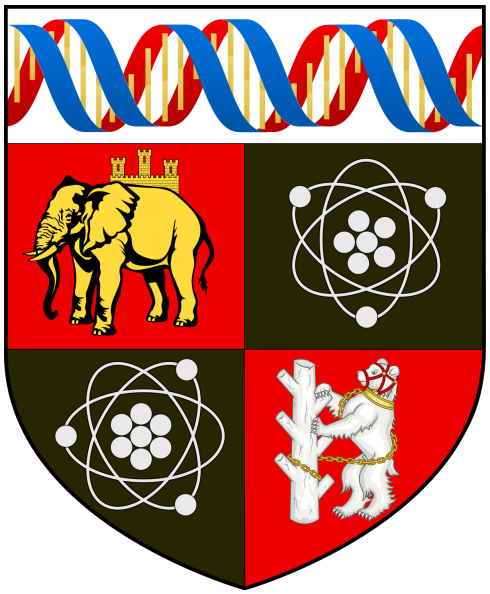
A language and proof-checker for two-way one-counter automata

Thomas Divers
Student no.: 2002735
Supervisor: Dmitry Chistikov

2 May 2023



Keywords
automata, formal languages, counter machines, programming languages, Rust, Hoare logic

Abstract

Two-way one-counter automata are a surprisingly powerful computational model, yet there does not exist a specific programming language capable of representing precisely these automata. This project aims to develop a programming language capable of representing two-way pushdown automata, particularly two-way one-counter automata, as well as a proof checker that recognises correct proofs for theorems about these automata.

Acknowledgements

Uncountably many thanks to Dmitry for his time, knowledge, help, guidance, and ability to answer even my stupidest questions eloquently enough to make me feel clever for asking them.

Also thanks to Stem City and associates for listening to me rant about automata theory for the better part of the last year.

All project code can be found at `https://github.com/tomdaboom/twoc`.

Note that some sections of this report have been adapted from work already submitted as part of the project specification, progress report and presentation.

Contents

Chapter 1

Introduction

1.1 Aims

This project's aims are as follows (in order of priority):

1. To design and implement a **programming language** (`twoc`) to represent and run two-way one-counter automata.
2. To design (and implement?) a **proof checker** (`twop`) to encode and prove theorems about `twoc` programs.

1.2 Problem Statement

We first begin by defining the class of automaton this project concerns.

A two-way one-counter automaton (fig. 1.1), for some finite alphabet Σ, receives some string $w \in \Sigma^*$ of size n stored on an input tape (which it reads one character at a time); executes some computation according to a finite state control; and decides whether or not to accept or reject w.

In order to decide whether or not to accept the input word, the automaton is allowed to move back and forth across its input string (thus making it a *two-way* machine). The automaton is also allowed access to as much finite-state memory as is needed (this can be embedded as part of its finite-state program). However, the only *infinite*-state memory it has access to is a single nonnegative counter value. The automaton is allowed to increment or decrement this counter at will, and is also allowed to check whether or not the counter contains 0 at any point during the computation. Note that the automaton is not allowed to write any values to the input tape during computation.

These automata can either be *deterministic* or *nondeterministic*. Throughout this report, the deterministic automata will be referred to as **2dc**, and their nondeterministic counterparts will be referred to as **2nc** (note that 2nc are more powerful than 2dc [Chr85]). The classes of languages that can be decided by a 2dc/2nc are called **2DC** and **2NC** respectively.

Now that we have defined the class of automaton that this project concerns, we reconsider the project's objectives.

The primary aim of this project is to design a *programming language* (`twoc`, standing for **t**wo-**w**ay **o**ne-**c**ounter) that can represent precisely this class of automata. We also provide an implementation of `twoc`, thus providing users with an environment to develop and test these automata.

A secondary objective is to design and implement a *proof checker* (`twop`, which doesn't stand for anything in particular) that allows users to formally prove theorems concerning these automata.

1.3 Motivation

The main motivation for this project is the surprising power of 2dc/2nc and the relatively sparse research into their computational power (much of the research into this kind of counter machine focuses on more powerful variants; e.g. machines with more than one counter, probabilistic and quantum variants, etc.).

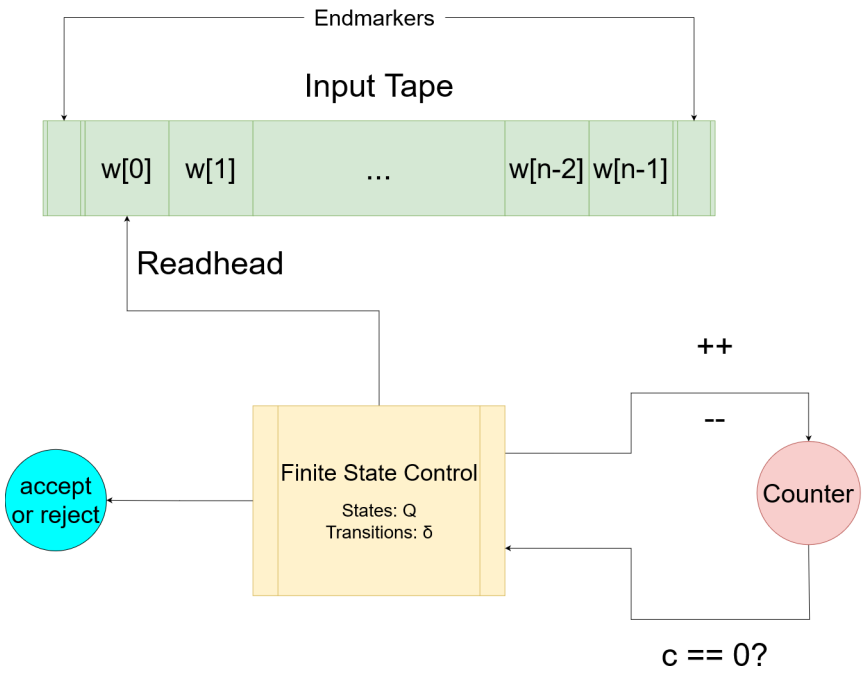


Figure 1.1: A two-way one-counter automaton

Two impressive results for this class can be found in [Pet94], which shows that 2dc can recognise the languages $\{0^n 1^{n^2} : n \geq 1\}$ and $\{0^n 1^{2^n} : n \geq 1\}$, proving that, for unary-encoded integers, 2dc/2nc are capable of multiplication and exponentiation. These results clearly show the impressive power of these automata, given how restrictive the class is.

The way in which Petersen proved these results was a key inspiration for this project. In order to specify the automata that recognise these languages, he defines these automata using Pascal-like pseudocode (fig. 1.2), as opposed to simply using natural language to describe their execution (as in [BY14], [Chr85], [Gal76], etc.).

Although the results presented in [Pet94] are undeniably correct, at first glance, they appear to be lacking in mathematical rigour. Many of the operations he uses (e.g. dividing the counter value by 2, testing whether or not the counter is odd, etc.) are not explicitly explained, and Petersen does not formally prove that the automata he describes decide precisely the languages claimed.

Another issue with Petersen's argument is the unintuitive nature of the programs he presents. Although he does describe their execution in natural language, the programs themselves are not easily understood. It is believed that these programs would be easier to explain to others if they were written to a formal, implementable standard.

This project aims to promote, augment and reinforce Petersen's style of argument concerning 2dc/2nc in two ways:

1. By formally defining a programming language that provably decides precisely the languages in 2DC/2NC.
2. By providing an efficient implementation of the programming language to ease the process of writing and understanding these programs.

The proof system is included to help provide more mathematical rigour to these constructive arguments, the idea being that others can prove that some language L is in 2DC/2NC by

1. defining a deterministic/nondeterministic automaton in the `twoc` language
2. formally proving that it decides L using the `twop` proof system

Hopefully, `twoc`/`twop` will encourage the discovery of more results pertaining to this class of automata,

```
function power(n:int, m:int):boolean;
begin
    if ((n=0)and(m=1)) or ((n=1)and(m=2)) or ((n=2)and(m=4)) or ((n=3)and(m=8))
    then accept;

    if (n<4) or (m<16) then reject;

    c := m;
    while not odd(c) do c := c div 2;
    if c > 1 then reject;

    c := m;
    c := c div 8;
    while not odd(c) do
    begin
        c := c+m div 2;
        while not odd(c) do c := c div 2;
        c := c + 2;
        while 2*c < m do c := c*2;
        c := c-m div 2;
        c := c div 2;
    end
    c := c div 2;
    if c+3=n then accept else reject;
end;
```

Figure 1.2: A program describing a 2dc that decides the language $L = \{0^n 1^{2^n} : n \geq 1\}$ *[Pet94]*

which, given its impressive and surprising power, could lead to interesting results in other areas of computer science.

The canonical example of research in automata theory being used to achieve surprising results in other areas is the Knuth-Morris-Pratt algorithm, which finds all the occurrences of a word $x \in \Sigma^*$ in another word $y \in \Sigma^*$ with $|x| \leq |y|$ in time $O(|x|+|y|)$ [KMP77]. This algorithm depends on an earlier algorithm designed by Stephen Cook, which decides if a two-way pushdown automata (see def. 1.5) accepts some arbitrary word w in time $O(|w|)$ [Coo71]. The potential for results like this could exist within 2dc/2nc; hopefully, such results could be found and formally verified using the systems described here, or other systems inspired by `twoc/twop`.

1.4 Formal definitions

Definition 1.1. *A* ***2nc*** *is a 5-tuple* $M = (Q, \Sigma, \delta, q_0, F)$ *where*

- Q *is the set of* ***states****.*
- Σ *is the* ***alphabet*** *of the input string (this implicitly includes two tape* ***endmarkers****, used to demarcate where the ends of input word are are; the left and right endmarkers are denoted* $\dashv$ *and* $\vdash$ *respectively).*
- $\delta : Q \times \Sigma \times \{0, 1\} \to \mathcal{P}(Q \times \mathbb{Z} \times \mathbb{Z})$ *is the* ***transition function*** *($\mathcal{P}(\cdot)$ is used to denote the powerset).*
- $q_0 \in Q$ *is the* ***start state****.*
- $F \subseteq Q$ *is the set of* ***accepting states****.*

The transition function δ describes what computation steps the automaton is allowed to take from some configuration.

More formally, $(q, \Delta i, \Delta c) \in \delta(p, x, 0)$ if the automaton can go from state p to state q while reading x from the tape and 0 from the counter (if δ's third argument is set to 1, the automaton instead reads any value > 0 from the counter). During this transition, the automaton changes the position of its readhead by Δi and changes the value of the counter by Δc.

Note that we assume, without loss of generality, that $\Delta i \in [-i, n - i]$ and $\Delta c \geq -c$, where $n = |w|$, i is the current index of the readhead on the tape, and c is the current value of the counter. This ensures that

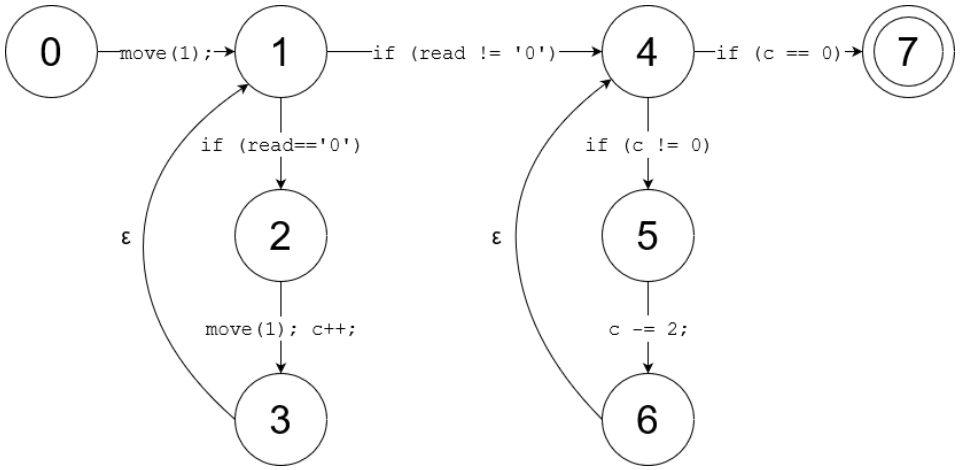


Figure 1.3: The state transition diagram of a 2dc that decides the language $L = \{0^{2n} : n \geq 0\}$

the counter value remains nonnegative and that the readhead does not move off of the tape at any point (these assumptions will lead to a minor consideration in sec. 3.3).

A **configuration** of our automaton is a 3-tuple containing precisely enough information for the finite state control to decide which transition(s) the automaton is allowed to take. A **snapshot** is a 3-tuple containing precisely enough information to completely describe the current state of the automaton's execution. More formally,

Definition 1.2. *A **(surface) configuration** of* M *on some input word* $w \in \Sigma^*, |w| = n$ *is some triple* $(q, i, c) \in Q \times [0, n-1] \times \{0, 1\}$*. We also define a **snapshot** of* M *to be some triple* $(q, i, c) \in Q \times [0, n-1] \times (\mathbb{N} \cup \{0\})$*.*

If, for a given configuration/snapshot, $q \in F$*, and the automaton halts immediately after reaching it, we say the configuration/snapshot is **accepting**. If* $q \notin F$ *and the automaton halts, we say that it is **rejecting**.*

Definition 1.3. *A given 2nc is also a **2dc** if, for all configurations* $(q, i, c) \in Q \times [0, n-1] \times \{0, 1\}$ *and inputs* $w \in \Sigma^*$*, the size of* $\delta(q, w_i, c)$ *is at most 1.*

Definition 1.4. *A **computation path** is a (potentially infinite) sequence of snapshots* $\gamma_0, \gamma_1, \ldots, \gamma_l$ *such that, for all* $k \in [0, l-1]$ *with* $\gamma_k = (p, i, c)$ *and* $\gamma_{k+1} = (q, i', c')$*, there exists some* $(q, \Delta i, \Delta c) \in \delta(p, w_i, \max\{c, 1\})$ *such that* $i' = i + \Delta i$ *and* $c' = c + \Delta c$*.*

Informally, a computation path completely describes the execution of a 2dc/2nc on some fixed input word w.

At the start of a computation, we assume the automaton to be in state q_0, with the readhead on $\dashv$, and with the counter set to 0 (i.e. the automaton begins all computations in the snapshot $\gamma_0 = (q_0, 0, 0)$).

Note that, for a fixed input, a 2nc may be able to choose between several different computation paths. In this case, we assume that the automaton always picks a computation path that leads it to an accepting snapshot, if one is reachable from γ_0. In this sense, we assume that the automaton's nondeterminism is *angelic* [Mam17].

Note that, throughout this report, we will often think of these automata as graphs (fig. 1.3), as opposed to the 5-tuples defined above. In these graphs, each vertex represents a state (accepting states are marked with two lines), and the edges encode the transition function (transitions labelled ϵ can be taken unconditionally and do not affect the readhead or the counter).

Also note that we will normally assume $Q \subseteq \mathbb{N} \cup \{0\}$ and $q_0 = 0$ (this convention simplifies some aspects of implementing these automata).

An important corollary of the fact that these automata can move both ways across their input is that these automata may infinitely loop (note that whether or not an automata does this on a given input is decidable [AHU68; Glü13]). If this occurs, the automaton is assumed to have rejected its input.

Another important corollary of these definitions is that any 2dc/2nc can be simulated by a **two-way pushdown automata** (a two-way automaton equipped with a *stack* as opposed to a counter [GHI67]; abbreviated as 2dpda/2npda) with a stack alphabet of size 1 [DG82] (this becomes relevant in sec. 5.4). The intuition for this is that the pushdown automata can use its stack to simulate a nonnegative counter.

We formally define these automata below (as in [AHU68]):

Definition 1.5. *A **2npda** is a 7-tuple $(Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ where*

- *Q is the set of states*
- *Σ is the tape alphabet*
- *Γ is the stack alphabet*
- *$\delta : Q \times \Sigma \times \Gamma \to \mathcal{P}(Q \times \Gamma^* \times \{-1, 0, 1\})$ is the transition function; if $(q, Z, d) \in \delta(p, x, Y)$ then the automaton can transition from state p to state q, push Z to the stack, and move the readhead d cells to the right upon reading x from the tape and popping Y from the stack.*
- *$q_0 \in Q$ is the starting state*
- *$Z_0 \in \Gamma$ is the symbol initially at the top of the stack*
- *$F \subseteq Q$ is the set of final states*

***Snapshots** are elements of $Q \times [0, n-1] \times \Gamma^*$, and **configurations** are elements of $Q \times [0, n-1] \times \Gamma$.*

*A 2npda is also a **2dpda** if $\delta(q, x, Z)$ has size at most 1 for all configurations $(q, i, Z) \in Q \times [0, n-1] \times \Gamma$ and inputs $w \in \Sigma^*$.*

Definition 1.4 for computation paths also applies to 2dpda/2npda.

1.5 Prior work

1.5.1 Programming languages for automata

Aside from the Pascal programs shown in [Pet94], there is very little literature concerning defining programming languages for specific classes of automata. In one such paper, Knuth and Bigelow specify a programming language capable of representing 2dpda/2npda, for the purposes of constructively proving inclusion properties for their corresponding classes (this is almost identical in purpose to what `twoc` aims to achieve) [KB67].

One key issue with [KB67] is its syntax, which seems to be inspired more by formal grammars than the languages that programmers typically write code in. This is likely a consequence of the age of this paper, which predates many modern programming languages (including C), and thus many syntactic and semantic conventions that contemporary programmers are familiar with.

Although the authors explain their language well, and avoid making confusing abstractions from the definition of the class of automata, programs in this language are still difficult to read at first glance. `twoc` aims to improve upon this by using more modern conventions, as well as providing some limited abstractions intended to ease the process of defining more complex automata.

Another key weakness of this paper concerns the implementation of their language. Their implementation does not leverage any efficient algorithms designed for 2dpda/2npda, and instead simply compiles the language down to assembly code, essentially treating their language equivalently to a Turing-complete programming language. In theory, significant optimisations can be made by applying polynomial-time simulation algorithms (see sec. 5.7 and C.4). This issue may also be a symptom of [KB67]'s age, which also predates some of the papers describing these algorithms [AHU68; Coo71]. A more thorough literature review of these algorithms is conducted in sec. 5.4.

Other work similar to [KB67] is primarily pedagogical in nature; most of these systems are intended as educational tools for students studying automata theory and formal languages, and are not intended to be used as research tools [CSK11].

```
begin
    class ab = a,b;
    class AB = A,B;
    input alphabet is ab;
    stack alphabet is AB,C,Z;
    input delimiters are start, stop;
    stack bottom is Z;

    [start => r],
    [
        a => r 'A', repeat;
        b => r 'B', repeat;
        stop => l 'C'
    ],
    [
        ab => l L, [ab => l], repeat;
        start => r
    ],
    [
        a.A => r R, repeat;
        b.B => r R, repeat;
        c => accept
    ]
end.

```

Figure 1.4: A program describing a 2dpda that decides the language $L = \{ww : w \in \{a, b\}^*\}$ [KB67]

1.5.2 The proof system

A key reason for describing an automaton in terms of its execution (either with natural language or an imperative program) as opposed to simply using its formal definition is that reasoning about its behaviour on arbitrary inputs becomes far easier. Another benefit is that formal systems for reasoning about imperative programs already exist, and can thus be adapted to the given language.

The proof system we design will be heavily based off of Hoare logic. The axioms described by Tony Hoare in [Hoa69] are already sufficient to prove theorems about deterministic programs, and these will be leveraged by the `twop` proof system. However, Hoare did not initially describe axioms sufficient to reason about nondeterministic programs.

Significant work has been done to extend Hoare logic with axioms for nondeterministic control structures [AO19; Apt83]. [Mam17] describes several axioms for nondeterministic control structures, some of which will be adapted to the control structures present in the `twoc` language. A section of [AO19] also discusses some of these issues. These will be elaborated on further in sec. 6.2.

[Sit21] describes an Haskell implementation of a Hoare-logic system for an imperative programming language capable of variable assignments and `if` and `while` statements (which will be sufficiently powerful to represent deterministic `twoc` programs). Any direct implementation of a proof checker for `twop` will likely be very similar to the one presented here, although significant adaptations will have to be made to translate this code from a functional language to an imperative one.

[Pie+20], while explaining how the judgements of Hoare logic work, describes how these rules can be embedded in a formal proof assistant; any embedding of the axioms described in chapter 6 will be extremely similar to the implementation provided here (in fact, [Pie+20] is essentially one large Coq[1] proof script).

[1] `https://coq.inria.fr/`

Chapter 2

Methodology

In this chapter, we discuss the methodology by which the project was undertaken, which we split into the following components:

- How the project was researched
- The high-level technical decisions made concerning the project's implementation (a thorough discussion of `twoc`'s implementation can be found in chapter 5)
- How the project was managed throughout its execution

We also briefly discuss the principal ethical concerns of this project (sec. 2.4).

2.1 Research

In this section, we discuss the research methodologies used to execute the project.

Sources were initially found through the following methods:

- The guidance of the project supervisor: Dmitry Chistikov's knowledge of and experience in this field served as an extremely useful resource; his guidance was used throughout the execution of the project in order to direct the author towards useful sources.
- Google Scholar:[1] Initially, very broad search terms ('two-way one-counter machines', 'Hoare-logic nondeterminism', etc.) were used to find initial broad resources. The information in these sources, as well as their bibliographies, were then used to locate more specific sources.

All resources found were collated into a list kept by the author. This included URLs referencing the resources, their authors, and brief descriptions of their contents, allowing them to be located and referenced more quickly thereafter.

2.2 Technical decisions

In this section, we discuss the initial, high-level decisions made about which tools to use concerning the implementation of `twoc`.

2.2.1 Choice of programming language

The author decided to write the codebase almost entirely in the Rust programming language[2] for the following reasons:

- Rust is an exceptionally performant language, on par with C++ [BA22].

[1] `https://scholar.google.com/`
[2] `https://www.rust-lang.org/`

- Rust is a very mature language, with good libraries and high-quality build tools available.[3]
- Rust has access to several functional features that make the development of programming languages easier (algebraic data types, pattern matching, etc.).
- Rust's memory-safety features ensured that the author did not encounter issues due to memory errors (memory leaks, null-pointers, etc.).
- Rust has access to the LALRPOP parser generator,[4] which drastically simplified the process of implementing a lexer and parser for the language (see sec. 5.1).
- This project served as a learning experience for the author: he did not know the Rust language beforehand, but is now confident with many of its notoriously subtle features.

These advantages made Rust a natural choice for this project. Of the other languages considered, Haskell[5] was the language that was most likely to have been selected instead of Rust, due to its functional features and the author's prior experience with it. However, Rust's performance advantages, as well as the author's past issues with Haskell development environments, ultimately led him to choose Rust over Haskell.

2.2.2 Other tools used

- Git and GitHub[6] were used for version control and to store a cloud backup of the project codebase.
- The Visual Studio Code integrated development environment[7] was used to edit the codebase and interact with Git/GitHub. This was due to the author's prior experience with VSCode, as well as its high-quality support for the Rust language through the rust-analyzer extension.[8]
- The Overleaf online LaTex editor[9] was used to write, edit and backup the reports and presentation slides. This was due to the author's prior experience with Overleaf.

2.3 Project management

In this section, we discuss the philosophies, methodologies, strategies and tools used to manage this project.

Note that, due to the relatively limited scale of this project, several components of it being predominantly theoretical, and the very small size of the team that completed it,[10] many of these techniques were used informally. In some cases, existing methodologies have been retroactively applied to the execution of the project in order to justify and formalise the processes used.

2.3.1 Planning

Prior to execution, during the project's specification stage, an initial plan of the project's execution was produced. This plan consisted of the technical decisions discussed previously, as well as the timetable contained in fig. 2.1.

This timetable was designed to be relatively loose, scheduling tasks for weeks of term as opposed to specific dates. It also aimed to parallelise the execution of the project, typically interleaving the execution of similar kinds of tasks with each other (e.g. designing `twoc` for 2dc is partially interleaved with its designing it for 2nc). The aim of this was to provide the author with some degree flexibility in terms of which specific tasks he would work on at a given time.

The objectives, although written as processes, were designed to be deliverable-oriented where possible [Con17]. Most of the tasks in this timetable correspond to some specific output, whether that be a theoretical concept, an implementation feature or a piece of documentation. Although a specific work breakdown

[3] `https://doc.rust-lang.org/cargo/`
[4] `https://github.com/lalrpop/lalrpop`
[5] `https://www.haskell.org/`
[6] `https://git-scm.com/` and `https://github.com/` respectively
[7] `https://code.visualstudio.com/`
[8] `https://rust-analyzer.github.io/`
[9] `https://github.com/overleaf/overleaf`
[10] $|team| = |\{Tom\}| = 1$

Stage	Start	End
Project specification	***T1 W1***	***T1 W2***
Understand the algorithms for simulating 2dc/2nc [AHU68; Coo71; Glü13]; read the literature concerning 2dc/2nc and other related automata; research about representing nondeterminism in automata languages	T1 W1	T1 W3
Design the programming language to represent simplistic 2dc	T1 W2	T1 W3
Design the programming language to represent more complex 2dc	T1 W3	T1 W5
Design the programming language to represent 2nc	T1 W4	T1 W6
Prove that the programming language is correct constructively and define an algorithm that can be used for these constructions	T1 W4	T1 W6
Implement a lexer/parser for the language	T1 W5	T1 W6
Implement a simple step-by-step interpreter for the language	T1 W5	T1 W6
Implement the 2dc/2nc construction and the Aho-Hopcroft-Ullman [AHU68] and Glück [Glü13] procedures for programs written in the language	T1 W7	T1 W9
Progress report	***T1 W8***	***T1 W9***
Begin writing the first draft of the final report	T1 W10	T2 W3
Christmas break	*T1 W10*	*T2 W1*
Design the proof checker for 2dc	T2 W1	T2 W2
Extend the proof checker with nondeterminism	T2 W2	T2 W3
Implement the proof checker for 2dc	T2 W3	T2 W4
Implement the proof checker for 2nc	T2 W4	T2 W6
Finish the first draft of the final report	T2 W5	T2 W7
Rewrite the project report based on supervisor feedback	T2 W7	T2 W10
Prepare the presentation	T2 W7	T2 W8
Presentation	***T2 W8***	***T2 W10***
Easter break	*T2 W10*	*T3 W1*
Final report	***T2 W10***	***T3 W1***

Figure 2.1: The initial project timetable, as provided in the specification

structure was not produced as part of this, it can be seen how one could be produced given this timetable [Ins13a].

