AN INFORMATION THEORETIC APPROACH TO THE EXPRESSIVENESS OF PROGRAMMING LANGUAGES

by

Joseph Ray Davidson

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy

University of Glasgow

College of Science and Engineering
School of Computing Science

February 2016

Abstract

The conciseness conjecture is a longstanding notion in computer science that programming languages with more built-in operators, that is more expressive languages with larger semantics, produce smaller programs on average. Chaitin defines the related concept of an elegant program such that there is no smaller program in some language which, when run, produces the same output.

This thesis investigates the conciseness conjecture in an empirical manner. Influenced by the concept of elegant programs, we investigate several models of computation, and implement a set of functions in each programming model. The programming models are Turing Machines, λ-Calculus, SKI, RASP, RASP2, and RASP3. The information content of the programs and models are measured as characters. They are compared to investigate hypotheses relating to how the mean program size changes as the size of the semantics change, and how the relationship of mean program sizes between two models compares to that between the sizes of their semantics.

We show that the amount of information present in models of the same paradigm, or model family, is a good indication of relative expressivity and average program size. Models that contain more information in their semantics have smaller average programs for the set of tested functions. In contrast, the relative expressiveness of models from differing paradigms, is not indicated by their relative information contents.

RASP and Turing Machines have been implemented as Field Programmable Gate Array (FPGA) circuits to investigate hardware analogues of the hypotheses above. Namely that the amount of information in the semantics for a model directly influences the size of the corresponding circuit, and that the relationship of mean circuit sizes between models is comparable to the relationship of mean program sizes.

We show that the number of components in the circuits that realise the semantics and programs of the models correlates with the information required to implement the semantics and program of a model. However, the number of components to implement a program in a circuit for one model does not relate to the number of components implementing the same program in another model. This is in contrast to the more abstract implementations of the programs.

Information is a computational resource and therefore follows the rules of Blum's axioms. These axioms and the speedup theorem are used to obtain an alternate proof of the undecidability of elegance.

This work is a step towards unifying the formal notion of expressiveness with the notion of algorithmic information theory and exposes a number of interesting research directions. A start has been made on integrating the results of the thesis with the formal framework for the expressiveness of programming languages.

Contents

Acknowledgements

This work would not have been possible without my supervisors Greg Michaelson and Phil Trinder. Greg's boundless enthusiasm and stalwart faith in my work kept me moralised and Phil's insightful comments and probing questions have improved the quality of this work enormously. Which isn't to say that either of them held a monopoly on these aspects.

I would also like to thank my family, friends and colleagues who have all played an enormous role in keeping me sane. Especially over the writeup. And to the strangers with whom I have talked at in regards to this. Those which ask questions. Those that just smile and nod. Thank you all.

This work has been funded by the European Union grant IST-2011-287510 'RELEASE: A High-Level Paradigm for Reliable Large-scale Server Software', and EPSRC EP/J001058/1 'The Integration and Interaction of Multiple Mathematical Reasoning Processes'. I am grateful to those who guard the respective purses for the money to keep myself alive even though the work here does not directly relate to the work of the grants.

Declaration

I declare that, except where explicit reference is made to the contribution of others, that this dissertation is the result of my own work and has not been submitted for any other degree at the University of Glasgow or any other institution.

Joseph Davidson

Chapter 1

Introduction

What is the result of adding together the numbers 5 and 8?

Nearly all tasks are not fully specified. When a task is given to a person or a machine, it is presented based on knowledge of the abilities of the assignee. If the assignee is versed in all pertinent aspects of a task, then they require no other information. If not, then they may need more specific instructions in order to carry out the task.

Examine the problem above. If one can read English, can count above 10, and knows how to perform addition, then one can obtain the correct answer: 13. If there is a gap in one's knowledge, one might have to learn how to read English, how to count above 10, or how to add two numbers together.

Not knowing English is an encoding problem. One does not have the ability to parse an English sentence into one's own internal representation[1], but one might be able to parse the same problem in a different encoding: $5+8$. If one is literate in Russian, a Cyrillic representation might be preferable to the English version: "Что такоерезультат сложения числа 5 и 8?"[2]

Not knowing how to add, or how the numerals behave above the number 10 requires some instruction in mathematics – the person doing the addition has to be told how to add. Assuming that the assignee can count up to 10 on their fingers, they can be instructed in how addition works by having them represent, say two on the left hand and three on the right. For each finger they lower

[1] However knowledge is represented in the mind.
[2] Courtesy of Google Translate.

on the left, a finger raises on the right. This is an algorithm for addition, and with enough examples can be generalised for any numbers as long as the assignee knows how to count up to them.

The point is: for any task to be completed, the assignee must have knowledge of how to perform the task and subtasks, and knowledge of the behaviour and effects of their actions upon the environment which contains the task. From the high level specification, down to the lowest level mechanical attributes of the assignee, each aspect of the above knowledges must be specified. The completion of a task is a culmination of combining the various pieces of knowledge to achieve the effect of a task.

When we discuss ourselves, or something to which we have ascribed anthropomorphic traits, we say that these knowledges are either "learned" or "implicit/inherent". Knowing how to tap something with a pen three times uses learned knowledge of how to hold a pen, how to count to three, what constitutes a 'tap' and so on. It also uses "implicit" knowledge of sending nerve impulses to contract muscles to manipulate the pen.

Constructing ontologies and taxonomies for knowledges and actions for living creatures is an extraordinary undertaking owing to their complexity, but such classifications for formal systems could be possible. Programming languages, which encapsulate the traits of some formal mathematical model, have a specified encoding (syntax), and a set of pre-defined functions which represent the knowledge of the language. The language initially "knows" how to perform these functions because the designer has decided that it should. The definitions of these functions, and algorithms to perform them, are defined in the *semantics* of the language as *implicit information*.

If a program is written in the language for a computational model A, and it is not in the correct encoding, or using functions not defined in the semantics, then A cannot compute this particular program. One would have to reformulate the program to use only the encoding and the functions defined in the semantics. If the programmer insists on a different encoding or the use of of some undefined function; then either the semantics of A has to be changed, or a program written in A to define the missing functions/translate the encodings. The computational

model A requires *more information*.

There are many models like A. While a lot of them can calculate the same set of functions, they all have a mixture of different encodings and pre-defined functions. The full mathematical description of these encodings and functions constitute the semantics of the model. Some models may have very large semantics with lots of pre-defined functions, and some may have very small semantics with few functions. If the size of the semantics of a computational model is taken into account when the program is measured, then we can ask which computational models require the least information to fully specify and compute a function.

1.1 Motivation

This thesis is an investigation into how the distribution of information in a computational model affects the sizes of programs written in that model. If the semantics of computational models are specified in a consistent manner (Section 3.4), and programs are written for each model in their respective encodings, then measurements of the size of semantics and programs can be taken. These measurements can be compared with the sizes of semantics and programs in other models to look for a relationship between semantics size and program size.

There is a high level intuition in Computer Science that languages which are more expressive (Section 2.5) have more pre-defined functions and thus larger semantics. Languages with larger semantics therefore produce smaller programs than languages with smaller semantics.

If this intuition holds true, then what is the nature of the relationship between the size of semantics and the size of programs? Can the relationship be generalised, or is it specific to each model? Additionally, questions can be asked about how the internal and external representations affect semantic and program sizes. This thesis is a preliminary investigation into these questions.

1.2 Investigation Overview

This investigation is conducted as an empirical study to compare multiple models of computation of varying paradigms. There are four models: the Turing Machine

(TM, Section 2.3.1.1), the SKI combinator calculus (Section 2.3.2.2), the Random Access Stored Program machine (RASP, Section 2.3.1.2), and the λ-calculus (Section 2.3.2.1).

Each of these models varies in how expressive they are (Section 2.5), so the mechanisms behind each one need to be formalised. This is done by writing down the semantics of each model using a common formalism. In this case, Structured Operational Semantics (SOS, Section 2.4.1) is used. In doing this, a baseline is established from which measurements of the information content of models and programs can be performed.

A set of functions is implemented sampling from both the primitive and the partial recursive functions (Section 4.1). This set covers problems as simple as addition up to more complicated functions like sorting a list and the universal machines. The results are presented and an analysis is performed.

There are shortcomings with the idea of measuring information at the semantic level. Even though the semantics are all specified in SOS, the question of how the functions which are pre-defined in SOS can be defined in another baseline can be asked. This further begs the question of how the functions of *that* baseline could be defined (Section 5.1). In an attempt to address this, the RASP and Turing models are reduced to the hardware level using Field Programmable Gate Arrays (FPGA, Section 5) which are configurable chips that can simulate the models at the logic gate level.

1.3 Hypotheses

As an empirical investigation, hypotheses are first formulated as a guide. These hypotheses are preliminary at this time, and shall be revised in the context of the literature review (Section 3.1).

Some notion of the size of a program or semantics is required. Information and algorithmic theory define the size of a piece of information as the number of characters required to write it down (Section 2.2). This is a useful definition which we adopt.

The information to compute a function in a model is split into the information

content of the semantics, and the information content of the program computes the function. These information values combined constitute the Total Information of the function.

Definition 1 (Semantic Information)**.** *Semantic information (SI) for a model is the size of the semantics of that model in characters.*

Definition 2 (Program Information)**.** *Program Information (PI) is the size of a program in characters.*

Definition 3 (Total Information)**.** *Total Information (TI) is SI + PI.*

It is expected that a model with more SI produces programs with less PI for the same functions in comparison to models with less SI. The intuition is that larger semantics are a consequence of defining more operators or constructs for a language or model. Sensibly defined operators ease the burden on the programmer, thus allowing them to write programs using less characters and therefore less PI.

Hypothesis 1P (Semantic Information)**.** *For two Turing Complete models (Section 2.1.2), if model A has more semantic information than model B, the average size of programs written for model A will be lower than the average for model B.*

For example, it is believed that a high level functional language is less of a chore to program in than assembler. The high level of abstraction afforded by the functional language allows the author of some program to focus their efforts on programming to the specification, rather than the minutiae of using the model. Conversely, writing the same program in assembler often requires that the programmer know what the layout of the registers are and their contents at any one time. Not only does the programmer have to solve the problem, but they have to manage resources intelligently, or risk bugs which break the program but do not directly relate to how the programmer has solved the problem.

