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.

INSERT DIAGRAM OF A SIMPLE TURING MACHINE THAT PERFORMS BINARY ADDITION

### 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.

INSERT DIAGRAM OF A TM W/ AN ORACLE SOLVING THE HALTING PROBLEM

The reason the oracle is not considered more powerful is because in practice (i.e. reality), is

because there is no such all-knowing source to retrieve information from. As a result, I will look only at TMs 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.

INSERT DIAGRAM OF A SIMPLE UNIVERSAL TURING MACHINE THAT PERFORMS BINARY ADDITION basically it just takes another input which are the instructions

### 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 when comparing the computing power of humans to TMs.

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.

[INSERT REFERENCE TO GANDY 1980 PAPER PG 123-148].

## 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 Brainfuck](https://gist.github.com/roachhd/dce54bec8ba55fb17d3a)

The 8 instructions are as follows:

| | |
|---|---|
| $>$ | 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

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
```

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/] Beyond 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

describe design of these languages DESCRIBE Java CAPABILITIES

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

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

#### 1.2.1.4 Mixed/Multi Paradigm Languages

describe design of these languages DESCRIBE PYTHON CAPABILITIES

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

DESCRIBE SOME ACTUAL USAGE OF THESE PROGRAMS IN MODERN SOFTWARE DEVELOPMENT

PYTHON HEADS THE CHARGE FOR AI AND ITS SUBDISCIPLINES OF ML,DL,AND GEN AI R is used for data science Javascript is still utilized for a large amount of web dev

NOT SURE IF I WANT TO KEEP THE FOLLOWING 2 SECTIONS OR NOT. I WOULD PROBABLY KEEP IT SIMPLE WITH 1-3 THINGS TO DESCRIBE ABOUT THE WAY THE LANGUAGE IS STRUCTURED. MAYBE 1-2 EXAMPLES WITH CODE AND EXPLANATION AS WELL (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 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. IT WOULD BE COOL TO SAY THOUGH, SINCE THE GOAL IS TO SHOW THIS LANGUAGE IS TC, WHICH REQUIRES ME TO LOOK AT THE LANGUAGE ON A DEEPER LEVEL.

#### 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

DESCRIBE WHAT PROTEUS IS, PURPOSE

### 1.3.1 Motivations for Turing Completeness

WHY DO WE WANT TO STUDY THIS LANGUAGE

In this thesis, we will look closely at the several methods to showing that a given system is TC. Afterwards, we will demonstrate that Proteus is in fact turing complete using some of the methods seen here.

# Chapter 2

# Different Approaches for Proofs to Demonstrate Turing Completeness

## 2.1 Turing Complete Proofs Overview

We will be exploring the different approaches to demonstrate TC

All proofs are equivalent in goal. The difference lies in their usability for specific domains.

## 2.2 Computer Engineering

### 2.2.1 Logical Design from a perspective of physical requirements

Gates & Wiring NOR, ADDER, OR, NOT gates

## 2.3 Computer Science

### 2.3.1 State Machines

#### 2.3.1.1 Formal Technical Writing

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

#### 2.3.1.2 Gamified Writing

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

### 2.3.2 Software Implementation

Demonstrate an example of a problem/scenario which shows TC.

#### 2.3.2.1 Conway's Game of Life

Classic example to implement

#### 2.3.2.2 Rule 110

Simpler example to implement

#### 2.3.2.3 Calculator with Store Value

More challenging to implement than the previous two.

### 2.3.3 Interpreter for a known Turing Complete language

Implement an interpreter for a known TC language in a given language. Must faithfully and accurately simulate the behavior of the given language to be interpreted.

typically, we will implement a simple TC language such as brainfuck

## 2.4 Mathematics

### 2.4.1 Lambda Calculus

Basically a simplistic view of math functions. can be shown as TC through construction of certain concepts.

# 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] Baker, N. 1966, in Stellar Evolution, ed. R. F. Stein & A. G. W. Cameron (Plenum, New York) 333