This timetable was used to guide the specific tasks executed by the author, and to give him a rough idea of when specific sub-deliverables should be completed. It was also used to identify whether or not the project was running on time or not.

Note that this timetable does not schedule any specific tasks for the Christmas and Easter breaks. The reasons for this are discussed in the next subsection.

Due to the theoretical nature of the project, it was decided not to execute several other project management processes typical of the planning stage (e.g. stakeholder analysis, critical path analysis, etc.)

2.3.2 Risks

Also as part of the planning stage, several negative risks were identified [Ins13b]:

1. Personal computer failure
2. Conflicting deadlines
3. Personal illness
4. Unanticipated delays

Strategies to mitigate the impacts of these risks were also specified:

- Ensuring all project code and documentation was frequently backed up (using GitHub and Overleaf respectively) minimised the amount of work he could risk losing if his personal laptop failed during development (thus mitigating risk 1).
- Loose timetabling minimised the probability that minor delays of a few days (such as those caused by minor personal illness) did not propagate into major delays (thus mitigating risks 2 and 3)

- Choosing not to allocate any tasks to be executed over the breaks ensured that more severe, unexpected delays did not propagate from one term to the next (thus mitigating 4).

Risks 2 and 3 resulted in minor delays throughout the project, which were mitigated accordingly. However, one instance of risk 2 in term 1 was severely underestimated,[11] and the impacts of this risk were not entirely mitigated by the strategies described above, leading to several activities scheduled for term 1 being executed in the Christmas break, as well as during term 2. This partially contributed to the implementation of the proof checker being removed from the scope of the project towards the end of term 2.

2.3.3 Execution

Although no specific project management philosophy was followed during execution, several principles of an agile scrum methodology were followed [Bec+01; Scr23]:

- Regular scrum meetings (in the form of weekly supervisor meetings) were used in order to analyse, focus, guide and plan the progression of the project.
- The week-based timetabling of the project and the weekly supervisor meetings naturally led to a system of week-long sprints during term time.
- The requirements analysis and timetabling of the project were both done very loosely and generally in order to provide high flexibility and to accommodate shifting requirements. In this case, requirements changes were the result of the author not initially having a complete knowledge of the subject matter and learning new information as the project progressed, as opposed to shifting stakeholder/client requirements.

These principles were necessary for the success of this project. This was due in part to the risks experienced (as discussed previously), but was also due to incredibly poor estimations for how long specific tasks would take. As execution progressed, the author discovered that virtually every estimation he made for the difficulty of specific tasks was wrong. This resulted in the proposed timetable (fig. 2.1) being used essentially as a checklist ordered roughly by dependency.[12]

For instance, an entire week was allocated to implement a lexer and parser for the language (as well as a step-by-step interpreter, which was later deemed unnecessary). However, this stage actually took less than 4 hours in total. Some other tasks (such as designing nondeterministic control structures, implementing the construction algorithm, and specifying the proof rules for deterministic programs) also took significantly less time than anticipated.

However, other tasks took significantly longer than expected. The most extreme example of this was the implementation of the simulation procedures. The author anticipated that, because this stage would solely consist of implementing algorithms that already existed, it would be relatively quick (taking less than two weeks of term 1). In practice, this took approximately a month in total, did not use the Aho-Hopcroft-Ullman algorithm at all [AHU68], and was executed non-consecutively, with some parts lasting into the Easter break. Other tasks (such as extending the proof system to nondeterministic programs) also took longer than expected.

Ultimately, the execution of the project is more accurately described by the table presented in fig. 2.2.

2.3.4 Evaluation

Overall, the management of the project was a success: almost all of its goals were completed successfully before its termination, and almost all of its negative risks were successfully identified and overcome. However, in hindsight, the project's execution suffered from three key weaknesses:

- The project's scope was too ambitious, resulting in some amounts of work and research (almost all concerning the proof system's implementation) being scrapped. If the scope had been sufficiently narrow at the start of execution, the time spent on these elements could have been spent elsewhere, resulting in higher-quality deliverables.

[11] Anyone familiar with popular 3rd year modules offered by the Department at the time of writing can probably hazard a guess as to which deadline caused this.

[12] The timetable is equivalent to a topological ordering of a directed acyclic graph of tasks.

Stage	Start	End
Project specification	***T1 W1***	***T1 W2***
Understand the algorithms for simulating 2dc/2nc; read relevant parts of the literature concerning 2dc/2nc and 2dpda/2npda	T1 W2	T1 W10
Design the programming language to represent simplistic 2dc	T1 W2	T1 W3
Design syntactic sugar/macro features of the programming language in order to represent more complex 2dc	T1 W2	T1 W4
Design the programming language to represent 2nc	T1 W3	T1 W4
Progress report	***T1 W8***	***T1 W9***
Implement a lexer/parser for the language	T1 W10	T1 W10
Implement the 2dc/2nc construction algorithm	Christmas W3	Christmas W4
Prove that the construction algorithm produces correct automata	T2 W1	T2 W3
Implement the simulation algorithms for 2dc/2nc *(continued sporadically throughout the Easter break)* [Glü13; Ryt85]	T2 W2	T2 W5
Implement syntactic sugar features	T2 W5	T2 W7
Design the proof system for 2dc and test it on some simple examples	T2 W7	T2 W8
Research and design the proof rules for 2nc *(continued sporadically throughout the Easter break)*	T2 W8	T2 W10
Presentation	***T2 W8***	***T2 W10***
Final report	***Easter W1***	***T3 W2***

Figure 2.2: The actual execution of the project

- The proposed timetable, while sufficiently detailed, was poorly estimated. Had this estimation been more accurate, the project could have made more use of the timetable, although the author is not sure how this estimation could have been improved at the time given his lack of knowledge at the start of term 1.
- The risk of one specific conflicting deadline was not sufficiently anticipated. Had the impacts of this been identified and appropriately mitigated earlier, it would have resulted in the project's scope being scaled back sufficiently early to avoid the aforementioned wasted efforts.

2.4 Ethical concerns

The only ethical concerns associated with this project are related to those of intellectual property.

The implementation heavily depends on algorithms designed by other people (particularly Robert Glück[13] and Wojciech Rytter[14]). In order to ensure that their intellectual property rights are satisfied, the papers in which they present these algorithms are cited in this report [Glü13; Ryt85], and are also referenced in the codebase. In fact, the functions that implement their algorithms are named for them.

This implementation of this project also heavily depends on several libraries written by other people. In order to ensure their intellectual property rights, their libraries are referenced in the footnotes throughout chapter 5. These libraries (except for the Rust standard library) can also be found in the `dependencies` section of the `twoc/Cargo.toml` file in the codebase.

In order to completely guarantee the intellectual property rights of all these people, as well as to allow others to benefit from and extend this work, the programmatic outputs of this project are completely open-source.

[13] `https://scholar.google.com/citations?user=cGfKulEAAAAJ&hl=en`
[14] `https://www.mimuw.edu.pl/ rytter/`

Chapter 3

Language design

The project aimed to specify a programming language which was adherent to the following design principles:

- `twoc` should be easy to learn and comfortable to use for users with prior programming experience (its syntax for almost all of its operations are identical to those present in the C language for precisely this reason)
- `twoc` should be as easy as possible to write programs in without abstracting too far from the definitions of 2dc/2nc

In sec. 7.1.1, we discuss how well `twoc`'s design adheres to these principles.

The design process was split into the following phases:

1. Design language features that are sufficient to represent 2dc
2. Design additional control structures to represent 2nc
3. Design syntactic sugar/macros as appropriate
4. Prove that any `twoc` program corresponds to an equivalent 2dc/2nc (this requires that we define the construction procedure; if this is true we state that the language/construction is **correct**). *This proof is presented in chapter 4.*

Proving that any 2dc/2nc can be represented by a `twoc` program equivalent is also an important result (we call this result **completeness**), and would allow us to assert `twoc` is sufficiently expressive to decide all of the languages in $2DC \cup 2NC$. However, this implication is far less important to `twoc`'s intended purpose than its correctness. Therefore, due to time considerations, this report does not provide a proof of `twoc`'s completeness (see sec. 4.5).

The semantics of these statements described in this chapter are formalised in the context of Hoare logic in chapter 6. In this chapter, we informally describe their semantics in order to ensure the reader understands how to read and write programs in the `twoc` language.

3.1 Deterministic programs

3.1.1 Basic instructions

In order to simulate 2dc, `twoc` needs to be able to represent moving the readhead left and right; incrementing and decrementing the counter; and accepting or rejecting the input string. In order to achieve this, the following basic instructions are included in the language:

- `move(i);` moves the readhead i cells to the right for some `i` $\in [-n, n]$, where $n = |w|$ (negative values of `i` indicate that the readhead should be moved left; if `i` $> n$ or `i` $< -n$ then the readhead is moved to the corresponding endmarker)
- `c += j;` increments the counter by `j` $\in \mathbb{N}$ (`c` is assumed to be initialised to 0)

- `c -= j;` decrements the counter by `j` $\in \mathbb{N}$ (the case where `j` > `c` will be discussed in sec. 3.3)
- `accept;` accepts the input string
- `reject;` rejects the input string

If the program terminates without either an `accept;` or `reject;` statement being encountered, then it is assumed to have rejected the input string.

For convenience, we introduce the following definition:

Definition 3.1. *A **basic block** is a linear sequence of any number of `move(i)`, `c += j` or `c -= j` instructions.*

Note that any basic block of any size can be represented by two instructions: one move instruction, and one increment instruction. The parameters of these two instructions is simply the sum of the parameters of each of the move and increment instructions respectively (we refer to this operation as *contraction*, which we demonstrate in fig. 3.7).

3.1.2 Logic

2dc are allowed to decide whether or not to take a transition based on the symbol underneath the readhead, and whether or not the counter is equal to 0. This necessitates the inclusion of the following logical instructions:

- `true` and `false` always evaluate to true and false respectively
- `c == 0` evaluates to true iff the counter is 0
- `c != 0` evaluates to true iff the counter is not 0
- `read == 'x'` evaluates to true iff the symbol located at the readhead is `x` for some `x` $\in \Sigma$ (the readhead is assumed to be at the left endmarker at the start of execution)
- `read != 'x'` evaluates to true iff the symbol located a the readhead is not `x` for some `x` $\in \Sigma$

The left and right tape endmarkers are represented by the keywords `lend` and `rend` respectively.

`twoc` also allows for logical and (`&&`), or (`||`) and not (`!`) instructions to be applied to other logical statements. `!` is given the highest precedence, followed by `&&` and `||`.

3.2 Nondeterministic control structures

The control structures defined in the previous section are sufficient to describe 2dc. However, they do not describe any nondeterministic execution that may exist in a 2nc.

[Dij75] describes two nondeterministic control structures: one which describes nondeterministic choice, and another which describes nondeterministic repetition. The two new control structures we introduce are designed to permit the same functionality.

3.2.1 `branch` statements

The `branch` statement (fig 3.1) is used to denote a nondeterministic choice between multiple blocks of code, and is based both on the **if** command in [Dij75] and on the $\sqcap$ operator presented in [SS92]. It is always assumed that the automaton executes whichever block leads it to an accepting snapshot, if such a block exists on the given input word (this convention is referred to as *angelic* nondeterminism [Mam17; Bod+10]).

To demonstrate it's semantics, consider the example in fig 3.2 (assume that, at the start of execution, `c` $= 0$).

In this block of code, we initially set the counter value to 7. We then let the program choose between three different computation paths. The first branch decrements the counter to 0; the second branch results in an infinite loop; and the third branch decrements the counter value to 1. Following the `branch` statement, the program accepts if and only if the counter is 0.

```
    branch {
        // Some code
    } also {
        // Some more code
    } also {
        // Even more code
    }

```

Figure 3.1: An example of a `branch` *statement*

We now consider the behaviour of each branch. If the automaton selects branch 2, then it infinitely loops (thus rejecting), and if it selects branch 3, it cannot enter the `if` statement, meaning it also rejects. However, if the automaton selects branch 1, the counter is set to 0, meaning that the program accepts once it reaches the `if` statement.

Because our automaton's nondeterminism is assumed to be angelic, we assume that it picked the first branch; therefore, this small program results in the automaton reaching an accepting snapshot.

```
    c += 7;

    // Nondeterministically choose a block of code to execute
    branch {
        // 1: Decrement c until it is 0
        while (c != 0) { c--; }
    } also {
        // 2: Loop forever
        while (true) { c++; }
    } also {
        // 3: Decrement the counter but leave it nonzero
        c -= 6;
    }

    if (c == 0) { accept; }
    reject;

```

Figure 3.2: A small nondeterministic program that utilises a `branch` *statement*

3.2.2 `while-choose` statements

The `while-choose` statement (fig 3.3) is used to denote a nondeterministic choice of how many times to execute a loop, and is directly inspired by the use of the nondeterministic `choose` variable presented in [Bod+10], as well as the **do** command in [Dij75].

Note that, for this statement, we utilise the same syntax as normal deterministic while loops; however, we use the `choose` keyword as opposed to a logical condition. This syntax was chosen in order to highlight the similarities between this structure and deterministic while statements.

As before, the nondeterminism here is angelic, meaning that we assume the automaton stays in the loop for as long as is required to reach an accepting snapshot, if one exists for the given input word.

```
    while (choose) {
        // Some code
    }

```

Figure 3.3: An example of a `while-choose` *statement*

To demonstrate its semantics, consider the example in fig 3.4 (also assume that, initially, $\mathtt{c} = 0$).

In this example, we initially set the counter value to 5. We then decrement the counter a nondeterministic number of times, and the program accepts if and only if the counter value at the end is 0. Because our

automaton's nondeterminism is presumed to be angelic, we assume that it executes the loop (at least) 5 times, meaning that this block of code reaches an accepting snapshot.

```
1     c += 5;
2
3     // Decrement the counter a nondeterministic number of times
4     while (choose) { c--; }
5
6     if (c == 0) { accept; }
7
```

Figure 3.4: A small nondeterministic program that utilises a `while-choose` statement

3.3 The structure of `twoc` programs

All `twoc` programs must be structured as in fig. 3.5.

```
1     // Reject if we try to decrement the counter when c = 0
2     decr_on_zero = false;
3
4     // Set the alphabet to {x, y}
5     alphabet = [ 'x', 'y' ];
6
7     // DYCK LANGUAGE OVER {x, y}
8     // ACCEPTS WORDS LIKE xxxyyy, xyxyxy, xxyyxxyxyy, etc.
9
10    twoc (string) {
11        // Read the whole input string
12        while (read != rend) {
13            move(1);
14
15            // Count a yet-unmatched x
16            if (read == 'x') { c++; }
17
18            // Match an x with a y
19            else if (read == 'y') { c--; }
20        }
21
22        // Accept iff there are no unmatched xs
23        if (c == 0) { accept; }
24    }
25
```

Figure 3.5: An example of a complete `twoc` program accepting the Dyck language over $\Sigma = \{x, y\}$

The declaration on line 2 solves a semantic issue briefly mentioned in sec. 1.4. The issue is as follows: what should the automaton do if it tries to decrement a counter that is currently set to 0? The following two answers are both useful conventions for such an instruction:

1. The counter remains set to 0
2. The automaton immediately enters a rejecting snapshot

When writing test cases for `twoc`'s implementation, it was discovered that, in some cases, the first convention resulted in more concise programs; and in other cases, the same was true of the second.

It was decided to allow the user to select between these two conventions by setting a flag at the start of a `twoc` program. If the `decr_on_zero` flag is set to `true`, we follow the first convention; if it is set to `false`, we follow the second.

The user also needs some way of defining the alphabet of the language their program decides. The user does this with the `alphabet` declaration demonstrated on line 5, where they specify the symbols in their alphabet (which are assumed to be alphanumeric characters). If the program receives an input containing a

character not contained in the declared alphabet, then the input is implicitly rejected.

We also note that the body of the program itself (lines 11-23) is surrounded by `twoc (string) {}` (this was inspired by C's function definitions). Here, the `string` keyword asserts that this `twoc` program does not assume that its input is of any particular format (the other case will be discussed in sec. 3.4.2 and sec. 5.3).

3.4 Macros and syntactic sugar

The features described in this section are not implemented as fundamental statements in the `twoc` language; instead, these are treated as shorthands for longer sequences of fundamental `twoc` statements, and are implemented as a 'desugaring' transformation made to `twoc` programs (this will be discussed in more detail in sec. 5.2).

3.4.1 Simple macros

The first macro we consider is the `goto` instruction. `goto(lend)` moves the readhead to the left endmarker, and `goto(rend)` moves the readhead to the right endmarker. Fig. 3.6 demonstrates the desugared equivalents of these two instructions.

```
1    // ORIGINAL
2    goto(lend)
3
4    // DESUGARED
5    while (read != lend) { move(-1); }
6
7
8    // ORIGINAL
9    goto(rend)
10
11   // DESUGARED
12   while (read != rend) { move(1); }
13
```

Figure 3.6: The desugared equivalents of both `goto` instructions

We also introduce the `repeat` macro, which functions as `twoc`'s equivalent of a `for` loop. A `repeat` statement simply indicates that a block of code should be repeated some constant number of times. We demonstrate this construct in fig. 3.7.

We also introduce counter assignments to our language, which allows the user to assign a specific nonnegative integer value to the counter. This is simply syntactic sugar for a `while` loop that decrements the counter to 0, and an increment instruction that sets the counter. We demonstrate this in fig. 3.8.

3.4.2 Input formatting

We first recall the programs described in [Pet94] and the languages they decide (e.g. the program in fig. 1.2 decides $L = \{0^n 1^{2^n} : n \geq 0\}$). Every word in these languages is a series of unary-encoded integers separated by alphabet symbol, and Petersen's programs heavily leverage this fact. Rather than explicitly describing the movement of a readhead across an input string, his programs are written as procedures with integer parameters. These parameters can then be used by the program to modify the counter.

`twoc` provides syntactic sugar features that support this, in order to help users write concise programs that decide languages like the ones decided by Petersen's programs (this is demonstrated in fig. 3.9; its desugared equivalent can be found in sec. C.2).

```
// ORIGINAL
repeat (3) {
    c -= 2;
    move(3);
}

// DESUGARED
c -= 2;
move(3);

c -= 2;
move(3);

c -= 2;
move(3);

// CONTRACTED
c += -6;
move(9);
```

Figure 3.7: An example `repeat` *statement and its desugared and contracted equivalents*

```
// ORIGINAL
c = 23;

// DESUGARED
while (c != 0) { c--; }
c += 23;
```

Figure 3.8: An example counter assignment and its desugared equivalent

```
decr_on_zero = false;

// One alphabet symbol per argument
alphabet = [ '0', '1', '2' ];

twoc (int X, int Y, int Z) {
    c  = X;
    c += Y;
    c -= Z;

    if (c == 0) { accept; }
}
```

Figure 3.9: An example program that decides $L = \{0^x 1^y 2^z : x + y = z\}$

3.5 The construction algorithm

It should be fairly clear that `twoc` programs can be used to define 2dc/2nc (`twoc` was designed with this in mind). Here, we will explicitly discuss a procedure that turns some input program `prog` into an automaton in linear time. Note that this procedure is remarkably similar to how compilers generate intermediate code (e.g. LLVM IR[1]) for control flow statements [Aho+06]. In fact, this procedure (as discussed in Warwick's Compiler Design module[2]) served as the main inspiration for the construction algorithm.

In chapter 4, it is proven that the following procedure constructs a correct automaton (the automaton accepts some $a \in \Sigma^*$ if and only if the `twoc` program it was constructed from also accepts a). In sec. 5.3, we will discuss how this algorithm is implemented.

[1] `https://llvm.org/`

[2] `https://warwick.ac.uk/fac/sci/dcs/teaching/modules/cs325/`. *Thank you Gihan, very cool.*

Note that this procedure maintains a global variable s, which stores the last state introduced to the automaton by the procedure.

Also note that we represent the automaton as the adjacency list of a graph, with the states corresponding to vertices and the transitions corresponding to edges (as in fig. 1.3).

3.5.1 Basic blocks

Trivially, each basic block corresponds to a single transition in the automaton. We convert a basic block `move(i); c += j;` into an equivalent automaton as follows:

1. Introduce a new state q
2. Introduce a new transition from s to q which moves the readhead `i` cells to the right and increments the counter by `j`
3. Update $s \leftarrow q$

3.5.2 Accept and reject statements

When the algorithm encounters an `accept;` statement, it simply makes s an accept state by updating $F \leftarrow F \cup \{s\}$. It also introduces a transition from s that decrements the counter by 1 and returns to s. This transition ensures that if the automaton can reach s, it can also reach s with an empty counter (this fact will be leveraged by the algorithms that simulate these automata; see sec. 5.4).

When the algorithm encounters a `reject;` statement, it records s as being a rejecting state and continues constructing the rest of the program. Once the automaton has been fully constructed, the algorithm removes all transitions that leave each rejecting state. This ensures that if the automaton reaches s, it must reject its input string, as it cannot possibly reach an accepting state from s.

3.5.3 Logical conditions

In order for the procedure to work for `if-else` and `while` statements, a procedure needs to be defined that can produce transitions that the automaton can take if and only if some arbitrary logical condition is true.[3] The procedure is defined inductively:

3.5.3.1 Base cases:

- A condition of the form `c == 0`, `c != 0`, or `read == 'x'` for some `x` $\in \Sigma$ can easily be transformed into a single transition (this follows trivially from the definition of 2dc/2nc).
- A transition of the form `read != 'x'` can be transformed into several transitions that check `read == 'y'` for each `y` $\in \Sigma - \{$`x`$\}$.

Note that this implies that transitions of the form `!cond` where `cond` is one of the base conditions can also be represented (`!(c == 0)` is equivalent to `c != 0`, etc.).

3.5.3.2 Inductive cases:

- A condition of the form `a || b` can be represented as 2 parallel transitions: one that checks `a` and one that checks `b`
- A condition of the form `a && b` can be represented as 2 sequential transitions: one that checks `a` and one that checks `b`
- A condition of the form `!(a || b)` can be transformed using de Morgan's law into a condition of the form `!a && !b`, which can then be represented in the automaton as above; the same applies for conditions of the form `!(a && b)`

[3] Note that this procedure is not actually utilised by the construction's implementation, and is included here in order to ensure that the construction conforms precisely to the definition of 2dc/2nc. A deprecated implementation that uses this can be found in the `construct_conditional_transitions()` function in `twoc/src/automaton/construction_old.rs`.

3.5.4 `if-else` statements

Consider an `if-else` statement of the following form (`cond` is an arbitrary conditional statement):

```
if (cond) {
  // if-block
} else {
  // else-block
}
```

Figure 3.10: An `if-else` statement

In order to convert this into an equivalent automaton, we do the following:

1. Introduce new states q_{if}, q_{else} and q_{end}
2. Add transitions that check `cond` from s to q_{if}, and transitions that check `!cond` from s to q_{else}
3. Update $s \leftarrow q_{if}$, and recursively construct an automaton for `if-block`. Save the value of s after this process as q_{ifEnd}.
4. Update $s \leftarrow q_{else}$, and recursively construct an automaton for `else-block`. Save the value of s after this process as $q_{elseEnd}$.
5. Update $s \leftarrow q_{end}$, and add ϵ-transitions from q_{ifEnd} and $q_{elseEnd}$ to q_{end}

This can be trivially adapted to handle `if` and `if-elseif-else` statements, and the procedures for the other control flow structures are also very similar to this process.

3.5.5 `while` and `while-choose` statements

Consider a `while` statement of the following form:

```
while (cond) {
  // while-block
}
```

Figure 3.11: A `while` statement

In order to convert this into an equivalent 2nc, we do the following:

1. Introduce new states q_{while} and q_{end} and save the value of s to q_{start}
2. Add transitions that check `cond` from s to q_{while}, and transitions that check !*cond* from s to q_{end}
3. Update $s \leftarrow q_{while}$ and recursively construct an automaton for `while-block`. Save the value of s after this process as $q_{whileEnd}$.
4. Add an ϵ-transition from q_{while} to q_{start} and update $s \leftarrow q_{end}$

The conversion procedure is almost identical for `while-choose` statements, the only difference being in step 2: instead of constructing conditional transitions from s to q_{while} and q_{end}, we simply introduce ϵ-transitions from s to q_{while} and q_{end}.

3.5.6 `branch` statements

Consider a `branch` statement of the following form:

```
branch {
  // branch-block-1
} also {
  // branch-block-2
} also {
  // ...
} also {
  // branch-block-k
}

```

Figure 3.12: A `branch` statement

In order to convert this into an equivalent 2nc, we do the following:

1. Introduce a new state q_{end}, and new states q_i for each $i \in [1, k]$
2. For each $i \in [1, k]$, add ϵ-transitions from s to each q_i
3. For each $i \in [1, k]$, update $s \leftarrow q_i$ and recursively construct a 2nc for `branch-block-i`. Save the value of s after this process as f_i.
4. For each $i \in [1, k]$, add a transition from f_i to q_{end}.
5. Update $s \leftarrow q_{end}$.

3.5.7 Runtime analysis

Although the time complexity of this procedure is not of enormous importance (in practice, these constructions are very fast), a brief analysis of its runtime is included.

Let *prog* be an input program stored in n bits. We make the following observations:

- Each program contains $\Theta(n)$ statements (this follows from the fact that each statement and logical operation in the language corresponds to $\Theta(1)$ characters)
- The algorithm considers each statement in the program exactly once
- Excluding the recursive calls, each statement can be constructed in time $O(1)$ (if we assume that adding states and transitions to an automaton takes time $O(1)$)

This yields a runtime of

$$\#\text{ statements} \times O(1) = O(n) \tag{3.1}$$

In practice, *prog* will be converted into an abstract syntax tree by the lexer and parser before it is constructed; this does not affect the asymptotic runtime of our algorithm, as the AST is of size $\Theta(n)$, which the parser constructs in time $O(n)$ (see sec. 5.1) [Leo91].