Extensionality is when a program is evaluated on its external effects rather than its internal structure. Two programs are the same in an extensional sense if they produce the same output for the the same inputs. The opposite of this is *intensionality*, which evaluates programs on how they compute something.

When applied to the same task, the extensionality of the functional program + semantics is equivalent to that of the assembler program + semantics. The semantics of the functional language are more complicated than those of the assembler, so its expected that the functional program will be appreciably smaller than the assembly program.

As the complexity of programs (Section 3.1.2) increases, so does their minimum size. If the SI hypothesis (1P) is correct, then this size increase will be more marked in languages with small semantics as opposed to languages with larger semantics. It is hypothesised that smaller models and simpler programs will contain less TI than simple programs in complex models. However as the size and complexity of the set of programs grows, the average TI of the complex models will be lower than that of the simple models.

Hypothesis 2P (Total Information)**.** *As the size and complexity of a program increases, the average total information of an implementation in a model with large semantics decreases relative to the total information of an implementation in a model with small semantics.*

Analogous hypotheses for FPGAs can be stated:

Hypothesis 3P (Semantic Circuit Size)**.** *A Model A with a larger set of semantics than model B will produce a larger circuit when converted into a hardware representation.*

Hypothesis 4P (Total Circuit Sizes)**.** *The average total circuit size (semantics + programs) of a more expressive model will be lower than that of a less expressive model.*

These hypotheses will be expanded in Section 3.1 which evaluates and refines the hypotheses in the context of the literature survey.

1.4 Contributions

This work makes the following contributions:

Empirical Comparison of Program Sizes in Computational Models For each model of computation considered in this thesis, semantics are defined in a common representation (Section 3.4) using Structured Operational Semantics (SOS). This representation is measured in the accepted information theoretic metric of characters (Section 2.2.1) and produces a representative set of functions which are as *elegant* (Section 2.2.2) and stylistically consistent as possible. The programs are measured and these measurements are analysed (Chapter 6). The analysis shows:

- In the same model paradigm, models with large semantics tend to produce smaller programs than models with small semantics (Sections 6.2.2 and 6.2.3) [19].
- When comparing models from differing paradigms, semantic size is not a reliable indicator of relative program size (Section 6.2.8).
- The information levels of the simpler models (e.g. SKI calculus and Turing machines) exhibit differing trends in the TI required to compute the set of chosen functions, compared to more complex models (the RASPs and λ-calculus). For the set of functions in this thesis, the simpler models have a significant increase in required information when the universal machines for the RASP and TM are included (Section 6.4).
- The encoding of the input to a function can drastically affect the size of the program to calculate the function (Section 6.6). Proposals are made to incorporate the information of encoding functions and input growths to the broader field of Algorithmic Information Theory.

FPGA Realisation of RASP and Turing Machines Comparisons founded on a character-based information theoretic encoding carry some problems as there is no account of the semantics of the SOS formalism in which the model semantics are defined (Section 5.1). Such implicitly defined operators in SOS may be used in the semantics of one model, but not in another. Furthermore, there may be consistency of the models within the confines of these information theoretic comparisons, but no guarantee that this consistency holds in another mode of comparison.

The mathematical models of the semantics can be physically grounded by translating the SOS of the models into a specification language for electronic circuits such as VHDL. This specification is synthesised into a circuit schematic suitable for implementation on a Field Programmable Gate Array (Chapter 5). These implementations provide a concrete comparison of the number of electronic components required to implement the semantics and programs of the models. The analysis shows:

- FPGA realisations are correlated with the TIs of the models. The TI can be used as an indicator of the number of components required to implement the semantics and program (Section 6.3) [19].
- FPGA realisations are a poor indicator of relative expressiveness. One cannot determine the expressiveness of the TM vs the RASP using the number of components of an FPGA implementation (Section 6.3.3).

Alternative proof of the undecidability of Elegance Chaitin's proof of the undecidability of elegance is based on the operation of programs. An alternative proof is obtained via proving that the information to calculate a function in some model is a *Blum complexity measure* (Section 3.1.1). For a Blum complexity measure, there exists a function where the information for a program and input can always be reduced for almost all inputs (Speed-up Theorem, [5]).

Universal RASPs In the course of this investigation, a number of programs drawing from the sets of primitive and partial recursive function have been written. One of these programs is the universal RASP machine, a program which takes the definition of a RASP and runs it according to the semantic rules of the RASP model (Section 4.4.2). A RASP machine, Turing machine, λ-calculus expression, and SKI combinator expression have all been written which perform this function. A suitably encoded RASP given as input to these programs will return the RASP in a halting state (if one exists) which is identical the halting state of the same machine executed according to the RASP semantics.

RASP Busy Beavers The Busy Beaver problem is that of finding a Turing Machine of a given size that runs for the longest number of steps, and/or prints the

most symbols before halting [73]. A variant of this problem had been developed for the finite RASP machine and an upper bound on the highest number of instructions executed, and the highest number of outputs, has been discovered for 2^3 by brute forcing all possible machines (Section A.3.1). Subsequent classes have also been investigated and lower bounds established through the use of seeded and non-seeded parallel genetic algorithms (Section A.3.2) [18].

1.5 Structure

This structure of this thesis is as follows: Chapter 2 is a survey of the literature, covering the history of computability, information theory, elegance, expressiveness and the models which are used.

Chapters 3, 4, 5, and 6 tackle the crux of the central question. Chapter 3 lays out the semantics of the models in Structural Operational Semantics, discusses the metrics and criteria which are used to gauge the written programs, and covers the method used in the investigation. Chapter 4 presents the programs from which comparisons are drawn and details their algorithms. Chapter 5 sets out the rationale and implementation of physically grounding the TM and RASP machines using FPGAs.

Chapter 6 provides a detailed analysis of the measured programs, semantics and circuits. By combining, contrasting and evaluating them in multiple contexts, insight is gained into the shape of the information landscape and how the information contents of models relate to each other.

Chapter 7 reflects on the investigation as a whole and concludes it. The chapter discusses particular topics of interest which may provide further insight into the results described herein. It proposes extensions to this work and explores ideas of information for computation.

1.6 Publications

The publications which have resulted from this work are:

- "Brute Force is not Ignorance", Joseph Davidson and Greg Michaelson, *The*

Informal Proceedings of Computability in Europe 2013, Milan, Italy.

- "Elegance, Meanings and Machines", Joseph Davidson and Greg Michaelson, *Computability*, 2015 (accepted subject to revision).

Chapter 2

Literature Review

This chapter outlines the literature behind this thesis in order to prepare us to understand the results that are to come. Figure 2.1 shows relationships between relevant topics in computer science. Each arrow shows the influence of one topic on another. This does not show all the relationships because in reality, the computability bubble influences almost all the other topics and should have a lot more arrows. Computability is where we start.

2.1 Computability

In the broadest sense, a function is computable if it can be translated into some kind of formal representation which is then executed on a model of computation. There are caveats to this, such as the model needs to predictably stop (halt) once the computation is finished. In computer science, computability is the discipline of determining if a function is computable [86].

2.1.1 Hilbert and Gödel

In 1900 the German mathematician David Hilbert had a dream. He actually had 23 dreams, each of which was a single problem that he believed was a important question for mathematics to address in the coming century [37]. At the time of writing, 11 are fully resolved, 7 are partially (or controversially) resolved, 4 are unresolved and one (the 4^{th})is thought to be stated too vaguely for any work to take place [31].

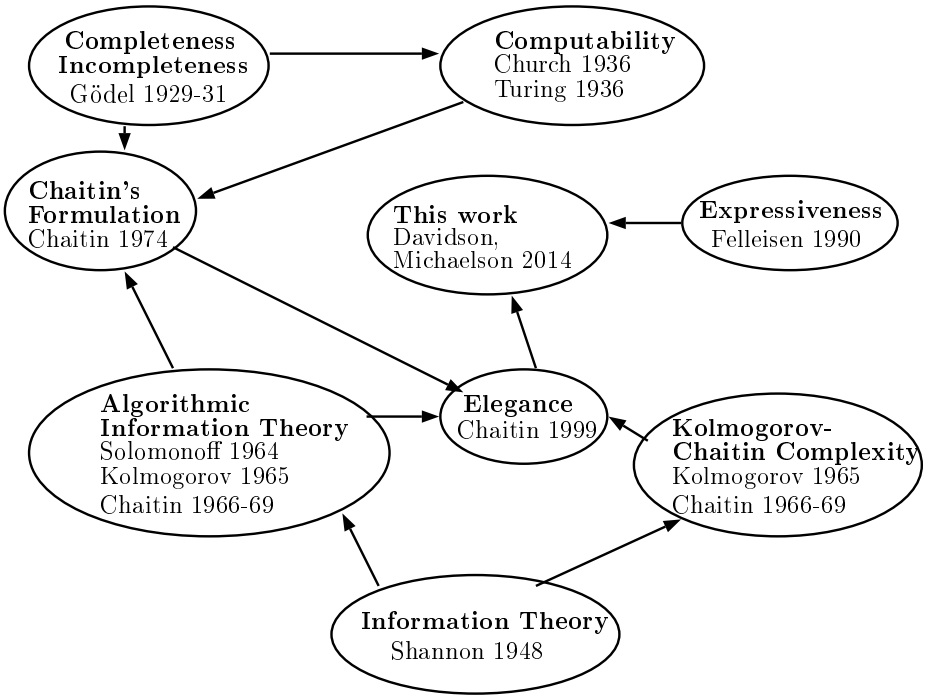


Figure 2.1: Overview of topics related to this thesis.

Of these problems, we focus on the second one. Hilbert wanted to formalise all of mathematics such that if someone were to write a mathematical statement in this formal system; "it shall be possible to establish the correctness of the solution by means of a finite number of steps based upon a finite number of hypotheses which are implied in the statement of the problem and which must always be exactly formulated." [37]

To put it in a more modern vernacular, Hilbert wanted a computer program which could take any set of axioms (a statement taken to be "self evident") and formulae provided by the user, and return a proof of the formulae starting from those axioms. This mechanisation of mathematics would allow us to formulate any unresolved question (such as the twin primes conjecture [107]), a set of basic axioms (such as the Peano or ZFC axioms [101]) and eventually get an answer. To do this however, needs a formal system which is is *complete* (able to express all possible mathematical formulae) and *consistent* (there are no two true formulae that contradict each other).

