CALIFORNIA STATE UNIVERSITY, NORTHRIDGE

Survey of Imperative Style Turing Complete proof techniques and an application to prove Proteus Turing Complete

A thesis submitted in partial fulfillment of the requirements
for the degree of Master of Science in
Computer Science

By

Isaiah Martinez

December 2024

The thesis of Isaiah Martinez is approved:

Kyle Dewey, PhD., Chair	Date
John Noga, PhD.	Date
Maryam Jalali, PhD.	Date

Acknowledgements

Lorem ipsum dolor sit amet, consectetur adipiscing elit. Vivamus lacinia odio vitae vestibulum vestibulum. Cras venenatis euismod malesuada. Maecenas vehicula felis quis eros auctor, sed efficitur erat suscipit. Curabitur vel lacus velit. Proin a lacus at arcu porttitor vehicula. Mauris non velit vel lectus tincidunt ullamcorper at id risus. Sed convallis sollicitudin purus a scelerisque. Phasellus faucibus purus at magna tempus, sit amet aliquet nulla cursus.

Table of Contents

List of Figures

This list must reference the figure, page it appears, and subject matter.

List of Tables

This list must reference the table, page it appears, and subject matter.

List of Illustrations

This list must reference the illustration, page it appears, and subject matter.

Abstract

TITLE GOES HERE

By

Isaiah Martinez

Master of Science in Computer Science

Abstract which will cover the contents of the entire paper in less than 350 words or so. 1.5 pgs double spaced

state research probem briefly describe methods and procedures used in gathering data or studying the problem give a condensed summary of the findings of the study

Chapter 1

Introduction

1.1 Turing Machines

Alan Turing is generally considered the father of computer science for his numerous contributions including: formalization of computation theory, algorithm design, complexity theory, as well as creating the idea of the Turing Machine. A Turing machine can be described as a machine/automata that is capable of performing operations towards some desired goal given an input. In a sense, it was designed to be capable of performing any single computable task, such as addition, division, concatenating strings, rendering graphics, etc. TMs are at the highest level of computational power, i.e. capable of handling any computation.

Figure 1.1 is an example of a Turing Machine designed using JFLAP, see https://www.jflap.org/ for more information on this software. The described machine takes an input string of 1's followed by a 0. The machine then outputs whether there is an even number of 1's or not. We will revisit this exact machine later in section 2.3.

1.1.1 Oracles

Of course there also exists the Oracle, sometimes called Turing Machines with an oracle, which is capable of solving problems that TMs cannot. It does so by having the ability to respond to any given problem from the TM it is connected to. For example, the oracle would be able to solve the Halting Problem for the associated Turing Machine, but not the Halting Problem in general for all Turing Machines. The reason the oracle is not considered more powerful is because in practice

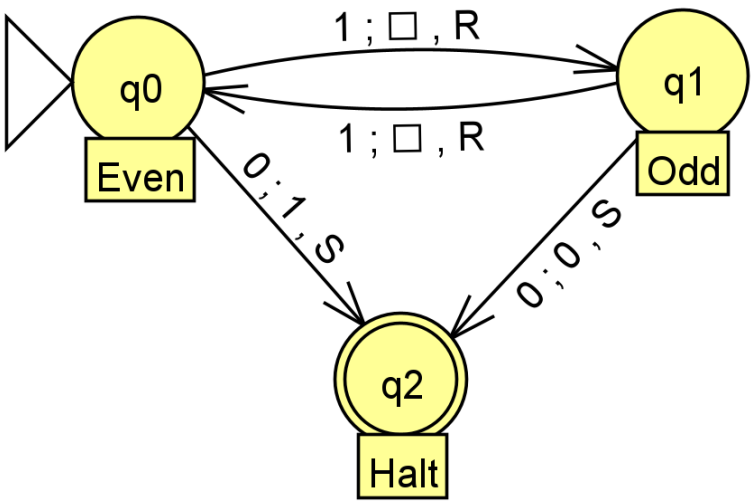

Figure 1.1: A TM that determines if there is an even number of 1's on the tape.

(i.e. reality), is because there is no such all-knowing source to retrieve information from. As a result, I will disregard the Oracles for the rest of the thesis.

1.1.2 Universal Turing Machines

A simple abstraction of the standard TM is a Universal Turing Machine. A UTM is capable of solving any computable problem, given the exact process/rules of the TM that will solve it. In essence it is a machine that is not hard-coded with what to perform when given input. The UTM will read the input, and respond based on the rules given. As a result, the UTM is equivalently as powerful as a TM. The only functional difference is the usability of the UTM towards a larger number of problems as opposed to the TM being created for a singular problem.

1.1.3 The Church-Turing Thesis

According to the Church-Turing Thesis, every effectively calculable function can be computed by a Turing Machine. As explained by Robin Gandy, the main idea was to show that there is an upper bound on the computation of TMs. This upper bound does not exist for humans and is therefore the basis of separation between computation power of TMs and humans.

Theorem 1. *(Church-Turing Thesis) Every effectively calculable function can be computed by a Turing Machine.*

He proposed a series of four Principles that we still use today as a basis for determining what one of the definitions of a TM is capable of computing. Any automata that violates any of these Principles is said to have ”free will”, which in context means being able to compute any non-computable function. As an example, the Oracle machine would be capable of computing a non-computable function, namely the halting problem, and thus would have ”free will”. We disregard such automata as there aren't any systems that exist in reality as of yet to display ”free will”. As a result, TMs are the most powerful automata that can compute any calculable function. This is why the Church-Turing thesis is generally assumed to be true [1].

1.2 Turing Completeness

Turing Completeness is a closely related term when discussing Turing Machines. For a system to be Turing Complete, it must be capable of performing any computation that a standard TM can perform. An equivalent description would be that for a system to be TC, it must simulate a UTM. By transitivity, if any system is proven to be TC, then it must be equivalent in power to all other systems that are TC. Therefore, all TC systems are considered the most powerful computation machine.

1.2.1 Considering the Practicality of Turing Complete Programming Languages

Despite all TC systems being equivalent in computation power, this does not mean that all are practically useful. This is because despite the TC being able to simulate any TM, it may have a more complex method for simulation or calculation of the same problem. This is considered a non-issue as the length of time needed for computation is not considered when discussing TMs and TC systems. This is only a factor for practical purposes, such as programming languages, space and time complexity are of major importance.

1.2.1.1 Esoteric Progamming Languages

Esoteric programming languages are designed to demonstrate a key concept with language design, but are often done so in a joking manner. An example of a highly simplistic well-known esoteric TC language is brainfuck. I will describe the way brainfuck operates and then provide several example programs with explanations.