Chapter 4

Language correctness

4.1 Additional definitions

4.1.1 Δ-snapshots

We first recall the definition of snapshots of 2dc/2nc (def. 1.2). Alongside these definitions, we also introduce the notion of Δ-snapshots, which describe how the automaton moves from one snapshot to the next. More formally,

Definition 4.1. *A* **Δ-*snapshot*** *is a 3-tuple* $(q', \Delta i, \Delta x) \in Q \times [-n, n] \times \mathbb{Z}$.

During a computation step moving from snapshot c to c', the automaton updates its state to q', moves to the readhead Δi cells to the right, and adds Δx to the value of the counter.

We will use the notation $c' = c + d$ to state that a Δ-snapshot d applied to a snapshot c yields a new snapshot c'. More formally,

$$(q, i, x) + (q', \Delta i, \Delta x) = (q', \text{clamp}(0, i + \Delta i, n), \max\{0, x + \Delta x\}) \tag{4.1}$$

where $\text{clamp}(0, i + \Delta i, n) = \max\{0, \min\{i + \Delta i, n\}\}$.

Similarly, we will state that $d = c' - c$ if the Δ-snapshot d takes us from snapshot c to snapshot c'. We will also state that $c' = c + n \cdot d$ for some $n \geq 0$ if applying d to c n times takes us to snapshot c'.

4.1.2 Snapshots of `twoc` programs

We first introduce the notion of labels for `twoc` programs; this concept is intended to equate to states in the automaton. Program labels and corresponding automaton states will be given the same names in the proof to emphasise this.

Every statement in `twoc` has two labels associated with it: one prior to its execution (called its entry label) and one following it (called its exit label). For two consecutive statements A and B in the same block, A's exit label is the same as B's entry label. We demonstrate this concept in fig. 4.1. Note that a basic block is considered to be one statement for the purposes of this definition.

We then define a snapshot of a deterministic `twoc` program P with labels L to be some $s \in L \times [0, n - 1] \times (\mathbb{N} \cup \{0\})$. Clearly, this is enough information to completely describe the current state of P's execution.

Δ-snapshots are defined in the same way as before. If a program is in a snapshot s after it passes an `accept`/`reject` statement, then the snapshot is accepting/rejecting (assume that reaching an `accept` statement means that the program clears its counter before terminating).

Additionally, if a block of code executes without accepting, rejecting or infinitely looping, we say that the block of code **terminates**. We also say that a block of code **halts** if it either accepts or rejects.

4.2 Overview

In order to prove the correctness of the programming language in the deterministic case, we prove that, for some arbitrary input $w \in \Sigma^*$, some arbitrary program P with labels L, and for the automaton M constructed by the procedure on P:

```
alphabet = ['0', '1'];
decr_on_zero = true;

twoc (string) {
    // q_0
    move(1);

    // q_1
    while (read == '0') {
        // q_2
        c++;
        move(1);
        // q_3
    }

    // q_4
    while (read == '1' && c != 0) {
        // q_5
        c--;
        move(1);
        // q_6
    }

    // q_7
    if (read == rend && c == 0) {
        // q_8
        accept;
        // q_9
    }
    // q_10
}

```

Figure 4.1: A `twoc` program with comments indicating where the labels are

- if P starts in snapshot $s = (q_s, i_s, x_s)$ and ends in snapshot $f = (q_f, i_f, x_f)$, then M, starting in snapshot s, ends in snapshot f
- if P reaches an `accept` statement (i.e. P accepts w), then M terminates in an accepting snapshot
- if P reaches a `reject` statement or the end of the program is reached (i.e. P doesn't accept w), then M halts in a rejecting snapshot or infinitely loops
- if P loops infinitely (i.e. it enters a `while` statement that it cannot exit), then the automaton enters some infinite cycle or a rejecting snapshot

We first prove that the algorithm is correct in the deterministic case (i.e. for programs that contain no `branch` or `while-choose` statements), via induction on the structure of `twoc` programs. We will then extend this result to nondeterministic programs.

4.3 The proof

Observation 4.1. *The construction algorithm introduces a new state q for each label $l \in L$.*

Proof. This can be seen from the construction procedure; for every statement in the language, it introduces transition(s) from the last state it introduced (this state corresponds to the statement's entry label). It also introduces a common final state for all the gadgets generated as part of the statement (this state corresponds to the statement's exit label). □

4.3.1 Base cases

Lemma 4.1. *The construction algorithm is correct for basic blocks.*

Proof. Consider a basic block of the form `move(a); c += b;` with $\texttt{a} \in [-n, n]$, $\texttt{b} \in \mathbb{Z}$ and entry and exit labels $q_s, q_f \in L$. When executing this basic block from snapshot s, the program clearly enters snapshot $f = (q_f, \text{clamp}(0, i_s + a, n), \max\{x_s + b, 0\})$.

In order to construct this, the algorithm introduces a transition from q_s to q_f that moves the readhead `a` cells to the right and adds `b` to the counter, which M takes unconditionally. Therefore, the automaton goes from s to f. □

Lemma 4.2. *The construction algorithm is correct for* `accept` *statements.*

Proof. Consider an `accept` statement with entry and exit labels $q_s, q_f \in L$. By definition, the snapshot $f = (q_f, i_f, 0)$ must be accepting, and the program accepts w.

In order to construct this statement, the algorithm makes q_s an accepting state, introduces a single transition from q_s, which decrements the counter by 1, and removes any other transitions out of q_s. If M is in snapshot $s = (q_s, i_s, x_s)$ with $x_s > 0$ once it reaches q_S, it must repeatedly take the decrementing transition until $x_s = 0$. Once $x_s = 0$, the automaton has reached snapshot $f = (q_f, i_f, 0)$. □

Lemma 4.3. *The construction algorithm is correct for* `reject` *statements.*

Proof. Consider a `reject` statement with entry and exit labels $q_s, q_f \in L$, and assume the program reaches it from the snapshot $s = (q_s, i, x)$. Upon executing this statement, the program's snapshot is unchanged. By definition, s must be rejecting, and the program rejects w.

In order to construct this statement, the algorithm removes all transitions from q_s, meaning that once M enters q_s, it cannot reach any other state from it, or change the position of the readhead or the value of the counter. By definition, M halts in s, which is rejecting. □

4.3.2 Logical conditions

Lemma 4.4. *If a logical condition Φ in P is checked on $s = (q_s, i_s, x_s)$, and is immediately followed by $f = (q_f, i_s, x_x)$ if and only if s satisfies Φ, then M can go from s to f via these transitions if and only if s satisfies Φ.*

Proof. The proof is by induction on the structure of conditional statements. (Note that any sequence of transitions in M that solely make conditional checks cannot affect the position of the readhead or the value of the counter.)

4.3.2.1 Base cases

For Φ of the form `c == 0`, `c != 0` and `read == 'a'` for some $\texttt{a} \in \Sigma$, the construction introduces a single transition from q_s to q_f that can, by the definition of 2dc, be taken if and only if s satisfies Φ.

For Φ of the form `read != 'a'` for some $\texttt{a} \in \Sigma$, the construction introduces transitions from q_s to q_f that check `read == 'b'` for each $\texttt{b} \in \Sigma - \{a\}$. Clearly, M enters f if and only if the symbol at the readhead is not `a`.

Note that these cases also mean that Φ of the form `!(c == 0)`, `!(c != 0)`, `!(read == 'a')` and `!(read != 'a')` also satisfy the lemma, as each of these statements can be trivially transformed into a statement of one of the previous forms.

4.3.2.2 Inductive cases

Assume that the lemma is true for conditional statements α and β, as well as their negations $!\alpha$ and $!\beta$.

For Φ of the form α `&&` β, the algorithm introduces a sequence of transitions from q_s to some new state q_α that check if s satisfies α. It also introduces a sequence of transitions from q_α to q_f that check if $s' = (q_\alpha, i_s, x_s)$ satisfies β.

Assume that Φ is satisfied for s_M, meaning that both α and β are also satisfied for s. If this is the case, by the induction hypothesis, M can take the transitions from q_s to q_α. Also by the induction hypothesis, it can then take the transitions from q_α to q_f. Therefore, Φ implies that M ends in snapshot f.

Assume that Φ is not satisfied for s, meaning that at least one of α and β is not satisfied for s. If α is unsatisfied, by the induction hypothesis, M cannot transition from q_s to q_α, and thus it cannot transition to q_f. If β is unsatisfied, also by the induction hypothesis, M may be able to enter q_α, but cannot transition from q_α to q_f. Therefore, $!\Phi$ implies that M does not end in snapshot f_M.

For Φ of the form α `||` β, M introduces a sequence of transitions from q_s to q_f that M can take if and only if s satisfies α. It also introduces a series of parallel transitions from q_s to q_f that M can take if and only if s satisfies β.

Assume that Φ is satisfied for s. Then either one of α or β must be satisfied for s. M can simply take the transitions associated with the one that is satisfied to reach f from s.

Assume that Φ is not satisfied form s. Then neither α nor β is satisfied for s, meaning that M cannot transition to f from s.

Note that in the case that α and β are both true, M ceases to be strictly deterministic, as it can simultaneously pick both sequences of transitions. However, both sets of transitions eventually take M to q_f; therefore, M does not actually behave nondeterministically (in a sense, M is *subtly* deterministic).

For Φ of the form `!`Ψ where Ψ is also inductively defined, we consider the following cases:

1. $\Psi =$ `!`α
2. $\Psi = \alpha$ `&&` β
3. $\Psi = \alpha$ `||` β

In case 1, $\Phi =$ `!!`α, which is constructed as a series of transitions that check α. It follows immediately from the induction hypothesis that this is correct.

In case 2, $\Phi =$ `!(`α `&&` β`)`. The construction transforms this statement, by de Morgan's law, into one of the form `!`α `||` `!`β. By the induction hypothesis, and the previous inductive case proving that `||` is constructed correctly, the construction is correct for case 2. Case 3 can also be proven correct using the same argument.

□

4.3.3 Inductive cases

Lemma 4.5. *The construction algorithm is correct for* `if-else` *statements.*

Proof. We first assume, for the purpose of induction, that the construction is correct for two subprograms P_{if} and P_{else}. Assume that P_{if} and P_{else}, if they terminate, have Δ-snapshots d_{if} and d_{else}.

We now consider an `if-else` statement of the following form:

```
1   // qs
2   if (cond) {
3     // if_s
4     P_if
5     // if_f
6   } else {
7     // else_s
8     P_else
9     // else_f
10  }
11  // qf
12
```

with $\{q_s, q_f, if_s, if_f, else_s, else_f\} \subseteq L$.

The construction algorithm introduces states if_s, if_f, $else_s$ and $else_s$ to M. It also introduces transitions that check `cond` from q_s to if_s, transitions that check `!cond` from q_s to $else_s$, and ϵ-transitions from if_f and $else_f$ to q_f. These conditional transitions behave correctly via Lemma 4.4.

Assume that `cond` satisfies $s = (q_s, i_s, x_s)$. In this case, the program executes P_{if}. If P_{if} terminates, the program stops at label l_f, meaning that $f_P = s_P + d_{if}$; and if P_{if} infinitely loops or halts, the program infinitely loops or halts.

In this case, by Lemma 4.4, M must take the transitions corresponding to `cond`, meaning it transitions to state if_s. By the induction hypothesis, if P_{if} infinitely loops or halts, then M infinitely loops or halts. Also

by the hypothesis, if P_{if} terminates, M arrives at state if_f, having executed the Δ-snapshot d_{if}. Finally, M q_f via the ϵ-transition, with snapshot $f = s + d_{if}$. Therefore, the construction is correct in this case.

Now assume that `cond` does not satisfy s_P and s_M; therefore, s_P and s_M satisfy `!cond`. If P_{else} terminates, the program reaches label l_f with $f = s+d_{else}$; and if P_{else} infinitely loops or halts, the program will infinitely loop or halt. By the exact same argument, the construction is also correct in this case, with M either looping infinitely, halting or reaching q_f with snapshot $f = s + d_{else}$.

□

Lemma 4.6. *The construction algorithm is correct for* `while` *statements.*

Proof. As before, for the purpose of induction, we assume that the construction is correct for a subprogram P_{while}. We also assume that, on iteration $i \in \mathbb{N}$, P_{while} has Δ-snapshot d_i if it does not halt or terminate.

We now consider a `while` statement of the following form:

```
// q_s
while (cond) {
  // while_s
  P_while
  // while_f
}
// q_f

```

with $\{q_s, while_s, while_f, q_f\} \subseteq L$.

The construction algorithm introduces states $while_s$ and $while_f$ to M. It also introduces transitions that check `cond` from q_s to $while_s$, `!cond` from q_s to q_f, and an ϵ-transition from $while_f$ to q_s. These conditional transitions behave correctly via Lemma 4.4.

Assume that P never enters the while loop (i.e. `cond` is initially false), meaning that the program stops at label q_f with snapshot $f = (q_f, i_s, x_s)$. By Lemma 4.4, the automaton must immediately go to state q_f with snapshot $f = (q_f, i_s, x_s)$. M must therefore be correct in this case.

Assume that P_{while} is executed by the program $n \in \mathbb{N}_0$ times before exiting to label q_f with snapshot $f = s + \sum_{i=1}^{n} d_i$. This implies that, for $j \in [1, n-1]$, $s + \sum_{i=1}^{j} d_i$ satisfies `cond`. It also implies that $s + \sum_{i=1}^{n} d_i$ does not satisfy `cond`.

By the induction hypothesis, P_{while} is constructed correctly in M; for each iteration i, the Δ-snapshot of the states and transitions from $while_s$ to $while_f$ in M is d_i. At the end of each of these passes, M must take the ϵ-transition from $while_f$ back to $while_s$.

By Lemma 4.4, if the automaton is in state q_s with snapshot $s + \sum_{i=1}^{j} d_i$ for $j \in [1, n-1]$, the automaton must go to state $while_s$. This implies that the loop is executed exactly n times.

After the nth iteration, M has snapshot $s + \sum_{i=1}^{n} d_i$. Also via Lemma 4.4, in this snapshot, M cannot return to $while_s$, and must take the transition corresponding to `!cond` to q_f, with $f = s + \sum_{i=1}^{n} d_i = f_P$. M must therefore be correct in this case.

Assume that P_{while} terminates or infinitely loops on iteration n. Trivially, with a similar argument to the previous case, M must also do the same.

Assume that the `while` loop in P never terminates, i.e. for all $i \in \mathbb{N}$, d_i satisfies `cond`. Trivially, with a similar argument to the previous two cases, M must also do the same. □

Theorem 4.7. *The construction algorithm is correct for deterministic programs.*

Proof. This follows by induction from the previous lemmas. □

4.4 Extending the proof to nondeterminism

So far we have only considered the problem in the deterministic case. The input programs have contained no nondeterministic control structures, and the output automaton has been assumed to be deterministic.

These lemmas are still valid for our overall computation, however, as none of the previous structures were nondeterministic, and thus could not have introduced any new branching behaviours (aside from those arising from `||` conditions, which do not actually behave nondeterministically in practice).

Lemma 4.8. *The construction algorithm is correct for* `branch` *statements.*

Proof. For the purposes of induction, we assume that we have correctly constructed subprograms $P_1, \cdots, P_k$ and P_{follow}. Let $X \subseteq [1, k]$ be the set of all i such that P_i terminates. Assume that for all $i \in X$, P_i terminates with Δ-snapshot d_i.

Consider a `branch` statement of the following form:

```
// q_s
branch {
  // b_s1
  P_1
  // b_f1
} ... also {
  // b_sk
  P_k
  // b_fk
}
// q_f
P_follow

```

with $\{b_{is}, b_{if}\} \subseteq L$ for all $i \in [1, k]$, and $\{q_s, q_f\} \subseteq L$.

The construction algorithm introduces states b_{si} and b_{fi} for each $i \in [1, k]$. It also introduces ϵ-transitions from q_s to each b_{si}, and from each b_{fi} to q_f.

We assume that the program starts execution in snapshot s_P.

Assume that, for some $i \in X$, P_i halts and accepts when executed from snapshot s_P. We then deduce that our program chooses to execute P_i. We can also deduce that M chooses to take the corresponding ϵ-transition to state b_{si}. By our induction hypothesis, the automaton must also halt and accept.

Assume that the above case is false, and that P_{follow} accepts if the program enters it in snapshot $s + d_i$ for some $i \in X$. Then the program will execute P_i, and enter configuration $s + d_i$. By the induction hypothesis, we can assume that the program accepts.

Assume that the above cases are false and that P_i halts and rejects for all $i \in X$. If this is true, then the automaton can only select subprograms that reject or infinitely loop, meaning the program must reject or infinitely loop. By the construction, M must also do the same.

Assume that the above cases are false (meaning that P_{follow} must halt or infinitely loop), and that, for some $i \in X$, P_i terminates without halting and rejecting. Then the program executes P_{follow} from snapshot s. By the construction, M can choose the same option by selecting the correct transition. By the induction hypothesis, M behaves the same as P_{follow} when being executed from snapshot $s + d_i$, and thus behaves correctly.

□

Lemma 4.9. *The construction algorithm is correct for* `while-choose` *statements*

Proof. As before, for the purpose of induction, we assume that the construction is correct for a subprograms P_{while} and P_{follow}. We also assume that, on iteration i, P_{while} has Δ-snapshot d_i if it does not halt or infinitely loop.

We now consider a `while-choose` statement of the following form:

```
// q_s
while (choose) {
  // while_s
  P_while
  // while_f
}
// q_f
P_follow

```

with $\{q_s, while_s, while_f, q_f\} \subseteq L$.

The construction algorithm introduces states q_s, $while_s$, $while_f$ and q_f. It also introduces ϵ-transitions from q_s to $while_s$, $while_f$ to q_s, and q_s to q_f.

Assume that P_{while} halts and accepts after being executed n times (i.e. snapshot $s + \sum_{i=1}^{n} d_i$ is accepting). Then we assume the program executes P_{while} exactly n times before terminating. Clearly, via the induction hypotheses and the construction, the automaton can also choose this computation path (the only case in which it may not is if P_{follow} halts and accepts on snapshot $s_M + \sum_{i=1}^{m} d_i$ for some $m < n$, which results in the same behaviour).

Assume that the above case is false, and that P_{while} terminates after being executed n times, and that P_{follow} halts and accepts from snapshot $s_M + \sum_{i=1}^{n} d_i$. Then the program must halt and accept on this block of code. Clearly, via the construction, the automaton can do the same by choosing the relevant ϵ-transitions to execute P_{while} n times, and then choosing the ϵ-transition to go to P_{follow}.

Assume that the above cases are false and that, for some number of iterations $n \geq 0$, P_{while} terminates (also assume that it may be the case that $n = \infty$). Also assume that P_{follow} rejects/loops on all configurations $s + \sum_{i=1}^{m} d_i$ with $m \leq n$. Then clearly the program must reject/loop. Clearly, via the construction, the automaton must either infinitely loop by executing P_{while} infinitely many times, or it must mimic P_{follow}'s behaviour by choosing to execute P_{while} at most m times, and then executing P_{follow}. □

Theorem 4.10. *The construction algorithm is correct for nondeterministic programs.*

Proof. This follows by induction from the previous lemmas. □

4.5 A note on `twoc`'s completeness

These theorems imply the following statements (here, TWOC_d and TWOC_n denote the classes of all languages decided by deterministic and nondeterministic `twoc` programs respectively):

- $\text{TWOC}_d \subseteq \text{2DC}$
- $\text{TWOC}_n \subseteq \text{2NC}$

Ideally, the equivalence of these classes would have also been proven in order to assert the expressiveness of `twoc` as well, by additionally proving that every 2dc/2nc could be transformed into an equivalent `twoc` program (i.e. $\text{2DC} \subseteq \text{TWOC}_d$). It is conjectured that this is the case: in this section, we discuss why.

We first note that, if we extend our language with `jump()` statements and labels[1] (equivalent to the branch statements in LLVM IR [Rod18; Pro03b]), then this theorem is trivial. Given some 2dc, for each $q \in Q$, we simply introduce a subprogram labelled `Lq`, and include `if` statements that check each element of $\Sigma \times \{0, 1\}$ to encode the automaton's behaviour for all alphabet and counter values. An deterministic example is presented in 4.2.

We can trivially do the same for 2nc through the use of `branch` statements; we simply nondeterministically choose between each element of $\delta(q, x, c)$ for each $(x, c) \in \Sigma \times \{0, 1\}$.

For fig. 4.2, assume the following:

- $\Sigma = \{a, b\}$
- $\{3, 4\} \subseteq Q$
- $\delta(3, a, 0) = \{(3, -3, 2)\}$
- $\delta(3, b, 1) = \{(4, 8, 0)\}$
- $\delta(3, a, 1) = \delta(3, b, 0) = \emptyset$

[1] It was decided not to include these in `twoc` due to the arguments presented by Dijkstra in [Dij68], as well as their absence from most modern structured programming languages.

```
...
L3 :
if      (read == 'a' && c == 0) { c += 2; move(-3); jump(L3); }
else if (read == 'a' && c == 1) { reject; }
else if (read == 'b' && c == 0) { move(8); jump(L4); }
else if (read == 'b' && c == 1) { reject; }

L4 :
...

```

Figure 4.2: A state constructed as a subprogram with labels and `jump()` statements

Therefore, if every `twoc` program with `jump()` instructions can be converted into a normal `twoc` program *without* jump instructions, then it follows that $\mathrm{TWOC}_d = \mathrm{2DC}$ and $\mathrm{TWOC}_n = \mathrm{2NC}$. It is conjectured that this is the case (note that this is true for Turing-complete programming languages via the Böhm–Jacopini theorem [BJ66]).

Chapter 5

Language implementation

Any implementation of the `twoc` language can be interpreted as a function $f : \texttt{TWOC} \times \Sigma^* \to \{0, 1\}$, where `TWOC` is the language of all well-formed `twoc` programs. $f(\texttt{prog}, w) = 1$ if and only if w is accepted by the automaton defined by `prog`.

In order to implement such a function, we first have to construct an automaton for a given `twoc` program. We do this according to the following transformations (also shown in fig. 5.1):

1. Convert the program from a string of characters into an abstract syntax tree (**lexing and parsing**)
2. Remove syntactic sugar/macros from the AST (**desugaring**)
3. Convert the AST into a graph representation of an automaton (**automaton construction**)

Once we have an automaton, we can simulate the automaton's execution on the input string to find out if `prog` accepts or rejects w.

Throughout this chapter, reference will be made to the project codebase.[1] If you wish to look through any of the source code yourself, it is recommended that you clone the repository to your own machine, and that you open the project with the Visual Studio Code text editor.[2] A list of recommended VSCode extensions, as well as build/run instructions, can be found in `twoc/README.md`.

Footnotes throughout this chapter specify relevant files in the codebase, and these also link to pages in `twoc`'s GitHub repository.

5.1 Lexing and parsing

The lexing and parsing stages of this implementation lean heavily on the LALRPOP parser generator,[3] which is demonstrated in fig. 5.2. To use LALRPOP, the programmer specifies a LR(1) grammar for the language they intend to parse, along with a simple syntax-directed translation scheme; the LALRPOP library then compiles this into a lexer and a bottom-up LR(1) parser, which can be invoked in order to construct an abstract syntax tree.

An LR(1) grammar for the `twoc` language in Backus-Naur form can be found in appendix A. The actual grammar given as input to LALRPOP (including the translation scheme) can be found in the `twoc` codebase.[4]

5.1.1 The grammar

We briefly discuss some decisions made when designing `twoc`'s LR(1) grammar.

In order to ensure that the conditions of an LR(1) grammar were satisfied when implementing one for `twoc`, it was decided that the grammar would be designed to the stricter constraints required by LL(k) grammars [Nij82]. This was due in part to the author's prior experience working with LL(k) grammars and parser generators. The only consequence of this decision was that the grammar contains no left-recursive productions.

[1] `https://github.com/tomdaboom/twoc`
[2] `https://code.visualstudio.com/`
[3] `https://github.com/lalrpop/lalrpop`
[4] Located in `twoc/src/parser/sugar/sugar_grammar.lalrpop`

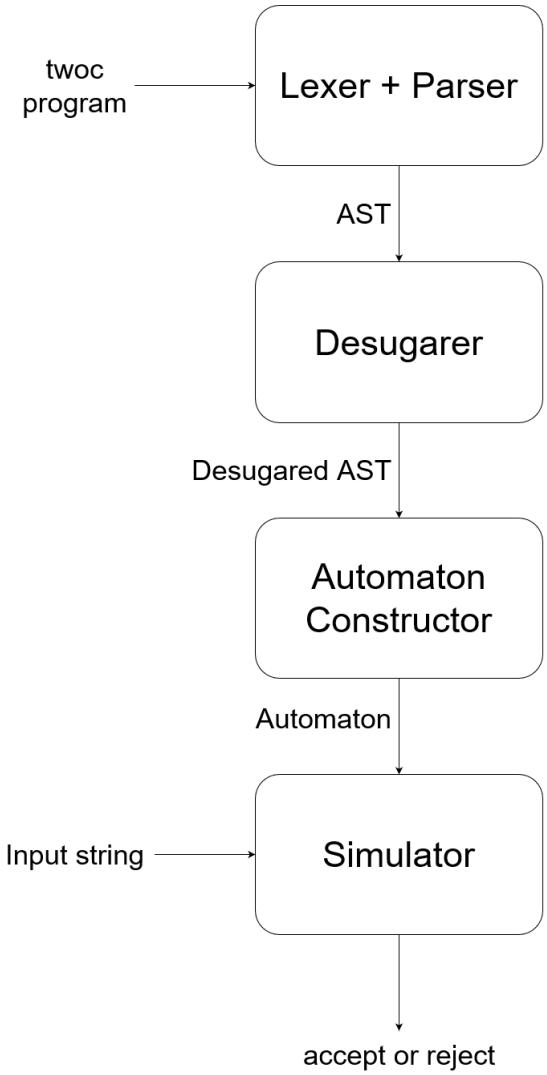


Figure 5.1: A pipe-and-filter diagram demonstrating an implementation of the `twoc` language

Another key decision was to embed the precedence of boolean operators into the grammar, as opposed to using a precedence table or some other method of encoding operator precedence [Pro03a]. This decision was made to ease the grammar's implementation in LALRPOP (using another method would have made this process significantly more complex). It is also worth noting the very small number of operators present in the `twoc` language; this meant that embedding this in the grammar only required the use of two extra productions (the `Cond` production separates boolean expressions by `||`s, and the `AndCond` production separates by `&&`s).