In 1931, Kurt Gödel proved that this was an impossible dream. The Incompleteness Theorems assert that even a simple formal system could express a formula which was the negation of itself [28]. He did this by constructing the

mathematical equivalent of the English sentence "This statement is false" using a scheme known as *Gödel numbering* or *Gödelisation* [64].

Gödel numbering is a method of mapping some finite alphabet to the natural numbers. As an example, say there are 3 symbols in an alphabet $\{(,), x\}$ and the sentences (x), $()x$, and $x()$ need to be numbered. A mapping of natural numbers to the individual letters is first defined, say $\{(\mapsto 1,) \mapsto 2, x \mapsto 3\}$. Then the numberings are constructed with care taken to preserve the structure of the formulae. The natural numbers from this alphabet can be concatenated together $(x) = 132$, but an alphabet of more than 9 symbols would present a problem. If $y = 11$, is $12113 = ()yx$ or $= ()((x$?

The *fundamental theorem of arithmetic* is an observation by Euclid that every natural (non-negative) number has a unique prime factorisation [24]. Take the number 523345 for instance:

$$523345 = 3 \times 17 \times 47 \times 131$$

Since we know that all prime numbers have only themselves and 1 as divisors, it is clear to see that we cannot substitute any other numbers for the factors above so it must be unique.

Prime factorisations are used to resolve the issue above. A number is constructed by using the prime numbers as position indicators for the formula with the exponents of the prime numbers indicating which character is in that position. For example:

$$(x) = 2^1 \times 3^3 \times 5^2 = 1350$$
$$()x = 2^1 \times 3^2 \times 5^3 = 2250$$
$$x() = 2^3 \times 3^1 \times 5^2 = 600$$

If $y = 11$, the two sentences $()yx$ and $()((x$ are as follows:

$$()yx = 2^1 \times 3^2 \times 5^{11} \times 7^3 = 301,464,843,750$$
$$()((x = 2^1 \times 3^2 \times 5^1 \times 7^1 \times 11^3 = 838,530$$

These are all unique, and so Gödel provided a mathematically straightforward method of mapping sentences to the natural numbers. Gödel uses this method

to not only map mathematical formulae of his chosen system, but also all *meta*-mathematical formulae. Doing this, he could substitute meta-mathematical assertions into his regular formulae which allowed him to construct a self-referential formula which stated its own negation.

The technical details of this are complex [64], but the implications are broad and deep across all fields of the mathematical sciences. Gödel essentially discovered the existence of problems that cannot be solved. To try to solve these will obtain a paradox. These problems are known as *uncomputable*, or *undecidable*.

Gödel's numbering technique has applications outside of his proof. An *enumeration* of programs is a size ordering using the alphabet of the programming language. Because any data drawn from a finite alphabet can be enumerated, there exists a Gödel numbering function which can enumerate all programs written in some language. The proofs and proof outlines in Sections 2.2.2, 7.3.1, and A.2 rely on this.

2.1.2 Church and Turing

In 1936, The American logician Alonzo Church and British mathematician Alan Turing were both concerned with the notion of what an algorithm is and how to formalise it. Church devised an abstract substitution system known as the λ-calculus [11] (Section 2.3.2.1) while Turing created a set of hypothetical machines [95] (Section 2.3.1.1).

Despite looking and operating in completely different manners, it can be shown that these two models of computation are equivalent. This means that every function that we can write in the λ calculus has a corresponding function in Turing machines. The most straightforward proof of this lies in the power of *universal machines*.

At its most basic level, a universal machine UX is a machine that will run any program which is written in some model X. For instance, Turing's seminal paper introduces the UTM, a Turing machine that takes as inputs on its tape, a description of another TM M and some input tape for M, say T. The UTM then executes the machine M against the tape T. In essence, Turing wrote an interpreter for Turing Machines in the language of Turing Machines.

Universal machines can be made to prove that the λ-calculus is equivalent to the TM model. Since we know that a universal machine for TMs can be written and that equivalent models can represent the same functions, let us assume we can write a UTM U in the λ-calculus. This is a λ term that takes a machine and tape encoded as λ terms and executes the machine on the tape.

Consider a hypothetical program P that can be written in the TM but not in the λ-calculus. The existence of a UTM λ term means that any TM can be encoded as a λ expression and then executed according the the rules of TMs. So if U can be written, then a TM program inexpressible in the λ-calculus such as P cannot exist.

Implementing a UTM in the λ-calculus is fairly straightforward [96] (Section 4.4.1.3), so we know that the λ-calculus can express all functions that a TM can. To show that the TM can express all the functions of the λ-calculus, the converse needs to be constructed. Writing a TM to evaluate any arbitrary λ expression is also achievable [96] so we can state with confidence that the λ calculus and Turing Machine computational models are equivalent.

This equivalence forms the basis of Church's (later the Church-Turing) thesis. This states that any function that can be computed is λ-*definable*, and by extension can be computed by the λ-calculus and Turing Machines [97]. Many other models of computation have been shown to be CT conformant such as Tag systems [72], Markov algorithms [57], RAM machines [63], and RASP Machines (Section 2.3.1.2).

The formalisation of this notion of computability ended a chapter of a search that started with Hilbert. It allows for an immediate and intuitive notion that if a problem is computable by a Turing machine, then it is computable in other models of computation equivalent in power to a Turing machine. If model A is equivalent in power to a Turing machine, then one can use Gödelisation to translate a TM encoding into an encoding suitable for A. A model equivalent in power to Turing machines is said to be *universal*.

2.1.2.1 Universal Machines

A universal Turing machine is a Turing machine that can simulate any universal system. The machines presented here follow a narrower definition in that they simulate the Turing machine model of computation. Each machine uses an internal TM representation that could be considered to be natural in that there is a clear mapping between the tuples of the machine to be simulated and the data/expression which is meant to represent the machine.

A machine is said to be universal if it simulates *any* universal system. Universal Turing machines can also be strong, semi-weak, or weakly universal. The tape of a a weakly universal machine has an infinitely repeated *word* (a string of symbols) extending to the left of the input machine (semi-weak), or a word extending to the left and another word extending off to the right (weak). In these machines, the tape is not a passive and initially informationless medium which is merely read from or written to, but is an active part of the information of the system. Strong universal machines do not have these repeated patterns, and the unbounded tape is always initially blank.

The universality of a machine does not make any guarantees about which universal system is simulated. One of the smallest strong universal machines is from Rogohzin. It is a four state, six symbol UTM of 22 tuples and it is not currently known if there is a smaller machine [77]. Universal though it is, Rogohzins machine does not directly simulate TMs. It simulates another universal model of computation known as 2-tag systems. In accordance with the Church-Turing Thesis, any arbitrary TM can be transformed a 2-tag system, but the process to do so is quite involved [66].

The universal machines measured in this thesis (Section 4.4) are so-called "direct simulation" machines. These machines simulate the universal machine UX of the model X by running a suitably encoded program for X using the semantic rules (Sections 2.4 and 3.3.1) of X. The machine UX can be written in any computational model as long as that model is as computationally powerful as the model X.

2.2 Information and Algorithmic Theories

Shannon first investigated the field of information theory in 1948 [83]. His work not only concerned the engineering required to transmit a message, but also the context of the message between the transmitter and receiver. This dual approach allowed him to also investigate encoding schemes for the English language as well as engineering aspects such as bandwidth and signal to noise ratios.

When transmitting information between two parties there are a number of assumptions made about the message. In the most general sense, we assume that both the sender and receiver have the same semantics with which to interpret the message. A natural example is the assumption of a common language between the sender and receiver.

This 'expected context' has implications for encoding and compressing information. As an example, we can examine the following scenario: Suppose that every day at the same time you get an email. That email can contain one of two different messages: "There has been an earthquake in the last 24 hours." or "There has not been an earthquake in the last 24 hours.". While each message is several words long, they contain surprisingly little information. Since the message only states whether there has been an earthquake, with no concern to location/magnitude/damage etc, we could replace the entire sentence with a "0" for no earthquake and "1" for an earthquake, with no information being lost.

The English language can be efficiently encoded by assigning a code to represent each letter. The length of the code is dependent on how frequently the letter will appear in a piece of text. In the English language, the letter "E" is the most common, then "T", "A", and so forth[1] down to "Q" and "Z" which are the least common [53].

A standard method of applying these variable length codes is Huffman encoding [41] which constructs a binary tree sorted by the letter frequencies. So for any given English text (with notable exceptions such as lipograms or constrained writings [104]), we can transmit the text in the most efficient way assuming that the frequencies used to construct the encoding are correct.

[1] The precise order can vary according to the texts studied, for instance, A and T are so very close to each other frequency-wise that some studies swap their position.

A natural consequence of the study of information theory is the idea of compression. If the author of the message can recognise the essential information which the message conveys, then they can write a brief message with only that information. Huffman encoding can compress text on a computer further by assigning a variable length code to every n bits (traditionally 8) which represent a single character.

2.2.1 Kolmogorov-Chaitin Complexity

Kolmogorov-Chaitin Complexity [47, 10] is the measure of randomness in a string. For a string s, the function $KC_L(s)$ returns the size of the most minimal, also known as "elegant", program in language L which will output s when run. The idea is that if s has some structure, then there will exist a computer program which is smaller than the length of s. If s is truly random, then $KC_L(s) \geq size(s)$ since the only way to express s will be to write it out. For example the string s:

$$s = xyzxyzxyzxyzxyzxyzxyzxyzxyzxyz$$

has a regular structure which consists of the repeated morpheme "xyz" 10 times. Writing a sentence like "xyz 10 times" is shorter than writing the string out in full. The information of the string is compressed into fewer characters without any loss of information so $KC_L(s) = 12$. In contrast the string:

$$s' = ss783hsh23sh24156ejflau356hqndgph03jaxfwhg0aqfhrfsry$$

has no discernible structure. So to convey all of the information in the string, it needs to be written out in full. $KC_L(s') \geq 52$. If there is no structure to a string, and all that the resultant program can do is just print the string as above, then it is *incompressible*. The above function can be generalised. $KC_L(s|x)$ is the function which returns the size of the most minimal program in L which returns the string s when run with the input x.