The language has only 8 instructions, a data pointer, and an instruction pointer. It uses a s-ingle dimensional array containing 30,000 byte cells, with each cell initialized to zero. The data pointer points to the current cell within the array, initialized to index 0. The instruction point-er points to the next instruction to be processed, starting from the first character given in the code. Any characters besides those used in the instructions are considered comments and will be ignored. Instructions are executed sequentially unless branching logic is taken via the '[' or ']' instructions. The program terminates when the instruction pointer moves beyond the final command. Additionally, it has two streams of bytes for input and output which are used for entering keyboard input and displaying output on a monitor using the ASCII encoding scheme. [Wikipedia citation brainfuck](https://en.wikipedia.org/wiki/Brainfuck) [Github basics of Brain-fuck](https://gist.github.com/roachhd/dce54bec8ba55fb17d3a)

The 8 instructions are as follows:

Each '[' or ']' must correspond to match with it's complement symbol, namely ']' and '[' respectively. Also, when input is read with the ',' command the given character from a keyboard input will have its value read as a decimal ASCII code (eg. '!' corresponds to 33. 'a' corresponds to 97, etc.). The decimal value is what is then converted to binary and stored within the current byte.

BRAINFUCK CITATION FROM STACK OVERFLOW

Here is a simple program that modifies the value of the first cell in the 30,000 byte array.

```
++      Add 2 to the byte value in cell 0
[-]     Decrement the value of the current cell until it reaches 0
```

$>$	Increments the data pointer by one. (This points to the next cell on the right).
$<$	Decrement the data pointer by one. (This points to the next cell on the left).
$+$	Increments the byte at the data pointer by one.
$-$	Decrements the byte at the data pointer by one.
.	Output the byte at the data pointer.
,	Accept one byte of input, storing its value in the byte at the data pointer.
[	If the byte at the data pointer is zero, then instead of moving the instruction forward to the next command, go to the matching ']' command. (Jump forwards).
]	If the byte at the data pointer is non-zero, then instead of moving the instruction forward to the next command, go to the matching '[' command. (Jump backwards).

Table 1.1: Brainfuck Instruction Set

In fact, we can remove the comments and put the code onto a single line to achieve the same result. Recall that comments include any character that is not listed as one of the 8 aforementioned instructions.

```
++[-]
```

An equivalent program in python is seen below:

```
# Let 'Array' be our 30,000 byte array
Array[0] += 2
while (Array[0] != 0):
    Array[0] -= 1
```

Below is an example program that outputs Hello World. At the end of each line is the end result of the operations done in the line as a comment. Each line prints a new character.

```
>++++++++[<+++++++++>-]<.       H
>++++[<+++++++>-]<+.            e
+++++++..                       l
```

```
+++.                            l
>>++++++[<+++++++>-]<++.        o
------------.                   [space]
>++++++[<+++++++++>-]<+.        W
<.                              o
+++.                            r
------.                         l
--------.                       d
>>>++++[<++++++++>-]<+.         !
```

For an in-depth breakdown of brainfuck with examples and guiding logic, see: [Github basics of Brainfuck](https://gist.github.com/roachhd/dce54bec8ba55fb17d3a).

One common technique utilized when creating programming languages is to bootstrap them. This means that the developers will write a compiler for the language, using the language itself. This is done for many reasons, but the reason for introducing it here is to show how brainfuck is capable of complex logic that is more practically useful than simple programs as seen above. Below is the current smallest bootstrapped compiler for brainfuck.

PLACE SAYING ITS THE SMALLEST BRAINFUCK COMPILER [SMALLEST BRAINFUCK COMPILER][https://brainfuck.org/dbfi.b]

```
>>>+[[-]>>[-]++>+>+++++++[<++++>>++<-]++>>+>+>+++++[>++>++++++<<-]+>>>,
<++[[>[->>]<[>>]<<-]<[<]<+>>[>]>[<+>-[[<+>-]>]<[[[-]<]++<-[<+++++++++>[
<->-]>>]>>]]<<]<]<[[<]>[[>]>>[>>]+[<<]<[<]<+>>-]>[>]+[->>]<<<<[[<<]<[<]
+<<[+>+<<-[>-->+<<-[>+<[>>+<<-]]]>[<+>-]<]++>>-->[>]>>[>>]]<<[>>+<[[<]
<]>[[<<]<[<]+[-<+>>-[<<+>++>-[<->[<<+>>-]]]<[>+<-]>]>[>]>]>[>>]>>]<<[>>
+>>+>>]<<[->>>>>>>>>]<<[>.>>>>>>>]<<[>->>>>>]<<[>,>>>]<<[>+>]<<[+<<]<]
```

As we quickly found out, using brainfuck in any practical sense is simply too much work due to its extreme inefficiency. It also is extremely difficult to understand without comments indicating

the goal of each step. Due to the simplicity of the language, it is very useful for studying Turing Completeness. There exist many other esoteric TC programming languages, but the reason for choosing brainfuck in particular is its simple instruction set. As a result, we will now look at more useful and practical programming language paradigms. These languages within these paradigms will be much more efficient and legible, at the cost of increased complexity in instruction set.

1.2.1.2 Procedural Languages

Procedural Programming Languages are designed to be read linearly in execution order, top to bottom. The main idea behind this design of languages is to create procedures and subprocedures (equivalently routines and subroutines), to achieve a larger goal. For example to calculate the sum of squares in code you may design it in the C code as follows:

```
float squareNumber(float a) {
    return a * a;
}
float findSumOfSquares (float a, float b) {
    return squareNumber(a) + squareNumber(b);
}
```

When working in Procedural Programming Languages, variables are used to store and modify data. These variables may be locally or globally defined, which is where we define the concept of scope. Scope refers to the current lens in which we view code and the system memory. It identifies which variables exist, what values they have, and what operations are being performed. The below example in C provides insight into the importance of scope.

```
float globalVariable;


float foo (float bar) {
    float localVariable;
```

```
    ...
}

float baz (float qux) {
    float localVariable;
    ...
}
```

Notice that we are able to utilize a variable named localVariable in the two functions foo and baz. This is allowed because when the scope is inside of either function, the other function does not exist. The variable globalVariable is available to both because it is outside of the scope of both functions. This means that any other function in the code in the same scope as the globalVariable is capable of accessing its value.

I will now describe the importance of functions in Procedural Programming Languages. Functions are designed to complete a single goal and return a single output. They are capable of accepting 0^+ inputs and outputting 0 or 1 outputs. These functions are capable of calling other functions, including themselves. When a function calls itself, this is called recursion. Here is an example constructing the fibonacci sequence in C in 2 ways: without recursion, and with recursion.

```
//Given an integer n, calculate the first n numbers
//of the fibonacci sequence without recursion

void sequentialFibonacci (int n) {
    if (n < 1) {
        printf("Input must be an integer greater than 0");
        return;
    }

    int sub1 = 0;
```

```
    int sub2 = 1;

    for (int i = 1; i <= n; i+=1) {
        if (i > 2) {
            int curr = sub1 + sub2;
            sub1 = sub2;
            sub2 = curr;
            printf("%d ", curr);
        }
        else if (i == 1) {
            printf("%d ", sub1);
        }
        else if (i == 2) {
            printf("%d ", sub2);
        }
    }
}

/*****************************************************************/

//Recursive case

//keep track of current Index, given amount of fibonacci numbers
//to print, and propogate the two subnumbers to the next step
void recursiveFibonacci (int currIndex, int n, int sub1, int sub2) {
    if (currIndex < n) {
        printf("%d ", sub1 + sub2);
```

```
        recursiveFibonacci(currIndex + 1, n, sub2, sub1 + sub2);
    }
    return;
}

//handle base cases (exit conditions)
//otherwise start the recursive process
void startRecursiveFibonacci (int n) {
    if (n < 1) {
        printf("Input must be an integer greater than 0");
    } else if (n == 1) {
        printf("%d ", 0);
    } else if (n == 2) {
        printf("%d %d ", 0, 1);
    } else {
        printf("%d %d ", 0, 1);
        recursiveFibonacci(0, n - 2, 0, 1);
    }
    return;
}
```

As shown above, recursion is a powerful tool to simplify the amount of lines needed to code the main functionality of the procedure. It simplifies the amount of lines written because the overall logical design is more complex.

