Cflat: A Turing Complete Language

by

William Sanville

A Project Submitted to the Graduate

Faculty of Rensselaer Polytechnic Institute

in Partial Fulfillment of the

Requirements for the degree of

MASTER OF SCIENCE

Major Subject: Computer Science

Approved:

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Eugene Eberbach, Thesis Adviser

Rensselaer Polytechnic Institute

Hartford, CT

August, 2011

(For Graduation December, 2011)

**CONTENTS**

[1. Introduction 7](#_Toc301192017)

[1.1 What is Cflat? 7](#_Toc301192018)

[1.2 Cflat Compiler 7](#_Toc301192019)

[1.3 Research Question 8](#_Toc301192020)

[1.4 Motivation 8](#_Toc301192021)

[2. Theory 9](#_Toc301192022)

[2.1 Turing Machine Definition 9](#_Toc301192023)

[2.2 Significance of Turing Machines 10](#_Toc301192024)

[2.3 Universal Turing Machine 10](#_Toc301192025)

[2.4 Turing Completeness 11](#_Toc301192026)

[3. Methodology 12](#_Toc301192027)

[3.1 Implementing a Universal Turing Machine 12](#_Toc301192028)

[3.2 Class Diagram 13](#_Toc301192029)

[3.3 Turing Machine File Format 14](#_Toc301192030)

[3.3.1 Example Transition Functions 15](#_Toc301192031)

[4. Results and Discussion 16](#_Toc301192032)

[4.1 Universal Turing Machine Inputs 16](#_Toc301192033)

[4.2 Example Turing Machines 16](#_Toc301192034)

[4.2.1 Machine A 17](#_Toc301192035)

[4.2.2 Machine B 19](#_Toc301192036)

[4.2.3 Machine C 21](#_Toc301192037)

[4.3 Running the Turing Machines 22](#_Toc301192038)

[4.3.1 Machine A 22](#_Toc301192039)

[4.3.2 Machine B 22](#_Toc301192040)

[4.3.3 Machine C 23](#_Toc301192041)

[4.4 Simulating Arbitrary Turing Machines 23](#_Toc301192042)

[4.5 Analysis 24](#_Toc301192043)

[5. Conclusion 25](#_Toc301192044)

[5.1 Summary of Results 25](#_Toc301192045)

[5.2 Significance 25](#_Toc301192046)

[5.3 Further Research 25](#_Toc301192047)

[6. References 27](#_Toc301192048)

[7. Appendix 28](#_Toc301192049)

[7.1 Complete Cflat Source Code 28](#_Toc301192050)

LIST OF TABLES

Table 3.1 – Turing Machine file format, lines 0 – 5 – p.14

Table 3.2 – Example transitions, lines 6+ – p.15

Table 4.1 – Inputs and results for Machine A – p.22

Table 4.2 – Inputs and results for Machine B – p.22, p.23

Table 4.3 – Inputs and results for Machine C – p.23

Table 4.4 – Core transition functionality – p.23

LIST OF FIGURES

Figure 3.1 – Class diagram for the Cflat Universal Turing Machine p.13

Figure 4.1 – Source for Machine A p.17

Figure 4.2 – Graphical representation of Machine A p.18

Figure 4.3 – Source for Machine B p.19

Figure 4.4 – Graphical representation of Machine B p.20

Figure 4.5 – Source for Machine C p.21

Figure 4.6 – Graphical representation of Machine C p.22

ABSTRACT

*Modern object oriented programming languages provide rich features such as classes, inheritance, abstraction, encapsulation, and more. As a result, new programming languages are assumed to have these features. While programmers of today are concerned about these features and solving real world problems, the humble beginnings of a programming language often go unnoticed. Given that the modern day model of computation is inspired so heavily by Alan Turing’s writings in the 30s, this begs the question, do new programming languages carry the same expressive power as the theoretical Turing Machine? The research addressed here will prove the Turing completeness of Cflat, a programming language implemented in 2010. That is, Cflat will be proved to have the same expressive power as a Turing Machine. By implementing a Universal Turing Machine in the Cflat language, we can prove the language’s status as a Turing complete language. In practice, implementing a Universal Turing Machine in Cflat should be no more difficult than implementing a comparable program in other general purpose languages, such as C++, C#, or Java, given that Cflat supports arrays, looping, classes, and dynamic memory allocation.*

# Introduction

## What is Cflat?

Cflat is a general purpose, object oriented programming language and compiler implementation written in C#. The goal for the language is to provide a simple, yet full featured implementation for an object oriented programming language that supports classes, abstraction, inheritance, information hiding (encapsulation), recursion and more. The language and compiler implementation was developed in 2010 by Tim Goric, Will Sanville, and Matt Simmons. Syntactically, Cflat is very similar to existing object oriented languages, such as Java or C#.

The language has built in support for basic I/O operations, such as printing to a console window, reading user input from the console, and reading text files. Cflat also supports primitive data types, built in global functions, support for classes with member variables and methods, arrays of primitive types and classes, and is a type safe language. Cflat has a number of built in data types, which include objects, integers, real numbers (decimals), boolean values, strings, characters, arrays, and classes. Cflat also has a number of built in functions for performing string manipulation and parsing, as well as methods for retrieving lengths of strings and arrays.