5.1.2 The abstract syntax tree

The main output of the parser (aside from syntax errors) is a tree representation of the input program (the abstract syntax tree), which is much easier to manipulate than the input program. An example AST for the program presented in fig. 3.5 can be found in appendix B.

Here, we discuss the design and implementation of this data structure, which is implemented as a collection of recursive algebraic data types (each type is implemented as an enum). This was chosen over implementing each AST node as a struct for the following reasons:

- An enum implementation is drastically more consice than a struct implementation. This is because each AST node can be defined using a single line in one large enum, while any struct implementation would require the implementation of one struct for each AST node (as well as an interface to encapsulate all AST nodes).

```
// Rule to parse else-if statements
ElseIf : ast::Stmt = {
    "else" "if" "(" <cond:Cond> ")" "{"
        <elif_body:StmtList>
    "}" <else_body:ElseBody?>
        => match else_body {
        None             => ast::Stmt::If(cond, elif_body, Vec::new()),
        Some(else_block) => ast::Stmt::If(cond, elif_body, else_block),
    },

    "else" "if" "(" <cond:Cond> ")" "{"
        <elif_body:StmtList>
    "}" <other_elif:ElseIf>
        => ast::Stmt::If(cond, elif_body, vec![other_elif]),
}
```

Figure 5.2: A production in `twoc/parser/sugar/sugar_grammar.lalrpop`

- enum types can very easily leverage Rust's pattern-matching statements (this is extremely useful when writing functions that take ASTs as input).

The `Stmt` type[5] corresponds to a single statement in the twoc language. Sequences of statements (corresponding to a block of code) are stored using Rust's inbuilt vector data structure (blocks of code are therefore of type `Vec<Stmt>`).

Instances of the `Stmt` type may also contain values of other types. These are either the parameters of instructions (how much to move/increment, etc.), logical conditions, or sequences of statements (hence making this type recursive).

Logical conditions are implemented using the `Cond` type;[6] its implementation is very similar to that of the `Stmt` type. Logical conditions can either contain the parameters of checks (e.g. `read == 'x'`, `c != 0`, etc.), or they can contain pointers to other logical instructions (e.g. `&&` and `||` statements).

Entire `twoc` programs are stored using the `Prog` type,[7] which is a struct containing:

- The program code (of type `Vec<Stmt>`)
- The alphabet (of type `HashSet<char>`)
- The value of the `decr_on_zero` flag (of type `bool`)
- A list of input parameters (of type `Vec<String>`)
- A map from input parameters to alphabet symbols (of type `HashMap<String, char>`)

This struct is shown in fig. 5.3.

The parameter map is constructed when this structure is initialised: if the programmer has provided any input parameters, the ith parameter maps to the ith character in the alphabet. For example, the program in fig. 3.9 (with `alphabet = ['0', '1', '2']` and input parameters `(int X, int Y, int Z)` would have its parameter map set to $\{(\texttt{X}, \texttt{'0'}), (\texttt{Y}, \texttt{'1'}), (\texttt{Z}, \texttt{'2'})\}$. This map is used by the desugarer to identify which part of the input string a parameter corresponds to.

5.2 Desugaring

The desugarer is implemented as a function `convert_sugar()`,[8] that receives a sugared program as argument (of type `SugarProg`; this is an alias for the `Prog` type discussed previously) and converts it to a program containing no syntactic sugar (of type `Prog`[9]).

[5] Located in `twoc/src/parser/sugar/ast.rs`
[6] Located in `twoc/src/parser/ast.rs`
[7] Located in `twoc/src/parser/sugar/program.rs`
[8] Located in `twoc/src/parser/sugar/convert_sugar.rs`
[9] Located in `twoc/src/parser/program.rs`

```rust
pub struct Program {
    // The actual program itself
    pub stmts  : Vec<ast::Stmt>,

    // The input alphabet
    pub alpha  : HashSet<char>,

    // Any parameters in our program
    pub pars   : Vec<String>,

    // A symbol table mapping input parameters to input characters
    pub parmap : HashMap<String, char>,

    // Whether or not c-- is a valid instruction when c == 0
    pub decr_zero : bool,
}
```

Figure 5.3: The `Program` struct in `twoc/parser/sugar/program.rs`

Note that a separate AST enum data type exists for programs that do not contain syntactic sugar.[10] This type contains each of the AST nodes that are directly constructed in an automaton (as well as one for basic blocks), and does not include any that should be removed by the desugarer.

The decision was made to implement the sugared and unsugared variants of these types separately in order to ensure correctness. By explicitly separating these types, we can ensure that `convert_sugar()` completely desugars its input, as it cannot possibly return a program containing AST nodes that need to be removed. We also ensure that any functions that should receive unsugared programs as argument cannot possibly accept unsugared programs as argument.

The actual desugaring procedure is simply a postorder traversal of the sugared AST, which constructs a `Vec<Stmt>` containing the result of applying each of the constructions discussed in sec. 3.4. Here, we will discuss the procedure for desugaring programs that include input formatting (e.g. fig. 3.9).

The first step of this process is to insert statements at the start of the program that reject the input string if it is not of the correct format. For example, a program with `alphabet = ['0', '1', '2']` and three corresponding input parameters should reject words that do not conform to the regular expression `0*1*2*`.

This can be done trivially by simply moving off of `lend`, reading each occurrence of the elements of the alphabet in the expected order, rejecting if the character at the end of this process is not `rend`, and returning to `lend`. For the same alphabet, the subprogram that does this is demonstrated in fig. 5.4.

```
// Move off of lend
move(1);

// Read the input variables in the expected order
while (read == '0') { move(1); }
while (read == '1') { move(1); }
while (read == '2') { move(1); }

// Reject if we don't reach rend
if (read != rend) { reject; }

// Go back to lend
while (read != lend) { move(-1); }

```

Figure 5.4: A program that rejects any input strings that do not match `0*1*2*`

We then need to desugar the instructions instructions such as `c = X`, `c += X` and `c -= X` for some input parameter `X`.

[10] Located in `twoc/src/parser/ast.rs`

The procedure for this is also relatively simple. For instructions of the form `c += X`, we simply move to `lend`, read along the input string until we read `parmap[X]` or `rend`, increment the counter for every `parmap[X]` in the input string, and return to `lend` (this is demonstrated in fig. 5.5). For instructions of the form `c -= X`, we follow the same process except for decrementing the counter for each `parmap[X]` we read. For instructions of the form `c = X`, we simply empty the counter before we increment for each input symbol.

```
// c += X;

// goto lend
while (read != lend) { move(-1); }

// read all the characters that aren't 1
while (read != '0' && read != 'rend') { move(1); }

// read and count each 1
while (read == '1') {
    move(1);
    c++;
}

// goto lend
while (read != lend) { move(-1); }

```

Figure 5.5: A program that increments the counter by `X`, where `parmap[X] = '0'`

5.3 Automaton construction

5.3.1 The `GenericAutom<>` structure

Before implementing any procedures to construct automata, we must first implement the data structure that will represent our automata.[11] Note that the `State` type is an alias for Rust's inbuilt 16-bit unsigned integer type (`u16`).

The `GenericAutom<>` type (fig. 5.6) is a struct containing the following attributes:

- The number of states in our automaton (of type `State`; recall that we use the convention $Q \subseteq \mathbb{N} \cup \{0\}, q_0 = 0$, therefore it is sufficient for us to store our set of states Q as a single integer)
- A list of accepting states (of type `Vec<State>`)
- A list of rejecting states (of type `Vec<State>`)
- The alphabet symbols (of type `std::collections::HashSet<char>` [Teae])
- The `decr_on_zero` flag (of type `bool`)
- The transition function

We store the transition function using an adjacency list, which consists of a map from the states in our automaton to the transitions off of it. This is represented using the `HashMap` struct contained within Rust's `std::collections` library (an automaton's adjacency list is therefore of type `HashMap<State, Vec<Transitions>>`) [Tead].

An adjacency list was chosen primarily because it allows for the transitions off of a state to be found in time $O(1)$ (in the average case). It also allows new transitions to be introduced to the automaton in constant time.

The `GenericAutom<>` type has several methods associated with it in order to facilitate construction and simulation. These include methods that:

[11] Located in `twoc/src/automaton/generic_autom.rs`

```rust
pub struct GenericAutom<Transition> {
    // Adjacency list of states to transitions off of states
    pub state_map : HashMap<State, Vec<Transition>>,

    // Counter to keep track of the number of states in the automaton
    pub state_total : State,

    // Vector to keep track of accepting states
    accepting : Vec<State>,

    // Vector to keep track of rejecting states
    rejecting : Vec<State>,

    // The tape alphabet (excluding the endmarkers)
    pub alpha : HashSet<char>,

    // Whether or not c-- is a valid instruction when c == 0
    pub decr_zero : bool,
}
```

Figure 5.6: The `GenericAutom<>` struct

- introduce new states (fig. 5.7)
- introduce new transitions
- find the transitions coming from a given state
- register states as accepting/rejecting
- add counter-emptying transitions to all accepting states
- remove all transitions from rejecting states
- check if a given state is accepting/rejecting

```rust
// Introduce a new state to the automaton
pub fn introduce(&mut self) -> State {
    // Add a new state to the adjacency list with no transitions
    self.state_map.insert(self.state_total, Vec::new());

    // Increment the total number of states
    self.state_total += 1;

    // Return the new state
    self.state_total-1
}
```

Figure 5.7: The `GenericAutom<>::introduce()` method

The `GenericAutom<>` type is parameterised over a generic type for transitions. Any transition type must implement `TransitionTrait` (traits in rust are equivalent to interfaces in other languages), which ensures that any transition types can represent basic-block and ϵ-transitions. This was originally done due to differences in the types of transitions for 2dc and 2nc; however, in the final version of `twoc`, these differences no longer exist. The `GenericAutom<>` type is still parameterised over a generic type to allow for modification and extension in future.

The `Transition` struct (fig. 5.8),[12] which represents transitions in 2dcs, contains the following attributes:

- The state the transition moves to (of type `State`)

[12] Located in `twoc/src/automaton/dautom.rs`

- How much the transition changes the counter by (of type `i32`)
- How far the transition moves the readhead (of type `i32`)
- The logical condition that must be satisfied for the automaton to take the transition, if one exists (of type `Option<Cond>`; the `Option` type is equivalent to the `Maybe` types found in many functional languages [Teab])

```rust
pub struct Transition {
    // The state we transition to
    pub goto : State,

    // Increment by
    pub incr_by : i32,

    // Move by
    pub move_by : i32,

    // Any conditionals required to take this transition if they exist
    pub condition : Option<Cond>,
}
```

Figure 5.8: The `Transition` struct in `twoc/parser/automaton/autom.rs`

5.3.2 Implementing the construction procedure

The construction functions[13] simply implement the procedure described in sec. 3.5. However, a few considerations had to be made when adapting it from its natural-language description to a Rust implementation.

The first issue that was identified when implementing this procedure was the fact that Rust does not support mutable global variables. This was solved by giving the recursive `construct_stmt()` function an argument of type `&mut State` (a mutable reference to a `State` variable). This allows us to modify the value of s as the automaton is constructed. `construct_stmt()` also accepts a mutable reference to the automaton that it builds.

Another issue concerns the procedure for constructing basic blocks. The simulation algorithms both assume that each transition changes the value of the counter by at most 1. This forces us to construct basic blocks (such as `move(i); c += j;`) as follows:

1. Introduce a new state q
2. Introduce a transition from s to q that moves the readhead `i` cells to the right
3. Update $s \leftarrow q$
4. For $x \in [1, |j|]$:
   (a) Introduce a new state p
   (b) If $j > 0$, introduce a transition from s to p that increments the counter by 1. Otherwise, introduce a transition from s to p that decrements the counter by 1.
   (c) Update $s \leftarrow p$

Another issue concerns the construction of transitions that do not increment/decrement the counter.

The algorithm that is used to simulate 2nc only works for automata whose transitions always change the value of the counter. Therefore, for 2nc, including some transition t from state p to state q which does not affect the counter must instead be done as follows:[14]

1. Introduce a new state r

[13] Located in `twoc/src/automaton/construction.rs` for 2nc and `twoc/src/automaton/determ_construction.rs` for 2dc
[14] This is implemented as the `add_transition_pop_push()` method in `twoc/src/automaton/construction.rs`

2. Introduce a transition from p to r that behaves the same as t while also incrementing the counter

3. Introduce a transition from r to q that decrements the counter

The final issue identified concerns the differences between the deterministic and nondeterministic `Transition` types.

In the deterministic case, the logical conditions that must be satisfied for a transition to be legal are stored directly as part of the transitions; therefore these logical conditions do not need to be constructed in the deterministic case. However, these constructions *do* need to be made in the nondeterministic case, as these transitions adhere much more closely to the formal definition of the transition function (sec. 1.4).

5.4 Automaton simulation

Once our program has been converted to an automaton, we can now find the result of executing the program on the given input string.

Recall that every 2dc/2nc can be represented by a 2dpda/2npda with a stack alphabet of size 1. This fact is extremely important to the design of `twoc`'s implementation, due to the fact that the efficient simulation of two-way pushdown automata is a very well-researched area.

In 1968, Aho, Hopcroft and Ullman presented an algorithm that could decide whether or not a word of length n is accepted by a given 2npda in time $O(n^3)$ [AHU68]. If the automaton is a 2dpda, this algorithm terminates in time $O(n^2)$. This result was slightly improved by Rytter in 1985, who provided a similar $O(n^3/\log n)$-time algorithm for 2npda [Ryt85]. Glück has also described algorithms for simulating 2npda in cubic time [Glü16; Glü13].

In 1970, Cook significantly improved this result for 2dpda, presenting an $O(n)$-time algorithm [Coo71].[15] Since then, other authors have presented alternative linear-time algorithms for 2dpda, including Jones [Jon77] and Glück [Glü13].

5.4.1 Choice of algorithms

We simulate 2dc using the procedure designed by Glück in 2013. This algorithm was chosen primarily due to its simplicity: the algorithm is concise, easily understood, and very well-explained in [Glü13] (see fig. 5.10). It also utilises a relatively loose definition for 2dpda, which provides us with more freedom when implementing our construction algorithm.

While still being relatively simple, Jones's procedure is still more complex than Glück's procedure; therefore, the decision was made not to use it.

We simulate 2nc them using the procedure designed by Rytter in 1985. This algorithm was chosen for similar reasons to Glück's procedure: it is also pleasantly concise, well-explained in [Ryt85], and also uses a loose definition for 2npda (although this is stricter than the one used by Glück).

The same cannot be said for the Aho-Hopcroft-Ullman procedure. This procedure, while similar to Rytter's, is far more complicated. It also makes the restriction that each transition must move the readhead by at most 1 cell to the left or right.

In [Glü13], Glück also provides a similar algorithm for simulating 2npdas in cubic time, an implementation of which is provided in the codebase.[16] This algorithm is not used by this implementation to simulate 2nc (unless the user explicitly chooses to) due to a key issue that Glück leaves unresolved.

When arguing the correctness of his algorithm, he identifies that, for 'left-recursive' automata (which *can* be constructed from `twoc` programs, including some programs featuring `while-choose` statements, such as the one in sec. C.3), his algorithm excludes potential computation paths and may reject strings that the automaton should accept. He posits that, as for one-way pushdown automata and context-free grammars, a procedure exists to convert left-recursive 2npdas to non-left-recursive 2npdas [Gre65]. However, he does not provide such a procedure in [Glü13], and designing such a procedure is well beyond the scope of this project.

Note that we do not apply the optimisations to Rytter's algorithm that reduce its runtime from $O(n^3)$ to $O(n^3/\log n)$. This decision was made due to the fact that, while these optimisations do reduce the algorithm's

[15] This paper has proven elusive, and the author has not been able to find evidence of its existence beyond references that appear in other publications. This is intriguing for a paper containing such a significant and applicable result (see [KMP77]).

[16] Located in `twoc/src/simulation/glueck_nondeterm.rs`

asymptotic runtime, they do not *significantly* reduce its runtime (note that $O(\log n)$ grows drastically slower than $O(n)$, let alone $O(n^3)$).

5.4.2 Snapshots and configurations

Both of the algorithms used for simulation manipulate configurations and snapshots of the automaton (recall sec. 1.4). To that end, we provide types for these, as well as functions to manipulate them.[17]

We introduce the following types:[18]

- `Config`: this is a struct storing snapshots of the automaton (fig. 5.9)
- `DeltaConfig`: this is a struct storing Δ-snapshots (see sec. 4.1) of the automaton (this is implemented as an alias for the `Config` type)
- `StrippedConfig`: this is a tuple corresponding to configurations (this is an alias for `(State, i32, bool)`, with the first element storing the state, the second storing the index of the readhead, and the third storing whether or not `c == 0`)

We also implement functions that

- find the legal transition(s) in an automaton from a given snapshot
- apply a transition to a snapshot
- convert snapshots to configurations
- compute the Δ-snapshot between two snapshots

```rust
pub struct Config {
    // The state the automaton is in
    pub state : State,

    // The index of the read head
    pub read : i32,

    // The value of the counter
    pub counter : i32,
}
```

Figure 5.9: The `Config` struct

5.4.3 Glück's algorithm [Glü13]

Glück's algorithm leverages the fact that 2dpdas can be constructed such that they completely empty their stacks upon termination. In our context, the procedure finds, from a given snapshot with counter value c, the next snapshot the automaton reaches before it decrements the counter to some value below c (we call this value the *terminator* of the original configuration). The algorithm recursively simulates the execution of the automaton in order to find the terminator of the starting configuration (if one exists), while remembering (or *memoizing*) the terminators of other intermediate configurations in order to avoid repeated computations (this optimisation allows the algorithm to terminate in time $O(n)$ for all 2dc).

[17] Located in twoc/src/simulation/config.rs.

[18] In hindsight, some of these are quite poorly named; this is due to the fact that the author implemented these before he specified the precise definitions he would use for snapshots and configurations.

```
procedure Sim(c: conf): conf;
  if defined(T[c]) then return T[c];   /* find shortcut */
  cond
    push(c):    d := Sim(follow(c,Sim(next(c))));
    op(c):      d := Sim(next(c));
    pop(c):     d := c;
    halt(c):    halt;
    accept(c):  accept;
  end;
  T[c] := d;                           /* memoize result */
  return d
```

Figure 5.10: Glück's procedure for simulating 2dpdas in linear time [Glü13]

5.4.3.1 Runtime analysis

Here, we re-analyse the runtime of this algorithm to take into account parameters of the automaton it simulates.

As can be seen in fig. 5.10, the procedure is defined in terms of two functions that accept configurations as argument and return new configurations (explained more thoroughly in [Glü13]):

- $next(c)$ returns the configuration the automaton enters in the next computation step after c, if the automaton either moves the readhead to a new position or pushes a symbol to the stack
- $follow(c, d)$ returns the configuration the automaton enters after it pops from the stack in configuration d, given that c was the most recent configuration that resulted in a push to the stack

Recall def. 1.5, and assume that finding $\delta(s, x, Z)$ takes time $O(1)$ for all $s \in Q$, $x \in \Sigma$ and $Z \in \Gamma$. Therefore, for configurations $c = (p, i, A)$ and $d = (q, j, B)$, finding $next(c)$ and $follow(c, d)$ must also take time $O(1)$, given that their values depend solely on δ.

In the worst case, the procedure will make a call to these operations $O(1)$ times for every possible configuration (i.e. once for every $c \in Q \times [1, n] \times \Gamma$). For each configuration, the procedure also queries a table (assume this can be done in time $O(1)$). The only other operations the procedure does for each configuration clearly take time $O(1)$ as well.

Also recall that every 2dc can be simulated by a 2dpda with $|\Gamma| = 1$. Therefore, for a 2dc, the procedure will run in time

$$O(|Q \times [1, n] \times \Gamma|) \cdot O(1) = O(|Q| \cdot |\Gamma| \cdot n) = O(|Q| \cdot n) \tag{5.1}$$

Note that the runtime of this algorithm scales with the size of the automaton; therefore, it is desirable for us to attempt to reduce the size of the constructed automata as much as possible.

5.4.3.2 Implementation

The code corresponding to Glück's algorithm is located in `twoc/src/simulation/glueck_array.rs`.

The algorithm is implemented as a method inside a struct (`GlueckSimulator`; fig. 5.12), which contains the following attributes (note that `Autom` is an alias for `GenericAutom<determ_autom::Transition>`, and is thus the type of 2dc):

- The automaton (of type `&'a Autom`[19])
- The input (of type `Input`, which is an alias for `Vec<Readable>`)
- The table mapping configurations to their precomputed terminators (of type `Vec<Option<DeltaConfig>>`; elements of this are set to `None` if the terminator has not yet been computed)
- A stack of configurations that have already been visited (of type `Vec<StrippedConfig>`)

[19] This is simply a pointer to a value of type `Autom`. `'a` is a lifetime parameter, which is required by Rust's borrow checker [Teaa].

```rust
pub fn glueck_procedure<'a>(autom : &'a Autom, input : &str) -> bool {
    // Convert the input into a list of Readables
    let readable_input = Readable::from_input_str(input);

    // Get the starting configuration of the automaton
    // Automaton always starts in state zero from lend with c=0
    let start_config = Config { state : 0, read : 0, counter : 0 };

    // Declare the GlueckSimulator object
    let mut simulator = GlueckSimulator::new(autom, readable_input);

    // Run the simulator to find the terminator of this config
    let final_config = simulator.simulate(start_config);

    // Return based on the final config
    match autom.check_if_halting(final_config.state) {
        None => false,
        Some(accept) => accept,
    }
}
```

Figure 5.11: The body of the `glueck_procedure()` function

We push the current configuration to the stack of visited configurations at the start of each recursive call, and we pop from it before we return. This allows us to detect infinite loops, which should result in the input string being rejected (we do this by returning the starting configuration which, for a program that infinitely loops and rejects, cannot be an accepting configuration).

Note that the original algorithm stores the memoized terminators in an array that is indexed by configurations, which can be accessed in time $O(1)$. However, declaring this array forces us to initially consider every possible configuration of our automaton (of which there are $|Q| \cdot n \cdot 2$).

An implementation utilising a hashmap (using the high-performance implementation provided in Rust's hashbrown library[20] as opposed to the default implementation in `std::collections`) was experimented with. In theory, this hashmap implementation still allows us to update/access memoized terminators in time roughly $O(1)$ (evidently, this is still greater than the time taken to index a vector), and allows us to avoid the expensive process of computing every single possible configuration at the start of computation.

Ultimately, despite these potential optimisations, it was discovered that this implementation was slightly slower than an array implementation (see sec. 5.7), and therefore it is not used in our implementation of `twoc`.

```rust
// Struct to hold variables for the Glueck procedure
struct GlueckSimulator<'a> {
    // Automaton being simulated
    autom : &'a Autom,

    // Input being simulated on
    input : Input,

    // Table that stores the previously computed terminators
    config_table : Vec<Option<DeltaConfig>>,

    // Number of configurations in total
    num_configs : usize,

    // Past configurations
    past_configs : Vec<StrippedConfig>,
}
```

Figure 5.12: The `GlueckSimulator` struct

[20] `https://github.com/rust-lang/hashbrown`

One issue with Glück's algorithm is that it makes $O(n)$ recursive calls to itself. In a theoretical context, this is fine; in practice, however, this leads to an interesting problem when running this algorithm on longer strings. On larger inputs, Glück's algorithm quickly causes stack overflow errors, as each recursive call requires the creation of a new stack frame by the operating system. An iterative version of this algorithm could have been developed in order to avoid this issue (or a completely different algorithm making fewer recursive calls could have been used [Coo71; Jon77]); instead, due to time and scope constraints, a simpler, less elegant solution was used.

When running Glück's algorithm on very long strings, we use Rust's `std::thread` library to spawn a thread with a stack of size $O(n)$, which we then use to run the algorithm (fig. 5.13) [Teac]. This approach is used extensively by the benchmarks located in `twoc/tests`, although it is not used when `twoc` is invoked on command-line arguments.

We also implement a variant of this algorithm that simulates 2ncs (although this isn't used by default as it can give incorrect results for some automata). This implementation was still included due to the fact that, in practice, it runs much faster than Rytter's algorithm, due to the fact that it often does not need to consider every possible configuration of the automaton throughout its execution, while Rytter's algorithm does.

```
    // autom and test_word are defined somewhere here

    // Run test with a massive stack of size O(n)
    thread::scope(|s| {
        thread::Builder::new().stack_size(0xFFFF * n) // Give the thread O(n) memory
        .spawn_scoped(s, || glueck_procedure(&autom, &test_word))
        .unwrap();
    });
```

Figure 5.13: How Glück's algorithm is invoked for very long inputs

5.4.4 Rytter's algorithm [Ryt85]

Rytter's algorithm operates by computing a relation $R \subseteq K^2$, where $K = Q \times [1, n] \times \Gamma$ is the set of all possible automaton configurations. R relates two configurations i and j if the automaton can reach j from i on the given input. Formally, the program in fig. 5.14 computes a set R that satisfies the following conditions:

- $(i, i) \in R$ for all $i \in K$ (this is enforced by 1 in fig. 5.14)
- If $(i, j) \in R$ then $\text{below}(i, j) \subseteq R$ (this is enforced by 2 in fig. 5.14)
- If $(i, k) \in R$ and $(k, j) \in R$ then $(i, j) \in R$ (this is enforced by 3 and 4 in fig. 5.14)

Rytter defines $(k, l) \in \text{below}(i, j)$ if and only if

- The automaton can go from k to i in one push move
- The automaton can go from j to l in one pop move
- k and l have the same stack top symbol

If $(s, m) \in R$, where s is the starting configuration and m is an accepting configuration, then the automaton can reach an accepting configuration, meaning it accepts the input string.