The invariance theorem for Kolmogorov-Chaitin complexity states that for a string s, the language we use LU and an ideal language LI (in which $KC(s)$ is

the most minimal for any L), there exists an overhead c such that:

$$\forall s : KC_{LI}(s) \leq KC_{LU}(s) + c$$

This is to say that to translate from one language to another requires a constant size program which performs the task. So the Kolmogorov-Chaitin complexity for any arbitrary string in some language is some constant c characters from the ideal size.

2.2.2 Elegance

Chaitin defines an *elegant* program p for the output/string s in language L as the shortest program written in L which outputs s. In other words, there is no smaller program (fewer characters) which can be written in L which outputs s:

$$KC_L(s) = size(p)$$

We cannot in general decide if a program p is elegant [10]:

Theorem 1 (Undecidability of Elegance (Chaitin))**.** *In general, it cannot be determined that a program p is an elegant program for the output s over a certain threshold of size.*

Proof. Assume there exists an 'elegant tester' program ET which takes a program P as input and returns true if P is an elegant program and false otherwise.

Consider the program B which takes a number n and enumerates (via some Gödel numbering method) all possible programs Pn which are longer than n. For each program in Pn, B runs ET against it until ET returns true. Once an elegant program K has been found, B runs K.

If $size(n)$ is the size of n encoded as an input of B, consider the case of B with $n > size(B) + size(n) + 1$ so that any Pn generated is greater in size than B with n. There are an infinite number of elegant programs, so ET will state that one (K) is elegant. However B runs K and therefore returns the result of K. The combined sizes of B and n are lower than the size of K, so the function ET cannot do what it is assumed it can do. □

The elegance of programs can only be proven up to a certain size ($size(B) + size(n)$), so elegance is undecidable in general. This formulation of elegance only refers to programs returning a singular output s, a so-called "constant" function. However, for any given language L there may exist some programs which are of a size below that of B and perform some general function such as addition. In other words, there may exist an elegant formulation p such that for an input x and output y in a function F:

$$KC_L(y|x) = size(p) + size(x) + size(y)$$

for all x and y in F. Section 3.1 proves that such a function cannot exist, and Section 6.6 gives a concrete example of programs which exhibit the contradiction obtained.

Despite these challenges, the concept of elegant programs has been drawn on as inspiration for comparisons. Elegance itself cannot be directly compared across languages because the semantics of languages are not included in the definition. The semantics of a language affect how easily arbitrary algorithms can be realised (expressiveness, Section 2.5), so we can question how the elegance of a set of functions realised in language A compares to the elegance of the function in language B with a different level of expressivity.

2.2.3 Other Measures of Complexity

Software Science, more colloquially known as Halsteads Complexity measures, is a field which attempts to characterise aspects of algorithms and programs in order to assess the difficulty of implementation, approximate length of a program, and even the time to implement such programs [34].

Halsteads model and others (like Cyclomatic Complexity [61]) are built on a series of mathematical formulae. These formulae use counting metrics of the program like number of unique variables, number of unique operands, total variable occurrences, and total operand occurrences. The formulae then purport that the complexity of the program can be calculated with respect to how easy it is to implement and understand in an arbitrary language.

If such a system of formulae were to exist, it would be very useful. However, such complexity metrics tend to fall short of their claims when subjected to theoretical and empirical scrutiny [85, 84]. It is hard to accept that nebulous concepts such as the complexity of a program, and how easy it is to understand and write can be ascribed to these metrics. So much depends on a programmer's style and skill.

Software metrics are an attractive idea, but their present immaturity and lack of rigour does not make them a suitable characterisation of the information contained in a program over a more simple metric such as the number of characters or bytes.

2.3 Models of Computation

A model of computation is an abstract formal system consisting of a set of operators, a grammar for forming statements and a semantics which evaluates the operators of the model in a consistent manner. Models have an associated language that is the result of combining the operators with the grammar. We shall use the terms "language" and "model" synonymously.

For a model to be considered Turing Complete, it must be capable of representing a UTM as described in 2.1.2. All of the models in this section are Turing Complete, and their respective UTMs are described in Section 4.4.

2.3.1 Imperative/Procedural Languages

Imperative models of computation have a structure much like a recipe. A program is a list of instructions which are executed in a sequential fashion.

Figure 2.2 shows a small imperative program which uses the procedure `add()` three times. The flow of control starts at the top of the`main()` procedure. Variable x is assigned with a call to `add(4,3)`, in which the flow of control 'jumps' into the `add()` procedure, and then 'jumps' back once the addition has been performed. Variable y is then assigned with another call to `add(2,7)`. With $x = 7$ and $y = 9$, the final call to `add()` finishes the program returning the value 16.

Imperative languages are typically easy to follow, but writing a program can

```
int main(){
  int x = add(4,3);
  int y = add(2,7);
  return add(x,y);
}

int add(int x, int y){
  return x + y;
}
```

Figure 2.2: A code snippet of a procedural program.

require that the programmer interact significantly with the underlying machine, especially in an older language like C or C++. Tasks like allocating and initialising memory may not be handled by the semantics of simpler imperative languages. This puts more stress on details which are not directly related to the problem.

Programming languages are either *pure* or *impure*. Functional languages are distinguished from imperative languages by exhibiting purity in the entirety of the language, or in a significant part. One of the important aspects of purity is *referential transparency*. A function, or sub-program is referentially transparent if the function can be replaced with its return value without affecting the rest of the program.

In other words, the function does not change any global state of the abstract machine running the program. In Figure 2.2, the `add()` function is referentially transparent. The calls in `main()` of `add(4,3)` and `add(2,7)` can be replaced with 7 and 9 respectively without affecting the rest of the program.

Consider a global variable t, which is a variable that can be accessed and used by any part of a program. If the `add()` function is Figure 2.2 were to change t when called, then the function would lose referential transparency, because the changing of t is a side effect. The `add()` function does not just return a value, it changes the global state of the program.

Modern functional languages often requires that the programmer specifies only which structures are used and how they are used to solve the problem. The semantics of the functional language dictate how this more abstract solution is to be implemented on the inherently stateful underlying machine without much,

if any, intervention from the programmer.

2.3.1.1 Turing Machines

The Turing machine (TM) is a model of computation introduced by Alan Turing [95]. Turing machines come in many variations, but the most common consists of a state machine with a read/write head positioned over a *tape* made up of *cells*. Each cell can hold a single symbol and can be overwritten as many times as needed. The tape is unbounded in both direction, so additional cells may be added as required.

The machine has a read/write head that can read a symbol from and write a symbol to a single cell of the tape. It can also move the tape one square to the left or one square to the right.

At any given moment, a TM can be in one of a number of states. A particular state and symbol pair informs the machine what to do next according to the symbol table. The symbol table is a function:

$$ST : \text{STATE} \times \text{SYMBOL} \mapsto \text{STATE} \times \text{SYMBOL} \times \text{DIRECTION}$$

which takes the current state of the machine: $state_{old}$ and the symbol currently under the head: $symbol_{old}$. It returns a new state to transist to: $state_{new}$, symbol to write: $symbol_{new}$, and direction in which to shift the tape: dir.

$$\langle state_{old}, symbol_{old} \rangle \mapsto \langle state_{new}, symbol_{new}, dir \rangle$$

It is possible that the function ST does not return a result for the current state and symbol pair. In this case, we have not defined what the machine should do next, so it just halts. As a convention in this thesis, Turing machines will start in state 1, the read/write head is initially positioned over the left hand side of our tape input (if not explicitly defined to be elsewhere), and a transition to state 0 halts the machine. The machine will also halt if it encounters an undefined state/symbol pair. There is no distinction between halting by 'legitimately' transisting to zero, or encountering an undefined state/symbol pair.

Consider a simple machine to invert a sequence. This sequence is defined as a string of either '1' or '0' ended with two instances of '1' in a row. For instance "1010100010011" is a sequence. A machine to invert this sequence will start at the left hand side of the sequence and proceed by overwriting any 1s with 0s and 0s with 1s. It will halt when the machine reads the second '1' in a row. The symbol table for this machine is:

$$\langle 1, 0\rangle \mapsto \langle 1, 1, R\rangle$$
$$\langle 1, 1\rangle \mapsto \langle 2, 0, R\rangle$$
$$\langle 2, 0\rangle \mapsto \langle 1, 1, R\rangle$$
$$\langle 2, 1\rangle \mapsto \langle 0, 0, R\rangle$$

This symbol table consists of four transitions, two for each state. Every time a '0' is read, the machine transists to state 1. If the machine is in state 1 and it reads a '1', it will transist to state 2. Reading another '1' while in state 2 will halt the machine by transiting to state 0.

2.3.1.2 The Random Access Stored Program Machine

The Random Access Stored Program (RASP) machine [23, 16, 36] is a register machine with a Von Neumann memory architecture [32]. A register machine can intuitively be thought of as a computer processor with a set of registers to hold both the program and data.

A Random Access Machine (RAM) is a register machine with two sets of registers, one set contains the program, and another set contains the data. The program can read and write to the data registers, but cannot write to the program registers [78]. This establishes a boundary between program and data which emulates a "traditional" idea of programming such that this memory model is supported by most mainstream languages by default.

In general, the RASP model makes no distinction between program and data which are combined into a single register space. It is therefore conceivable that instructions can be considered as data and vice versa.

The RASP machine was conceived by Elgot and Robinson [23] as an attempt to introduce the notion of an extensible model which can be discussed from a

semantic viewpoint. They define a RASP as an ordered sextuple:

$$P = \langle A, B, b_0, K_o, h^1, h^2 \rangle$$

The first four items are described below:

- A and B are possibly infinite, overlapping, or coinciding sets of *addresses* and *words* respectively.
- $b_0 \in B$ is the empty word.
- $K_o \subseteq K$ is the set of *content functions* such that $k(a) = b$, where $k \in K_o$, $a \in A$, and $b \in B$

For each $k \in K$, every $a \in A$ such that $k(a) \neq b_0$ is part of a set known as the *support* of k. Finally every k with a finite support is a member of the set K_f. K_o is a subset of K and is *finitely supported* if $K_o = K_f$.