## Cflat Compiler

The Cflat compiler is a multi-pass compiler, which performs three passes over the source code in order to verify the input program is indeed a valid Cflat program. The Cflat compiler relies on a scanner generator and parser generator developed by Queensland University of Technology, which are both licensed under a Free-BSD license.

The scanner generator used by the Cflat compiler is the Gardens Point Scanner Generator (GPLEX) [1]. GPLEX the takes an input file similar to the format used with the traditional implementation of LEX, and is used to generate a scanner in the C# language. The parser generator used by the Cflat compiler is the Gardens Point Parser Generator (GPPG) [2]. GPPG takes an input file similar to that used in other popular parser generators such as YACC and BISON. GPPG generates a LALR(1) parser in C#. The result of running the scanner and parser on a sample Cflat program is an instance of an abstract syntax tree (AST), in which a node is created for each production in the grammar. The AST nodes represent all of the constructs that make up the input program, such as a loop statement, variable or method declaration, and more. The AST is then walked several times in both the semantic analysis and code generation phases.

The Cflat compiler generates code in the target language CIL. CIL, or Common Intermediate Language, is a low level, stack based intermediate language developed by Microsoft [3]. CIL is the target language for many of the existing .NET languages used today, such as C#, F#, and Visual Basic.NET, and therefore, Cflat may interoperate with these languages seamlessly. CIL, in concept, is very similar to the bytecode generated by a Java compiler. The resulting CIL code from the Cflat compiler will run on any machine with Microsoft’s .NET Framework, or on a machine with the Mono runtime. Thus, a Cflat program, when run, is JIT (just in time) compiled into the target machine code by the .NET Framework (or Mono Framework).

## Research Question

Given that Cflat was designed to be a general purpose programming langue, it should give a programmer the expressive power needed to solve any computable problem. Thus, the goal of the research addressed here is to provide a yes or no answer to the question “Is Cflat a Turing complete language?” The approach used to answer this problem is to implement a Cflat program that can do exactly what a Turing Machine can do. That is, implement a Universal Turing Machine.

## Motivation

Implementing a Universal Turing Machine in Cflat was chosen because it involves writing a real program in Cflat. The goal was to provide a proof of Turing completeness, and at the same time, discover and fix as many bugs as possible in the Cflat compiler implementation. The resulting Cflat program involves classes with member variables and functions, arrays, looping, file I/O, string parsing and manipulation, and more. Putting these features to use uncovered flaws, and fixing them greatly improved the quality of the Cflat compiler.

# Theory

## Turing Machine Definition

Before determining whether or not Cflat is a Turing complete language, we first must settle upon a rigid definition of what a Turing Machine is, and what it means to be Turing complete. The Turing Machine definition used throughout the research is based off of the definition provided by Hopcroft, Motwani, and Ullman [4]. The definition of the Turing Machine has the following properties:

* A finite set of states that the machine can be in
* Unlimited storage, known as a tape
  + The tape is divided into cells, each which hold a single symbol
* A finite length input, that is made up of the symbols from the tape alphabet, which denotes the legal symbols that the tape can hold
* The tape initially contains the input
  + The tape has a head, which denotes the current cell on the tape that is being operated on
  + The tape head is initially at the left most input symbol
* The machine has a set of transition functions, which is a cross product of the current state and the symbol denoted by the tape head
  + A transition function will specify a new state to change to, a new symbol to write in the current tape cell, and a direction to move the tape head, either left or right.

Formally, a Turing Machine can be defined as a tuple made up of the following components:

Turing Machine, M = (Q, ∑, Γ, δ, q0, B, F)

Where:

**Q** is a set of states.

**∑** is the set of valid input symbols.

**Γ** is the set of valid symbols for the tape, such that∑ is a subset of Γ.

**δ** is the transition function. The transition function takes two arguments, the current state, and the current tape symbol, and it returns a new state to transition to, a symbol to replace the current tape symbol, and a direction to move the tape head. The direction is either Left or Right.

**q0** is the starting state of the machine. q0 must be an element of Q.

**B** is the blank symbol for the tape. B must not be an element of ∑.

**F** is the accepting states. F must be a subset of Q.

The above definition for a Turing Machine for our purposes is implemented in a text file format described in section 4.1.

## Significance of Turing Machines

The concept of a Turing Machine was first coined by Alan Turing, a mathematician in 1930s [5]. The original purpose of a Turing Machine was to serve as an abstraction to reason about what can be solved by computation. Since then, it has been used extensively by computer scientists and has inspired many different models for computation. Since its inception, the Church-Turing thesis states that any computable problem can be done so by a Turing Machine. While it is possible to disprove the Church-Turing thesis with a counterexample, no example has been shown, and many computer scientists accept the Church-Turing thesis as correct [6].

A Turing Machine is equivalent to a modern day computer, except for the notion that a Turing Machine has infinite memory. Clearly, a modern computer has a finite amount of memory, and thus, a Turing Machine is equivalent to a modern day computer which has infinite memory [7]. Despite the infeasibility of infinite memory for a real world computer, the Turing Machine still proves a useful abstraction about what is feasibly computed.

## Universal Turing Machine

A Turing Machine that is universal is a special subset of a Turing Machine that accepts an algorithm and input data. A Universal Turing Machine must be powerful enough to simulate the behavior of the given algorithm on its input data [8]. If we suppose that the first input to the Universal Turing Machine is the encoding of another Turing Machine, we can conclude that a Universal Turing Machine is a machine that can emulate any Turing Machine.