5.4.4.1 Runtime analysis

As before, we re-analyse this algorithm's runtime as a function of the automaton being simulated.

First, we note that each possible pair of configurations $(i, j) \in K^2$ can appear in the queue at most once. Therefore the while loop executes at most $|K^2|$ times.

```
begin
1:  Q:= R:= {(i, i)|i ∈ K};
while Q ≠ ∅ do
  begin
    (i, j):= delete(Q);      {the pair (i, j) is to be processed}
2:  for each (k, l) ∈ below(i, j) do
    if (k, l) ∉ R then insert ((k, l), R) and insert((k, l), Q);
3:  for each (k, i) ∈ R such that (k, j) ∉ R do
      insert((k, j), R) and insert((k, j), Q);
4:  for each (j, k) ∈ R such that (i, k) ∉ R do
      insert((i, k), R) and insert((i, k), Q)
  end;
if (0, m − 1) ∈ R then ACCEPT
end.
```

Figure 5.14: Rytter's procedure for simulating 2npdas in cubic time [Ryt85]

Next (as in [Ryt85]), we assume that each configuration is numbered from 0 to $|K| - 1$. This allows us to implement R as a boolean matrix (i.e. $R \in \{0,1\}^{|K| \times |K|}$), where $R[i,j] = 1$ if and only if $(i,j) \in R$. Note that R contains $|K|^2$ entries, and assume that reading/updating a cell in R takes time $O(1)$. Also recall that we can simulate a 2nc using a 2npda with $|\Gamma| = 1$.

We consider each of the sub-loops executed by the while loop (numbered 2-4 in fig. 5.14) and sum their total contribution to the runtime of the while loop.

We first consider the size of below(i,j) for fixed $i, j \in K$ where $i = (p, x, A)$ and $j = (q, y, B)$ (2 in fig. 5.14). Assume that the automaton is a directed graph with vertices Q and edges δ; also assume that it is stored as an adjacency list. In the worst case, every transition that moves to state p may result in an element of below(i,j), as well as every transition that moves from state q. Therefore,

$$|\text{below}(i,j)| = O(deg^-(p)) + O(deg^+(q)) \tag{5.2}$$

where $deg^-(\cdot)$ and $deg^+(\cdot)$ denote the out-degree and in-degree functions respectively.

In the worst case, we compute below(i,j) for every element of K^2. Summing $|\text{below}(i,j)|$ across all $(i,j) \in K^2$ yields

$$\sum_{(i,j) \in K^2} (O(deg^-(p)) + O(deg^+(q))) = \sum_{i \in K} \sum_{j \in K} O(deg^-(p)) + \sum_{i \in K} \sum_{j \in K} O(deg^+(q)) = X + Y \tag{5.3}$$

We rewrite X as follows:

$$X = \sum_{i \in K} \sum_{j \in K} O(deg^-(p)) = \sum_{j \in K} \sum_{i \in K} O(deg^-(p)) = |K| \cdot |[1,n] \times \Gamma| \cdot \sum_{p \in Q} O(deg^-(p)) \tag{5.4}$$

Via Euler's handshaking lemma [Wes01], $\sum_{p \in Q} O(deg^-(p)) = O(|\delta|)$.[21] Therefore,

$$X = |Q| \cdot n^2 \cdot |\Gamma|^2 \cdot O(|\delta|) = O(|Q| \cdot |\Gamma|^2 \cdot |\delta| \cdot n^2) \tag{5.5}$$

Via the same argument, we yield the same value for Y; therefore, $X + Y$ is also equal to eq. 5.5, giving us the total size of all the sets below(i,j).

For each element $(k,l) \in$ below(i,j), we add it to the queue and set $R[k,l] = 1$. These operations take time $O(1)$; therefore, the total runtime contributed by elements of below(i,j) for all $(i,j) \in K^2$ is

$$O(|Q| \cdot |\Gamma|^2 \cdot |\delta| \cdot n^2) \cdot O(1) = O(|Q| \cdot |\Gamma|^2 \cdot |\delta| \cdot n^2) = O(|Q| \cdot |\delta| \cdot n^2) \tag{5.6}$$

We can rewrite $|\delta|$ by observing that our construction algorithm does not produce automata with asymptotically more transitions than states. In this sense, we can assume that our automaton is *sparse* (i.e. $|\delta| = O(|Q|)$). This yields a total runtime of

$$O(|Q|^2 \cdot n^2) \tag{5.7}$$

[21] In this context, δ is the edge-set of our graph as opposed to a function.

We now consider the runtime of 3 in fig. 5.14 (note that this will also be equivalent to the runtime of 4; therefore the asymptotic runtime of 3 will be equal to the sum of the runtimes of 3 and 4).

Fix configurations $i, j \in K$. In the worst case, for all $k \in K$, $R[k, i] = 1$ and $R[k, j] = 0$. For each of these k, we set $R[k, j] = 1$ and add (k, j) to the queue. As before, these operations take time $O(1)$; therefore, the runtime of 3 for fixed configurations i and j is

$$O(1) \cdot |K| = O(|Q| \cdot |\Gamma| \cdot n) \tag{5.8}$$

In the worst case, we run this operation for each $(i, j) \in K^2$. This yields a total running time of

$$|K|^2 \cdot O(|Q| \cdot |\Gamma| \cdot n) = O(|Q|^3 \cdot |\Gamma|^3 \cdot n^3) = O(|Q|^3 \cdot n^3) \tag{5.9}$$

Summing eq. 5.7 and eq. 5.9 yields the following:

$$O(|Q|^2 \cdot n^2) + O(|Q|^3 \cdot n^3) \tag{5.10}$$

Clearly, eq. 5.9 dominates eq. 5.7, and thus the algorithm has a total runtime of

$$O(|Q|^3 \cdot n^3) \tag{5.11}$$

Note that this grows even more quickly with $|Q|$ than eq. 5.1.

5.4.4.2 Implementation

The code corresponding to Rytter's algorithm is located in `twoc/src/simulation/rytter.rs`.

As with Glück's algorithm, we utilise a struct (`RytterSimulator`; fig. 5.15) to store the global variables. These are as follows:

- The automaton (of type `&'a Autom`[22])
- The input (of type `Input`, which is an alias for `Vec<Readable>`)
- The size of the input (of type `StrIndex`; this is an alias for the `i32` type)
- All the possible configurations of the automaton (of type `Vec<StrippedConfig>`
- How many configurations there are in total (of type `usize`)
- The queue of pairs of configurations to be processed (of type `VecDeque<(usize, usize)>` [Teaf])
- The matrix of configurations (of type `(HashMap<usize, Vec<usize>>, HashMap<usize, Vec<usize>>)`; its type will be discussed later)
- The inverse of the automaton's adjacency list (i.e. if state p is in state q's adjacency list, then q will be in p's inverse adjacency list; of type `HashMap<State, Vec<Transition>>`)

Here, as with the array implementation of Glück's algorithm, we encode our configurations as integers (these are all of type `usize`). The `configs` vector is used to find a configuration given its index (the config with index `x` is simply `configs[x]`).

The `get_index()` method is used to convert from configurations to their indices. This is a fast arithmetic operation that leverages the order that configurations are inserted into `configs` when the `RytterSimulator` object is initialised.

The first challenge of this implementation was efficiently computing the set below(i, j) (this is computed by the `RytterSimulator::below()` method).

[22] This is simply a pointer to a value of type `Autom`. `'a` is a lifetime parameter, which is required by Rust's borrow checker [Teaa].

```rust
struct RytterSimulator<'a> {
    // The automaton being simulated
    autom : &'a Autom,

    // The input being simulated on
    input : Input,

    // The size of the input
    n : StrIndex,

    // All the possible configurations of autom on input
    configs : Vec<StrippedConfig>,

    // How many configs there are in total
    num_configs : usize,

    // The queue of config pairs being considered
    queue : VecDeque<(usize, usize)>,

    // The boolean matrix R (represented as a pair of hashmaps)
    conf_matrix : (HashMap<usize, Vec<usize>>, HashMap<usize, Vec<usize>>),

    // The inverse of the automaton's adjacency list
    inverse_state_map : HashMap<State, Vec<Transition>>
}
```

Figure 5.15: The `RytterSimulator` struct

```rust
fn get_index(&self, conf : StrippedConfig) -> usize {
    // Get the state, index and counter
    let (state, index, counter_zero) = conf;

    // Compute the relevant offsets based on these values
    let counter_offset = if counter_zero {0} else {1} as usize;
    let index_offset = (index * 2) as usize;
    let state_offset = ((state as i32) * self.n * 2) as usize;

    // The index is the sum of the offsets
    let index = counter_offset + index_offset + state_offset;

    // Panic if this config is too big
    if index > self.num_configs {
        panic!("Config {:?} doesn't exist!", conf);
    }

    // Return
    index
}
```

Figure 5.16: The `RytterSimulator::get_index()` method

Computing all the configurations $l \in K$ such that the automaton can pop from j to l is fairly trivial (we simply use the functions in `twoc/src/simulation/config.rs`). However, computing all the configurations $k \in K$ that the automaton can push while moving to configuration i is not as simple.

In order to compute these configurations quickly, we compute the inverse of the automaton's adjacency list when initialising the simulator (fig. 5.17). This allows us to find the transitions that move onto configuration i in time roughly $O(1)$ (as opposed to time $O(|\delta|)$) without this pre-processing step, as we would have to search through every transition in the graph).

One key difference between this implementation and [Ryt85] is in how we represent the set R.[23]. Instead of representing R as a boolean matrix, we represent it as a pair of hashbrown hashmaps (A, B), each of type

[23] An alternative implementation more adherent to [Ryt85] can be found in `twoc/src/simulation/rytter_matrix.rs`, which is included to evaluate these optimisations. Note that this implementation relies on the array2d library, whic can be found at `https://docs.rs/array2d/latest/array2d/`.

```rust
    // Initialise the inverse state map with the correct keys
    let mut inverse_state_map = HashMap::new();
    for state in 0..autom.state_total {
        inverse_state_map.insert(state, Vec::new());
    }

    // Populate the inverse state map
    for from_state in 0..autom.state_total {
        for trans in autom.get_transitions(from_state) {
            let to_state = trans.goto;

            // Create the inverse transition
            let inverse_trans = Transition {
                // swap to_state and from_state
                goto : from_state,

                // leave everything else alone
                incr_by   : trans.incr_by,
                move_by   : trans.move_by,
                condition : trans.condition,
            };

            // Put this transition into the map
            let search_map = inverse_state_map.get_mut(&to_state).unwrap();
            search_map.push(inverse_trans);
        }
    }
```

Figure 5.17: The code that computes `inverse_state_map` in `RytterSimulator::new()`

`HashMap<usize, Vec<usize>>`. Throughout execution, we maintain that $(i, j) \in R$ if and only if $j \in A[i]$ and $i \in B[j]$.

Storing our matrix like this allows us to significantly optimise operations 3 and 4 of fig. 5.14. Without this optimisation, finding all of the configurations $k \in K$ such that $(k, i) \in R$ requires that we search through each configuration k and read the corresponding value of $R[k, i]$, which takes time $O(|K|)$. With this optimisation, the same computation takes much less time, as we simply query $B[i]$ to find all of the configurations k that satisfy this condition (which takes time $O(1)$ on average). The same applies to operation 4, but instead we query $A[j]$.

Note that this optimisation does not improve the asymptotic runtime of our algorithm; in the worst case, each $A[i]$ could still be of size $O(|K|)$. Consequently, in practice, this optimisation may not improve the algorithm's runtime (and often worsens it in practice), especially on programs with smaller numbers of states in the constructed automaton (see sec. 5.7).

Also note that this optimisation may decrease the amount of memory the algorithm uses. When stored as a boolean matrix, R is stored in $\Theta(|K|)$ bits, while this hashmap representation uses $O(|K|)$ bits.

5.5 Command line interface

Most of the code corresponding to the implementation of `twoc`'s command line interface can be found in `twoc/src/main.rs`.

Our command line interface receives the following values from the user:

- The name of the file containing the `twoc` program
- The input string (if this value is set to `//EMPTY//`, the input string is set to ϵ)
- An optional `--verbose` flag (if this is present, the user can see the outputs of each transformation applied to their input program)
- An optional `--use-glueck-nondeterm` flag (if this is present, Glück's algorithm (in [Glü13]) will be used to execute nondeterministic programs instead of Rytter's algorithm)
- An optional `--use-rytter-matrix` flag (if this is present, Rytter's algorithm will represent the set R using a boolean matrix, as opposed to a pair of hashmaps)

The command line interface makes use of Rust's Clap library.[24] This library allows programmers to specify the command line arguments for their program inside a struct (called `CliArgs` in our case), and automatically implements a function that parses them.

We consider some examples of commands used to invoke `twoc` below:[25]

- `twoc --file example.twoc --word 001122` executes the program in `example.twoc` on the string `"001122"`
- `twoc -f example.twoc -w 001122` also executes `example.twoc` on `"001122"`
- `twoc --file example.twoc --word //EMPTY//` executes `example.twoc` on ϵ
- `twoc --file example.twoc --word 001122 --verbose` executes `example.twoc` on `"001122"`, while also displaying the program's AST, the transformations applied to it, and the constructed automaton
- `twoc -f example.twoc -w 001122 -v` also verbosely executes the `example.twoc` on `"001122"`
- `twoc -f example.twoc -w 001122 --use-glueck-nondeterm` executes `example.twoc` on `"001122"`. If the program is nondeterministic, it uses Glück's algorithm instead of Rytter's.
- `twoc -f example.twoc -w 001122 --use-rytter-matrix` executes `example.twoc` on `"001122"`. If the program is nondeterministic, it uses Rytter's algorithm with a matrix as opposed to a hashmap.

Once `twoc` has been invoked on the command line, it simply outputs whether or not the input string was accepted or not (e.g. `"001122" is rejected`), unless the user used the `--verbose` flag. In this case, the following are also outputted:

- The AST with syntactic sugar
- The AST without syntactic sugar
- The constructed automaton

A simple preorder traversal can be used to convert ASTs to their string representations,[26] while the automaton can be printed by simply displaying each of its states and the transitions off of them.[27]

5.6 Testing

Before we consider how this implementation was tested, we note the following:

- The first two stages (lexing/parsing and desugaring) are relatively trivial.
- The construction procedure is proven to be correct (see chapter 4).
- Verifying that an automaton is equivalent to a given program is impossible without being able to simulate the automaton.
- Specifying automata through the use of a programming language is can be done much more quickly than designing their finite state control graphs and converting these to adjacency lists.
- Almost all of the bugs identified during implementation were caused by errors in the simulation procedures.

Due to these considerations (as well as due to the time constraints of the project) writing comprehensive unit tests of the first three subroutines was deemed to be unnecessary, and these parts of the implementation were instead tested informally (mostly by invoking `twoc` through the command line). Instead, formal tests have been conducted on the entire implementation, which we discuss here.

All of the code that executes test cases can be found in `twoc/tests`.

[24] `https://docs.rs/clap/latest/clap/`

[25] Note that you will need to change `twoc` to `cargo run --` when invoking `twoc` using Cargo

[26] See the `Stmt::print()` methods in `twoc/src/parser/sugar/ast.rs` and `twoc/src/parser/ast.rs`.

[27] See the `GenericAutom<>::print()` method in `twoc/src/automaton/generic_autom.rs`.

5.6.1 Deterministic test cases

All of the test programs used can be found in `twoc/twocprogs/determ`. These test programs were designed to test all the different deterministic features of the `twoc` language.

In total, 70 test cases were run across 7 different test programs using the hashmap and array implementations of Glück's algorithm, totalling $70 \times 2 = 140$ tests. The implementation returned correct results for all of these.

5.6.2 Nondeterministic test cases

All of the nondeterministic test programs used can be found in `twoc/twocprogs/nondeterm`. These test programs were designed to test how `twoc`'s nondeterministic features interact with its deterministic ones.

In total, 34 test cases were run across 4 different test programs using the hashmap and matrix implementations of Rytter's algorithm, totalling $34 \times 2 = 68$ tests. Additionally, the previous 59 deterministic tests were also run on both algorithms, totalling $68 + 70 \times 2 = 208$ tests. The implementation returned correct results for all of these.

5.7 Benchmarking

In order to evaluate the efficiency of the implementation, we discuss several benchmarks applied to the simulation algorithms. All benchmarks were run on the machines in the Department's computer labs, although these have varying specifications.

The results of our benchmarks have been plotted with the help of a short Python script,[28] which receives a `.csv` file containing the benchmark results as input. The ith entry of one of these files takes the form $(n_i, t(n_i), \Delta t(n_i))$, where n_i is the size of the input string for that test, $t(n_i)$ is the time it took to simulate it, and $\Delta t(n_i) = t(n_i) - t(n_{i-1})$ is an empirical measure of $t'(n)$ (the derivative of t with respect to n).

This script reads these values from the result of the benchmark, filters excessive values for Δt, and plots the values of $t(n)$ and $\Delta t(n)$ using the Matplotlib library. It also computes a polynomial regression (either linear, quadratic or cubic) of these results and evaluates the corresponding r^2 values using the NumPy and scikit-learn libraries.[29]

5.7.1 Deterministic programs

The benchmarks of deterministic algorithms were run on the program found in sec. C.4. This program was chosen because a naive simulation of it should execute in time $O(n^2)$ (this is demonstrated in fig. 5.18; note the much higher values of $t(n)$, especially for $n \geq 1000$). Despite this, our implementation should execute this program in time $O(n)$; determining whether or not this is the case allows us to evaluate the efficiency of our implementation.

All benchmarks of deterministic programs were done on a machine containing an Intel i5-7500 CPU[30] and 32GB of memory.

We first discuss the benchmark of the array implementation (fig. 5.19). The test program was executed on strings of n 0s for each $n \in \{0, 30, 60, 90, \cdots, 12000\}$.

This result closely matches a linear regression on it (with $r^2 \approx 0.938$), showing that, in practice, this simulation algorithm runs efficiently for input strings of a reasonable size. However inspection of the graphs of $t(n)$ and $t'(n)$ reveal that this procedure seems to run in time $\omega(n)$ (i.e. super-linear time).

The reason for this growth requires rigorous investigation that the time constraints of this project did not allow; however it may be due to the solution used to deal with stack overflow errors (fig. 5.13). The operating system may be struggling to allocate larger and larger amounts of memory to the thread running the simulation algorithm as n increases, thus resulting in an overhead [Wei12].

We also briefly discuss the benchmark for the hashmap implementation on the same inputs (fig. 5.20). This curve is of an extremely similar shape to fig. 5.19 ($r^2 \approx 0.937$ in this case; note how similar this is to the previous case). However, note that $t_{array}(12000) \approx 5$ seconds while $t_{hashmap}(12000) \approx 6.5$ seconds;

[28] Located in `twoc/tests/bench_results/result_plotter.py`

[29] `https://matplotlib.org/`, `https://numpy.org/` and `https://scikit-learn.org/stable/` respectively

[30] Specifications for this model can be found at `https://www.intel.co.uk/`

it appears that $t_{hashmap}$ has been scaled up by some constant factor (i.e. $t_{hashmap}(n) = k \cdot t_{array}(n)$ for some $k \geq 1$; for these benchmarks, $k \approx 1.3$). This multiplicative increase is almost certainly due to the `hashbrown::HashMap` data structure being slower to access than the `std::Vec` data structure by a constant factor. These two implementations are compared in fig. 5.21.

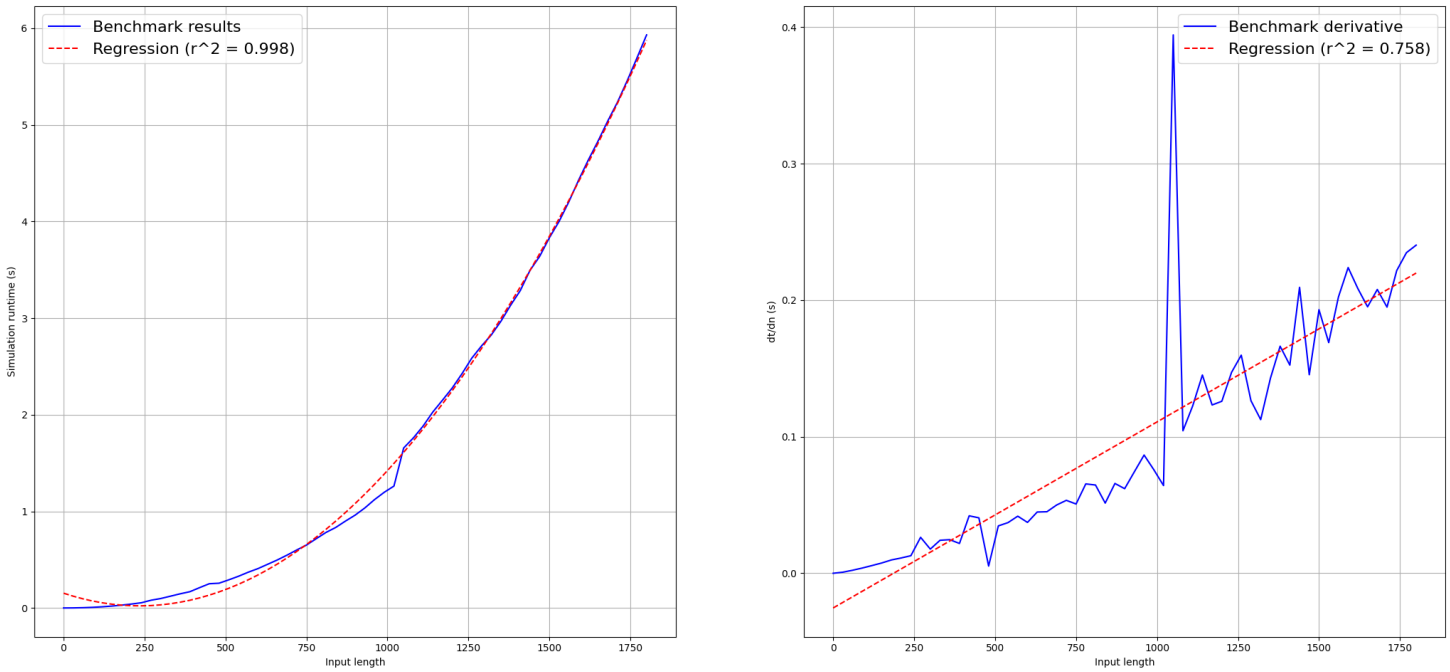


Figure 5.18: The results of a benchmark of a naive simulation using a quadratic regression

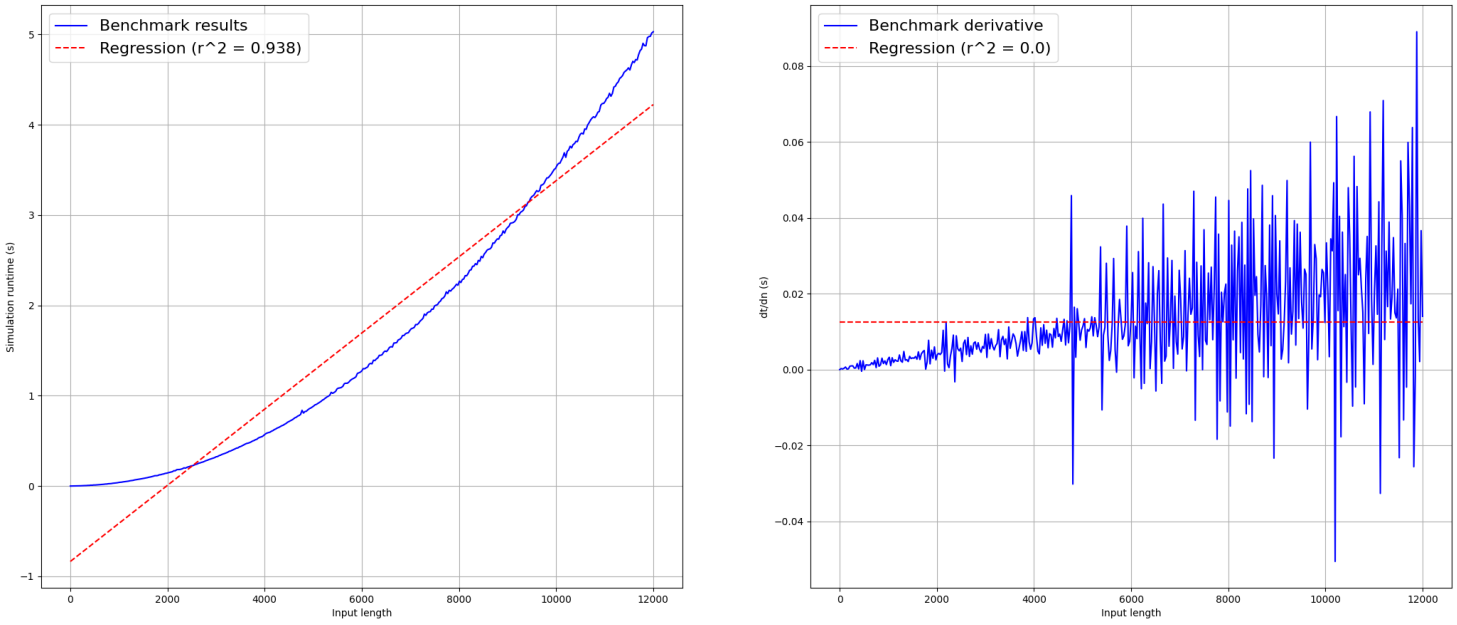


Figure 5.19: The results of a benchmark of Glück's algorithm using an array to store terminators

5.7.2 Nondeterministic programs

For nondeterministic programs, we also run benchmarks on the program in sec. C.4. This program was executed on strings of n 0s for each $n \in [0, 100]$. Note that, for the 2nc constructed from this program, $|Q|$

