to create an abstract syntax tree so it can be evaluated and reversed.
- As a user, I would like to see the contents of the store after evaluating my tree, so I can see the effect of the input program.
- As a user and a developer, I would like to see the reversed tree of my input tree, so I can

understand some of the computation.

- As a user, I want to see the contents of the store after evaluating the reverse of my tree, so I can see if the store has returned to its initial state.

4.2 Design

4.2.1 Command evaluation

The way the interpreter was designed is on the basis of the big-step semantics (from 2.1 to 2.12).

Big-step semantics, differently from abstract machines or small-step semantics, describe the meaning of a language with no middle steps. Meaning, from an operation, the big-step semantics tells us what the state before execution of that operation looks like, and what it looks like after. Everything that happens in-between is not relevant to the description of the operation. These rules allow very easy translation into code, even better if it is functional code with pattern matching, which is what will be used here. So technically, the decision made was to implement an imperative programming language like RIMP using a functional programming language like Scala [7], with some instances of imperative computation here and there. More information about the differences between imperative and functional paradigms can be found in [1]

4.2.2 Reversibility

As mentioned previously, reversibility also has different meanings and alternative designs. It can be thought about in a deterministic way, like RIMP is, or in a non-deterministic way, which opens up to very different interpretations.

The design of RIMP also gives us the freedom to choose if reversibility should be intended as a step-by-step concept, or as an inversion of final states. In other words, reversibility can be implemented with an abstract machine, which allows inversion of small program steps but is more low level, or with big-step semantics, as will be done here, which will reverse the entire program at once.

Chapter 5

Implementation

This section will explain in detail how RIMP and the two added features were implemented in code. An interpreter that can read a parse tree and output the reversed tree, as well as evaluate the results of both trees, was created, using the language Scala [7]. Scala was chosen for its powerful functional capabilities and immutability, that make it a very elegant language to read and code in.

5.1 Writing the abstract syntax tree

To write down the parse trees of a RIMP program, the first step is to define what the sections of the grammar of RIMP are, through the abstract syntax trees.

This way we are simply defining in code what the nodes of our parse trees can be: so for example an if-statement or an assignment. And of course, for-loops, procedure calls and procedure definitions as well.

For clarity purposes, expressions E were divided into two classes: arithmetic expressions called AExp and boolean expressions called BExp. This is because, in the semantics of RIMP, nearly all rules that have an E are specific to either booleans or arithmetic operators. For example, in if statements (2.9), the E that is used as a condition must of course be a boolean expression, because an arithmetic expression would have a numeric result, which couldn't be used as a condition. In the same way, when it comes to assignments (2.7), we know from the rules of RIMP, and SIMP before it [2], that variables can only be assigned to integers, so having a variable l assigned to a boolean wouldn't work.

The only rule that works the same for arithmetic expressions and boolean expressions is the

op rule (2.4), but it is assumed that the right operations are used for the right expressions. For example, the rule is not applicable if we use the + operation for two booleans. So in this case, for the implementation, the *op* rule was also separated into two rules, AOp and BOp, for arithmetic expressions and boolean expressions respectively. While AOp only accepts arithmetic expressions as member, with an arithmetic operator in the middle (+, -, *, /). BOp on the other hand, accepts only arithmetic expressions as member as well, but there needs to be a boolean operator in the middle except for *not* and *and* (less than, greater than, or =), as they will be considered separate classes extending BExp, together with True and False.

So in this implementation, there are three abstract classes that represent everything in RIMP instead of the original 2: Commands *Cmd*, Arithmetic expressions *AExp* and Boolean expressions *BExp*. Every node in the abstract syntax tree has to belong to one of these types.

5.2 Writing the evaluator

After having established what the abstract syntax trees of RIMP are, there needs to be a function that is able to take the parse tree of a RIMP program and obtain a result. So in practice, an interpreter, or evaluator, needs to be built.

The way the interpreter works is by having an environment, or store s, as it is called in the RIMP background above, that contains the runtime values of all the variables used in the program, next to the recursive value v that is basically a history of changes. Thanks to this tuple, reversibility is achieved.

The initial state will usually be with no variables present in the environment. As variable assignments happen, the state of the environment changes, until the final state, at the end of the program. The final state of the store is the output of the evaluator given as input a parse tree P. This interpreter will show how the store has changed as a consequence of running a specific RIMP program. To prove reversibility, it will then evaluate the reverse of parse tree P (which is also shown as output to the user), and if the resulting state is the same as the initial state, then the program was evaluated in a reversible way.