The concept of a Universal Turing Machine has served as the basis for modern day machines with stored programs [8]. For example, modern day machine running an operating system contains many stored programs loaded by the end user, and the operating system of course itself is made up of stored algorithms and programs.

## Turing Completeness

The notion of Turing completeness is a classification of languages which demonstrates whether or not a language is equivalent to the computational power of a Turing Machine [9]. Thus, we can say that for a language to be Turing Complete, it must be able to emulate what a Turing Machine can do. A Universal Turing Machine is trivially Turing Complete, by definition. With the classification of Turing complete languages, we can definitively state whether or not a given programming language can be used to solve real world computational problems.

# Methodology

## Implementing a Universal Turing Machine

The approach used to prove the Turing completeness of Cflat is to implement a Universal Turing Machine in Cflat. This will produce an executable program which takes two inputs on the command line, a Turing Machine representation, and an input string. The Cflat Universal Turing Machine will then simulate the given Turing Machine’s behavior by first writing the input string on the simulated machine’s tape, and then will simulate each step of the machine until it halts, or the physical machine which runs the Cflat executable runs out of memory.

The first part of the Universal Turing Machine implementation is to simulate infinite tape storage. In Cflat, this was implemented using a doubly linked list of “Cells.” A Cell holds a payload, which is a member of the tape alphabet, and a reference to its neighboring cells on the left and right. The linked list keeps track of a pointer to the current tape cell, and has functions to move the head left and right, as well as overwrite the current cell value. If the tape head needs to move in a given direction, but no cells exist, a new cell is simply instantiated that holds the blank symbol.

By using a linked list rather than an array to implement the tape storage, the Cflat Universal Turing Machine does not have any theoretical limit on the storage. Instead, the Cflat program is bounded by the target machine on which it runs, which has finite memory. However, this limitation of the physical machine does not change or reduce the expressive power of the Cflat language itself.

The second main component of the Cflat implementation is the Turing Machine parser. In order to pass the encoding of a Turing Machine to the Cflat executable, a text file format was created to specify the implementation of any Turing Machine. The file format, discussed in detail in section 3.3 was chosen for its readability and ease of understanding for both humans and computer programs. The Turing Machine parser must validate the input file to make sure that all of the required information to define a Turing Machine is present (that is, the states, tape alphabet, input alphabet, blank symbol, starting state, final state, and transition functions), and also must validate that each transition in the machine is valid. For example, a transition must be defined using states that exist, and the resulting tape head move must be valid.

The final component of the implementation is the simulator. After the tape has been filled by the initial input string and the Turing Machine file format has been parsed, the simulation can begin. The “Machine” class searches for the transition to use based off of the current state and the symbol in the current tape cell. If a transition is found, it is applied, that is, the machine transitions to a new state, a symbol is written to the current tape cell, and the head moves. If no transition is found, the machine halts and rejects. If a transition in the machine specifies that the machine should halt, it does so, and accepts or rejects whether or not the machine is in the final state.

The full source code for the implementation is included in the Appendix section.

## Class Diagram

The following is a class diagram representing all of the classes, member variables and methods that make up the Cflat implementation of the Universal Turing Machine.

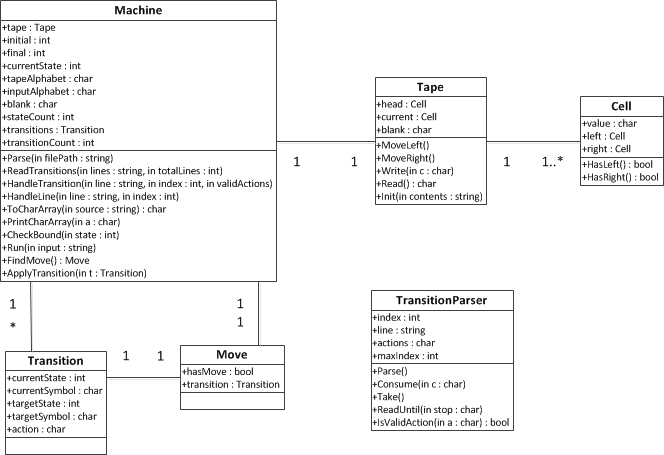


Figure 3.1 – Class diagram for the Cflat Universal Turing Machine

## Turing Machine File Format

A Turing Machine is represented as a text file in a predefined format. Each of the first 6, non-commented lines of the file are reserved for special purposes, while all remaining lines of the file are used to denote the transition function. Comments may be included, which are denoted by a line beginning with the pound (#) symbol. The first 6 lines of the file (numbered 0 – 5 here) are broken down as follows:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Line Number** | **Description** | **Data Type** | **Example Value** | **Example Description** |
| 0 | The number of states in the machine. **\*** | Integer | 10 | 10 states, numbered 0 – 9 |
| 1 | The tape alphabet. | String, all characters consecutively | abx| | The language {a, b, x, |} |
| 2 | The blank symbol. | Character | \_ | The blank tape cells will hold \_ |
| 3 | The input alphabet. | String, all characters consecutively | ab | The language {a, b} |
| 4 | The initial state. | Integer | 0 | The machine will start in state 0 |
| 5 | The final state | Integer | 9 | The machine will accept when halted in state 9 |

Table 3.1 – Turing Machine file format, lines 0 – 5

**Note \*:** By specifying the total number of states in the machine, we simplify the task of defining the set of states. Conceptually, the set of states is the range [0, total).  
  