Let $\Sigma = K \times A$ and $\Sigma_o = K_o \times A$ be sets of machine states. The function $h^1 : \Sigma_o \times B \mapsto K_o$ executes a word in B to obtain a new content function. The function $h^2 : \Sigma_o \times B \mapsto A$ executes a word in B to obtain the next address. These mappings can be combined into $h : \Sigma_o \times B \mapsto \Sigma_o$ such that given a machine state and word to execute, the machine derives both the next content function (via h^1) and next address(h^2) which is the new state:

- $h^1 : \Sigma_o \times B \mapsto K_o$ executes a word to obtain a new content function.
- $h^2 : \Sigma_o \times B \mapsto A$ executes a word to obtain a new address.

Elgot and Robinson's first order and set theoretic treatment of the RASP describes the implementation of general recursive functions and introduces the idea of language extensions termed *definitional extensions*. It is clear that they intended to use the RASP model as a basis for the implementation of semantics of programming languages and studying how the addition of new definitions would affect the languages. This initial treatment of semantics influenced the development of PL/I [55] and (by means of the Vienna Definition Language) SOS [71]. However, using the RASP machine to specify these semantics never really gained traction.

The RASP has been used to study computational complexity. Cook, Reckhow and Hartmanis [16, 36] have investigated the time complexity of self modifying

programs relative to fixed ones. Hartmanis discovered that RASPs have the potential to be faster than a RAM or Turing machine due to this self modification.

Hartmanis defines a RASP as a pair $\langle M, I \rangle$ of a machine M and set of instructions I. M contains two special registers; an instruction counter (IC), and the accumulator (AC). These two registers are at the beginning of the memory, and the rest of the memory consists of an unbounded sequence of registers. Each register can hold an arbitrarily sized but finite binary sequence.

Register #	Content
...	...
R_5	1
R_6	5
...	...

Figure 2.3: Indirection, accessing the address stored in R_6: $\ll 6 \gg = <5> = 1$

The contents of a register R_n is denoted $<n>$, similarly $<IC>$ and $<AC>$ refer to the contents of the instruction counter and accumulator. Indirection is indicated with $\ll n \gg$ which is explained in Figure 2.3.

There are 7 instructions in the instruction set I, some of which can take different types of parameters. For example the ADD instruction can add a natural number to the accumulator, but it could also add the contents of another register to $<AC>$, or even the contents of the address held in some other register. Each register in this model holds a single instruction + data and after an instruction (except HALT) is executed, $<IC>$ is incremented for the next register. The instructions I of Hartmanis are explained in Table 2.1.

The instruction set is at first quite appealing, but the minutiae of implementation would prove to be quite finicky. Consider for example the case of instructions taking one of several types of input, we see that we would either have to devise an encoding scheme that indicates if the parameter to functions are direct or indirect, or we would have to split the instructions out into special cases (i.e ADD, ADDi, ADDd for the cases of n, $<n>$, and $\ll n \gg$). Furthermore, since a register holds both the instruction and data, there is no clear way to change one or the other so that the machine can self modify. If there exists some Gödelesque encoding for each $\langle instruction, data \rangle$ pair, we would have to load the contents of that register and carefully edit it to change either the instruction, or the data.

Name	Meaning
TRA n, TRA $\langle n \rangle$	Transfer control to register n or $\langle n \rangle$ respectively. i.e. $\langle IC \rangle = n$ or $\langle IC \rangle = \langle n \rangle$.
TRZ n, TRZ $\langle n \rangle$	If $\langle AC \rangle = 0$, transfer control to register n or $\langle n \rangle$ respectively.
STO n, STO $\langle n \rangle$	Store $\langle AC \rangle$ in register n or register $\langle n \rangle$ respectively.
CLA n, CLA $\langle n \rangle$, CLA $\ll n \gg$	The values n, $\langle n \rangle$ or $\ll n \gg$ respectively are stored in AC. The contents of R_n and $\langle R_n \rangle$ are not altered.
ADD n, ADD $\langle n \rangle$, ADD $\ll n \gg$	$\langle AC \rangle$ is replaced by $\langle AC \rangle + n$, $\langle AC \rangle + \langle n \rangle$, or $\langle AC \rangle + \ll n \gg$ respectively.
SUB n, SUB $\langle n \rangle$, SUB $\ll n \gg$	$\langle AC \rangle$ is replaced by $\langle AC \rangle - n$, $\langle AC \rangle - \langle n \rangle$, or $\langle AC \rangle - \ll n \gg$ respectively.
HALT	The machine stops and no further instructions are executed.

Table 2.1: Instructions of Hartmanis

These issues lead us to believe that Hartmanis was defining his RASP as more of a RAM machine, where the data is simply appended to the end of the program and where the program does not actually modify itself, but does modify the same piece of memory which holds the program and data. This implementation is formally congruent to the specification of Elgot and Robinson, but is not as interesting as a RASP which can modify its own program.

In contrast to the above, the model used in this thesis is predominately finite through the restriction of sets A and B. RASP sizes are specified in terms of "n-bits" and an n-bit RASP has 2^n registers, each of which can hold a single natural number up to 2^{n-1}. The registers themselves are numbered in the range 0 to 2^{n-1}.

Registers, 0, 1 and 2 have specific functions which are used to keep track of the state of the machine. Register 0 is the Program Counter (PC, analogous to the IC) which points to the current register being executed. Register 1 is the Instruction Register (IR) where the contents of the address in the PC is copied for decoding and execution, Register 2 is the Accumulator (ACC, analogous to the AC) upon which all of the arithmetic instructions operate. When a RASP

Natural	Command	Effect
0	HALT	Halt the machine.
1	INC	M[ACC] $\leftarrow$ M[ACC]+1
2	DEC	M[ACC]$\leftarrow$M[ACC]-1
3	LOAD x	M[ACC]$\leftarrow x$
4	STO x	M[x]$\leftarrow$M[ACC]
5	JGZ x	IF M[ACC] > 0 THEN M[PC] $\leftarrow x$
6	OUT	Output the current value of the accumulator.
7	CPY x	M[ACC] $\leftarrow$ M[x]

Figure 2.4: The effects of each instruction on a RASP machine M

machine is parsed by the semantics (Section 3.4.2), the PC, IR, and ACC are initialised to 3,0,0 which can be thought of setting the IR and ACC to 0, while the PC points to the first instruction of the program.

There are 8 instructions in the RASP machine with each instruction mapped to a natural number. Figure 2.4 shows the effects of each instruction on a RASP machine M, where M[y] is the value stored in address y of the machine. This instruction set borrows from Cook and Reckhow's definition in [16], but has some notable differences:

- No negative numbers.
- Finite number of registers and the size of a number which can be stored.
- INC and DEC rather than ADD/SUB.
- No READ for external input.
- Explicit CPY instruction for indirection.

In the event of an over- or underflow due to the execution of INC and DEC statements or the incrementing of the PC, the machine will carry on as normal. An overflow will set the the affected register back to 0 and an underflow will set it to $2^n - 1$. If the machine attempts to decode and execute a natural number that is not in the range 0-7, the machine will halt.

The RASP machines of this thesis operate according to the fetch execute cycle shown in Algorithm 1. If a machine were to execute the LOAD instruction it would first copy the instruction from the memory address pointed to by the PC into the IR. Decoding the LOAD would prompt an increment of the PC and a further fetch of the parameter into the IR. Once this has been done, the LOAD command will be fully executed by setting the ACC to the value which

```
while not halted do
    M[IR] ← M[M[PC]];
    if M[IR] > 7 then
        Halt;
    end
    if instruction requires a parameter then
        M[PC] ← M[PC]+1;
        M[IR] ← M[M[PC]];
    end
    Execute instruction;
    if last executed instruction was not a successful jump then
        M[PC] ← M[PC]+1;
    end
end
```

Algorithm 1: RASP Fetch-Execute cycle.

Instr	Data	I Label
3		:PC
0		:IR
0		:ACC
STO	'here	:here
INC		
JGZ	'here	

Figure 2.5: An example of a RASP that will self modify in order to halt.

is currently held by the IR. The machine increments the PC again and continues on to the next instruction.

The most prominent feature of the RASP is the ability to self modify and change the running program. Figure 2.5 shows an example of a machine which does this. RASP machines are displayed using this form to make them readable. A RASP machine which is to be executed by the semantics is expressed as a linear array of natural numbers. For example the above machine (ignoring the initial values for the PC, IR, and ACC) is: 4,3,1,6,3. Each number represents an instruction, piece of data, or both. And while compact, this form is difficult for a reader to parse. This thesis will primarily deal with the more readable form as shown in Figure 2.5.

Labels come in two forms: instruction labels and data labels. These labels are prefixed with a ':' and a ';' respectively and are used as pseudo-variables/comments and refer to the memory address of the instruction or data to which it is attached. Labels can be referred to by a prefixed ' which should be read as "the memory

address of the labelled information in the machine". The machine in Figure 2.5 uses a label ":here" to refer to the address holding the STO instruction. This address is 3, so when "STO 'here" is executed, the machine really executes "STO 3".

The first action of the machine in Figure 2.5 is to store the contents of the ACC at address 2 (0) in register 3, overwriting the STO command. Then the machine increments the ACC, changing it to 1, and jumps back to register 3 due to the ACC being greater than 0. At register 3, the instruction 0 is decoded and executed and the machine halts.

While the ⟨instruction,data⟩ pairs and labels are used as representations in this thesis to aid of understanding, the RASPs are measured in the comma delimited form: $3, 0, 0 \ldots$ as described in Section 3.3.1.

Recalling the canonical definition of Elgot and Robinson above, we now map the RASP of this thesis on to that definition. The sets A and B of an n-bit RASP machine are: $A = B = \{0, \ldots, 2^n - 1\}$ and the empty word b_0 is the HALT instruction, or 0.