The way the environment was implemented is as a simple Map in Scala, from Strings to a pair of type (Integer, V). The first value, the integer, is k, the current runtime value of the variable. While V is the recursive class mentioned previously, which can either be the base case [0] or it can be +(n, v), where n is an integer.

Each variable is assigned a name in RIMP, and that name, or that string specifically, is the key to the Map that contains the variable values. For each variable name, the map will store its

corresponding (k, v) pair.

Because there are three types of nodes in our abstract syntax trees, the evaluator is also separated into three functions: one to evaluate arithmetic expressions, one to evaluate boolean expressions, and one to evaluate commands.

The functions to evaluate arithmetic expressions and boolean expressions work exactly how they would work in an irreversible programming language like SIMP [2], nothing changes when it comes to reversibility. The way they are implemented in Scala is through pattern matching, or in other words through a partial function. For example, if the input is a *!l* (2.2), called Var(l) in the code, the result will be s1(l), where s is the map storing the variable values, and l, the variable name, is the key. For s1, we mean we obtain the first element of the pair, because that is the runtime value of the variable.

This is slightly different for the function to evaluate commands though, as some commands, assign(2.7) and unassign (2.8) in particular, will need additional use of the rightmost member of the pair, v, to maintain reversibility. Like all other commands, this is done by strictly following the evaluation rules described in the semantics for those two operations.

5.3 Writing the inversion function

To evaluate the function definition command (and as we will see later on, to reverse the program) the evaluator needs the inversion function.

The way this is implemented is through a partial function that follows the definitions given in the background from 2.13 to 2.20, as well as the entries that were added in the Technical Contributions section (3.17 and 3.19).

In the RIMP implementation, expressions and commands are extensions of a class Prog, as programs can be either a command or an expression. So the function *rev* takes as input a type Prog, which just outputs the same program p whenever that is an arithmetic expression AExp or a boolean BExp (as per the rule at 2.13), while in case it's a command *Cmd*, it follows the inversion rules mentioned previously, acting on a case by case basis.

As mentioned in the entry for while loops (2.19) and for loops (3.15), the *rev* function will also have an additional argument T, which is the table which stores the conditions E for each loop, so they don't get "lost". The conditions are added to this table inside the augmentation function, when while loops and for loops are augmented.

5.4 Writing the augmentation function

The augmentation function is necessary for the reversibility of while loops, for-loops and if-statements. The way it works is exactly as described in the background:
For the loops, it modifies the structure of the two constructs with a counter which will memorize the loop iterations. To do this in the most convenient way, two additional classes were created: *AugWhile* and *AugFor*, that are the augmented versions of while and for. The reason why a different class was created is to increase the number of arguments by one, to include a variable i which is different for each loop. The i is what is assigned to the counter of each loop, as well as the key of the entry that is added to the table T. The variable i is crucial for the inversion function to recognize the loop it is reading.
Of course, another thing that needs to happen is the update of the table T. As a loop is augmented, the condition of that loop is added to T (with key i) so that after inversion the condition E is easily recoverable from T.

For the if-statement the implementation was very different, as two new functions needed to be created: one called *createAssignments*, which takes the boolean condition E of the if-statement, and returns the sequence of assignments l' := l, where l is a variable used in E, and does it for all variables, creating the duplicates. The second function, called *replaceExp*, taking as an input the condition E, generates the new condition E', which contains all the duplicate variables instead of the original ones. The sequence of the created assignments followed by the new if-statement with condition E' is returned by the *aug* function.

5.5 Writing the main function

The last function to write is the one that puts all the other functions together. It takes as input an abstract syntax tree of type *Prog*, and then creates an empty store, the definitions table D and the conditions table T.
Because our tree P can be either a command C or an expression E, it checks the instance of the tree and if it is a command, it augments the tree P and feeds it to the evaluation function, outputting the result. After that, it inverts P using the inversion function, outputs the new tree, and then evaluates the reverse of P, printing the result again. These three outputs are the result of our full interpreter.
In case it is an expression (for example 3+5) it is much simpler. It prints out three results: the

unchanged store (as an expression doesn't change anything within the program), the tree P (as the reverse of an expression E is just E), so the reversed tree is the same as the original, and the store, unchanged again.

Chapter 6

Results/Evaluation

This section will explain how the written software was tested, and the results of that testing, with an analysis of what it can do and what its limitations are.

6.1 Software Testing

The software was tested thoroughly in a manual way, by writing several methods, each with a different abstract syntax tree.