All subsequent lines in the file will denote a transition. A transition takes the following form:

(**state**,**symbol**) -> (**state**,**symbol**) **action**

“**state**” is an integer, such that 0 < state < number of states in the machine.

“**symbol”** is a member of the input alphabet.

“**action**” is a single character in { L, R, H }.

All other characters in the transition must appear verbatim. The syntax used for transitions was inspired by the format used Mike Davey’s machine implementation [10]. For full examples of a Turing Machine encoding, see section 4.2.

Given that the number of transitions in a Turing Machine is fixed and known at runtime, the actual implementation of the Universal Turing Machine stores the transition functions in an array with a fixed size. When the input file is parsed, the total number of non-commented lines is counted. Then, the number of transitions in the Turing Machine is simply 6 less than the number of non-commented lines, in order to account for the special purpose lines in Table 3.1. There is no theoretical limit to the number of states or transitions in a Turing Machine imposed by Cflat. Instead, the Turing Machines are bounded by physical machine constraints, and by the range of representable signed numbers using 32 bits of storage.

The Cflat compiler stores all non-decimal numbers as 32 bit integers, meaning the range of representable numbers is from -2,147,483,648 through 2,147,483,647. Therefore, a Turing Machine file must not contain more than 2,147,483,641 transitions or have more than 2,147,483,647 states. This limit however, does not change the expressive power of the language itself, because this is a limitation imposed by the Cflat compiler. This limitation can be loosened by allowing more precision in the integer representation. However, it is uncommon in practice for a programming language to provide numbers larger than 64 bits.

### Example Transition Functions

|  |  |
| --- | --- |
| **Example Transition** | **Explanation** |
| (1,1) -> (3,x) R | When in state 1 and the current tape symbol is “1”, transition to state 3, write the character “x” to the current tape cell, and move the tape head right. |
| (4,0) -> (6,x) L | When in state 4 and the current tape symbol is “0”, transition to state 6, write the character “x” to the current tape cell, and move the tape head left. |
| (8,\_) -> (0,\_) H | When in state 8 and the current tape symbol is “\_”, transition to state 0, write the character “\_” to the current tape cell, and halt. |

Table 3.2 – Example transitions, lines 6+

# Results and Discussion

## Universal Turing Machine Inputs

The Cflat executable takes two command line parameters. The first input is the path to the Turing Machine file. The second input is used to specify the input string to the given Turing Machine. The Cflat Universal Turing Machine will then simulate the behavior of the given machine by writing the input string on the emulated Turing Machine’s tape. Execution of the program then continues, processing each step of the Turing Machine until it terminates, or the physical machine on which the Universal Turing Machine executes on depletes its memory resources.

## Example Turing Machines

Three example Turing Machines were implemented in order to test the abilities of the Cflat Universal Turing Machine. We shall refer to them as Machine A, Machine B, and Machine C. The full file format representation is included in order to clarify the file format, and provide concrete examples.

### Machine A

Machine A decides the language { w#w | w in {0, 1}\* }. Intuitively, this machine will accept a string made up of 0s and 1s, followed by a pound sign (#), then followed by the exact sequence of 0s and 1s, and will reject otherwise. This machine has 9 total states, and uses the underscore (\_) character as the blank symbol.

|  |
| --- |
| 9  01#x\_  \_  01#  1  0  (1,1) -> (3,x) R  (1,0) -> (2,x) R  (1,#) -> (8,#) R  (2,0) -> (2,0) R  (2,1) -> (2,1) R  (2,#) -> (4,#) R  (3,0) -> (3,0) R  (3,1) -> (3,1) R  (3,#) -> (5,#) R  (4,x) -> (4,x) R  (4,0) -> (6,x) L  (5,x) -> (5,x) R  (5,1) -> (6,x) L  (6,0) -> (6,0) L  (6,1) -> (6,1) L  (6,x) -> (6,x) L  (6,#) -> (7,#) L  (7,0) -> (7,0) L  (7,1) -> (7,1) L  (7,x) -> (1,x) R  (8,x) -> (8,x) R  (8,\_) -> (0,\_) H |

Figure 4.1 – Source for Machine A

Graphically, Machine A is depicted in the following diagram, where a transition of the form “x / y, R” denotes that “x” is the current tape symbol, it should be replaced by “y”, and the head should move right.