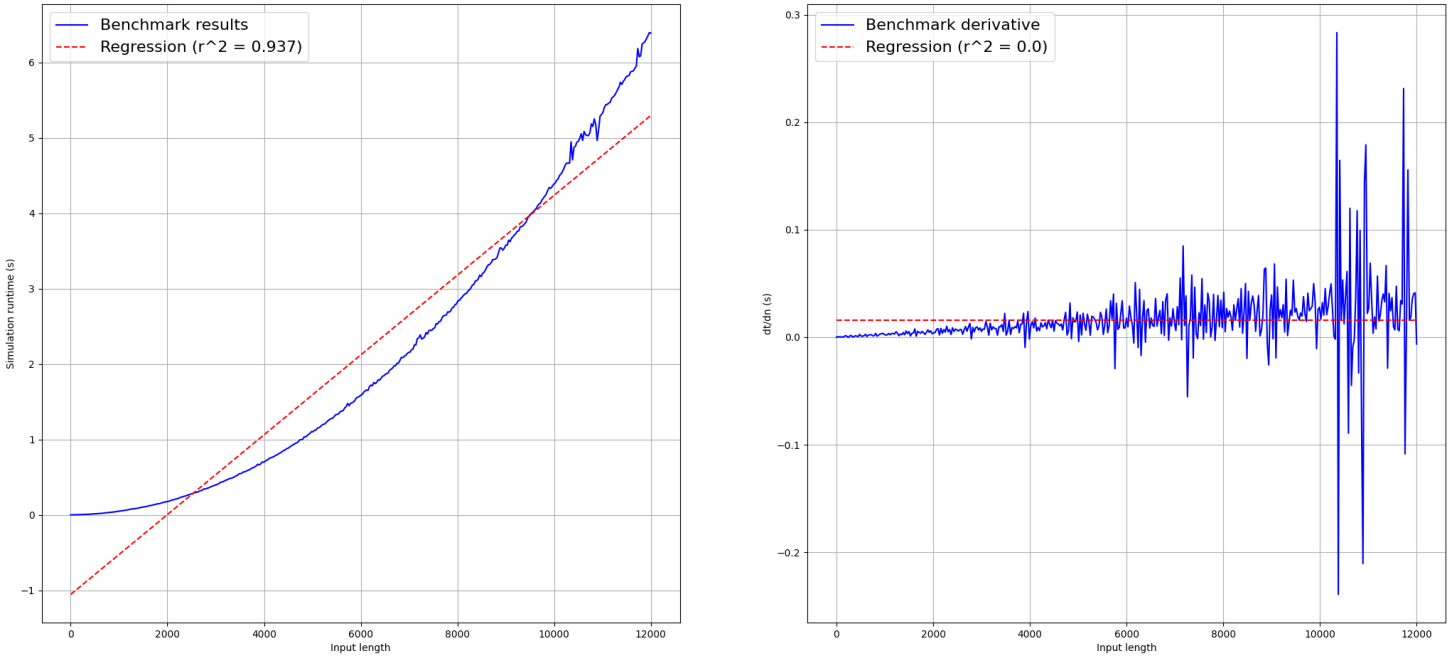


Figure 5.20: The results of a benchmark of Glück's algorithm using a hashmap to store terminators

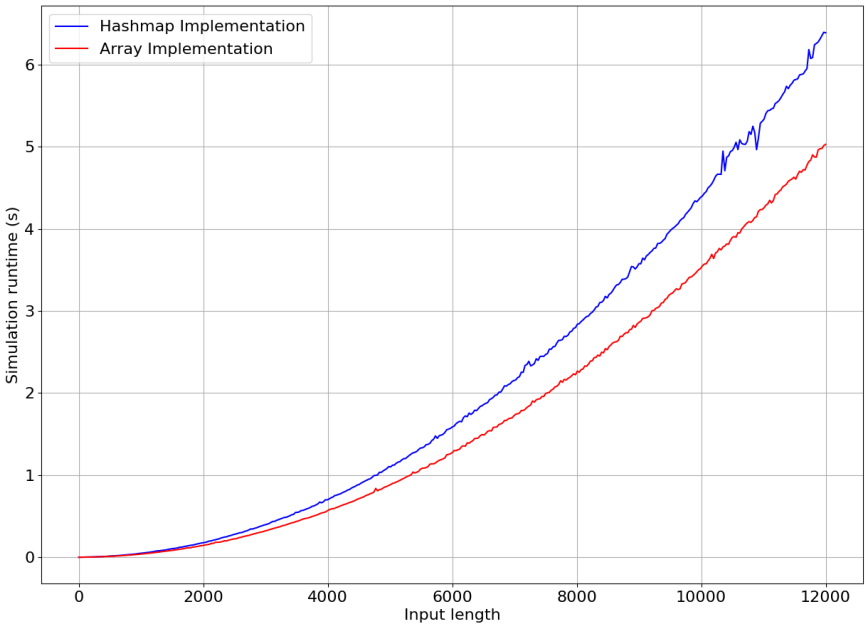


Figure 5.21: A comparison of the hashmap and array implementations of Glück's algorithm

is relatively small.

This set of benchmarks were done on a machine containing an Intel i5-6500 CPU[31] and 16GB of memory.

We first analyse the results of this benchmark on Rytter's algorithm implemented with a hashmap (fig. 5.22). Note that, unlike before, this graph very closely matches the expected regression ($r^2 = 0.999$), which confirms that our implementation almost certainly runs in time $O(n^3)$. The same can also be said of the

[31]Specifications for this model can be found at `https://www.intel.co.uk/`

alternative matrix implementation (fig. 5.23).

Also note that, despite the fact that our hashmap implementation is (in theory) slightly faster than the matrix implementation, the difference in access times between hashmaps and arrays has, again, resulted in the matrix algorithm being much faster as n grows high (especially on this example, where $|Q|$ is relatively small). In fig. 5.24, we compare the results of these two algorithms, as well as comparing them to Glück's algorithm. This is done in order to justify why we allow the programmer to opt into using it despite its tendency to produce false negatives.

In order to demonstrate why we use the hashmap implementation by default, we consider two benchmarks of the program in sec. C.5 for $n \in [0, 77]$ (fig. 5.25). Note that, for the automaton constructed from this program, $|Q|$ is much higher than in the previous example.

In this comparison, note that, for $n < 70$, our hashmap implementation *does* slightly outperform the matrix implementation. We also note that, in practice, users are very likely to write programs that result in very large values of $|Q|$, while they are also likely to execute these programs on relatively small values of n. This, as well as fig. 5.25, justify why we use the hashmap implementation by default.

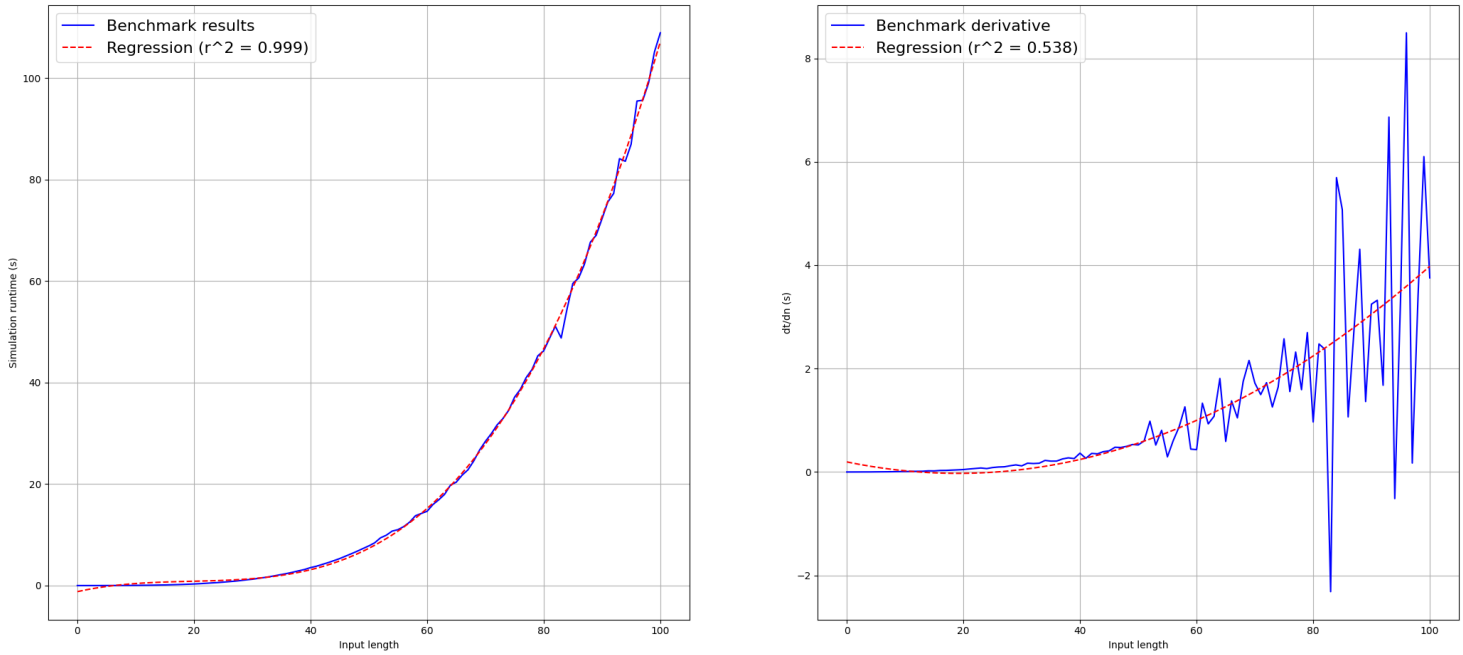


Figure 5.22: The results of a benchmark of Rytter's algorithm using a pair of hashmaps

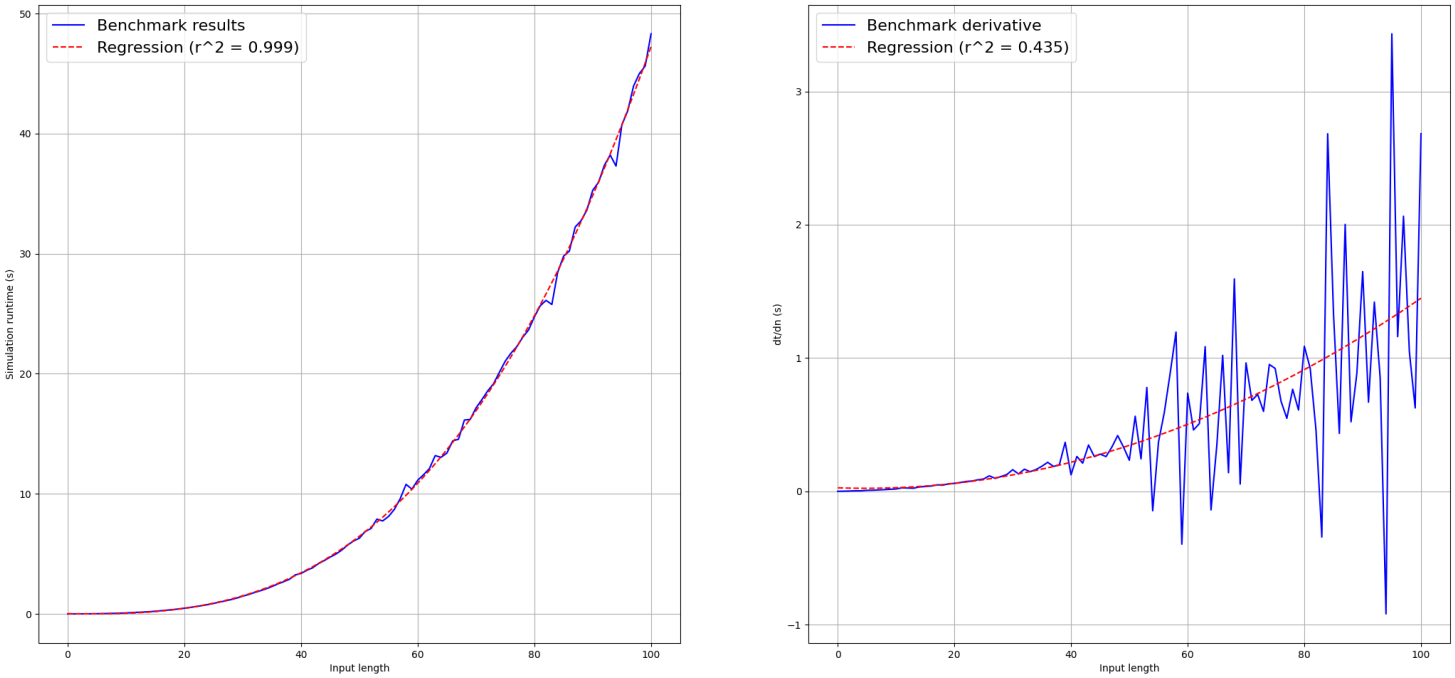


Figure 5.23: The results of a benchmark of Rytter's algorithm using a boolean matrix

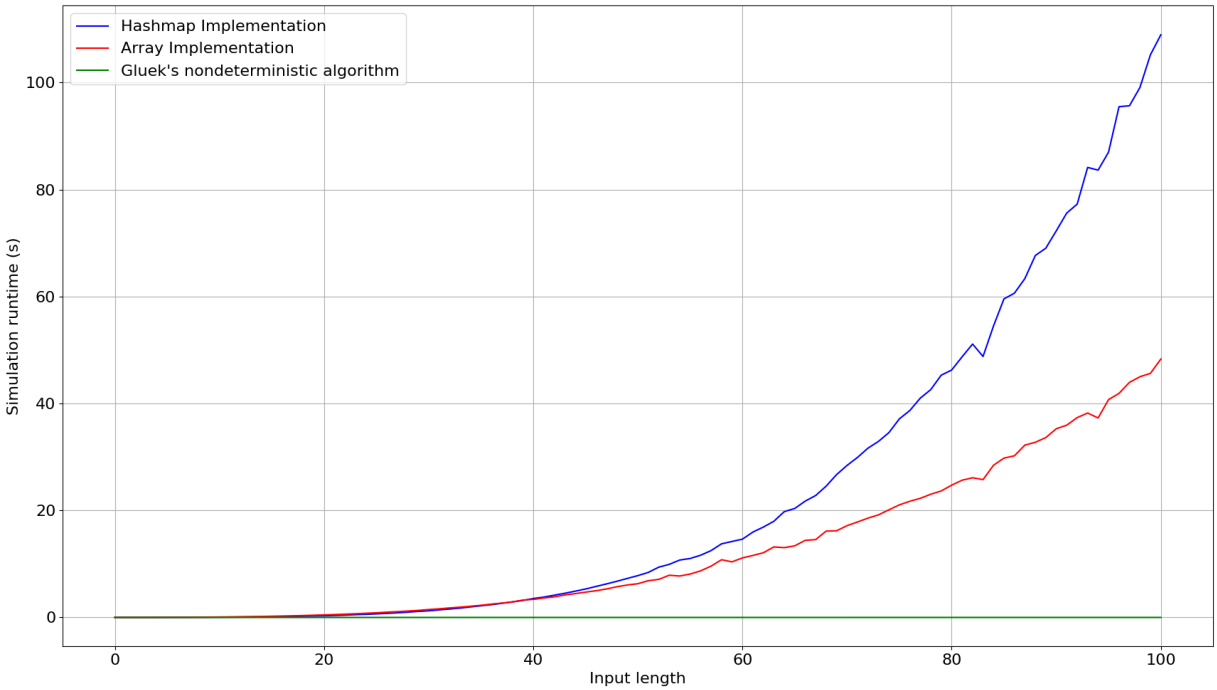


Figure 5.24: A comparison of the results of a benchmark on Glück's and Rytter's algorithms

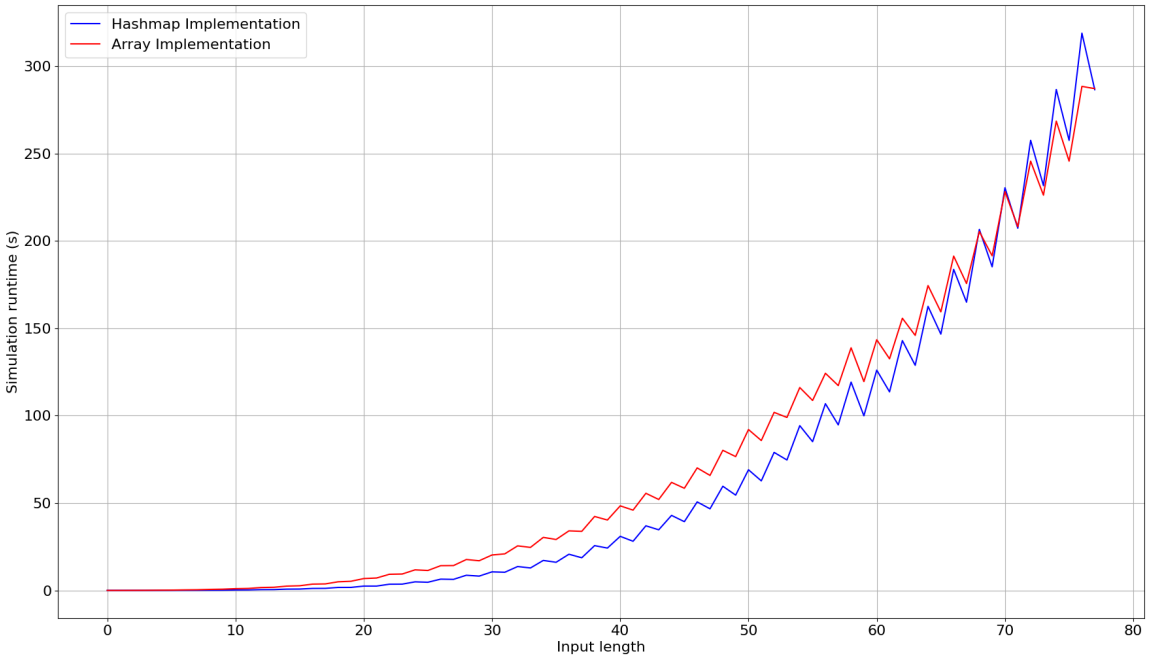


Figure 5.25: A comparison of the results of a benchmark on both implementations of Rytter's algorithm

Chapter 6

The proof system

In this chapter, we define axioms and rules that can be used to prove theorems about `twoc` programs. We also demonstrate it by proving theorems on small example programs. Note that, in the following, $\wedge$ denotes logical conjunction, $\equiv$ denotes logical equivalence and $\Rightarrow$ denotes logical implication.

Due to time constraints, the author was unable to provide a complete implementation of this proof system; however, it is expected that such a system would not be enormously difficult to embed in an existing proof assistant (e.g. Coq, Agda, HOL, etc.), as in [Pie+20]. It would also not be extremely difficult to embed this system in a programming language (as was originally intended for this project), similarly to the Haskell implementation described in [Sit21].

6.1 Adapting Hoare logic to `twoc`

We first consider the axioms we will need to reason about deterministic `twoc` programs.

The framework typically used to reason about imperative programs is that of Hoare logic [Hoa69]. Hoare logic consists of assertions that relate programs to logical statements. These assertions are of the form $\{P\}\texttt{prog}\{Q\}$, where `prog` is a sequence of `twoc` statements, and P and Q are logical formulae, referred to as the precondition and postcondition respectively. This assertion states that, if P is true prior to the execution of `prog`, then Q will be true afterwards. An assertion of the form $\{true\}\texttt{prog}\{Q\}$ asserts that Q is true when executing `prog` from any precondition.

6.1.1 Logical formulae

We first discuss the logical formulae that we can make reasonable assertions regarding `twoc` programs. Typically, for some program `prog`, we will want to prove formulae corresponding to statements such as:

- `prog` accepts on inputs of the form $w = 0^k1^k$ for all $k \geq 0$ (note that w may be subject to input formatting (see sec. 3.4.2), and may instead be represented as a series of integers $X, Y, Z \in \mathbb{N} \cup \{0\}$)
- `prog` rejects all inputs of the form $w = a^{2^k+1}$ for all $k \geq 0$
- `prog` halts on all inputs

In order to represent these statements, we will need to consider logical expressions over the following variables:

- $w \in \Sigma^*$ is the input string; assume this is of length n and that it is indexed by the values $[0, n-1]$ (i.e. if $w = abcde$ then $w_2 = c$)
- $c \in \mathbb{N} \cup \{0\}$ is the value of the counter
- $i \in [0, n-1]$ is the position of the readhead on the tape

We also need the ability to assert that a program accepts or rejects. For some program `prog`, we will assert that $\{P\}\texttt{prog}\{accepts\}$ if and only if `prog` halts and accepts from precondition P, and we will likewise assert that $\{P\}\texttt{prog}\{rejects\}$ if `prog` halts and rejects from P.

Our logical formulae will consist of assertions about the domains of these variables, or of the assertions that their corresponding programs accept or reject. More formally (assuming we do not introduce any other variables in our proofs), a logical formula P describes a domain

$$D(P) \in \mathcal{P}(\Sigma^* \times [0, n-1] \times \mathbb{N})) \cup \{accepts, rejects\} \tag{6.1}$$

where $D(P) = (W, C, I)$ if P asserts that $w \in W$, $c \in C$ and $i \in I$; and $D(P) \in \{accepts, rejects\}$ if P asserts that the corresponding program accepts or rejects.

6.1.2 Initial assumptions

Before discussing the axioms that will allow us to deduce theorems about programs, we first discuss the statements that we can assume to be true at the start of the execution of the program.

Firstly, we discuss our **static** assumptions: these are statements that we assume to be true throughout the execution of the automaton on a given input w. For any proof, our static assumptions are

- $c \geq 0$ and $i \in [0, n-1]$ (these follow from the domains given above and from the definition of 2dc/2nc)
- any assumptions we wish to make about the structure of w

Note that all `twoc` programs stardw with $c = 0$ and $i = 0$. We thus define $init \equiv c = 0 \wedge i = 0$. An assertion of some postcondition Q following the execution of an entire program `prog` will thus take the form of $\{init\}\texttt{prog}\{Q\}$.

6.1.3 Axioms and rules for deterministic programs

[Hoa69] introduces deduction rules sufficient to derive these assertions for programs consisting of variable assignments, sequential computations, and if and while statements. The addition of axioms dealing with `accept;` and `reject;` statements will allow these axioms to be sufficiently powerful to prove useful statements about deterministic `twoc` programs, which we describe here.

The following rules are written as natural deduction-style implications. These are of the form

$$\frac{A_1 \quad A_2 \quad \cdots \quad A_m}{B} example \tag{6.2}$$

which denotes that we can deduce the succedent B from the conjunction of the antecedents $\bigwedge_{j=1}^{m} A_j$, and that we call this rule *example*. A rule of this form is also an *axiom* if it contains no antecedents.

Also note that $P[x/y]$ is the result of substituting every free instance of y in P with x.[1]

The following axiom asserts that an empty program does nothing (assume that `skip` is a program containing no statements):

$$\frac{}{\{P\}\texttt{skip}\{P\}} skip \tag{6.3}$$

The following axioms encode the semantics of `accept` and `reject` statements:

$$\frac{}{\{P\}\texttt{accept;}\{accepts\}} accept \tag{6.4}$$

$$\frac{}{\{P\}\texttt{reject;}\{rejects\}} reject \tag{6.5}$$

The following axioms encode the semantics of assignments made to the counter (here, $X \in \mathbb{N} \cup \{0\}$ is some constant integer; if X is an input variable, append $[i/0]$ to each of the preconditions of these rules):

$$\frac{}{\{P[X/c]\}\texttt{c = X;}\{P\}} assign\_counter \tag{6.6}$$

$$\frac{}{\{P[c + X/c]\}\texttt{c += X;}\{P\}} increment\_counter \tag{6.7}$$

[1] The author suggests that you read $P[x/y]$ as P with x *instead of* y.

These first two axioms are relatively straightforward; however, designing rules to encode the semantics of the counter being decremented are slightly more complex:

$$\frac{}{\{P[\max\{0, c - X\}/c]\}\texttt{c -= X;}\{P\}}\textit{decrement\_counter\_a} \tag{6.8}$$

$$\frac{P[c - X/c] \Rightarrow c \geq X}{\{P[c - X/c]\}\texttt{c -= X;}\{P\}}\textit{decrement\_counter\_b} \tag{6.9}$$

$$\frac{P[c - X/c] \Rightarrow c < X}{\{P\}\texttt{c -= X;}\{\textit{rejects}\}}\textit{decrement\_counter\_c} \tag{6.10}$$

Note that we do not use the *decrement_counter_a* axiom in conjunction with the *decrement_counter_b* and *decrement_counter_c* axioms. If the `decr_on_zero` flag has been set to `true`, then we use *decrement_counter_a*; otherwise, we use the other two rules (for the sake of simplicity, all the programs we prove theorems about will have this flag set to true).

The following axioms encode the semantics of `move();` and `goto();` instructions (recall that $\text{clamp}(x, y, z) = \max\{x, \min\{y, z\}\}$):

$$\frac{}{\{P[\text{clamp}(0, i + k, n - 1)/i]\}\texttt{move(k);}\{P\}}\textit{move\_readhead} \tag{6.11}$$

$$\frac{}{\{P[0/i]\}\texttt{goto(lend);}\{P\}}\textit{goto\_lend} \tag{6.12}$$

$$\frac{}{\{P[n - 1/i]\}\texttt{goto(rend);}\{P\}}\textit{goto\_rend} \tag{6.13}$$

The following rules allow for the composition of sequential instructions (the first of these is taken directly from [Hoa69]; the others are included to propagate `accept;` and `reject;` statements through the program):

$$\frac{\{P\}\texttt{prog1}\{Q\} \quad \{Q\}\texttt{prog2}\{R\}}{\{P\}\texttt{prog1;prog2}\{R\}}\textit{sequence} \tag{6.14}$$

$$\frac{\{P\}\texttt{prog1}\{\textit{accepts}\} \quad \{Q\}\texttt{prog2}\{R\}}{\{P\}\texttt{prog1;prog2}\{\textit{accepts}\}}\textit{sequence\_accept} \tag{6.15}$$

$$\frac{\{P\}\texttt{prog1}\{\textit{rejects}\} \quad \{Q\}\texttt{prog2}\{R\}}{\{P\}\texttt{prog1;prog2}\{\textit{rejects}\}}\textit{sequence\_reject} \tag{6.16}$$

Here, `prog1;prog2` denotes the program obtained by appending `prog2` to the end of `prog1`.