Through the use of scope and compartmentalizing procedures, Procedural programming is a very straightforward and capable design paradigm for software development. Some well known languages that are Turing Complete from this paradigm are:

- C

- Pascal
- COBOL
- Fortran
- ALGOL
- Basic

Although these languages are very old, with some coming from the 1960s, some still find modern use. Linus Torvalds, the creator of the Linux kernel and git, chose C to be the main language for developing both of these well known pieces of software. Both are still actively developed and improved to this day and remain majorly written in C. [SOURCE TO SEE THAT FILES FOR GIT ARE IN C][https://git.kernel.org/pub/scm/git/git.git/tree/] [SOURCE TO SEE THAT LINUX KERNEL FILES ARE IN C][https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/tree/] Additionally, Richard Stallman led the development for the GNU operating system using C. SOURCE TO SHOW C IS PREFERRED Although most users are on the Windows or Apple platform for PCs, the GNU operating system with the Linux kernel is still a popular choice amongst users looking for a different experience. [SOURCE SHOWING MARKET SHARE OF OS][https://gs.statcounter.com/os-market-share/desktop/worldwide/] Besides C, COBOL remains a language that is used professionally for banking. Many banks still use COBOL their business application and management. [COBOL USED BY BANKS][https://increment.com/programming-languages/cobol-all-the-way-down/]

1.2.1.3 Object Oriented Languages

A different scheme altogether for a programming language is an Object Oriented Language. Developing in this language paradigm is known as a Object Oriented Programming. OOP is structured entirely different than Procedural Programming. Instead of defining procedures to solve the problem, we utilize a new idea of coding. We outline classes, which are representations of some

system that we wish to design. Classes contain 3 parts: Data Members, Constructors, and Methods. Data Members are used to describe what the Class is. For example, if I want to model a school, an important data member would be the amount of students enrolled. Constructors are ways to create an instance of the class. This is where the object is created. In the school example, perhaps there would be two ways to create a school: with a total amount of students enrolled already, and another without. Both are valid as adding an existing school to the digital system would use the first constructor, while creating a new school would utilize the second constructor. Methods are ways we modify the attributes of the objects. Perhaps a certain amount of students enroll into the newly created school. We must have a way to update the amount of students for any school. These 3 parts form the basis of what OOP looks like. Below is a snippet of Java code demonstrating these principles.

```
class School {
    private int numEnrolledStudents;

    public School () {
        this.numEnrolledStudents = 0;
    }

    public School (int numAlreadyEnrolled) {
        this.numEnrolledStudents = numAlreadyEnrolled;
    }

    public int getNumEnrolled() {
        return this.numEnrolledStudents;
    }

    public void setNumEnrolled(int numStudents) {
```

```
        this.numEnrolledStudents = numStudents;
        return;
    }
}

class RunCode {
    public static void main(String[] args) {
        School NewSchool = new School();

        System.out.println("The number of students enrolled
            in the new school is: " + NewSchool.getNumEnrolled()); // 0

        NewSchool.setNumEnrolled(100);

        System.out.println("The number of students enrolled
            in the new school is: " + NewSchool.getNumEnrolled()); // 100

        School CSUN = new School(32172);

        System.out.println("The number of students enrolled
            at CSUN is: " + CSUN.getNumEnrolled());                // 32172
    }
}
```

[SRC FOR CURRENT NUM CSUN STUDENTS][https://www.usnews.com/best-colleges/california-state-university-northridge-1153]

There are more advanced features such as Inheritance that allow for more complex design models. Furthermore, Java contains Modifiers which are used to change the permission of which

piece of code is capable of being accessed by another piece of code. In the above example, only the school object is capable of managing the data of numEnrolledStudents. Through the use of the methods getNumEnrolled and setNumEnrolled, any other class can modify the value of the class, but only through the reference of the school object,

Some well known languages that are Turing Complete from this paradigm are:

- Java
- C++
- Scala
- PHP
- Perl
- Swift

1.2.1.4 Multi Paradigm Languages

Some languages allow for the combination of OOP and Procedural Programming. In such paradigms, the code allows for both to be run at the same time and enjoys the benefits of both approaches, at the cost of increased design overhead of the project. Here is an example snippet of Python code that demonstrates both at the same time:

```
def findSumOfSquares(num1, num2):
return (num1 ** 2) + (num2 ** 2)


class Homework:
    def __init__(self, problem):
        self.problem = problem
```

```
    def problem(self, problem):
        self.problem = problem

HW = Homework("What is the sum of squares of 2 and 3?")
print(HW.problem)

print(findSumOfSquares(2, 3))

#####       Printed to Terminal     #####

What is the sum of squares of 2 and 3?
13
```

In the code example, we utilize Procedural Programming to create the findSumOfSquares function. Through the usage of OOP, we create a Homework object that has a single data member, a single problem. By accessing the problem within the Homework object, we are able to print it out, then use the function to solve it.

Some well known languages that are Turing Complete from this paradigm are shown below. Notice that some languages mentioned in the previous sections may show up in the list:

- JavaScript
- C++
- Python
- R
- Perl
- Fortran

Multi Paradigm languages have a lot of flexibility for the applications that they can be used to create. The top frontend frameworks for web development use Javascript as their main language including React, Vue, Svelte, and more. [JS IN FRONTED][https://www.simform.com/blog/javascript-frontend-frameworks/] Python is very popular for its legible and flexible code. With its libraries such as Tensorflow and Keras, Machine Learning and other AI subgenres are easier to implement than in other languages. The popular LLM ChatGPT is primarily written in python. [chatgpt uses python][https://enjoymachinelearning.com/blog/is-chatgpt-written-in-python/] R is another language that is popular for its data science capabilities. It is heavily used within the Sciences (alongside python) because of its simplistic syntax, as well as its numerous libraries for data analysis. [LINK TO MY PROJECT ON ML AND DL IN PYTHON AND R][INSERT LINKS] [R for data science][https://www.simplilearn.com/what-is-r-article]

NOT SURE IF I WANT TO KEEP THE FOLLOWING 2 SECTIONS OR NOT: FUNCTIONAL PROGRAMMING LANGUAGES, LOGIC PROGRAMMING LANGUAGES. I WOULD PROBABLY KEEP IT SIMPLE WITH 1-3 THINGS TO DESCRIBE ABOUT THE WAY THE LANGUAGE IS STRUCTURED W/ 1-2 CODE EXAMPLES AND EXPLANATIONS. (I ESTIMATE 3-5 PAGES PER SECTION DEPENDING ON SIZE OF EXAMPLES). I THINK I'VE SHOWN MY POINT, WHICH IS THAT THE DIFFERENT PARADIGMS ARE CAPABLE OF BEING TC DESPITE BEING SO DIFFERENT FROM EACH OTHER. IT MIGHT BE COOL/INTERESTING TO SHOW THE DIFFERENT THINKING INVOLVED FOR LANGUAGES LIKE PROLOG. I COULD ALSO COME BACK TO ACKNOWLEDGE CERTAIN ASPECTS OF LANGUAGE DESIGN PRINCIPLES HERE, WHICH WOULD REQUIRE FURTHER READING. I DO ADMIT THAT THIS IS SOMEWHAT OUTSIDE THE SCOPE OF THIS THESIS BECAUSE IM NOT EXACTLY LOOKING AT THE STYLE OF LANGAUGE OF PROTEUS, NOR HOW IT COMPARES TO THESE PARADIGMS TO CATEGORIZE IT.