All the different operations (assignment, if-statements, for-loops etc) were individually tested with methods running very basic trees, and subsequent methods were written to run trees that combine the operations using sequences. A table containing some of the tested trees and the correctness of their results can be found below. I decided to include in the table the basic cases for all different operations, and then a final complex case combining many commands, a program computing the Fibonacci sequence [3] of length 10. The left column will just specify which tree was used, written in the form of a RIMP program. As the evaluator has three outputs, the right column in the table has values from 0 to 3, which corrispond to the number of outputs that are correct, or it will state if it crashes. So if the program behaves the way it should for abstract syntax tree a, it should have a *correct results* score of 3.

Program	Correct results
5	3
!l	Correct error
3 + 5	3
true	3
7 < 9	3
true and false	3
not(false)	3
skip	3
l := 15	3
l := 5; if !l > 5 then z := 2 else z := 4	3
i := 0; j := 1; while !i < 5 do (j := !j * 2; i := !i + 1)	3
j := 1; for i := 0 cond !i < 5 step i := !i + 1 do j := !j * 2	3
def i-reset as i := 0	3
call i-reset	Correct error
def i-reset as i := 0; call i-reset	3
i := 5; def i-decr as (if !i = 0 then skip else (i := !i - 1; call i-decr)); call i-decr	2
f1 := 0; f2 := 1; f3 := 0; for i := 3 cond !i < 11 step i := !i + 1 do (f3 := !f1 + !f2; f1 := !f2; f2 := !f3)	3

6.2 Results and analysis

The result of this project is an interpreter which can evaluate and give results of RIMP trees, including especially for-loops and procedure calls, as well as prove the reversibility of the calculations by going back to the initial state.

One of the first things to notice when analyzing the program is the way the store works. All possible variables in the store are automatically initialized to 0. They are technically not visible until they are used, but they exist as "initial variables" from the start. So for example, a program like the one in the second line of the table (*!l*) will crash because it is asking for the value of a variable l that has never been declared. In this implementation of RIMP, assignment works as variable declaration. So when a tree is run, if any assignments are present, the initial state of the store will contain all the variables used in the program, assigned to 0.

This is crucial for the way programs are reversed. If we take an assignment, like in line 9(l := 15) of the table, the initial store will contain a variable l with the value of 0. After evaluation

happens, the store will change the runtime value of l to 15. When inversion happens, the variable l will go back to its value at the initial state, which is 0. Therefore the third result, the one after inversion, will be a store that contains variable l with runtime value 0, despite l not being present in the store before the evaluation of the tree. As previously said, this is because it is assumed that every variable is initialized to 0 in the initial state, so inverting variable l back to value 0 corresponds to the same state as having a store with no variable l present, therefore reversibility is maintained. 0 can be considered a default.

Another important things to say is that the evaluator assumes well written trees. This means that it will throw a custom error before evaluation if, for example, the simbol "@" is used as an operand for an arithmetic expression, or similar mistakes. As it is a simple evaluator and does not implement a parser or lexer, any tree is fed directly into the interpreter and there is no complex syntax check before it happens, so it is important to ensure there are no typing mistakes in the input trees. The interpreter will give out personalized error messages that describe as accurately as possible what caused the exception, but as it is not a compiler, these will of course happen mid-computation. Another thing to be careful of is the use of protected words. As explained multiple times in this report, variable names like *counter* or *step* are protected names, and the user is not allowed to use them. These kind of checks would normally occur at lexing level, but because this program takes as input the parse trees already, it will assume the user will be following these rules, or unexpected behavior could happen.

Additionally, being interpreted directly instead of translated into machine code, the software can take exponentially long for very long sequences, and is generally a little slower than a compiler would be, as that would not combine compilation time and runtime, so it could optimize the code a lot at compile time. Regardless, the interpreter performs quite quickly across all commands.

As for the two main additions in this paper, for-loops and procedure calls, the results are satisfying, as reversibility was maintained in all the test cases. When it comes to for-loops it was expected, as they are simply evaluated as if they were while loops, which were proven to be reversible in [2]. There will be an error if the command written as the initial assignment isn't an assignment, as explained previously in the *for-loops notation* section.