Because of the strict co-incidence A and B, the set of content functions K, for these RASPs is slightly different from the original definition. The concepts of content functions, states, and the state transition functions $h^1$and h^2 are mixed up in this definition. For the RASPs of this thesis, the state of the memory provides all the information required to obtain the next state. Thus a state is not a combination of $K \times A$, but is just the content function $k \in K$. If K represents every possible mapping of $A \mapsto B$, the set K_o is the set of states that the machine running a particular program can be in. We can see that the Σ term is not required to describe the state, as it will be $\sigma = \langle k, k(0) \rangle$ for every k.

This has a knock-on effect for h. Given a RASP state, a fetch determines the next instruction to be executed. In doing so, the state of the machine is set to an intermediate state (as the IR changes). Execution then changes the state again as it applies the instruction in the IR to the machine.

We can coerce the fetch execute cycle in terms of h and σ, but can rewrite all

of the functions in terms of k:

$$\begin{aligned} h(\sigma, b) &= \langle h^1(\sigma, b), h^2(\sigma, b) \rangle \\ &= \langle h^1(\langle k, k(0) \rangle, k(k(0))), h^2(\langle k, k(0) \rangle, k(k(0))) \rangle \end{aligned}$$

A better alternative for the functions h^1 and h^2 is a single function $f : K \mapsto K$ which takes a state k, and evaluates using the fetch-execute to produce k'. The updated expression for an n-bit variant of our RASP (taking HALT as 0) is therefore:

$$P = \langle A : \{0 \ldots 2^n - 1\}, 0, K_o, f \rangle$$

The specifics of the function f are described by the semantics of the RASP machine which are explored in detail in Section 3.4.2.

2.3.1.3 Variations of the RASP

While the RASP is perfectly usable as a model of computation, addition and subtraction are laborious processes. If there are multiple case of addition/subtraction in a large program, encapsulating add/sub in a pseudo-function and calling this function when required can save time and space.

The calling is performed by copying the data and the return address into the relevant memory, jumping to the first instruction in this function and then retrieving the final value once the function returns.

Figure 2.6 shows an example of a reusable addition function. The first block of instructions store the numbers 6 and 5 in the second block, store where the function should jump back to and jump to the start of the addition function. The addition function itself adds the two parameters together and jumps back to the indicated location once Param1 is zero.

This approach works reasonably well for moderately sized programs, but for very large programs with many such calls it would be preferable to also implement an execution stack which can generalise the function call.

We can iterate on the basic RASP in two different ways by replacing INC and DEC with ADD x and SUB x. Table 2.2 states the effects of the new instructions. RASP2 will use ADD x and SUB x, where x is a value, such that ADD 3 will add the value of 3 to the accumulator. RASP3 will also use ADD x and SUB x, but

Instr	Data	I Label	D Label
LOAD	6		
STO	'Param1		
LOAD	5		
STO	'Param2		
LOAD	'retAddress		
STO	'returnAddr		
JGZ	'AddStart		
CPY	'Param2	:retAddress	
HALT			
LOAD	0	:AddStart	;Param1
JGZ	'add		
LOAD	1		
JGZ	0		;returnAddr
DEC		:add	
STO	'Param1		
LOAD	0		;Param2
INC			
STO	'Param2		
LOAD	1		
JGZ	'AddStart		

Figure 2.6: An example of a RASP pseudo function and calling code

Integer	Command	RASP2	RASP3
1	ADD x	M[ACC]←M[ACC]+x	M[ACC]←M[ACC]+M[x]
2	SUB x	M[ACC]←M[ACC]-x	M[ACC]←M[ACC]-M[x]

Table 2.2: The ADD and SUB instructions for a RASP2/3 machine M

```
factorial 0 = 1
factorial n = n * factorial(n-1)
```

Figure 2.7: A Haskell program for computing factorials.

the x is a memory address where the value is held. ADD 3 is akin to ADD M[3] which adds the contents of the memory at address 3 to the accumulator. In doing this, we eliminate the requirement for a generalised function for addition in the RASP programs. This means that a RASP2 or 3 program will be significantly shorter than a RASP program which performs additions.

2.3.2 Functional Language

Informally, functional languages put the onus on *specifying* a problem rather that the minutiae of solving it [100]. Programs written in a functional language tend to resemble mathematical formulae rather than the 'recipe' of instructions of an imperative language.

For instance, the mathematical definition of the factorial function is:

$$fact(n) = \begin{cases} n = 0 & : 1 \\ n > 0 & : n \times fact(n-1) \end{cases}$$

This is a recursive function. `fact(n)` will call itself until $n = 0$ and then the resulting product will combine $n \times n-1 \times n-2 \times \ldots \times 1$ to return the answer. Figure 2.7 shows the definition of the factorial function in Haskell, a functional programming language [40]. There are many different ways to express this function in Haskell, including using an if/then/else structure – similar to what you might find in an imperative language, or using a fold function over a list of 1 to n, but this method (*pattern matching*) captures the simplicity of the mathematical definition.

Functional languages are often more abstract than imperative ones. Modern functional language implementations process a number of aspects of a users program like allocating memories, performing pattern matching, and determining the flow of control. The automated handling of these tasks eases the burden on the programmer and reduces areas in which bugs can occur [42]. Requiring the pro-

grammer to only mathematically specify the problem can lead to more (Chaitin) elegant programs compared to imperative languages, which require much more interaction with the machine. This abstraction comes at a cost however. The automation of interaction with the underlying machine are contained in the semantics of the language making them larger than their imperative counterparts.

2.3.2.1 λ-Calculus

The λ-calculus was devised by Church [12, 11] and is a model of computability that relies on substitution and abstraction. The abstract syntax for this language is:

$$E := \lambda v.E | (E\ E) | v$$
$$v \in \{a \dots z\}^+$$

The calculus is made up of λ terms generated from this grammar which are evaluated via some evaluation strategy. Evaluation is performed by substituting expressions and values in for variables, also known as β-reduction, each of which is a computation step. As an example, consider a very simple λ term:

$$(\lambda x.xxy)P$$

This term consists of a λ term $(\lambda x.xxy)$ and an atom P (which could potentially be another λ term). We say that the variable x in the term is *bound* by the λ, and that the variable y is *free*. A step of β reduction will replace all occurrences of x in the term with the atom P, but leave the y as it is. There are two occurrences of x in the body of the expression, so we remove the $\lambda x.$ and replace each (newly freed) x with P. This is a single step of β reduction and results in the term PPy.

Consider:

$$(\lambda x.\lambda y.y)PQ$$

This λ term has two bound variables: x and y, and two atoms: P and Q. The first step of β reduction replaces all occurrences of x with P. There are no occurrences of x, so P is effectively "deleted" from the expression giving:

$$(\lambda y.y)Q$$

We then execute the next reduction to obtain Q. When performing β reduction, we substitute for the very leftmost bound variable first. If there is no expression with which to substitute for the leftmost bound variable, then the sub-expressions are evaluated. This is known as *normal order/leftmost outermost* evaluation and an expression which cannot be further evaluated is in *normal form*.

There do exist other evaluation strategies like *applicative order/leftmost innermost*, where a term such as $(\lambda x.(\lambda a.a)(\lambda b.b)x)(\lambda y.y)$ reduces $(\lambda a.a)(\lambda b.b)$ first, and reduction to *weak head normal form*, where evaluation stops when the leftmost abstraction does not have an available reduction ($(\lambda x.(\lambda a.a)(\lambda b.b)x)$ in weak head normal form). However full normal order reduction is the only reduction strategy considered in this thesis.

The term $(\lambda x.x)$ is known as the *identity function* which takes a single argument and returns it. $(\lambda x.\lambda y.x)$ and $(\lambda x.\lambda y.y)$ are known as the *true* and *false* functions. They both take two arguments and *true* returns the first argument while *false* returns the second:

$$\begin{aligned}
\text{TRUE A B} &\equiv (\lambda x.\lambda y.x)\text{A B} \\
&\Rightarrow_\beta (\lambda y.\text{A})\text{B} \\
&\Rightarrow_\beta \text{A} \\
\text{FALSE A B} &\equiv (\lambda x.\lambda y.y)\text{A B} \\
&\Rightarrow_\beta (\lambda y.y)\text{B} \\
&\Rightarrow_\beta \text{B}
\end{aligned}$$

They can also be though of as the *select first* and *select second* functions.

Application is left-associative, so the reduction of a λ term (ABC) proceeds with A applied to B, then the result applied to C. The fully bracketed notation is $((AB)C)$, but we omit the extra ones for brevity. Brackets inside an expression denote the application order if not left-associative as described above.

The natural numbers in the λ-calculus can be represented by the "Church numerals" [11], which are *higher order functions* (HOFs). HOFs take another function as an argument or return some function as an output. While every

lambda term with an abstraction is a HOF, the Church numerals are a particularly good example of the higher order property.

Church numerals are functions which take two λ terms. A number n applies the first argument n times to the second one.

$$\begin{aligned}
\text{ZERO} &\equiv \lambda f.\lambda x.x \\
\text{ONE} &\equiv \lambda f.\lambda x.fx \\
\text{TWO} &\equiv \lambda f.\lambda x.f(fx) \\
\text{THREE} &\equiv \lambda f.\lambda x.f(f(fx)) \\
n &\equiv \lambda f.\lambda x.f^n x
\end{aligned}$$

Church numerals can be combined using other λ terms to produce the arithmetic functions. The successor function $s()$ adds one to a number n:

$$s(n) = n + 1$$

The implementation of $s()$ in the λ-calculus adds an extra 'f' to the left of a numeral n to obtain $n + 1$:

$$\begin{aligned}
\text{SUCC ZERO} &\equiv (\lambda n.\lambda f.\lambda x.f(nfx))(\lambda f.\lambda x.x) \\
&\Rightarrow_\beta (\lambda f.\lambda x.f((\lambda f.\lambda x.x)fx) \\
&\Rightarrow_\beta (\lambda f.\lambda x.f((\lambda x.x)x) \\
&\Rightarrow_\beta (\lambda f.\lambda x.fx) \\
&\equiv \text{ONE}
\end{aligned}$$

Using SUCC, numerals can be defined in terms of other numerals:

$$\begin{aligned}
\text{TWO} &\equiv (\text{SUCC ONE}) \equiv (\text{SUCC ZERO}) \\
n &\equiv \text{SUCC}^n\text{ZERO}
\end{aligned}$$

The opposite of the successor $s()$ is the predecessor $p()$:

$$p(n) = \begin{cases} 0 & : n = 0 \\ x & : n = (s(x)) \end{cases}$$

The predecessor function decrements a natural number n if $n > 0$ otherwise it will return 0: $(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda i.i))$. Given a numeral, the function replaces the variable n and then applies the sub-expressions $(\lambda g.\lambda h.h(gf))$ and $(\lambda u.x)$ to the numeral.