1.2.1.5 Functional Programming Languages

describe design of these languages Some well known languages that are Turing Complete from this paradigm are:

- Lisp
- Haskell
- Elixir
- OCaml
- Go
- Rust

DESCRIBE SOME ACTUAL USAGE OF THESE PROGRAMS IN MODERN SOFTWARE DEVELOPMENT

RUST IS BEING USED TO TRY TO CREATE A NEW AND DIFF LINUX KERNEL. GO IS NEW POPULAR PROGRAMMING LANGUAGE FOR WEB DEVELOPMENT, DEVOPS, ETC.

1.2.1.6 Logic Programming Languages

The most well known Logic Programming Language is Prolog. describe design of prolog Describe how it basically relies on recursion to do stuff. Other languages exist, but are not popular when compared to the aforementioned languages in the previous sections.

IS PROLOG EVEN USED TODAY FOR STUFF IN INDUSTRY???

1.3 Proteus

The main goal of this thesis is to outline a proof demonstrating that a novel prototype language, Proteus, is TC. In this section, I will describe in detail what Proteus is.

1.3.1 Proteus Description

Proteus is a programming language and compiler being developed as a project for CSUN's Autonomy Research Center for STEAHM in collaboration with the NASA Jet Propulsion Laboratory (JPL). JPL system engineers needed a safer language to develop autonomous systems reliably, which is why Proteus was created. Proteus allows for the creation of different models: actors and hierarchical state machines. It is compiled to C++ with the C++17 standard [2].

Proteus is a programming language that follows the Actor model paradigm, which is somewhat related to the OOP paradigm. The difference lies in that Actor model allows for concurrent computation, while OOP generally runs sequentially. This means that parallelism is inherently existent in the language. [NEED SOURCE HERE] Furthermore, because of the design of the events and event queue for Actors, any code involving them is run sequentially. This means that Proteus also supports the Procedural Programming langauge paradigm. Thus, Proteus is a Multi Paradigm language that enjoys the ability to utilize features such as scope, recursion, and so forth.

1.3.1.1 Actors

Actors are independent entities within concurrent systems. By allowing several actors to operate independently, there is: no sharing of resources, concurrent runtime, and only interact amongst each other via a message system. Communication is asynchronous because the messages get buffered by the system until the recipient can handle them. Actors can send messages, modify local state, or create more actors based on the message handling.

1.3.1.2 Hierarchical State Machines

Hierarchical State Machines allow developers to model the system that they are developing for. These HSMs are an extension to the standard definition of a state machine as HSMs allow states to be HSMs themselves. This allows for simplification of the states and transitions amongst states allowing simpler models for usage in the real-world.

Actors have 0^+ HSMs while each HSM belongs to exactly one actor. Event Handlers are the

way that messages are sent amongst machines as well as how the machine perform state transitions. Actors and states are statically defined, which means that they cannot be created nor destroyed at runtime. When compiled, Actors and states are created as C++ structs.

1.3.2 Proteus Grammar

Below is the Grammar for Proteus. It outlines the command followed by the definition for writing the command. Anything outlined in single quotations indicates text to be written explicitly. It is to be read as: 'OPERATION' can be written as 'HOW TO WRITE THE OPERATION'.

```
Program: DefEvent* DefGlobalConst* DefFunc* DefActor+
DefActor: 'actor' ActorName '{' ActorItem* '}'
ActorItem: DefHSM | DefActorOn | DefMember | DefMethod
DefActorOn: 'on' EventMatch OnBlock
DefHSM:   'statemachine' '{' StateItem* '}'
DefState: 'state' StateName '{' StateItem* '}'
StateItem: DefOn | DefEntry | DefExit | DefMember |
            DefMethod | DefState | InitialState
DefOn: 'on' EventMatch OnBody
EventMatch: EventName '{' [VarName (',' VarName)*] '}'
OnBody: GoStmt | OnBlock
OnBlock: Block
DefEntry: 'entry' '{' Block '}'
DefExit: 'exit' '{' Block '}'
DefMember: Type VarName '=' ConstExpr ';'
DefMethod: 'func' FuncName FormalFuncArgs ['->' Type] Block
InitialState: 'initial' StateName ';'
Block: '{' Stmt* '}'
Stmt: IfStmt | WhileStmt | DecStmt | AssignStmt | ExitStmt |
```

```
            ApplyStmt | SendStmt | PrintStmt | PrintlnStmt
DefEvent: 'event' EventName '{' [Type (',' Type )*] }' ';'
DefFunc: 'func' FuncName FormalFuncArgs ['->' Type] Block
DefGlobalConst: 'const' Type VarName '=' ConstExpr ';'
ExitStmt: 'exit' '(' NUMBER ')' ';'
ReturnStmt: 'return' Expr ';'
DecStmt: Type VarName '=' Expr ';'
AssignStmt: VarName '=' Expr ';'
ApplyStmt: ApplyExpr ';'
SendStmt : HSMName '!' EventName ExprListCurly ';'
PrintStmt : 'print' ExprListParen ';'
PrintlnStmt : 'println' ExprListParen ';'
FormalFuncArgs : '(' [Type VarName (',' Type VarName)*] ')'
ExprListParen :'(' [Expr (',' Expr)*] ')'
ExprListCurly :'{' [Expr (',' Expr)*] '}'
Type: 'int' | 'string' | 'bool' | 'actorname' | 'statename' |
            'eventname'
GoStmt: JustGoStmt | GoIfStmt
JustGoStmt: 'go' StateName Block
GoIfStmt: 'goif' ParenExpr StateName Block
            ['else' (GoIfStmt | ElseGoStmt)]
ElseGoStmt: 'go' StateName Block
IfStmt: 'if' ParenExpr Block ['else' (IfStmt | Block)]
WhileStmt: 'while' ParenExpr Block
ParenExpr: '(' Expr ')'
ConstExpr: IntExpr | BoolExpr | StrExpr
Expr: ValExpr | BinOpExpr | ApplyExpr
```

```
BinOpExpr: ValExpr BinOp Expr

BinOp: '*' | '/' | '%' | '+' | '-' | '<<' | '>>' | '<' | '>' |
            '<=' | '>=' | '==' | '!=' | '^' | '&&' | '||' | '*=' |
            '/=' | '%=' | '+=' | '-=' | '<<=' | '>>=' | '^='

ApplyExpr: FuncName ExprListParen

ValExpr: VarExpr | IntExpr | StrExpr | BoolExpr | ActorExpr |
            StateExpr | EventExpr | ParenExpr

VarExpr: VarName

IntExpr: NUMBER

StrExpr: STRING

BoolExpr: BOOL

ActorExpr: 'actor' ActorName

StateExpr: 'state' StateName

EventExpr: 'event' EventName

StateName: NAME

ActorName: NAME

FuncName: NAME

VarName: NAME

EventName: NAME
```

Looking at the grammar is similar to looking at the pieces of a puzzle without actually arranging the pieces together. Below is an example written in Proteus code that showcases Actors and HSMs in a system.