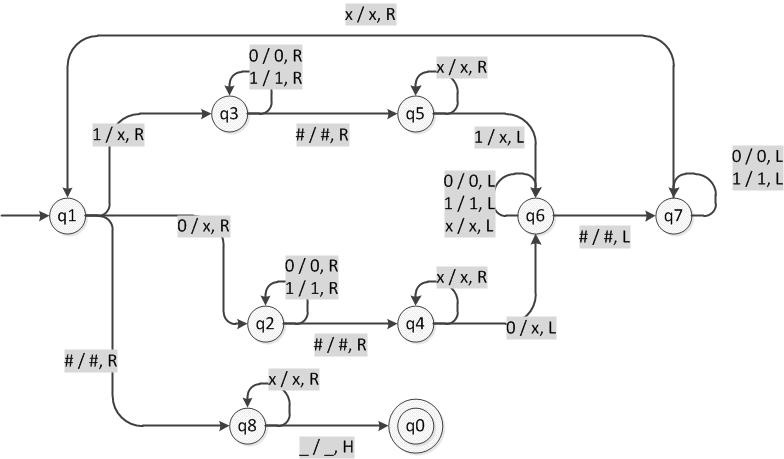


Figure 4.2 – Graphical representation of Machine A

### Machine B

Machine B decides the language { w | w in {0, 1}\* and w is a palindrome }. Intuitively, this machine will accept a sequence of 0s and 1s that reads exactly the same forwards as it does backwards and reject otherwise. This machine has 8 total states, and uses the underscore (\_) character as the blank symbol.

|  |
| --- |
| 8  01x\_  \_  01  0  7  (0,0) -> (1,x) R  (0,1) -> (4,x) R  (0,x) -> (7,x) H  (1,0) -> (1,0) R  (1,1) -> (1,1) R  (1,\_) -> (2,\_) L  (1,x) -> (2,\_) L  (2,0) -> (3,x) L  (3,0) -> (3,0) L  (3,1) -> (3,1) L  (3,x) -> (0,x) R  (4,0) -> (4,0) R  (4,1) -> (4,1) R  (4,\_) -> (6,\_) L  (4,x) -> (6,x) L  (6,1) -> (5,x) L  (5,0) -> (5,0) L  (5,1) -> (5,1) L  (5,x) -> (0,x) R  (2,x) -> (7,x) H  (6,x) -> (7,x) H |

Figure 4.3 – Source for Machine B

Graphically, Machine B is depicted as follows:

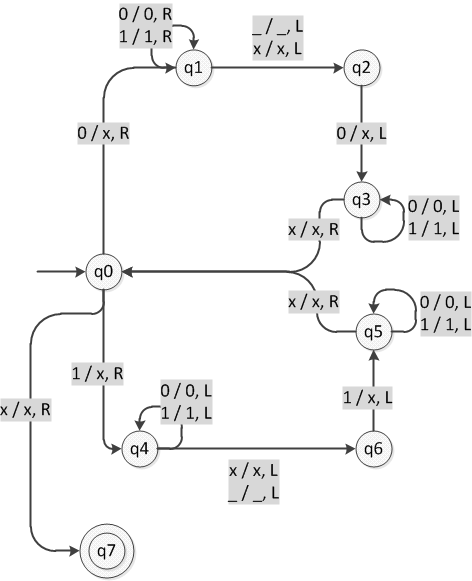


Figure 4.4 – Graphical representation of Machine B

### Machine C

Machine C decides the language { anbn | n >= 1 }. Intuitively, Machine C will accept a string with one or more occurrences of the character “a” followed by the same number of “b” characters, and will reject all other strings. This machine has only 6 states and uses the underscore character (\_) as the blank symbol.

|  |
| --- |
| 6  abx\_  \_  ab  0  5  (0,x) -> (4,x) R  (0,a) -> (1,x) R  (1,a) -> (1,a) R  (1,b) -> (1,b) R  (1,x) -> (2,x) L  (1,\_) -> (2,\_) L  (2,b) -> (3,x) L  (3,a) -> (3,a) L  (3,b) -> (3,b) L  (3,x) -> (0,x) R  (4,\_) -> (5,\_) H  (4,x) -> (5,x) H |

Figure 4.5 – Source for Machine C

Graphically, Machine C is depicted as follows:

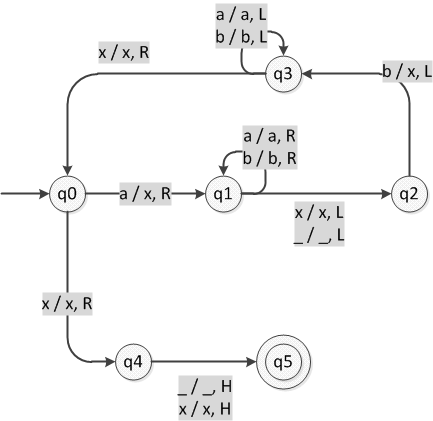


Figure 4.6 – Graphical representation of Machine C

## Running the Turing Machines

In order to verify the functionality of the Cflat Universal Turing Machine, the following test cases were used on each machine. Results are provided for verification purposes.

### Machine A

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected Result** | **Actual Result** |
| 01#01 | Accept | Accept |
| 000111#000111 | Accept | Accept |
| 1011#101 | Reject | Reject |
| 11001100 | Reject | Reject |
| 101#101 | Accept | Accept |

Table 4.1 – Inputs and results for Machine A

### Machine B

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected Result** | **Actual Result** |
| 1 | Accept | Accept |
| 1001001 | Accept | Accept |
| 110 | Reject | Reject |
| 1111 | Accept | Accept |
| 10010 | Reject | Reject |

Table 4.2 – Inputs and results for Machine B

### Machine C

|  |  |  |
| --- | --- | --- |
| **Input** | **Expected Result** | **Actual Result** |
| a | Reject | Reject |
| aaabb | Reject | Reject |
| aaaaabbbbb | Accept | Accept |
| aaabbba | Reject | Reject |
| bbbaaa | Reject | Reject |

Table 4.3 – Inputs and results for Machine C

## Simulating Arbitrary Turing Machines

The Cflat Universal Turing Machine can be proved to have the expressive power of any arbitrary Turing Machine by employing the same strategy used to implement the three example machines. The Turing Machine file format provides the means to specify the exact components of a Turing Machine as defined in section 2.1 (that is, a set of states, the set of input symbols, the set of tape symbols, the transitions, the blank character for the tape, and the initial and final states). Thus, we can specify any number of input states, any set of ANSI characters for the input alphabet, tape alphabet, and blank symbol, the initial and finals states, and any number of transitions in an arbitrary machine.