If the numeral is zero, the first of the terms is deleted leaving $(\lambda f.\lambda x.(\lambda t.t)(\lambda u.x)(\lambda u.u))$. This is reduced, bearing in mind that ABC is $((AB)C)$, to $(\lambda f.\lambda u.u)$.

A non-zero numeral N produces N copies of the first term and proceeds to apply the second term to the first, and removes the third term. The $(g\ f)$ structure keeps the $(\lambda u.x)$ close to the rear of the expression. Observe the application of PRED to TWO:

$$
\begin{aligned}
\text{PRED TWO} &\equiv (\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda i.i))(\lambda f.\lambda x.f(fx)) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(\lambda f.\lambda x.f(fx))(\lambda g.\lambda h.h(gf))(\lambda u.x)(\lambda i.i)) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(\lambda x.(\lambda g.\lambda h.h(gf))((\lambda g.\lambda h.h(gf))x))(\lambda u.x)(\lambda i.i)) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(\lambda g.\lambda h.h(gf))((\lambda g.\lambda h.h(gf))(\lambda u.x))(\lambda i.i)) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(\lambda h.h(((\lambda g.\lambda h.h(gf))(\lambda u.x))f))(\lambda i.i)) \\
&\Rightarrow_\beta (\lambda f.\lambda x.((\lambda i.i)(((\lambda g.\lambda h.h(gf))(\lambda u.x))f))) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(\lambda g.\lambda h.h(gf))(\lambda u.x)f) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(\lambda h.h((\lambda u.x)f))f) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(f((\lambda u.x)f))) \\
&\Rightarrow_\beta (\lambda f.\lambda x.(f(x))) \\
&\equiv \text{ONE}
\end{aligned}
$$

Note that lines 6-9 have the sub-expression $((\lambda u.x)f)$ close to the end of the term. The final reduction applies the f to $(\lambda u.x)$ to eliminate it and therefore decrement the numeral.

PRED is more complex than the successor function because it contains redundant clauses which do not affect the ZERO term, but subtract an 'f' from any numeral which is not zero. The subtractive functions which make use of PRED are therefore larger than the additive functions which use SUCC.

The addition function nominally adds two numbers x and y together by recursively decrementing x to zero while incrementing y:

$$add(x, y) = \begin{cases} y & : x = 0 \\ add(p(x), s(y)) & : x \neq 0 \end{cases}$$

Addition in the λ-calculus with Church numerals does not follow this recursive definition however, as the higher order nature of the Church numerals can add n and m by applying SUCC m times to n:

$$\begin{aligned} \text{ADD TWO ONE} &\equiv (\lambda m.\lambda n.m \text{ SUCC } n)\text{TWO ONE} \\ &\Rightarrow^*_\beta \text{TWO SUCC ONE} \\ &\Rightarrow^*_\beta \text{SUCC(SUCC(ONE))} \\ &\Rightarrow^*_\beta \text{THREE} \end{aligned}$$

We can test for ZERO:

$$iszero(x) = \begin{cases} 1 & : x = 0 \\ 0 & : x \neq 0 \end{cases}$$

$$\begin{aligned} \text{ISZERO ONE} &\equiv (\lambda n.n(\lambda x.(\lambda a.\lambda b.b))(\lambda a.\lambda b.a)) \text{ ONE} \\ &\Rightarrow_\beta (\lambda f.\lambda x.fx)(\lambda x.(\lambda a.\lambda b.b))(\lambda a.\lambda b.a) \\ &\Rightarrow^*_\beta (\lambda x.(\lambda x.(\lambda a.\lambda b.b))x)(\lambda a.\lambda b.a) \\ &\Rightarrow_\beta (\lambda x.(\lambda a.\lambda b.b))(\lambda a.\lambda b.a) \\ &\Rightarrow_\beta (\lambda a.\lambda b.b) \end{aligned}$$

$$\begin{aligned} \text{ISZERO ZERO} &\equiv (\lambda n.n(\lambda x.(\lambda a.\lambda b.b))(\lambda a.\lambda b.a)) \text{ ZERO} \\ &\Rightarrow_\beta (\lambda f.\lambda x.x)(\lambda x.(\lambda a.\lambda b.b))(\lambda a.\lambda b.a) \\ &\Rightarrow^*_\beta (\lambda x.x)(\lambda a.\lambda b.a) \\ &\Rightarrow_\beta (\lambda a.\lambda b.a) \end{aligned}$$

The resulting function from ISZERO is either TRUE $\equiv (\lambda x.\lambda y.x)$ or FALSE $\equiv (\lambda x.\lambda y.y)$. Both functions take two arguments and TRUE returns the first, while FALSE returns the second.

The HOF properties of Church numerals can be leveraged to create succinct 'additive' functions (addition, multiplication, exponentiation). Conversely, subtractive functions (subtraction, division, square root) are large in comparison to

their additive counterparts because the predecessor function (PRED) applied to ZERO is still ZERO and PRED has to take this into account.

Lists are constructed pairwise. They are nested lambda expressions for pairs with the innermost pair including an end marker. This end marker will allow an expression to test for it so that we know when we reach the end of the list.

$$\begin{aligned}
\text{PAIR} &\equiv \lambda x.\lambda y.\lambda z.zxy \\
\text{NIL} &\equiv \lambda x.\lambda a.\lambda b.a \\
\text{HEAD} &\equiv (\lambda p.p(\lambda a.\lambda b.a)) \\
\text{TAIL} &\equiv (\lambda p.p(\lambda a.\lambda b.b)) \\
\text{NULL} &\equiv (\lambda p.p(\lambda q.\lambda r.(\lambda a.\lambda b.b)))
\end{aligned}$$

Here, NIL is the end marker and NULL is a test for that marker which returns TRUE if it is applied to NIL and FALSE if it is applied to PAIR. Additionally, HEAD returns the first element of the list and TAIL returns everything except for the first element. A three element list can be constructed with the expression (PAIR A (PAIR B (PAIR C NIL))).

Other logical connectives can be constructed to make use of the TRUE and FALSE expressions:

$$\begin{aligned}
\text{AND} &\equiv \lambda p.\lambda q.pqp \\
\text{OR} &\equiv \lambda p.\lambda q.ppq \\
\text{NOT} &\equiv \lambda p.\lambda a.\lambda b.pba
\end{aligned}$$

The fixed point combinator $\text{Y} \equiv (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))$, is a λ term with an unusual property. Given an argument term k, (Y k) will reduce to k(Y k) in some number of reduction steps. If left unchecked, the reductions will continue forever: $(\text{Y}k) = k(k(\ldots(Yk)\ldots))$. Essentially, what Y does is copy the function k to the front of the expression and apply k to (Y k).

$$\begin{aligned}
(\text{Y k}) &\Rightarrow_\beta (\lambda f.(\lambda x.f(xx))(\lambda x.f(xx)))k \\
&\Rightarrow_\beta (\lambda x.k(xx))(\lambda x.k(xx)) \\
&\Rightarrow_\beta k((\lambda x.k(xx))(\lambda x.k(xx))) \\
&\Rightarrow_\beta \ldots
\end{aligned}$$

The use of Y is the general method of implementing recursive functions. The

copying behaviour of Y allows k to accept a copy of itself as a parameter. If k is a recursive function, then a 'call' to k will begin with a copy of k being made which is to be passed into the function itself. Consider the DIV function from Section 4.2.5:

$$\mathrm{Y}(\lambda g.\lambda q.\lambda a.\lambda b.\mathrm{LT}a\ b(\mathrm{PAIR}\ q\ a)(g(\mathrm{SUCC}\ q)(\mathrm{SUB}\ a\ b)b))\mathrm{ZERO}$$

The initial reduction is the application of the fixed point combinator to the expression, producing DIV(Y DIV)ZERO. The abstraction g moves the (Y DIV) into the leading DIV which completes the recursive call.

Two λ expressions are equivalent if they have the same effect. This is a property known as *extensionality* where we care only about how the term interacts with other terms, rather than how the inside of the term is evaluated (*intentionality*). Working out if two arbitrary terms are equivalent is generally uncomputable [12]. But we have tools, known as α and η (and β reduction if the terms are not in a normal form) conversion, which we can use to convert similar terms to test for equivalence.

Consider the two terms $A = (\lambda p.(\lambda a.\lambda q.a)p)$ and $B = (\lambda a.\lambda b.a)$. These two terms could possibly be equivalent, but we have to use both α and η conversion to make sure. A term $(\lambda x.Mx)T$, where there are no free occurrences of x in M, will *always* reduce to MT for all M and all T. The abstraction over x is superfluous as it neither duplicates, nor moves T in any way. The abstraction over p in λ expression A can therefore be removed such that $A = (\lambda a.\lambda q.a)$.

We may naïvely believe that two terms abstracting over different names cannot be equivalent. This is where renaming or α conversion is called for. Renaming the variables in a term is the process of changing the name of the bound variable and the name of every variable which is bound by that λ. The expression $(\lambda x.x((\lambda x.xx)x)x)$ binds the variable x in two different expressions. The inner expression binds x twice, and the outer binds x three times.