There are a total of 3 events: POWER_ON which accepts a boolean as input, POWER_OFF, and NEXT with the latter two not accepting an inputs. There are 2 actors: Main and Driver. Main has a single state machine with 2 states: On and Off. Main defines an internal boolean for whether Mode2 is enabled. By default it is initialized to false. The HSM within Main is initalized to Off, and switches to On when the POWER_ON event is registered. Furthermore, it updates the value for

Mode2 being enabled with the input for POWER_ON. When turned On, there is a defined Mode1 that is the initial mode of the machine. It then defines what the machine does when it is turned on and off. In both cases, it outputs a message indicating the status of the power state of the machine (on prints on, and off prints off). Mode1 prints to the output the current Mode, and then has logic determining what to do when the NEXT message is received. If the machine has mode2_enabled set to true, then it should should go to Mode2. Mode2 simply prints the current mode when it is entered. The second actor is the Driver. It determines the actions to be taken by Main in a series of messages (events) that are broadcasted from its internal statemachine. Upon turning on the machine, it will send the event for Main to turn on with an input of true. Then it sends Main the NEXT event twice. It then tells Main to power off with the POWER_OFF event.

```
event POWER_ON {bool};
event POWER_OFF {};
event NEXT {};
actor Main {
    bool mode2_enabled = false;
    statemachine {
        initial Off;
        state Off {
            on POWER_ON {x} {go On {mode2_enabled = x;}}
        }
        state On {
            initial Mode1;
            entry {println(\turning on");}
            exit {println(\turning off");}
            on POWER_OFF {} {go Off {}}
            state Mode1 {
                entry {println(\mode 1");}
```

```
                on NEXT {} {goif(mode2_enabled) {Mode2 {}}}
            }
            state Mode2 {
                entry {println(\mode 2");}
            }
        }
    }
}
actor Driver {
    statemachine {
        entry {
            Main ! POWER_ON {true};
            Main ! NEXT {};
            Main ! NEXT {};
            Main ! POWER_OFF {};
        }
    }
}
```

From the above example, we can see the OOP and Procedural Programming properties that Proteus is capable of. In fact, we can see the property of scope in action. The local variable mode2_enabled is only accessible within Main and the HSM within Main. Furthermore, we see that the order of events is run in sequence. Main accepts these events in the order received in the internal event queue, and is capable of responding based on the internal state conditions as well as the event received. When this code is run, the output is seen below:

```
turning on
mode1
mode2
```

```
mode1
turning off
```

The goal of this Thesis is to analyze this language to prove that it is TC. This is done by looking at the grammar and type of programming paradigm. With this understanding, we will also look at different approaches for proving Turing Completeness. Afterwards, we will demonstrate that Proteus is in fact turing complete using some of the methods seen in the next chapter.

Chapter 2

Different Approaches for Proofs to Demonstrate Turing Completeness

2.1 Overview

We will be exploring the different approaches to demonstrate TC for different systems. I have outlined the approaches based on their respective discipline, increasing in abstraction. With each discipline comes a more theoretical view and understanding of TMs and TC systems. My intention is to add clarity on the logic for these proofs/techniques. For example, in the Computer Engineering perspective, TM is created from its mechanical properties through the usage of logic gates. This is vastly different compared to how Mathematicians show TC, which is through the use of Lambda calculus – a model for representing mathematical logic. All proofs are equivalent in goal, however. These are not the only perspectives and types of proofs for showing Turing Completeness as well as TMs. This is simply a survey into what TMs and Turing Completeness looks like across the disciplines.

2.2 Computer Engineering

In this section, we will analyze what a TM looks like from a physical perspective. This may seem contradictory because the TM is described as a theoeretical machine. But in fact, the very computers that we use today are are capable of processing TC systems through the usage of programming languages. This means that they are limited TMs, because they are bounded only in memory. In this approach, we will look at the core components of Computer Engineering to create a TM.

2.2.1 Logical Design of a TM

To define what a TM does, we must explore what it is capable of. Recall Theorem 1 in 1.1.3, ”Every effectively calculable function can be computed by a Turing Machine.” Every effectively calculable function, as Turing and Church understood, was any mathematical calculation. This means that a TM must have some ability to perform any operation on numbers, such as the basic operations of addition, subtraction, multiplication, and division. Furthermore, they must be capable of combining these together to form more complex operations such as exponential arithmetic. Beyond the mathematical aspect, they must allow for some sort of logic handling. [BIOLOGY PAPER ON TM]

2.2.1.1 Architecture

Looking at modern day computer architecture, there are several components that work independently but operate concurrently. It is based off of the Modified Harvard Structure which is a variation of the Harvard computer architecture and Von Neumann architecture. It combines both approaches towards computer architecture to handle many tasks that were difficult to handle using one of either architecture.

The von Neumann Architecture which has a centralized CPU to handle tasks for the computer. All processes are handled by the CPU directly. It contains several parts inside for processing data. Inside the CPU is an ALU with registers, as well as a Control Unit. The ALU processes arithmetic and logical computation, with the assistance of registers to store data at each step. The Control unit determines the commands to be given to the ALU and other parts of the computer. There is an associated Memory Unit which is where the bulk of memory storage lies. Outside of the CPU are the Input and Output devices.

citation for von neumann architecture image

The von Neumann architecture has several limitations, with one of the biggest criticisms being that it is bottlenecked by the throughput between the CPU and memory. Essentially, the CPU will eventually have more processing power than the bus can handle to write/read from memory. This

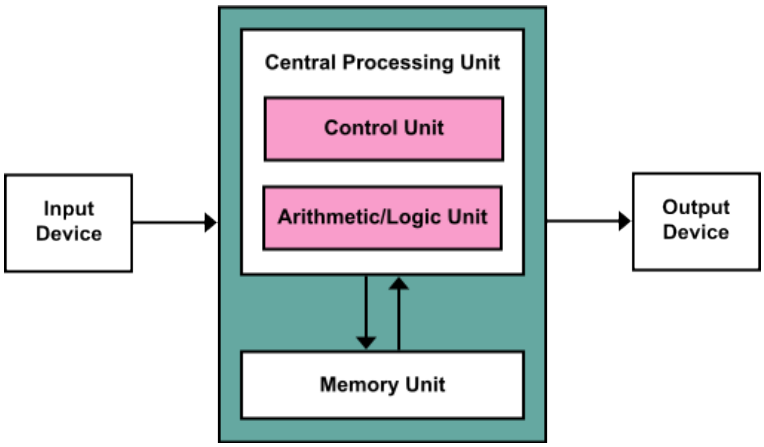

Figure 2.1: Von Neumann architecture.

causes the CPU to wait until the bus is freed to continue processing. As an alternative, we will now look at the Harvard Architecture

The Harvard architecture looks at the problem as says that if the CPU is too large and complex, then each individual component should be separated. This allows for the tasks to be distributed evenly amongst the several smaller components like the ALU and Instruction memory as opposed to having them live inside the CPU. The CPU is capable of simultaneous reads and writes. However, a similar bottleneck occurs where the bus connecting each of the components is the limiting factor. Below is a diagram demonstrating the Harvard Architecture.