The following rule allows for implications in the precondition and postcondition to be used to form new valid Hoare triples:

$$\frac{P' \Rightarrow P \quad \{P\}\texttt{prog}\{Q\} \quad Q \Rightarrow Q'}{\{P'\}\texttt{prog}\{Q'\}}\textit{consequence} \tag{6.17}$$

The following rules encode the semantics of `if-else` statements (these were not originally defined in [Hoa69], but are now a standard feature of Hoare logic [AO19; Pie+20]):

$$\frac{\{P \wedge B\}\texttt{prog1}\{Q\} \quad \{P \wedge \neg B\}\texttt{prog2}\{Q\}}{\{P\}\texttt{if (B) \{prog1\} else \{prog2\}}\{Q\}}\textit{if} \tag{6.18}$$

$$\frac{P \Rightarrow B \quad \{P\}\texttt{prog1}\{Q\}}{\{P\}\texttt{if (B) \{prog1\} else \{prog2\} }\{Q\}}\textit{if\_true} \tag{6.19}$$

The *if_true* rule can actually be derived form the *while* rule described below (note that a while statement can be rewritten as an if statement containing a while statement). It can also be trivially modified to produce a corresponding *if_false* rule that applies if $P \Rightarrow \neg B$. These rules can also be used to encode the semantics of `if` statements and `if-elseif-else` statements, as both of these can be rewritten as (potentially several nested) `if-else` statements.

The following rule encodes the semantics of *deterministic* `while` statements (i.e. not `while-choose` statements):

$$\frac{\{B \wedge \Psi\}\texttt{prog}\{\Psi\}}{\{\Psi\}\texttt{while (B) \{prog\}}\ \{\neg B \wedge \Psi\}} while \tag{6.20}$$

Ψ is referred to as a **loop invariant**. Loop invariants are notoriously difficult to find, and result in almost all of the difficulty when finding Hoare-logic proofs.

We also provide a rules to propagate `accept` and `reject` statements out of `while` loops:

$$\frac{\{P\}\texttt{prog}\{accepts\} \quad P \Rightarrow B}{\{P\}\texttt{while (B) \{prog\}}\{accepts\}} while\_accepts \tag{6.21}$$

This rule can be converted to the *while_rejects* rule by swapping the *accept* assertions with *reject* assertions. Note that these rules may invalidate eq. 6.1, depending on how many times we choose to execute the `while` loop before we arrive at an execution

Note that this rule only works if `prog` accepts or rejects after being executed once; however, if we transform a while statement into an infinite series of nested `if` statements, we can apply this rule to any iteration of the loop.

Finally, we provide a rule to assert that an infinite loop that never accepts results in a `while` loop rejecting:

$$\frac{\{P\}\texttt{prog}\{P\} \quad P \Rightarrow B \quad P \Rightarrow \neg accepts}{\{P\}\texttt{while (B) \{prog\}}\{rejects\}} while\_rejects \tag{6.22}$$

These rules are sufficient to prove useful theorems about 2dc, and we demonstrate such a proof in sec. 6.1.4.

6.1.4 An example proof

Consider the simple program in fig. 6.1 (assume it is called `prog`). We prove a small theorem about it in order to demonstrate `twoc`'s utility.

```
decr_on_zero = true;
alphabet = ['0', '1'];

twoc (string) {
    // Move off of lend
    move(1);

    // Count the 0s
    while (read == '0') {
        c++;
        move(1);
    }

    // Count the 1s while the counter is nonempty
    while (read == 1 && c != 0) {
        c--;
        move(1);
    }

    // Accept iff number of 0s equals number of 1s
    if (c == 0 && read == rend) {
        accept;
    }
}

```

Figure 6.1: A simple program that decides the language $L = \{0^i 1^i : i \geq 0\}$

Theorem 6.1. *The program accepts words of the form $w = 0^k 1^k$ for all $k \geq 1$.*

Proof. We prove this statement by verifying the Hoare triple $\{init\}$`prog`$\{accepts\}$.

We begin by stating our static assumptions:

- $w = 0^k1^k$ for all $k \geq 1$ (note that $|w| = n = 2k + 2$ including the endmarkers)
- $c \geq 0$, $i \in [0, n-1] = [0, 2k+1]$

We can assume these statements to be true at any point in any of the logical formulae in the proof.

We first consider line 6, and apply the *move_readhead* rule to yield a postcondition of $c = 0 \wedge i = 1$.

We now consider the `while` loop on lines 9-12, which requires we prove the following lemma:

Lemma 6.2. $\{B \wedge \Psi\}$`c++; move(i);`$\{\Psi\}$ *where* $B \equiv w_i = 0$ *and* $\Psi \equiv c + 1 = i \wedge i \leq k + 1$.

Proof. First note that $B \Rightarrow i \leq k + 1$ (this follows from $w = 0^k1^k$). We begin with the precondition $B \wedge \Psi$, and we apply the *increment_counter* rule to yield

$$\{B \wedge \Psi\}\texttt{c++;}\{B \wedge c = i \wedge i \leq k + 1\} \tag{6.23}$$

We then use the *move_readhead* rule to yield

$$\{B \wedge c = i \wedge i \leq k + 1\}\texttt{move(i);}\{B \wedge c + 1 = i \wedge i \leq k + 2\} \tag{6.24}$$

Note that, because $B \Rightarrow i \leq k + 1$, our postcondition implies $B \wedge \Psi$. Therefore, via the *consequence* rule,

$$\{B \wedge c = i \wedge i \leq k + 1\}\texttt{move(i);}\{B \wedge \Psi\} \tag{6.25}$$

From the *sequence* rule, we can assert $\{B \wedge \Psi\}$`c++;move(i);`$\{\Psi\}$. □

From lemma 6.2, via the *while* rule, we can deduce that

$$\{\Psi\}\texttt{while (read == 0) \{c++; move(1);\}}\{\neg B \wedge \Psi\} \tag{6.26}$$

We also make the following observations:

- $c = 0 \wedge i = 1$ implies Ψ
- $\neg B \wedge \Psi$ implies $c + 1 = i \wedge i = k + 1$

Using these, as well as the *consequence* and *sequence* rules, we can deduce that, after executing this `while` loop, we have a postcondition of $c + 1 = i \wedge i = k + 1$. This implies that $c = k \wedge i = k + 1$.

We now consider the `while` loop on lines 15-18, which requires the proof of another lemma:

Lemma 6.3. $\{A \wedge \Phi\}$ `c--; move(1);` $\{\Phi\}$, *where* $A \equiv w_i = 1 \wedge c > 0$ *and* $\Phi \equiv c \leq 2k + 1 - i \wedge i \geq k + 1$.

Proof. First note that $A \Rightarrow i \geq k + 1$ (this also follows from $w = 0^k1^k$). We begin with the precondition $A \wedge \Phi$, and we apply the *decrement_counter_a* rule to yield

$$\{A \wedge \Phi\}\texttt{c--;}\{c \leq 2k + 1 - i \wedge i \geq k + 1\} \tag{6.27}$$

We note that $c \leq 2k + 1 - i$ implies that $c \leq 2k + 1 - (i - 1)$. Applying the *consequence* rule yields

$$\{A \wedge \Phi\}\texttt{c--;}\{c \leq 2k + 1 - (i - 1) \wedge i \geq k + 1]\} \tag{6.28}$$

We then use the *move_readhead* rule to yield

$$\{c \leq 2k + 1 - (i - 1) \wedge i \leq 2k + 1\}\texttt{move(1);}\{c \leq 2k + 1 - i \wedge i \geq k + 1\} \tag{6.29}$$

Via the *consequence* and *sequence* rules, we obtain $\{A \wedge \Phi\}$ `c--; move(1);` $\{\Phi\}$. □

As before, from lemma 6.3 and the *while* rule, we can deduce

$$\{\Phi\}\texttt{while (read == '1' \&\& c != 0) \{ c--; move(1); \}}\ \{\neg A \wedge \Phi\} \tag{6.30}$$

We also make the following observations:

- $c = k \wedge i = k + 1$ implies Φ
- $\neg A \wedge \Phi$ implies $c = 0 \wedge i = 2k + 2$

Using these, as well as the *consequence* and *sequence* rules, we can deduce that, after executing this while loop, we have a postcondition of $c = 0 \wedge i = 2k + 2$. This implies that $c = 0 \wedge w_i = \vdash$.

We now consider the `if` statement on line 21. From the if_true and *accept* rules, we can deduce that we reach a postcondition of *accepts*, which completes the proof. □

Note that this proof is trivial to extend to $k \geq 0$: we simply demonstrate the execution of the program on $0^0 1^0 = \epsilon$ to satisfy the case where $k = 0$, and consider this in conjunction with the theorem in order to prove that the program accepts all strings $0^k 1^k, k \geq 0$.

Also note that this proof is not a complete proof that the program *decides* $L = \{0^k 1^k : k \geq 0\}$ (i.e. that the program is correct), as we have not proven that the program rejects all strings of the form $w = 0^k 1^l, k \neq l$.

6.2 Rules for nondeterminism

The rules described in the previous section are sufficient to prove theorems about deterministic `twoc` programs, but we have not yet introduced axioms that handle `branch` or `while-choose` statements.

We first note that both of these commands are equivalent to the two control structures presented by Dijkstra in [Dij75] (the **if** and **do** commands respectively). These commands form the basis for most of the research done into programmatic nondeterminism, and are discussed in several papers concerning Hoare-logic proofs of nondeterminstic programs [AO19; Apt83].

Before we discuss the proof rules, we make note of the semantics and context of our nondeterminism. While in many contexts (such as in Warwick's Formal Languages module[2]), it is useful to think about nondeterminism as a program concurrently executing several different computation paths, for our purposes it is far more useful to think about nondeterminism as providing our automaton with *choice*.

For `branch` statements, our programs *choose* between multiple blocks of code, and they choose whichever block of code leads it to an accepting snapshot, if such a snapshot can be reached (recall that our nondeterminism is assumed to be *angelic*). For `while-choose` statements, our programs *choose* how many times to execute a block of code, always executing them however many times they need to in order to reach an accepting snapshot, if one exists.

6.2.1 Branch statements

In the following, $\texttt{branch } \{\texttt{Bj}\}_{\texttt{j}=1}^{\texttt{k}}$ is shorthand for `branch {B1} also {B2} ... also {Bk}`.

Branch statements are the easier of the two control statements to design proof rules for, and the proof rules we design for them will help us to write proof rules for `while-choose` statements. The first of these rules follows directly from the formal semantics for the **if** command discussed in [Dij75]:

$$\frac{\forall \texttt{j} \in [1,k] : \quad \{P\}\texttt{Bj}\{Q\}}{\{P\}\texttt{branch } \{\texttt{Bj}\}_{j=1}^{k}\{Q\}} branch\_choice\_independent \tag{6.31}$$

We refer to this rule as *choice-independent* because it makes an assertion about `branch` statements that does not depend on which branch we choose. Note that, if we want to prove that a `branch` statement results in the program rejecting, we must use this rule.

Also note that this rule does not encode anything about the angelicism of our `branch` statements, and we can derive rules that allow us to prove nondeterministic acceptance by considering this. Note that these are heavily inspired by the rules presented in [Mam17].

The first of these rules is as follows:

$$\frac{\exists \texttt{j} \in [1,k] : \quad \{P\}\texttt{Bj}\{accepts\}}{\{P\}\texttt{branch } \{\texttt{Bj}\}_{j=1}^{k}\{accepts\}} branch\_internal\_accept \tag{6.32}$$

This rule asserts that, if any sub-branch `Bj` reaches an accepting snapshot from P, then the program must select that branch (or some other accepting branch) in any `branch` statement containing `Bj`.

[2] `https://warwick.ac.uk/fac/sci/dcs/teaching/modules/cs259/`

The second of these rules is as follows:

$$\frac{\exists \texttt{j} \in [1,k]: \quad \{P\}\texttt{Bj;C}\{accepts\}}{\{P\}\texttt{branch } \{\texttt{Bj}\}_{j=1}^{k}\texttt{;C}\{accepts\}} branch\_external\_accept \tag{6.33}$$

This rule asserts that if, for blocks `Bj` and `C`, if we can execute `Bj` followed by `C` from P to yield an accepting snapshot, then the program must select branch `Bj` (or some other accepting branch) in any `branch` statement containing `Bj` that is immediately followed by `C`.

These additional rules can be thought of as giving our proofs access to the same power to make angelic non-deterministic choices that these programs have (i.e. when given a choice, we are allowed to pick whichever block of code leads us to an accepting state). Note that the index `j` is simply an index used to assist in writing these rules, and the use of this variable does not require the introduction of any additional variables to our logical formulae.

6.2.2 While-choose statements

In the following, `prog*k` is shorthand for `prog;prog; ... ;prog` repeated `k` ≥ 0 times.

First note that a `while-choose` statement of the form `while (choose) {prog}` is equivalent to the infinite `branch` statement in fig. 6.2. This observation allows us to trivially adapt the rules for `branch` statements.

```
branch { prog*0; }
also   { prog*1; }
also   { prog*2; }
also   { prog*3; }
also   { prog*4; }
also   { prog*5; }
also   { prog*6; }
// etc.
```

Figure 6.2: An infinite branch statement equivalent to a while-choose statement

$$\frac{\forall \texttt{j} \in \mathbb{N} \cup \{0\}: \quad \{P\}\texttt{prog*j}\{Q\}}{\{P\}\texttt{while (choose) } \{\texttt{prog}\}\{Q\}} while\_choose\_choice\_independent \tag{6.34}$$

$$\frac{\exists \texttt{j} \in \mathbb{N} \cup \{0\}: \quad \{P\}\texttt{prog*j}\{accepts\}}{\{P\}\texttt{while (choose) } \{\texttt{prog}\}\{accepts\}} while\_choose\_internal\_accept \tag{6.35}$$

$$\frac{\exists \texttt{j} \in \mathbb{N} \cup \{0\}: \quad \{P\}\texttt{prog*j;C}\{accepts\}}{\{P\}\texttt{while (choose) } \{\texttt{prog}\}\texttt{;C}\{accepts\}} while\_choose\_external\_accept \tag{6.36}$$

Note that, unlike with our rules for branch statements, these rules *do* require that our logical formulae range across more variables than c, i and w (thus invalidating eq. 6.1), as our proofs now may need to introduce a new free variable $\texttt{j} \in \mathbb{N} \cup \{0\}$ for each `while-choose` statement encountered in a proof.

6.2.3 An example proof

Consider the simple program in fig. 6.3.

Theorem 6.4. *This program accepts all words $w \in \Sigma^*$.*

Proof. As before, we first state our static assumptions: $c \geq 0 \wedge i \in [0, n-1]$. Note here that we do not make any assertions about w (aside from $|w| = n$).

We begin with the precondition *init*, which implies $c = 0$. Via the *branch_accepts* rule, we choose the second block of code in the `branch` statement to yield a postcondition of $c = n + n \Rightarrow c = 2n$. Assume that the rest of the program is called `prog'`.

```
1     decr_on_zero = true;
2     alphabet = [ '0' ];
3
4     twoc (int X) {
5         // Set c = X or c = 2*X
6         branch {
7             c  = X;
8         } also {
9             c  = X;
10            c += X;
11        }
12
13        // Decrement the counter however many times we like
14        while (choose) {
15            if (c != 0) {
16                c -= 2;
17            }
18        }
19
20        // Accept if c was even
21        // this is always true
22        if (c == 0) {
23            accept;
24        }
25    }
26
```

Figure 6.3: A simple nondeterministic program that always accepts

We then move on to the `while-choose` statement; via the *while_choose* rule, we transform the program to the if statement on lines 15-17 repeated n times. Trivially, after one of these if statements is executed, the counter is decreased by 2 (i.e. $\{c = k > 0\}$`if (c != 0) c -= 2;`$\{c = k - 2\}$). Also trivially, for each iteration, c is initially greater than 0. Therefore, after executing the loop n times, we have a postcondition of $c = 2n - 2 - 2 \cdots - 2 = 2n - 2n = 0$.

With this as our precondition, we apply the if_true, *accepts* and *sequence* rules to assert that $\{init\}$`prog`$\{accepts\}$. □

6.3 A proof about a non-trivial example program

Consider the program in sec. C.1, and assume it is called `prog`.

Theorem 6.5. *This program accepts all words $w = a^{2^k}$ for all $k \geq 2$.*

Proof. We first begin by stating our static assumptions: $w = a^{2^k}$ for some $k \geq 2$, as well as the domains of i and c. As before, our initial precondition is $init \equiv c = 0 \wedge i = 0$.

Next, we apply the *move_readhead* and if_false rules to lines 7 and 8 in order to yield the postcondition $i = 1 \wedge c = 0$. Similarly, we apply the *move_readhead* and if_false rules to lines 12-15 to yield a postcondition $i = 3 = 2^1 + 1 \wedge c = 0$ (recall that the repeat macro simply repeats a block of code).

We apply the *while_accepts* rule $k - 2$ times, which allows us to ignore the while loop for now. We now state some lemmas about the body of the loop; we call this block of code `body`.

Lemma 6.6. *For all $l \in [1, k-2]$, $\{i = 2^l + 1 \wedge c = 0\}$`body`$\{i = 2^{l+1} + 1 \wedge c = 0\}$.*

Lemma 6.7. $\{i = 2^{k-1} \wedge c = 0\}$`body`$\{accepts\}$

The conjunction of these two lemmas, as well as the *while_accepts* rule, implies the theorem. □

Proof of lemma 6.6. We start with the precondition $i = 2^l + 1 \wedge c = 0$. We then apply the *move_readhead* rule to line 21 to yield $i = 2^l \wedge c = 0$.

Now consider the `while` loop on lines 22-27. We define $\Psi \equiv c = 2^{l+1} - 2i$ (the loop invariant) and $B \equiv i \geq 1$

(the loop condition). Note that $i = 2l \wedge c = 0$ implies $\Psi \wedge B$. We then apply the *move_readhead* rule to yield

$$\{\Psi \wedge B\}\texttt{move(-1);}\{i \geq 0 \wedge c = 2^{l+1} - 2(i+1) = 2^{l+1} - 2i - 2\} \tag{6.37}$$

We also apply the *increment_counter* rule to yield

$$\{i \geq 0 \wedge c = 2^{l+1} - 2i - 2\}\texttt{c += 2;}\{i \geq 0 \wedge c = 2^{l+1} - 2i\} \tag{6.38}$$

We then apply the *sequence* and *consequence* rules to yield

$$\{\Psi \wedge B\}\texttt{move(-1);c += 2;}\{\Psi\} \tag{6.39}$$

This allows us to use the *while* rule to assert that this loop terminates in postcondition $\Psi \wedge \neg B$, which implies that $i = 0 \wedge c = 2^{l+1}$.

Executing line 30 results in $i = 1 \wedge c = 2^{l+1}$. We then consider the `while` loop on lines 31-34. We define $\Phi \equiv i = 2^{l+1} - c + 1$ (the loop invariant) and $B \equiv i \leq 2^k \wedge c \geq 1$ (the loop condition). Note that $i = 1 \wedge c = 2^{l+1}$ implies $\Phi \wedge B$. We then apply the *decr_zero_a* rule to yield

$$\{\Phi \wedge B\}\texttt{c--;}\{i = 2^{l+1} - c\} \tag{6.40}$$

We then apply the *move_readhead* rule to yield

$$\{i = 2^{l+1} - c\}\texttt{move(1);}\{i = 2^{l+1} - c + 1\} \tag{6.41}$$

We then apply the *sequence* and *consequence* rules to yield

$$\{\Psi \wedge B\}\texttt{move(-1);c += 2;}\{\Psi\} \tag{6.42}$$

This allows us to use the *while* rule to assert that this loop terminates in postcondition $\Phi \wedge \neg B$, which implies that $i = 2^{l+1} + 1 \wedge c = 0$. We use the if_false rule to show that we do not accept on this iteration. □

Proof of lemma 6.7. We repeat the previous argument until the end of the second while statement, which gives us a postcondition of $i = 2^k + 1 \wedge c = 0$, which implies that $w_i = \vdash \wedge\, c = 0$. Using the if_true rule, we reach a postcondition of *accept*. □

Chapter 7

Evaluation

In this chapter, we discuss the outputs of this project and evaluate their success. We also discuss their limitations and detail how they would have been improved had this project been allowed more time. Finally, we conclude the project and discuss how this work can be applied and extended in the future.

7.1 The language

7.1.1 Language design

We begin by evaluating the design of the language, recalling the following design principles outlined at the start of chapter 3:

- `twoc` should be easy to learn and comfortable to use for users with prior programming experience
- `twoc` should be as easy as possible to write programs in without abstracting too far from the definitions of 2dc/2nc

It is the author's opinion that the `twoc` language successfully accomplishes both of these criteria.

It is clear that, for deterministic programs, the language is extremely easy to learn for users with prior programming experience (particularly those with experience in langauges like C, C++, Java, etc.). In this case, users will be familiar with all of the language's syntactic features (increments/decrements, if and while statements, etc.).

The design of this language also successfully contextualises the features of this class of automaton that do not fit neatly into a programmatic environment (namely the automaton posessing a traversable input tape). This is done through treating the readhead as a readable variable (e.g. `read != '0'`) and by associating the concept of move instructions to that function calls (e.g. `move(j);`), which are both concepts that experienced users will be very comfortable with.

The author also believes that the syntax of the two nondeterministic control structures (branch and while-choose statements) is intuitive, both for users with prior experience with nondeterministic programs, and for those who have never encountered nondeterministic programs before. Users with experience in this area will immediately be able to identify the similarities between these commands and those presented by Dijkstra and others [Dij75; SS92; Bod+10].

The macros and syntactic sugar provided as part of this implementation also help to meet these two goals. These macros are also designed to correspond to familiar programming features (input formatting associates to function definition, repeat statements correspond to for loops, etc.). These features also allow for significantly more concise programs.

Finally, this language provably accomplishes its primary goal of representing programs that can be transformed into 2dc/2nc.

7.1.2 Language implementation

We now discuss efficacy of the implementation of the language.

First and foremost, our implementation of the `twoc` language is very likely correct. Users can be highly confident that, for any input program, it produces correct outputs. If any bugs have slipped through,

the project codebase is left open-source in order to allow for the open-source community to identify and fix these, as well as to implement new features if desired. Our implementation also demonstrably runs sufficiently quickly for its intended use cases, and (for deterministic programs) quickly enough for it to be practically useful even for excessively long input strings.

Due to the nature of this project, it was not appropriate to implement any kind of graphical user interface; however, the command-line interface provided is simple and effective for users with sufficient experience to learn.

7.1.3 Future improvements

Firstly, more language features could have been provided in order to allow for the description of more complex automata (e.g. [Pet94; Gal76]). Examples of some of these features are included below:

- The ability to test whether or not parts of the input string adhere to given regular expressions. For instance, `r"0*1?"` could be a valid logical expression, asserting that the input string to the right of the readhead matches the regex `0*1?` (note that $R \subseteq 2DC$, where R is the set of all regular languages).
- The ability to declare variables that inhabit finitely many values (note that such variables can be embedded as part of the automaton's finite state control).
- The ability to test whether or not the counter value is even or odd (this feature can also be embedded in the automaton's finite control, a fact is used extensively in [Pet94]).
- Additional nondeterministic features designed to adhere to those presented in [Dij75] more closely. For instance, a combination of the branch and if statements would adhere almost precisely Dijkstra's guarded **if** command.

As well as this, the author expects that, had he had enough enough time, he could have formally proven that there exists a transformation that turns 2dc/2nc into equivalent programs. This theorem is left as a conjecture (although a potential proof strategy is provided in sec. 4.5).

The implementation could be subject to several improvements, mostly in the form of further optimisations.

The first of these optimisations could be to implement procedures designed to reduce the size of the constructed automata while preserving their behaviour. If feasible, this would result in significant speedups to the simulation algorithms (especially Rytter's algorithm, which we show runs in cubic time with respect to the size of the automaton.

The simulation algorithms could also be significantly optimised. Recall that the source of Glück's algorithm's slightly super-linear performance may be related to the number of recursive calls it makes [Glü13]. Other algorithms that simulate 2dpda in linear time (e.g. [Coo71; Jon77]) are not subject to these issues, as they are not recursive; therefore, they may run significantly faster than Glück's algorithm for larger input strings.

Also note that we do not apply the optimisation that results in Rytter's algorithm running in sub-cubic time [Ryt85]. Doing this would also be beneficial to the implementation, as well as exploring other algorithms for simulating 2npda (e.g. [AHU68; Glü16]; Glück's 2016 algorithm for this problem is of particular interest, due to its similarities to Glück's 2013 algorithm, which we demonstrate is significantly more efficient than Rytter's cubic-time algorithm).

Several quality-of-life improvements could also be applied to the implementation.

The first and most important of these would be the introduction of sufficient error reporting during the parsing stage. Currently, our implementation does not modify the parse errors produced by LALRPOP into anything approaching a user-friendly format (as required by the practical coursework assigned as part of Warwick's Formal Languages module[1]). Doing this, as well as providing VSCode extensions to support syntax highlighting for `twoc`, would drastically improve the usability of the implementation.

A GUI component could also have been implemented to allow users to see the state transition diagrams (e.g. fig. 1.3) of the automata constructed from their programs. Time constraints, as well as the author's critical lack of experience in GUI development, resulted in this being infeasible for this project, but given more time, the author may have deemed it worthwhile.

[1] `https://warwick.ac.uk/fac/sci/dcs/teaching/modules/cs259/`

7.2 The proof system

The proof system presented in this report is demonstrably powerful enough to formally verify several results concerning 2dc/2nc. It is expected that it could be used to formally verify several important results concerning this automata.

7.2.1 Future improvements

The most obvious future improvement to the proof system would be to provide an implementation for the axioms and rules provided (as was originally planned). As previously stated, there exist multiple possible approaches for this [Sit21; Pie+20]. In order to demonstrate `twop`'s power, it would also be appropriate to embed the proofs presented in chapter 6 in such a formal system.

Although we demonstrate it on some smaller examples, it would also be useful to provide alternate `twop` proofs for other existing theorems concerning 2dc/2nc. Formally verifying the programs in [Pet94] would be particularly useful applications of `twop`.