Procedure calls on the other hand work perfectly well linearly, but they don't work recursively for specifically designed statements, like the one in the last example in the table, because one of the three outputs is wrong. The incorrect output is the third one, the result of the store after running the reversed tree. So in practice, while they execute perfectly, they don't reverse correctly. Why this happens can easily be seen in the reversed tree: taking our example in the table, the problem is the inversion of the if-statement. After running the tree forwards, the variable i decreases from 5 to 0, and then leaves the recursion. But the condition *if i=0 then skip* remains the same even in the reversed tree, and that means than when computing back, because i is 0 from the forward computation, the recursion is exited immediately, instead of undoing all the previous steps. This became evident only thanks to the implementation, while it was hard to spot just from the theory.

After analyzing the issue a little more, I came to the conclusion that this would work with local variables, but it can't possibly work the way it was designed here, with global variables used to break out of the recursion. This is because local variables and arguments don't interfere with each other during different instances of a procedure call, while it's exactly what happens in our case. Every time the procedure is called again, the variable used should be unique, not the same one as before. So to correctly implement recursion, each instance of the same local variable would need to be stored separately (with a counter for example, which identifies which procedure instance it belongs to).

Given the time and resources, I chose to scrap recursion, and only allow iteration through while loops and for-loops, like a strict imperative language. So procedures act purely as synthetic pieces of code, which can be defined to avoid duplication. Of course, because there are no local variables in RIMP, global variables are still the ones accessed inside the procedures, so it is important to write code which does not mistakenly assume those variables will reset after each call.

Chapter 7

Legal, Social, Ethical and Professional Issues

The work that has been carried on this project strictly follows the Code of Conduct & Code of Good Practice issued by the British Computer Society. It is fully understood that not following these rules can even lead to prosecution, as they serve to protect intellectual property.

7.1 Licensing

To build the main program, the language Scala [7] was used, which is open source, while the freeware editor Visual Studio Code, published by Microsoft, was used for coding and also to write this main report, using LaTex typesetting.
Aside from this, when it comes to the code, no libraries were used. The interpreter was created from scratch using the knowledge obtained throughout my degree, in particular in the module about Compilers and Formal Languages studied this year, and of course with information found in [2], which is the main paper this report extends.
Whenever any content belonging to other sources was mentioned or used in the report, it was always appropriately referenced using the Harvard referencing style.

7.2 Ethical clearance and issues

As this project did not involve human participants or any access to human data, it was never given ethical clearance, therefore there are no issues in this sense to be concerned with. Every-

thing within the program is standalone, and doesn't require any information to run, aside from an abstract syntax tree as input, which the user has total freedom to create.

7.3 Accessibility options

This program was created to be used by anyone who sees fit, and anyone who would like to play around with a reversible program language to test its functionality and results. It does not discriminate in any way on the grounds of sex, sexual orientation, marital status, nationality, colour, race, ethnic origin, religion, age or disability.

Unfortunately though, it does not include any accessibility options for people with disabilities, so for anyone willing to use the program with seeing or typing disabilities, it is recommended to seek the assistance of a second party in order to help them through its use.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

The main thing that can be derived from this work is how a reversible programming language, and in this specific case, one containing for-loops and procedure calls, can be implemented by modifying the underlying semantics of how the language works, and without changing anything in the way a user would write a standard, irreversible language.

Using RIMP, we have seen that by adding just one new operation to the semantics (the reverse assignment) we are able to create a function that can reverse every single operation, including operations that are combined through the use of sequences. This is not done just by the *rev* function of course, as we have seen it is also helped by code augmentation and look-up tables. Nevertheless, the point stands that imperative languages like RIMP (for more on what an imperative language is you can check out [1]) base their entire computation on assignment statements. Each state transition is caused exclusively by a variable in the store changing its value, or in other words, by an assignment happening. So by using an operation like *unassign* (2.8) with a meaning that was designed (in [2]) to be the total opposite of *assign*, the entire reversible language can be built on the basis of these two operations driving computation in one direction or the other. For this reason, anything that proceeds linearly is easily implementable. For-loops, for example, are in practice just a differently formatted type of while-loop, and they could be simply be transformed into one.

Procedure calls are much tougher to implement though, because of what recursion entails. Recursion destroys linearity in assignment statements, and makes it incredibly difficult to go back in computation and find the same kind of linear structure that would be found in loops. It is

very hard to find out what values variables had in a specific point in time when the computation is not ”set in stone”, and this is why recursion was omitted for the time being.