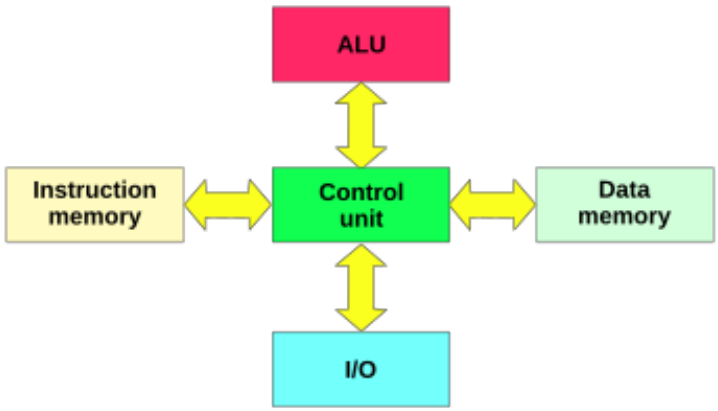

Figure 2.2: Harvard architecture.

citation for harvard architecture image

However, modern day computers utilize a mixed computer architecture called the Modified Harvard architecture. It combines both architectures into a single model. This usually is of the form of separating the components of the computer, but allowing several smaller memory caches for the CPU. This is why modern computers utilize several components such as the CPU, GPU, Main Memory (SDD or HDD) and so forth. Furthermore, this advancement allows for paralellism or multi-core systems to arise. Some notable examples include the NVIDIA RTX 4080 which has over 8000 cores SOURCE or for the intel i9-13900ks CPU to have 16 cores SOURCE. By allowing each one to independently operate, but still have the CPU as the ”brains” of the operation. We will now dive slightly deeper into the discussion to see what lies beneath these components within modern computer systems.

2.2.1.2 Logic Gates

The basic building blocks for devices such as the ALU, CPU, and such are logic gates. These are simplistic logical components that allow for processing of data and performing operations on them.

The core of the CPU relies on the ALU. The ALU is contains registers which simply hold data inside. Registers are associated with an address for referencing purposes.

Within the ALU there are smallers components that perform specific operations such as addition and subtraction. These smaller components utilize logical gates such as the OR gate to compute the result with the given input. See the example below for a 2-Bit ALU that can process OR, AND, XOR, and addition calculations.

img src

2.2.2 Constructing the TM

With the ability to construct logic gates, we can create more complex components such as the ALU, CPU, and more. Accompanied with the ability to store memory, as well as have a way to interact with the system through Input and Output, we are able to create a functional computer. In

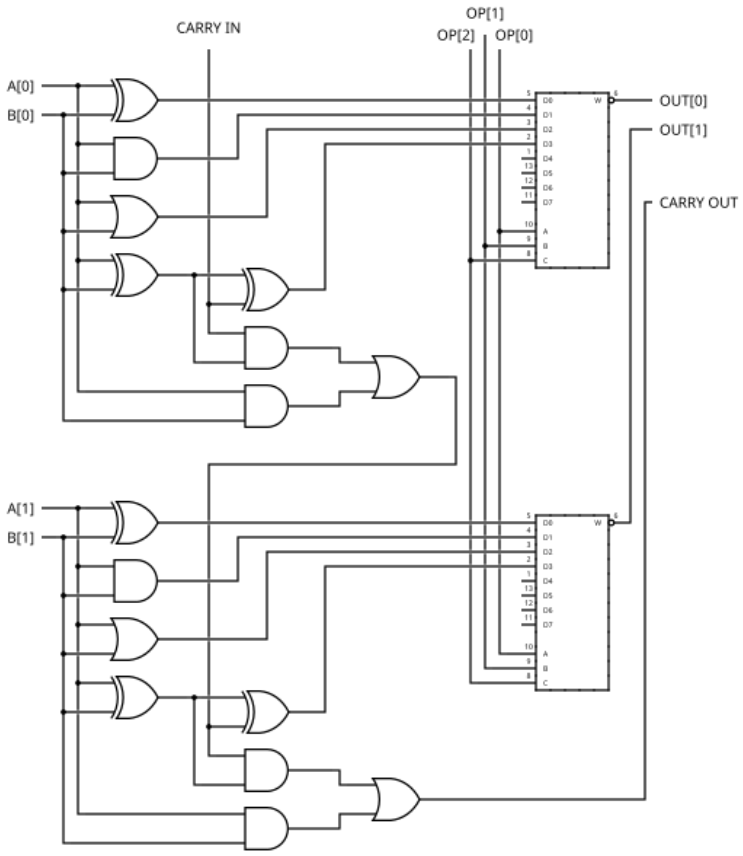

Figure 2.3: A 2-Bit ALU.

the following examples, developers have successfully created computers in the well-known video games of Minecraft and Terraria respectively, Computer in Minecraft Computer in Terraria. In fact, we will see later on that the computer built inside Terraria is verifiably TC via a method discussed in 2.3.2.1.

This means that to construct a TM on a physical level (of course alleviating the restriction of unbounded memory), these would be the minimum requirements.

SRC - https://www.nand2tetris.org/ SRC -

2.3 Computer Science

In this section, we will conceptualize what a TM looks like under the lens of Computer Science. There are 2 main perspectives: that of the Automata Theory and the Software Engineering

approach. The Automata Theory approach utilizes theoretical designs more reminiscent of those listed by Turing and Gavin. The Software Engineering approach instead applies it to a problem to showcase Turing Completeness via programs and code.

2.3.1 Automata Theory

We will now abstract from the physical understanding of how to create a TM, to creating a theoretical one using Automata theory. In automata theory, Turing Machines are described using logical notation. The definition of a TM is stated within [3].