Regarding the transition function, we can closely examine the example Turing Machines provided to show each possibility is implemented. Below is a subset of the transitions from Machine C, with a description of their significance.

|  |  |
| --- | --- |
| **Transition** | **Significance** |
| (3,a) -> (3,a) L | Demonstrates moving the tape head to the left only. |
| (1,b) -> (1,b) R | Demonstrates moving the tape head to the right only. |
| (1,x) -> (2,x) L | Demonstrates moving to a different state. |
| (0,a) -> (1,x) R | Demonstrates writing a new tape symbol and moving to a new state. |

Table 4.4 – Core transition functionality

## Analysis

Given that the Cflat Universal Turing Machine is able to simulate arbitrary Turing Machines, as well as example Turing Machines that demonstrate the core functionality (that is, changing state, moving the tape head left and right, replacing input symbols on the tape), we conclude that Cflat is indeed powerful enough to solve any computable problem, given enough time and space. Because Cflat is capable of implementing a Universal Turing Machine, we can conclude that Cflat is a Turing complete language.

# Conclusion

## Summary of Results

We have shown Cflat’s ability to express a program that functions as a Universal Turing Machine. That is, a Cflat program exists that takes a Turing Machine and a string as its input, and performs all steps that the given Turing Machine would perform. Example Turing Machines were provided that demonstrate this ability, for example, a machine that decides whether or not a string is a palindrome made up of 0s and 1s.

This proves that the Cflat language can be used to solve any problem that can be solved by a Turing Machine. Therefore, programmers can feel confident that Cflat does not lack the expressiveness to be a useful language.

## Significance

By proving that the Cflat language is Turing Complete, this addresses any uncertainties as to whether or not the language can be used to solve real world computation problems. A programming language which supports uncommon or new features can be beneficial to the programming community by bringing forth new ideas in how to express a programmer's intentions clearly and succinctly. Because Cflat provides the basic implementation for OO features, it serves as a base for experimentation by providing a means for adding new or unconventional object oriented features. The language can be extended with new ideas, in addition to the language's new keywords.

## Further Research

Now that Cflat has been validated as a Turing complete general purpose programming language, there are several follow-up questions to address. Future research could address questions like:

* Does Cflat have the expressive power of Super-Turing computation models?
* What language features or extensions would need to be implemented to give Cflat more expressive power than a Turing Machine?

Further research can be done to investigate how Cflat compares against existing Super-Turing models of computation, such as Persistent Turing Machines, $-Calculus, and π-Calculus [11].

# References

[1] Gough, John, 2008, The Gardens Point Scanner Generator (GPLEX), Queensland University of Technology (<http://plas.fit.qut.edu.au/gplex/>)

[2] Kelly, Wayne, 2008, The Gardens Point Parser Generator (GPPG), Queensland University of Technology (<http://plas.fit.qut.edu.au/gppg/>)

[3] Barnett, Granville, 2009, Common Intermediate Language and its role in .NET/Mono (<http://dotnetslackers.com/articles/net/Common-Intermediate-Language-and-its-role-in-NET-Mono.aspx>)

[4] Hopcroft, John E., Rajeev Motwani, and Jeffrey D. Ullman, 2007, Introduction to Automata Theory, Languages, and Computation, Boston: Pearson/Addison Wesley

[5] Barker-Plummer, David, 2011, Turing Machines, The Stanford Encyclopedia Philosophy (<http://plato.stanford.edu/entries/turing-machine/>)

[6] Révész, György E., 1991, Introduction to Formal Languages, Courier Dover Publications

[7] Lebanon, Guy, 2006, The Turing Machine, Georgia Institute of Technology (<http://www.cc.gatech.edu/~lebanon/notes/TuringMachine.pdf>)

[8] Kumar, Rajendra, 2010, Theory of Automata Languages and Computation, New Delhi: Tata McGraw-Hill

[9] Tanaka-Ishii, Kumiko, 2010, Semiotics of Programming, Cambridge University Press

[10] Davey, Mike, 2010, A Turing Machine In The Classic Style (<http://aturingmachine.com/>)

[11] Teuscher, Christof, 2004, Alan Turing: Life and Legacy of a Great Thinker, Springer

# Appendix

## Complete Cflat Source Code

The following is the exact source code used to generate the test case results. The included source code is meant to be used with revision 179 of the compiler. The Cflat compiler, as well as the Cflat Universal Turing Machine source code, and example Turing Machines can be downloaded at the following address:

<http://code.google.com/p/cflat/>

class Cell

{

char value;

Cell left;

Cell right;

Cell(char c)

{

value = c;

}

bool HasLeft()

{

return isnull(left) == false;

}

bool HasRight()

{

return isnull(right) == false;

}

}

class Tape

{

Cell head;