8.2 Future work

So as said above, for anyone interested and willing to continue this work, recursion is where they should start. The idea I had was to create a separation between global variables and local variables first, and after this, necessarily, add arguments to the procedure definitions. This alone is not a simple task, even without considering recursion, as it would probably need another table, storing a label for each local variable in each different procedure definition. After doing that, the same way loops are augmented with counters, procedures could be as well, with a hidden global variable that will just count how many times a procedure has been run.
While some of these will make sense, some are very raw ideas that probably have many faults. The paper at [4] is a very good one to read if interested in the matter, as it implements recursion in a multi-threaded reversible programming language, differently from RIMP which of course has no parallelism.

Speaking of the implementation itself, the program is also quite low level, as user-friendliness was never a priority in the conception of the interpreter. The way this evaluator is designed to be used and played with by the user is by simply allowing them to acess the script source code, and create their own abstract syntax tree at the top, or just modify the test ones. While this works quite well, having the user access the source code is not always the best idea.
If someone were to improve on this, one option could be to allow the user to click a button to randomly generate a tree. This is actually a very complex task, as randomness is never that easy to execute, and the amount of permutations all the commands can form combined is probably very high, so it could also be quite slow.

An alternative is to allow the user to input the abstract syntax tree from the command line or text, and simply read the input and lex/parse it. Of course, by also giving the user instructions on how to correctly write the trees. While this seems simple, I personally think it just becomes another lexing and parsing problem, and the whole idea of starting directly from the abstract syntax tree was to avoid the slower lexing and parsing sections that come with compilers and interpreters. So at this point, instead of having the user write down the

trees, the most complete thing to do would be to let them write down the full program, and then just lex and parse the program to our abstract syntax trees, to make a complete interpreter.

One last thing that could be changed is not really an addition, but quite a big structural modification to to implementation of RIMP, and it's the choice of what formal language specification to implement. As previously explained, the choice for this project was to implement RIMP using big-step semantics, but one could also implement small-step semantics or the abstract machine (more on what these are can be found in [1]).
There isn't necessarily a winner on the best specification to implement, as each one has its advantages or disadvantages. In our case, with big-step semantics you don't need to worry about what the middle steps are, and reversibility matters only for initial and final state, which makes it more straightforward. Nonetheless, if someone wanted to extend this with a debugger, modifying it to implement abstract machines instead of big-step semantics would be the best idea, as abstract machines are much lower level, and work with step by step computation. The code would be quite similar, as explained in [2] (and briefly mentioned in the background) with the addition of two stacks, a front stack containing the tree commands, and an empty back stack. As each command happened, it would get removed from the front stack and its reverse would be added to the back stack. At any point, to reverse computation, the two stacks would just be swapped.

References

[1] Maribel Fernández. *Programming Languages and Operational Semantics - A Concise Overview.* Undergraduate Topics in Computer Science. Springer, 2014.

[2] Maribel Fernández and Ian Mackie. A reversible operational semantics for imperative programming languages. In Shang-Wei Lin, Zhe Hou, and Brendan P. Mahony, editors, *Formal Methods and Software Engineering - 22nd International Conference on Formal Engineering Methods, ICFEM 2020, Singapore, Singapore, March 1-3, 2021, Proceedings*, volume 12531 of *Lecture Notes in Computer Science*, pages 91–106. Springer, 2020.

[3] VK Gupta, Yashwant K Panwar, and Omprakash Sikhwal. Generalized fibonacci sequences. *Theoretical Mathematics and Applications*, 2(2):115–124, 2012.

[4] James Hoey, Irek Ulidowski, and Shoji Yuen. Reversing parallel programs with blocks and procedures. In Jorge A. Pérez and Simone Tini, editors, *Proceedings Combined 25th International Workshop on Expressiveness in Concurrency and 15th Workshop on Structural Operational Semantics, EXPRESS/SOS 2018, Beijing, China, September 3, 2018*, volume 276 of *EPTCS*, pages 69–86, 2018.

[5] Rolf Landauer. Irreversibility and heat generation in the computing process. *IBM J. Res. Dev.*, 5(3):183–191, 1961.

[6] Christopher Lutz and Howard Derby. Janus: a time-reversible language. *Letter to R. Landauer.*, 1986.

[7] Martin Odersky, Philippe Altherr, Vincent Cremet, Burak Emir, Stphane Micheloud, Nikolay Mihaylov, Michel Schinz, Erik Stenman, and Matthias Zenger. The scala language specification, 2004.

[8] Kalyan S. Perumalla. *Introduction to Reversible Computing.* Chapman & amp; Hall/CRC, 1st edition, 2013.

[9] Eli Rose. Arrow: A modern reversible programming language. volume 270 of *Honors Papers*, 2015.