Definition 1. *A Turing Machine M is defined by:*

$$M = (Q, \Sigma, \Gamma, \delta, q_0, \square, F)$$

where:

Q *is the set of internal states,*

Σ *is the input alphabet,*

Γ *is the finite set of symbols called the tape alphabet,*

δ *is the transition function,*

$\square \in \Gamma$ *is a special symbol called the blank,*

$q_0 \in Q$ *is the initial state,*

$F \subseteq Q$ *is the set of final states.*

The transition function δ is defined as

$$\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}.$$

This means that for a given δ transition with inputs $q \in Q$ and $a \in \Gamma$, the tape will move to another state $x \in Q$, write nothing to the tape (indicated by $\square$) or some symbol $y \in \Gamma$, and choose to move the tape head Left one cell, Right one cell, or to Stay at the current cell. An example transition can

be written:

$$\delta(q_0, a) = (q_1, d, R)$$

where the internal state is q_0, and we read input token a. After the transition, we have internal state q_1, wrote symbol d onto the tape, and moved to the right one cell. See the below diagram demonstrating this change:

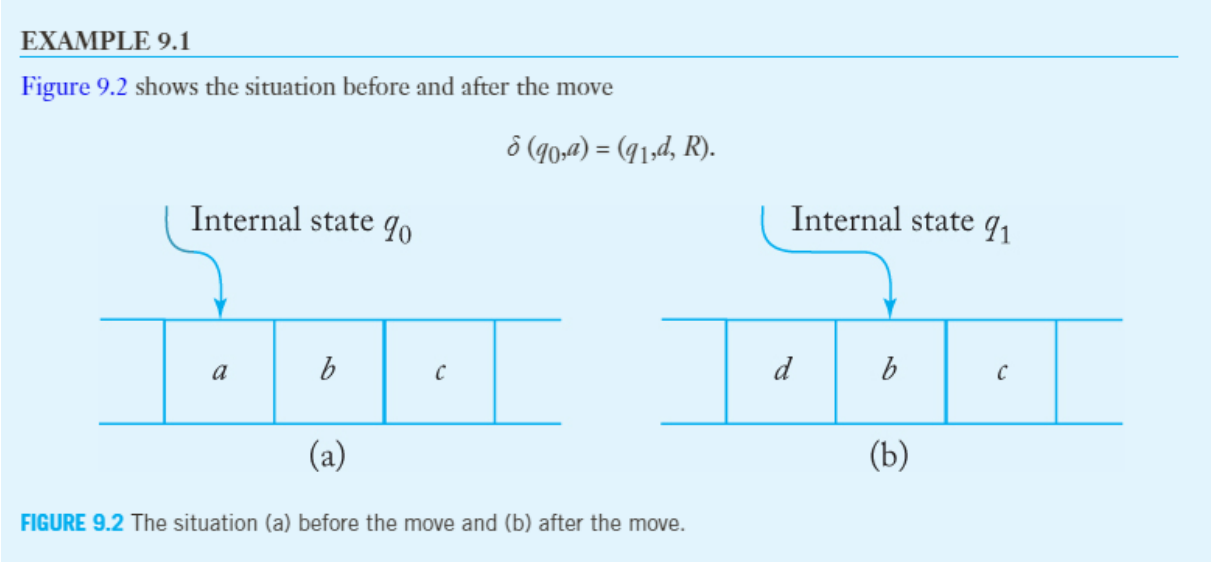

Figure 2.4: Delta transition example from [3].

Recall Figure 1.1 which represents a simplistic TM. In formal nomenclature, it can be written as follows:

$Q = \{q_0, q_1, q_2\}$ with associated labels {Even, Odd, Halt}

$\Sigma = 0, 1$

$\Gamma = 0, 1$

$F = \{q_2\}$

$q_0 \in$ Q as the initial state

and

$$\delta(q_0,0) = (q_2,1,S),$$

$$\delta(q_0,1) = (q_1,\square,R),$$

$$\delta(q_1,0) = (q_2,0,S),$$

$$\delta(q_1,1) = (q_0,\square,R).$$

With this now understood, I will talk about the two general lines of thought for demonstrating a system TC.

SRC - The Analysis and Research on Computational Complexity

2.3.1.1 Formal Technical Writing

$\lambda\gamma\tau$ etc. being used to describe TC

SRC - The Game Description Language is TC

SRC - Java Generics are Turing Complete

SRC - Turing Completness and Sid Meier's Civilization

2.3.1.2 Gamified Writing

$\lambda\gamma\tau$ etc. being used to describe TC, but based on a very verbose and interactive style

SRC - Magic: The Gathering is TC

2.3.2 Software Implementation

As opposed to the various theoretical approaches seen previously in section 2.3.1, this section outlines a different perspective. Instead of constructing a TM within the compounds of the system, an equivalent proof is to implement a program that demonstrates TC. By implementing any known TC program successfully implies that the overall system is TC.

One such implementation would be to create a functional example of a known TC cellular automata. Cellular automata are models of computation which use grids of cells. Each cell contains a finite number of states, belonging to only one at any given time. There are rules that determine

what state a cell should become. These rules are applied to all cells simultaneously, and thus form the next step in the sequence. These steps are made sequentially to show the changes over time. This makes all cellular automata 0-player games, meaning after an initial configuration there is no further input from the user. With the work from Stephen Wolfram and other researchers such as Matthew Cook, some of these rules of cellular automata have been shown to be TC. Famous examples of cellular automata include Conway's Game of Life and Rule 110.

https://mathworld.wolfram.com/CellularAutomaton.html wiki cell automata

Cellular automata are sorted into 4 classes:

- Class 1: Nearly all initial patterns evolve quickly into a stable, homogenous state. Any randomness in the initial pattern disappears.
- Class 2: Nearly all initial patterns evolve quickly into stable or oscillating structures. Some of the randomness in the initial pattern may filter out, but some remains. Local changes to the initial pattern tend to remain local.
- Class 3: Nearly all initial patterns evolve in a pseudo-random or chaotic manner. Any stable structures that appear are quickly destroyed by the surrounding noise. Local changes to the initial pattern tend to spread infinitely.
- Class 4:Nearly all initial patterns evolve into structures that interact in complex and interesting ways, with the formation of local structures that are able to survive for long periods of time. Class 2 type stable or oscillating structures may be the eventual outcome, but the number of required to reach this state may be very large, even when the initial pattern is realtively simple. Local changes to the initial pattern may spread indefinitely.

Wolfram conjectured that many class 4 cellular automata are capable of universal computation. Both Conway's Game of Life and Rule 110 exhibit "Class 4 behavior" and have been proven to be Turing Complete.

Wiki src SRC - Cellular Automata: A Discrete Universe by Andrew Ilachinski

2.3.2.1 Conway's Game of Life

Conway's Game of Life is a 2D grid of cells extending infinitely in the x and y directions. Each cell contains only 2 states: Alive (On) or Dead (Off). The rules of CGoL are simple:

1. Any live cell with fewer than two live neighbors dies. (Underpopulation)
2. Any live cell with two or three live neighbors lives on to the next generation. (Survival)
3. Any live cell with more than three live neighbors dies. (Overpopulation)
4. Any dead cell with exactly three live neighbors becomes a live cell. (Reproduction)

They are demonstrated in the following graphic:

src for graphic

CGoL is considered undecidable. This is because given any initial pattern and a desired pattern at some later generation, there is no algorithm to determine whether the desired pattern will exist. As such, it is analogous to the Halting Problem.