Cell current;

char blank;

Tape(char b)

{

head = new Cell(b);

current = head;

blank = b;

}

void MoveLeft()

{

if (current.HasLeft())

{

current = current.left;

}

else

{

Cell temp = new Cell(blank);

temp.right = current;

current.left = temp;

current = temp;

}

}

void MoveRight()

{

if (current.HasRight())

{

current = current.right;

}

else

{

Cell temp = new Cell(blank);

temp.left = current;

current.right = temp;

current = temp;

}

}

void Write(char c)

{

current.value = c;

}

char Read()

{

return current.value;

}

void Init(string contents)

{

int length = slen(contents);

for (int i = 0; i < length; i++)

{

char c = charAt(contents, i);

Write(c);

MoveRight();

}

/\* reset the tape to the beginning \*/

current = head;

}

/\* Prints out the tape contents from left most to right. \*/

void Debug()

{

int cellNumber = 0;

Cell temp = head;

bool done = false;

/\* find the left-most cell \*/

while (done == false)

{

if (temp.HasLeft())

{

cellNumber--;

temp = temp.left;

}

else

{

done = true;

}

}

/\* loop through all cells again \*/

done = false;

while (done == false)

{

println(cellNumber & ": " & temp.value);

if (temp.HasRight())

{

cellNumber++;

temp = temp.right;

}

else

{

done = true;

}

}

}

}

class Transition

{

int currentState;

char currentSymbol;

int targetState;

char targetSymbol;

char action;

Transition(int cState, char cSymbol, int tState, char tSymbol, char a)

{

currentState = cState;

currentSymbol = cSymbol;

targetState = tState;

targetSymbol = tSymbol;

action = a;

}

void Debug()

{

println("(" & currentState & "," & currentSymbol & ") -> (" & targetState & "," & targetSymbol & ") " & action);

}

}

class TransitionParser

{

char LPAREN;

char RPAREN;

char SPACE;

char COMMA;

char HYPHEN;

char GT;

int index;

string line;

char[] actions;

int maxIndex;

TransitionParser(string l, char[] a)

{

LPAREN = '\u0028';

RPAREN = '\u0029';

SPACE = '\u0020';

COMMA = '\u002c';

HYPHEN = '\u002d';

GT = '\u003e';

line = l;

actions = a;

maxIndex = slen(line) - 1;

}

/\* Parses a line of the form: (state,symbol) -> (state,symbol) action \*/

Transition Parse()

{

Consume(LPAREN);

string currState = ReadUntil(COMMA);

Consume(COMMA);

char currSymbol = Take();

index++;

/\* There's no real need for the extra symbols here, other than human readability. \*/

Consume(RPAREN);

Consume(SPACE);

Consume(HYPHEN);

Consume(GT);

Consume(SPACE);

Consume(LPAREN);

string targetState = ReadUntil(COMMA);

Consume(COMMA);

char targetSymbol = Take();

index++;

Consume(RPAREN);

Consume(SPACE);

char action = Take();

/\* Make sure the action is valid \*/

if (!IsValidAction(action))

{

die("Invalid transition action. Got: " & action);

}

/\* We have the primitives to make a transition, so return it! \*/

Transition t = new Transition(parseInt(currState), currSymbol, parseInt(targetState), targetSymbol, action);

return t;

}

/\* Advances the index if the current character matches the input \*/

void Consume(char c)

{

char current = Take();

if (current == c)

{

index++;

}

else

{

die("Invalid transition. Expected: " & c & " Got: " & current);

}

}

/\* Checks string bounds, and returns the current character if everything's ok. Otherwise, this will exit the application. \*/

char Take()

{

if (index > maxIndex)

{

die("Invalid transition. Unexpected end of line.");

}

return charAt(line, index);

}

/\* Concats a string of the characters and stops at the given symbol \*/

string ReadUntil(char stop)

{

string result = "";

char current = Take();

while (current != stop)

{

result = result & current;

index++;

current = Take();

}

return result;

}

/\* Makes sure the action matches the set of valid actions passed in \*/

bool IsValidAction(char a)

{

int l = alen(actions);

for (int i = 0; i < l; i++)

{

if (actions[i] == a)

{

return true;

}

}

return false;

}

}

class Move

{

bool hasMove;

Transition transition;

Move() { }

}

class Machine

{

Tape tape;

int initial; /\* initial state number \*/

int final; /\* accepting state number \*/

int currentState;

char[] tapeAlphabet;

char[] inputAlphabet;

char blank;

int stateCount;

/\* all of the transitions in the machine \*/

Transition[] transitions;

int transitionCount;

/\* Characters that correspond to the actions in a transition \*/

char LEFT;

char RIGHT;

char HALT;

char COMMENT;

Machine(string filePath)

{

/\* Set the characters for L, R, and H \*/

LEFT = '\u004c';

RIGHT = '\u0052';

HALT = '\u0048';

COMMENT = '\u0023';

/\* Parse the input file \*/

Parse(filePath);

/\* more initialization \*/

tape = new Tape(blank);

}

void Parse(string filePath)

{

string[] lines = readFile(filePath);

int totalLines = alen(lines);

int lineIndex = 0;

for (int i = 0; i < totalLines; i++)

{

/\* check for a comment in the input file \*/

string currentLine = lines[i];

char first = charAt(currentLine, 0);

if (first != COMMENT)