Formal soundness/completeness proofs for `twop` would also be of useful (these may follow directly from the equivalent results in [Mam17]).

7.3 Future work

We conclude by discussing the future work that could follow from the outputs of this project.

The most obvious of these, which follows directly from the project's motivation (sec. 1.3), is the use of `twoc` and `twop` to produce new inclusion theorems for the classes $2DC$ and $2NC$. Hopefully, the outputs of this project provide a framework for others to discover new theorems about these classes of automata, through their description (using `twoc`) and their formal verification (using `twop`).

Another avenue of potential future work would be the design, implementation and verification of new languages for different classes of automata.

The most obvious of these would be two-way pushdown automata, providing a competitor (in a sense) for the language described in [KB67]. Trivial modifications to this language and its implementation would directly result in a far more modernised and applicable variant of this language.

This approach could also be applied to various other classes of other automata. For instance, languages could be developed to represent other kinds of counter machine (machines with multiple counters, machines with multiple readheads, probabilistic and quantum counter automata, vector addition systems, etc.). However, note that the optimisations that make `twoc` efficient would not be possible in these cases.

References

[Aho+06] A.V. Aho et al. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Addison Wesley, Aug. 2006, pp. 399–410. ISBN: 0321486811. URL: https://www.google.co.uk/books/edition/_/dIU_AQAAIAAJ?hl=en. Also known as *The Dragon Book*.

[AHU68] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. "Time and tape complexity of pushdown automaton languages". In: *Information and Control* 13.3 (1968), pp. 186–206. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(68)91087-5. URL: https://www.sciencedirect.com/science/article/pii/S0019995868910875?via%3Dihub.

[AO19] Krzysztof R. Apt and Ernst-Rüdiger Olderog. "Fifty years of Hoare's logic". English. In: *Formal aspects of computing* 31.6 (2019), pp. 751–807. DOI: 10.1007/s00165-019-00501-3. URL: https://dl.acm.org/doi/10.1007/s00165-019-00501-3.

[Apt83] Krzysztof R. Apt. "Ten years of Hoare's logic: A survey— part II: Nondeterminism". In: *Theoretical Computer Science* 28.1 (1983), pp. 83–109. ISSN: 0304-3975. DOI: 10.1016/0304-3975(83)90066-X. URL: https://www.sciencedirect.com/science/article/pii/030439758390066X.

[BA22] William Bugden and Ayman Alahmar. *Rust: The Programming Language for Safety and Performance*. 2022. DOI: 10.48550/arXiv.2206.05503. arXiv: 2206.05503 [cs.PL]. URL: https://arxiv.org/abs/2206.05503.

[Bec+01] Kent Beck et al. *Manifesto for Agile Software Development*. 2001. URL: https://agilemanifesto.org/. Accessed April 2023.

[BJ66] Corrado Böhm and Giuseppe Jacopini. "Flow Diagrams, Turing Machines and Languages with Only Two Formation Rules". In: *Commun. ACM* 9.5 (May 1966), pp. 366–371. ISSN: 0001-0782. DOI: 10.1145/355592.365646. URL: https://dl.acm.org/doi/10.1145/355592.365646.

[Bod+10] Rastislav Bodik et al. "Programming with Angelic Nondeterminism". In: *SIGPLAN Not.* 45.1 (Jan. 2010), pp. 339–352. ISSN: 0362-1340. DOI: 10.1145/1707801.1706339. URL: https://dl.acm.org/doi/10.1145/1707801.1706339.

[BY14] Marzio De Biasi and Abuzer Yakaryilmaz. *Unary languages recognized by two-way one-counter automata*. 2014. DOI: 10.48550/arXiv.1311.0849. arXiv: 1311.0849 [cs.FL]. URL: https://arxiv.org/abs/1311.0849.

[Chr85] Marek Chrobak. "Variations on the technique of Ďuriš and Galil". In: *Journal of Computer and System Sciences* 30.1 (1985), pp. 77–85. ISSN: 0022-0000. DOI: 10.1016/0022-0000(85)90005-4. URL: https://www.sciencedirect.com/science/article/pii/0022000085900054.

[Con17] Adroit Project Consultants. *The Two Main Types of Project Work Breakdown Structures (WBS)*. Apr. 2017. URL: https://www.adroitprojectconsultants.com/tag/deliverable-oriented-wbs/. Accessed April 2023.

[Coo71] Stephen A. Cook. "Linear Time Simulation of Deterministic Two-Way Pushdown Automata". In: *Information Processing, Proceedings of IFIP Congress 1971, Volume 1 - Foundations and Systems, Ljubljana, Yugoslavia, August 23-28, 1971*. Ed. by Charles V. Freiman, John E. Griffith, and Jack L. Rosenfeld. North-Holland, 1971, pp. 75–80. URL: https://dblp.org/rec/conf/ifip/Cook71.bib. *The author was unable to locate this paper, and the given URL is for its DBLP entry.*

[CSK11] Pinaki Chakraborty, P. C. Saxena, and C. P. Katti. "Fifty Years of Automata Simulation: A Review". In: *ACM Inroads* 2.4 (Dec. 2011), pp. 59–70. ISSN: 2153-2184. DOI: 10.1145/2038876.2038893. URL: https://dl.acm.org/doi/10.1145/2038876.2038893.

[DG82] Pavol Duris and Zvi Galil. “Fooling a two way automation or one pushdown store is better than one counter for two way machines”. In: *Theoretical Computer Science* 21.1 (1982), pp. 39–53. ISSN: 0304-3975. DOI: 10.1016/0304-3975(82)90087-1. URL: https://www.sciencedirect.com/science/article/pii/0304397582900871.

[Dij68] Edsger W. Dijkstra. “Letters to the Editor: Go to Statement Considered Harmful”. In: *Commun. ACM* 11.3 (Mar. 1968), pp. 147–148. ISSN: 0001-0782. DOI: 10.1145/362929.362947. URL: https://dl.acm.org/doi/10.1145/362929.362947.

[Dij75] Edsger W. Dijkstra. “Guarded Commands, Nondeterminacy and Formal Derivation of Programs”. In: *Commun. ACM* 18.8 (Aug. 1975), pp. 453–457. ISSN: 0001-0782. DOI: 10.1145/360933.360975. URL: https://dl.acm.org/doi/10.1145/360933.360975.

[Gal76] Zvi Galil. “Some open problems in the theory of computation as questions about two-way deterministic pushdown automaton languages”. In: *Mathematical systems theory* 10.1 (1976), pp. 211–228. DOI: 10.1007/BF01683273. URL: https://link.springer.com/article/10.1007/BF01683273.

[GHI67] James N. Gray, Michael A. Harrison, and Oscar H. Ibarra. “Two-way pushdown automata”. In: *Information and Control* 11.1 (1967), pp. 30–70. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(67)90369-5. URL: https://www.sciencedirect.com/science/article/pii/S0019995867903695.

[Glü13] Robert Glück. “Simulation of Two-Way Pushdown Automata Revisited”. In: *Electronic Proceedings in Theoretical Computer Science* 129 (Sept. 2013), pp. 250–258. DOI: 10.4204/eptcs.129.15. URL: https://arxiv.org/abs/1309.5142v1.

[Glü16] Robert Glück. “A Practical Simulation Result for Two-Way Pushdown Automata”. In: *Implementation and Application of Automata*. Ed. by Yo-Sub Han and Kai Salomaa. Cham: Springer International Publishing, 2016, pp. 113–124. ISBN: 978-3-319-40946-7. DOI: 10.1007/978-3-319-40946-7_10. URL: https://link.springer.com/chapter/10.1007/978-3-319-40946-7_10.

[Gre65] Sheila A. Greibach. “A New Normal-Form Theorem for Context-Free Phrase Structure Grammars”. In: *J. ACM* 12.1 (Jan. 1965), pp. 42–52. ISSN: 0004-5411. DOI: 10.1145/321250.321254. URL: https://dl.acm.org/doi/10.1145/321250.321254.

[Hoa69] C. A. R. Hoare. “An Axiomatic Basis for Computer Programming”. In: *Commun. ACM* 12.10 (Oct. 1969), pp. 576–580. ISSN: 0001-0782. DOI: 10.1145/363235.363259. URL: https://dl.acm.org/doi/10.1145/363235.363259.

[Ins13a] Project Management Institute. *A Guide to the Project Management Body of Knowledge: PMBOK Guide*. 5th. PMBOK® Guide Series. Project Management Institute, 2013. Chap. 5.4, pp. 125–132. ISBN: 9781935589679. URL: https://books.google.co.uk/books?id=FpatMQEACAAJ.

[Ins13b] Project Management Institute. *A Guide to the Project Management Body of Knowledge: PMBOK Guide*. 5th. PMBOK® Guide Series. Project Management Institute, 2013. Chap. 11.2, pp. 319–327. ISBN: 9781935589679. URL: https://books.google.co.uk/books?id=FpatMQEACAAJ.

[Jon77] Neil D Jones. “A note on linear time simulation of deterministic two-way pushdown automata”. In: *DAIMI Report Series* 75 (1977). DOI: 10.7146/dpb.v6i75.6492. URL: https://tidsskrift.dk/daimipb/article/view/6492.

[KB67] Donald E. Knuth and Richard H. Bigelow. “Programming Language for Automata”. In: *J. ACM* 14.4 (Oct. 1967), pp. 615–635. ISSN: 0004-5411. DOI: 10.1145/321420.321421. URL: https://dl.acm.org/doi/10.1145/321420.321421.

[KMP77] Donald E. Knuth, James H. Morris Jr., and Vaughan R. Pratt. “Fast Pattern Matching in Strings”. In: *SIAM Journal on Computing* 6.2 (1977), pp. 323–350. DOI: 10.1137/0206024. URL: https://epubs.siam.org/doi/10.1137/0206024.

[Leo91] Joop M.I.M. Leo. “A general context-free parsing algorithm running in linear time on every LR(k) grammar without using lookahead”. In: *Theoretical Computer Science* 82.1 (1991), pp. 165–176. ISSN: 0304-3975. DOI: 10.1016/0304-3975(91)90180-A. URL: https://www.sciencedirect.com/science/article/pii/030439759190180A.

[Mam17] Konstantinos Mamouras. “Synthesis of Strategies Using the Hoare Logic of Angelic and Demonic Nondeterminism”. In: *Logical Methods in Computer Science* Volume 12, Issue 3 (Apr. 2017). DOI: 10.2168/LMCS-12(3:6)2016. URL: https://lmcs.episciences.org/2017.

[Nij82] Anton Nijholt. "On the relationship between the LL(k) and LR(k) grammars". In: *Information Processing Letters* 15.3 (1982), pp. 97–101. ISSN: 0020-0190. DOI: 10.1016/0020-0190(82)90038-2. URL: https://www.sciencedirect.com/science/article/pii/0020019082900382.

[Pet94] H. Petersen. "Two-Way One-Counter Automata Accepting Bounded Languages". In: *SIGACT News* 25.3 (Sept. 1994), pp. 102–105. ISSN: 0163-5700. DOI: 10.1145/193820.193835. URL: https://dl.acm.org/doi/10.1145/193820.193835.

[Pie+20] Benjamin C. Pierce et al. *Programming Language Foundations*. Vol. 2. Software Foundations. Version 5.8. Electronic textbook, 2020. Chap. Hoare. URL: http://softwarefoundations.cis.upenn.edu. The specific chapter referenced can be found *here*.

[Pro03a] LLVM Project. *Kaleidoscope: Implementing a Parser and AST*. 2003. URL: https://llvm.org/docs/tutorial/MyFirstLanguageFrontend/LangImpl02.html. Accessed April 2023.

[Pro03b] LLVM Project. *LLVM Language Reference*. 2003. URL: https://llvm.org/docs/LangRef.html. Accessed April 2023.

[Rod18] Michael Rodler. *Simple "if-then-else" Branching*. 2018. URL: https://mapping-high-level-constructs-to-llvm-ir.readthedocs.io/en/latest/control-structures/if-then-else.html. Accessed April 2023.

[Ryt85] Wojciech Rytter. "Fast recognition of pushdown automaton and context-free languages". In: *Information and Control* 67.1 (1985), pp. 12–22. ISSN: 0019-9958. DOI: 10.1016/S0019-9958(85)80024-3. URL: https://www.sciencedirect.com/science/article/pii/S0019995885800243?via%3Dihub.

[Scr23] Scrum.org. *What is Scrum?* 2023. URL: https://www.scrum.org/resources/what-scrum-module. Accessed April 2023.

[Sit21] Boro Sitnikovski. *Tutorial implementation of Hoare logic in Haskell*. Dec. 2021. DOI: 10.48550/arXiv.2101.11320. arXiv: 2101.11320 [cs.PL]. URL: https://arxiv.org/abs/2101.11320.

[SS92] H. Søndergaard and P. Sestoft. "Non-determinism in Functional Languages". In: *The Computer Journal* 35.5 (Oct. 1992), pp. 514–523. ISSN: 0010-4620. DOI: 10.1093/comjnl/35.5.514. URL: https://academic.oup.com/comjnl/article/35/5/514/402509.

[Teaa] Rust Team. *Lifetimes*. URL: https://doc.rust-lang.org/rust-by-example/scope/lifetime.html. Accessed April 2023.

[Teab] Rust Team. *Module std::option*. URL: https://doc.rust-lang.org/std/option/. Accessed April 2023.

[Teac] Rust Team. *Module std::thread*. URL: https://doc.rust-lang.org/std/thread/. Accessed April 2023.

[Tead] Rust Team. *Struct std::collections::hash_map::HashMap*. URL: https://doc.rust-lang.org/std/collections/hash_map/struct.HashMap.html. Accessed April 2023.

[Teae] Rust Team. *Struct std::collections::hash_set::HashSet*. URL: https://doc.rust-lang.org/std/collections/hash_set/struct.HashSet.html. Accessed April 2023.

[Teaf] Rust Team. *Struct std::collections::VecDeque*. URL: https://doc.rust-lang.org/stable/std/collections/struct.VecDeque.html. Accessed April 2023.

[Wei12] Rickey C. Weisner. *How Memory Allocation Affects Performance in Multithreaded Programs*. Mar. 2012. URL: https://www.oracle.com/technical-resources/articles/it-infrastructure/dev-mem-alloc.html. Accessed April 2023.

[Wes01] D.B. West. *Introduction to Graph Theory*. Featured Titles for Graph Theory. Prentice Hall, 2001, p. 26. ISBN: 9780130144003. URL: https://books.google.co.uk/books?id=TuvuAAAAMAAJ.

Appendix A

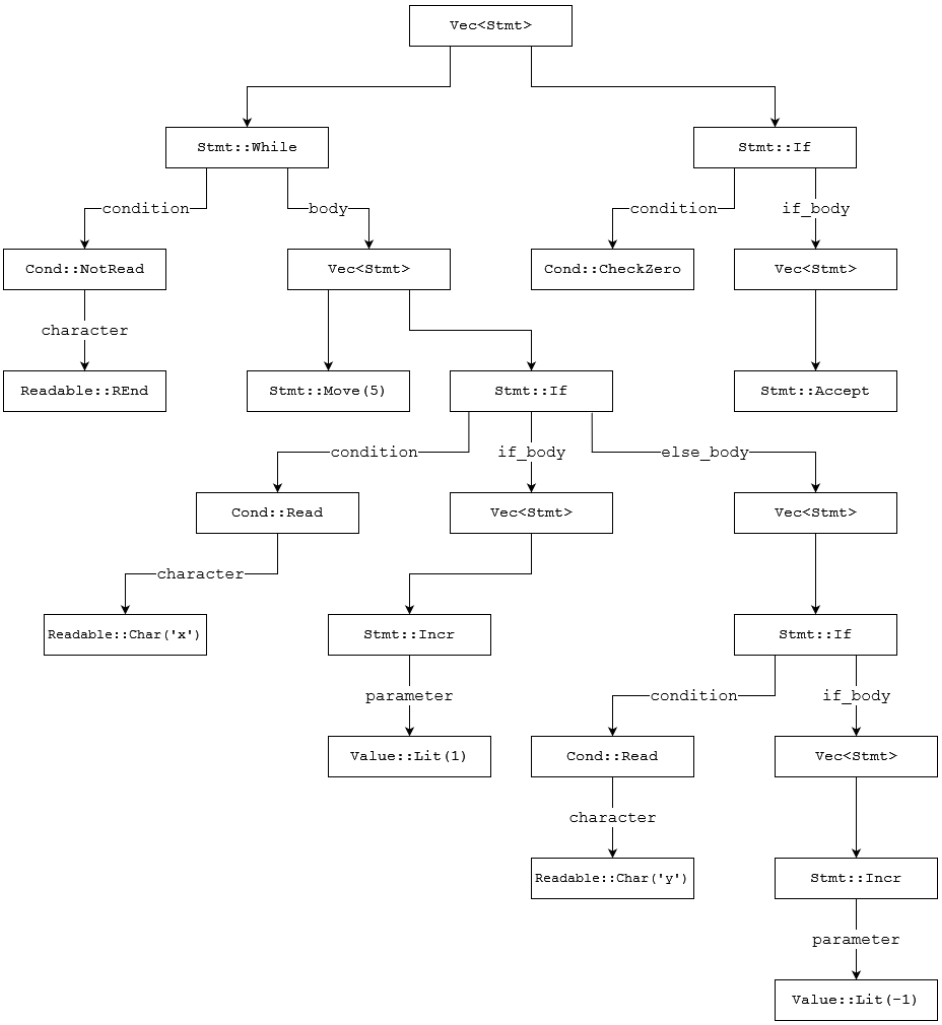
LR(1) grammar for the `twoc` language

```
// Terminals are written "like this"
// (these may be regular expressions; if so they are marked with an r)
// Nonterminals are written LikeThis
// The end-of-file character is denoted $

// Top-level rule
Twoc ::= ZeroDecr AlphabetDef "twoc" "(" Params ")" "{" StmtList "}" $

// Rule to parse decr_on_zero flag
ZeroDecr ::= "decr_on_zero" "=" "true" ";" | "decr_on_zero" "=" "false" ";"

// Rules to parse alphabet definition
AlphabetDef ::= "alphabet" "=" "[" Letters "]" ";"
Letters     ::= r"'[a-zA-Z0-9]'" | r"'[a-zA-Z0-9]'" "," Letters

// Rules to parse parameters
Params  ::= "string" | ParList
ParList ::= Param | Param "," ParList
Param   ::= "int" r"[a-zA-Z]+"


// Rule to parse lists of statements
StmtList ::= Stmt*

// Rule to parse staements
Stmt ::=
    "accept" ";" | "reject" ";" |
    "move" "(" r"-?[0-9]+" ")" ";" |

    "c" "++" ";" | "c" "--" ";" |
    "c" "+=" r"-?[0-9]+" ";" | "c" "-=" r"-?[0-9]+" ";" |
    "c" "+=" r"[a-zA-Z]+" | "c" "-=" r"[a-zA-Z]+" |

    "c" "=" r"-?[0-9]+" ";" | "c" "=" r"[a-zA-Z]+" ";" |

    "goto" "(" r"(lend|rend)" ")" ";" |

    "if" "(" Cond ")" "{" StmtList "}" ElseBody? |
    "if" "(" Cond ")" "{" StmtList "}" ElseIf |

    "while" "(" Cond ")" "{" StmtList "}" |
    "while" "(" "choose" ")" "{" StmtList "}" |
```

```
    "branch" "{" StmtList "}" AlsoBody* |

    "repeat" "(" r"-?[0-9]+" ")" "{" StmtList "}"

// Rule to parse else statements
ElseBody ::= "else" "{" StmtList "}"

// Rule to parse else-if statements
ElseIf ::=
    "else" "if" "(" Cond ")" "{" StmtList "}" ElseBody? |
    "else" "if" "(" Cond ")" "{" StmtList "}" ElseIf

// Rule to parse also statements
AlsoBody ::= "also" "{" StmtList "}"

// Rules to parse logical expressions
// Operator precedence embedded in the structure of the grammar (! then && then ||)
Cond     ::= AndCond | AndCond "||" Cond
AndCond  ::= BaseCond | BaseCond "&&" Cond
BaseCond ::=
    "true" | "false" |

    "c" "==" "0" | "c" "!=" "0" |

    "read" "==" r"'[a-zA-Z0-9]'" | "read" "!=" r"'[a-zA-Z0-9]'" |
    "read" "==" r"(lend|rend)" | "read" "!=" r"(lend|rend)" |

    "!" BaseCond | "(" Cond ")"
```

Appendix B

An example AST of a `twoc` program



Appendix C

Example `twoc` programs

Other example `twoc` programs can be found in `twoc/twocprogs`.

C.1 A nontrivial example program

This deterministic `twoc` program decides the language $\texttt{UPOWER}(k) = \{a^{k^n} : n \geq 1\}$ for $k = 2$, which demonstrates the capacity of 2dc to compute exponentials (this is significantly strengthened by the results in [Pet94], which we do not implement a program for) [BY14].

Note that this automaton can be generalised for any $k \geq 2$; the changes that need to be made in order to accomplish this have been identified with comments. Also note how this program leverages an infinite loop in order to correctly reject inputs.

```
decr_on_zero = true;
alphabet = [ 'a' ];

// UPOWER(2)
twoc (string) {
    // reject if w = epsilon
    move(1);
    if (read == rend) { reject; }

    // accept if w = a^1 or w = a^2
    // change this from 2 to change k
    repeat (2) {
        move(1);
        if (read == rend) { accept; }
    }

    // Loop
    // This loop does not terminate if w is not in UPOWER(2)
    while (true) {
        // Set c = 2^i
        move(-1);
        while (read != lend) {
            move(-1);

            // change this from 2 to change k
            c += 2;
        }

        // Check if w = a^c
        move(1);
        while (read != rend && c != 0) {
            c--;
            move(1);
        }

        // Accept if w = a^c
        if (c == 0 && read == rend) {
            accept;
        }
    }
```

```
}
```

C.2 Desugared example of fig. 3.9

```
    decr_on_zero = false;

    // One alphabet symbol per argument
    alphabet = [ '0', '1', '2' ];

    twoc (int X, int Y, int Z) {
        // Check that the input is in L(0*1*2*)
        while (read == '0')  { move(1); }
        while (read == '1')  { move(1); }
        while (read == '2')  { move(1); }
        if    (read != rend) { reject;  }

        // goto lend
        while (read != lend) { move(-1); }


        // c = X;

            // empty counter and goto lend
            while (c != 0) { c--; }
            while (read != lend) { move(-1); }

            // read all the characters that aren't 0
            while (read != '0' && read != rend) { move(1); }

            // read and count each 0
            while (read == '0') {
                move(1);
                c++;
            }

            // goto lend
            while (read != lend) { move(-1); }


        // c += Y;

            // goto lend
            while (read != lend) { move(-1); }

            // read all the characters that aren't 1
            while (read != '1' && read != rend) { move(1); }

            // read and count each 1
            while (read == '1') {
                move(1);
                c++:
            }

            // goto lend
            while (read != lend) { move(-1); }


        // c -= Z;

            // goto lend
            while (read statements), his algorithm excludes potential computation paths and
    may reject strings that the!= lend) { move(-1); }

            // read all the characters that aren't 2
            while (read != '2' && read != rend) { move(1); }

            // read and count each 2
            while (read == '2') {
                move(1);
```

```
                c--;
            }

            // goto lend
            while (read != lend) { move(-1); }


        if (c == 0) { accept; }
    }
```

C.3 A 'left-recursive' program

The automaton defined by this program is 'left-recursive' due to the `while-choose` statement on line 9. In the constructed automaton, the transition that enters the loop appears in the adjacency list before the transition that exits it; therefore, Glück's algorithm considers it first, executes it once, and returns to the same configuration. Because of this, his algorithm assumes that it leads to an infinite loop, meaning that it discounts the path which executes the loop exactly 10 times, which leads the automaton to an accepting configuration.

```
1   decr_on_zero = false;
2   alphabet = [ 'x' ];
3
4   twoc (string) {
5       // Set c to 10
6       c += 10;
7
8       // Decrement c however much we want
9       while (choose) { c--; }
10
11      // Accept if we could have decremented c to 0
12      // (trivially, this program will always accept)
13      if (c == 0) { accept; }
14  }
```

C.4 A deterministic program that runs in time $O(n^2)$

```
1   decr_on_zero = true;
2   alphabet = [ '0' ];
3
4   twoc (string) {
5       // Set c = n
6       while (read != rend) {
7           c++;
8           move(1);
9       }
10
11      // Go back to lend
12      while (read != lend) { move(-1); }
13
14      // Move back and forth across the entire input n times
15      // Each iteration should take time O(n)
16      // We run it n times, therefore this program takes time O(n^2)
17      // to simulate naively
18      while (c != 0) {
19          while (read != rend) { move(1); }
20          while (read != lend) { move(-1); }
21          c--;
22      }
23
24      // This program always accepts
25      accept;
26  }
```

In order to highlight the efficiency of `twoc`'s deterministic implementation, consider this Python program:

```
def quad_performance_test(input):
    # @ = lend, £ = rend
    input = "@" + input + "$"
    i = 0
    c = 0

    while input[i] != '$':
        c += 1
        i += 1

    while input[i] != '@':
        i -= 1


    while c != 0:
        while input[i] != '$':
            i += 1
        while input[i] != '@':
            i -= 1
        c -= 1

    return True

```

Clearly, these two programs are computationally equivalent; however, the `twoc` variant runs significantly faster than the Python variant. On an input of size $n = 12000$, the `twoc` program above executes in around 6 seconds, while the Python variant executes in around 20 seconds. The author suggests invoking this function in a Python REPL on the input `'0'*12000` to demonstrate this.

C.5 A nondeterministic program with lots of states

Note that the below program does not conduct any particularly interesting computation, and is simply a test case that results in a sufficiently complicated test example for benchmarking purposes.

```
decr_on_zero = false;
alphabet = [ '0' ];

twoc (int X) {
    c = X;
    goto(lend);

    while (read != rend) {
        branch {
            move(1);
            if (read == rend) { reject; }
        }

        also {
            c-=2;
            goto(lend);
            if (c == 0) { accept; }
        }
    }
}
```