2.3.2.2 Rule 110

Whereas CGoL is created on a 2D plane, Rule 110 lives in the 1D space. There is an infinite tape of cells that each may exist in one of two states: 0 or 1. By looking at three cells in series, one can find what the next state of the middle cell will be. Below are the rules for Rule 110:

1. 111 makes 0
2. 110 makes 1
3. 101 makes 1
4. 100 makes 0
5. 011 makes 1

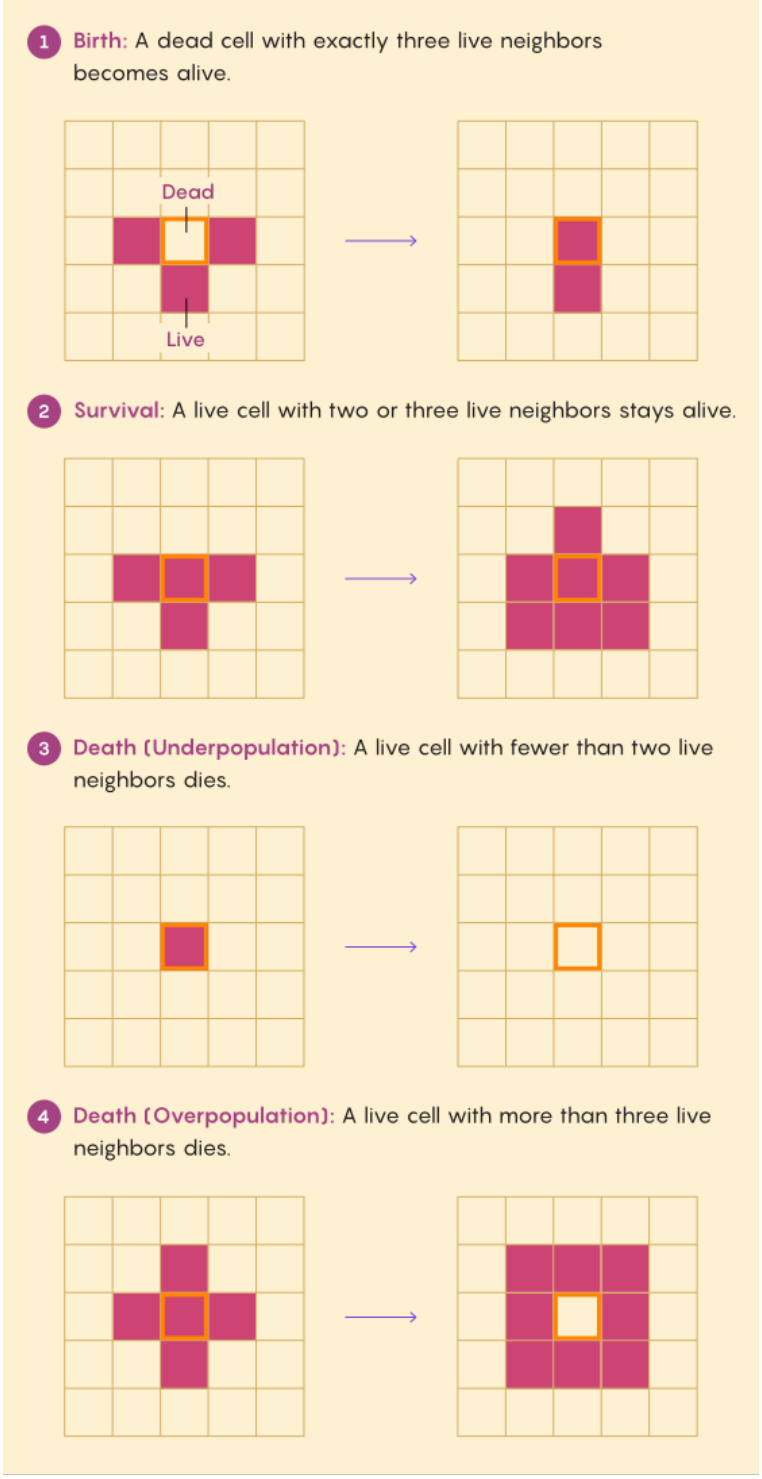

Figure 2.5: rules of conway's game of life visualized

6. 010 makes 1

7. 001 makes 1

8. 000 makes 0

Here is an associated graphic:

Figure 2.6: Rules for Rule 110

Rule 110 is one of the simplest TC system that is known. This makes it a relatively easy system to create to demonstrate Turing Completeness as opposed to CGoL.

SRC - WIKI page for graphic

2.3.2.3 Programmable Calculator

An entirely different approach to create a program that demonstrates Turing Completeness is to model the behavior of TMs directly. This means that you create a system that does everything that a TM can do. Recalling the Church-Turing Thesis (Theorem1), it must be able to calculate any function. In a basic sense, this means that the system is capable of:

- Reading/Writing memory
- Elementary Arithmetic/Logical operations
- Conditional Logic
- Looping Logic

Programmable Calculators meet all of these requirements. By being able to store values into variables which can be referenced later, it can read/write memory. Because it is a calculator, it is capable of performing arithmetic operations. If statements and while loops are sufficient for handling the conditional and looping logic. A programmable calculator thefore is TC.

A simplistic set of steps is outlined below:

1. Start by making a basic arithmetic calculator.

2. Then add the ability to store values into variables.

3. Afterwards, create functionality for if statements, allowing boolean logic.

4. Finally, create looping logic with while statements.

TC language in Ruby using Bable-Bridge

This much simpler approach is easier to digest and reason out with some software engineering design principles.

2.3.2.4 Interpreter for a known Turing Complete language

Alternatively, to show a programming language is TC, one can create an interpreter for a known TC language. Many programming languages feature complex grammars and rulesets, which is why TC esoteric programming languages are preferred. In fact brainfuck, as seen in section **??**, is used to demonstrate TC for its concise ruleset.

SRC - Add 2 sources showing a system is TC by creating an interpreter.

2.4 Mathematics

In this section, I will take a look at the mathematical system that is most well known for being TC, Lambda Calculus. This is an abstract form of understanding functions and their capabilities. It was actually shown to be TC by Church himself.

SRC - CHURCH SHOWING LAMBDA CALC IS TC.

2.4.1 Lambda Calculus

go though explanation of Lambda calc. Discuss the proof and how it can model a TM.

Chapter 3

Proteus is Turing Complete

This section will describe how we will construct 1+ proofs for showing that Proteus is TC.

3.1 The Proteus Grammar

List out the grammar and describe how it can be read, line by line. Probably have a small example of a program that shows most of (maybe all) of the functionality with Proteus

3.2 Initial Thoughts based off of the grammar

List out some ideas that led to the idea that Proteus could be TC

3.3 Proof Outline

Theorems that would be used Steps for proof outline

3.4 Proof v1

Formal proof that follows the outline

3.5 Implementing Rule 110

Implementation of Rule 110 that is a demonstration of TC.

Chapter 4

Conclusion

Succinctly describe what was the several techniques. Compare and constrast them? Perhaps a discussion section?

Bibliography

[1] R. Gandy, “Church’s Thesis and Principles for Mechanisms,” The Kleene Symposium, pp. 123–148, Jun. 1980.

[2] B. McClelland, “Adding Runtime Verification to the Proteus Language,” CSUN, May 2021.

[3] P. Linz, An Introduction to Formal Languages and Automata. Jones & Bartlett Learning, 2016.