{

/\* process each line of the file \*/

HandleLine(currentLine, lineIndex);

lineIndex++;

}

}

transitionCount = lineIndex - 6;

/\* loop through the lines again, after we have the number of transitions (it's needed to initialize the array) \*/

ReadTransitions(lines, totalLines);

}

void ReadTransitions(string[] lines, int totalLines)

{

transitions = new Transition[transitionCount];

/\* build a set of characters for the Transition parser \*/

char[] validActions = new char[3];

validActions[0] = LEFT;

validActions[1] = RIGHT;

validActions[2] = HALT;

/\* loop through the text file again \*/

int lineIndex = 0;

int transitionIndex = 0;

for (int i = 0; i < totalLines; i++)

{

/\* exclude comments \*/

string currentLine = lines[i];

if (lineIndex > 5)

{

HandleTransition(currentLine, transitionIndex, validActions);

transitionIndex++;

}

char first = charAt(currentLine, 0);

if (first != COMMENT)

{

lineIndex++;

}

}

}

void HandleTransition(string line, int index, char[] validActions)

{

TransitionParser parser = new TransitionParser(line, validActions);

Transition t = parser.Parse();

/\* make sure the state numbers are within bounds \*/

CheckBound(t.currentState);

CheckBound(t.targetState);

/\* valid, add to the array \*/

transitions[index] = t;

}

void HandleLine(string line, int index)

{

/\*println(index & ": " & line);\*/

if (index == 0)

{

/\* the number of states in the machine \*/

stateCount = parseInt(line);

}

else if (index == 1)

{

/\* the tape alphabet \*/

tapeAlphabet = ToCharArray(line);

}

else if (index == 2)

{

/\* the blank symbol \*/

blank = charAt(line, 0);

}

else if (index == 3)

{

/\* the input alphabet \*/

inputAlphabet = ToCharArray(line);

}

else if (index == 4)

{

/\* initial state \*/

initial = parseInt(line);

if (initial >= stateCount)

{

die("Input state is not valid. States are zero based, and must be less than the total number of states - 1.");

}

}

else if (index == 5)

{

/\* accepting state \*/

final = parseInt(line);

if (final >= stateCount)

{

die("Final state is not valid. States are zero based, and must be less than the total number of states - 1.");

}

}

}

/\* Copies each char in a string to a char array. \*/

char[] ToCharArray(string source)

{

int l = slen(source);

char[] result = new char[l];

for (int i = 0; i < l; i++)

{

result[i] = charAt(source, i);

}

return result;

}

/\* Used for debugging. \*/

void PrintCharArray(char[] a)

{

string s = "";

int l = alen(a);

for (int i = 0; i < l; i++)

{

s = s & a[i];

}

println(s);

}

/\* Checks if the given state number is within the number of states in the machine, zero based. \*/

void CheckBound(int state)

{

if (state >= stateCount)

{

die("Invalid transition. State number " & state & " is out of range.");

}

}

void Run(string input)

{

tape.Init(input);

/\* debugging \*/

println("Initial tape contents:");

tape.Debug();

/\* Begin processing \*/

currentState = initial;

println("Running...");

bool done = false;

while (done == false)

{

/\* debugging \*/

println("Current: (" & currentState & "," & tape.Read() & ")");

Move m = FindMove();

if (m.hasMove)

{

ApplyTransition(m.transition);

if (m.transition.action == HALT)

{

done = true;

}

}

else

{

done = true;

}

}

/\* We're done looping, check if we accept or reject! \*/

if (currentState == final)

{

println("Accept!");

}

else

{

println("Reject.");

}

}

/\* Returns whether or not we have a valid transition in the machine \*/

Move FindMove()

{

char symbol = tape.Read();

for (int i = 0; i < transitionCount; i++)

{

Transition t = transitions[i];

if ((t.currentState == currentState) && (t.currentSymbol == symbol))

{

Move found = new Move();

found.hasMove = true;

found.transition = t;

return found;

}

}

/\* not found, return an empty move object \*/

Move none = new Move();

none.hasMove = false;

return none;

}

/\* Changes the state, input symbol in the current cell, and moves the tape head if needed. \*/

void ApplyTransition(Transition t)

{

/\* debugging \*/

print("Applying Transition: ");

t.Debug();

currentState = t.targetState;

tape.Write(t.targetSymbol);

if (t.action == LEFT)

{

tape.MoveLeft();

}

else if (t.action == RIGHT)

{

tape.MoveRight();

}

}

void Debug()

{

println("Number of states: " & stateCount);

println("Tape Alphabet:");

PrintCharArray(tapeAlphabet);

println("Blank character: " & blank);

println("Input Alphabet:");

PrintCharArray(inputAlphabet);

println("Initial state: " & initial);

println("Final state: " & final);

println("Tape Contents:");

tape.Debug();

int l = alen(transitions);

println("Transitions, count = " & l);

for (int i = 0; i < l; i++)

{

Transition t = transitions[i];

t.Debug();

}

}

}

class program

{

void main(string[] args)

{

/\* command line input should be [file path] [turing machine input] \*/

int argLength = alen(args);

if (argLength != 2)

{

die("Usage: turingmachine.exe [file path] [input]");

}

Machine tm = new Machine(args[0]);

tm.Run(args[1]);

}

}