This expression is also hard to read. So we can rename either (or both) abstractions to something different. $(\lambda y.y((\lambda x.xx)y)y)$ is a little bit easier to read, clears up any possible ambiguities and maintains the intentionality of the term.

$$\begin{aligned} Ix &\equiv x \\ Kxy &\equiv x \\ Sxyz &\equiv xz(yz) \end{aligned}$$

(a) Effects of combinators

$$\begin{aligned} I &\equiv \lambda x.x \\ K &\equiv \lambda x.\lambda y.x \\ S &\equiv \lambda x.\lambda y.\lambda z.xz(yz) \end{aligned}$$

(b) Combinator λ terms

Figure 2.8: Combinator effects and corresponding λ terms

Applying this procedure to terms A and B, we rename the bound q in A to match the b in B. Thereby showing that $A = B = (\lambda a.\lambda b.a)$.

2.3.2.2 SKI Combinator Calculus

Combinatorial logic is a simple functional model of computation developed by Schönfinkel in 1924 [79] and independently re-discovered by Curry in 1927 [82]. The SKI combinator calculus consists of three titular combinators: S, K and I. The I combinator is the identity combinator. For any x, which could be another combinator or bracketed expression, Ix is x. The K combinator takes two arguments, x and y, and returns x which is just like the TRUE function from above. The S combinator takes three arguments and reorders them: $Sxyz = xz(yz)$. Figure 2.8 lists the three principal combinators of the calculus and the λ-calculus expressions which correspond to them.

The SKI combinators have simple λ-calculus counterparts as shown above. Interestingly, these three combinators are Turing Complete. This can be shown via a process known as *bracket abstraction*[98, 17, 94] which "eliminates" bound variables by replacing the abstraction mechanisms with combinators to copy and position parameters.

In this thesis, the SKI expressions for the tested set of functions (Chapter 4) are obtained via bracket abstraction of λ-calculus terms. There are multiple methods of bracket abstraction available [98] and a recent version by Tromp [94] is an effort to reduce the size of the resultant combination as much as possible.

Bracket abstraction is a process which converts λ-calculus terms into SKI terms. It was first coined by Curry with his abstraction rules [17]. These rules

work well for expressions with a single variable to be abstracted, but the resultant SKI expression grows in size quadratically with the number of variables in the term.

Turner noticed this and created his own algorithm [98] which uses new combinators to parse out particular patterns of nested expressions to reduce the size of the resulting term. However this method uses combinators other than the standard S, K, and I.

Tromp has devised a bracket abstraction algorithm which produces succinct combinations without the use of combinators other than S, K and I [94]. Tromp's rules are applied in decreasing order as follows:

$$
\begin{aligned}
&1.)\lambda x.(SKM) && \equiv SK \text{ [for all } M\text{]}\\
&2.)\lambda x.M && \equiv KM[x \notin M]\\
&3.)\lambda x.x && \equiv I\\
&4.)\lambda x.(Mx) && \equiv M[x \notin M]\\
&5.)\lambda x.(xMx) && \equiv \lambda x.(SSKxM)\\
&6.)\lambda x.(M(NL)) && \equiv \lambda x.(S(\lambda x.M)NL)[M, N \text{ are combinators}]\\
&7.)\lambda x.((MN)L) && \equiv \lambda x.(SM(\lambda x.L)N)[M, L \text{ are combinators}]\\
&8.)\lambda x.((ML)(NL)) && \equiv \lambda x(SMNL)[M, N \text{ are combinators}]\\
&9.)\lambda x.(MN) && \equiv S(\lambda x.M)(\lambda x.N)
\end{aligned}
$$

Rules 2, 3, 4, and 9 are borrowed from Curry's original algorithm. Much like Turner's new combinators, the extra rules focus on un-nesting abstracted expressions (rules 6, 7, and 8). Rule 1 takes advantage of the fact that $SKMT \implies T$ so we are saving time and space by getting rid of M. Rule 5 avoids the introduction of a term of the form II. This bracket abstraction algorithm is the one we use to produce SKI combinations from λ terms.

With this abstraction method in mind, we can define numerals and functions like those of the λ calculus:

		OR	$\equiv SII$
ZERO	$\equiv KI$	NOT	$\equiv S(SI(K(KI)))(KK)$
ONE	$\equiv I$	TRUE	$\equiv K$
TWO	$\equiv S(S(KS)K)I$	FALSE	$\equiv KI$
THREE	$\equiv S(S(KS)K)(S(S(KS)K)I)$	AND	$\equiv SSK$

These combinations can be tested for the desired behaviour. For example, a Church numeral n takes two functions, f and x, as parameters and returns the result of f applied to x n times:

$$\begin{aligned}
\text{(TWO f x)} &\equiv S(S(KS)K)Ifx \\
&\Rightarrow_S S(KS)Kf(If)x \\
&\Rightarrow_S KSf(Kf)(If)x \\
&\Rightarrow_K S(Kf)(If)x \\
&\Rightarrow_S Kfx(Ifx) \\
&\Rightarrow_K f(Ifx) \\
&\Rightarrow_I f(fx)
\end{aligned}$$

Tromp has confirmed that the most elegant Y combinator (via brute force search [94]) for SKI corresponds to the λ-calculus expression $(\lambda x.\lambda y.yx)(\lambda y.\lambda x.y(xyx))$ and is $SSK(S(K(SS(S(SSK))))K)$ via an exhaustive search. When obtaining a SKI expression from a λ term, this combinator will first be substituted for any occurrence of Y before bracket abstraction takes place.

2.4 Semantics

A program written for a computational model M is a string of characters generated from set of grammatical rules [63]. The semantics of M are a set of rules which describe the operations of M. When semantics are applied to a program and input (typically thought of as "running the program with input i"), the semantic rules of M are executed against the data i in accordance with the program [26].

Semantics can be specified in any formal system which is powerful enough to

$$\frac{\begin{array}{c} T(h) = sy \\ \delta(st, sy) = \langle st', sy', d \rangle \\ d = L \\ T'(h) = sy' \\ h' = h - 1 \end{array}}{E(st, T, h) \implies E(st', T', h')}$$

Figure 2.9: Semantics for the TM on a left shift.

express the operations of the language. Elgot and Robinson used first order logic and set theory to initially specify the RASP [23], a methodology which helped inspire the Vienna Definition Language and Structured Operational Semantics (SOS) [71].

There are many different semantic formalisms. Each formalism tends to focus on a particular aspect of models:

- SOS are concerned about *how* an operation is performed.
- Denotational Semantics explore the *effect* of an operation [80].
- Axiomatic Semantics are often used to prove properties of the model [38].

Given the various specialities of these semantic systems, it is often required to implement a model in multiple semantic formalisms in order to fully reason about the models properties.

2.4.1 Structured Operational Semantics

Structured operational semantics define an abstract machine that can execute a program written for the model. SOS is a mathematical programming language in which we define a universal machine for the model [70]. The semantic rules for the models are often (and will be in this thesis) represented as:

$$\frac{\text{Premises}}{\text{Conclusions}}$$

where the conclusions are satisfied if and only if all of the premises are. The specification of models in this thesis have a set of state variables defined where some or all of the variables change according to the semantic rules defined.

Figure 2.9 shows a semantic rule for the Turing Machine (TM). The variables for a TM are; the current state of the TM (st), the current tape (T), and the position of the read/write head on the tape (h). These are all arguments to the E (evaluation) function shown in the conclusion of the rule. If E is executed, then the state of the machine, tape and head position will all be affected.

There are five premises for this rule. These premises are a mixture of preconditions (statements which must hold before the changes in the conclusion) and postconditions (statements which must hold after the changes).

The first three lines are preconditions: On the tape T at position h there is the symbol sy. In the symbol table δ there is an entry for the current state st and read symbol sy. The direction d in the matched entry is a left shift L.

The next two lines are postconditions: The new tape T' has the symbol sy' at position h, and the new head h' is the predecessor of the previous head. If a TM makes a state transition which includes a left shift, then all of these pre- and postconditions will be met and E will have been executed.

Say there are two rules; rule A has three premises and rule B has four. If the model matches all of the conditions of rules A and B, which rule is followed? In a situation such as this, we execute the rule which has the most premises. The full semantics for each model are presented in Section 3.4.

2.4.1.1 Parsing

Structured Operational Semantics typically does not deal with the parsing of programs [70]. The assumption being that only well formed statements which can be determined from the abstract syntax provided in the semantics are executed and that any whole or part expression is syntactically valid in the context of the rule.

This is a perfectly reasonable approach to take. Usually the parsing of the program takes a secondary role to the execution of the rules in that program. Assuming that the language can be parsed (after all, why would you write mechanical semantics for a language that *cannot* be parsed) allows one to focus on the rules rather than specifying a parser.

However an even handling of all possible models requires that expressions and

programs are first parsed before execution. Consider the array-like description of the RASP machine and the string like description of the λ-calculus. The RASP has an intuitive mapping of one number to one register that is easy to manipulate. In contrast reducing a λ expression in the string form is hard because we would have to iteratively shift parts of the expression around to make room for substitution and so forth.

It is much easier to parse a λ term into a tree structure and perform *graph reduction* (Section 3.4.3) on it which simplifies the process of reduction to moving nodes in a tree rather than shuffling characters. This transformation of the external representation to the internal representation needs to be specified though and that specification is part of the semantics.

The parsers are specified along with the semantics of the models in Section 3.4. The RASP and TM parsers are relatively succinct in comparison to the SKI and λ-calculus parsers, as they facilitate a less extreme transformation between representations.

2.5 Expressiveness

Asserting that one languages is "more expressive" than another is a problematic proposition. Intuitively, we believe that a language A, which satisfies the Church-Turing thesis, is more expressive than language B which does not. This makes sense, because we can then define a program p which can be written in A, but not B. In other words p can be expressed in A, but cannot be expressed in B.

As neat as this definition is, it is too narrow to be very useful. As we saw earlier, Turing machines can compute any function that can be computed. The C programming language [45] is one of the most widely used languages in the world. One of its primary applications is in the development of operating systems [4] and can be considered the lingua franca of imperative languages. C programs use keywords, variables and structured logic blocks in order to make the program understandable for those versed in the syntax.

We would like to draw a distinction between the languages of C and TMs, and our intuition is to say that C is more expressive given the wider range of operators

```
int i;
for(i=0;i<10;i++){
  X;
}
```

```
int i = 0;
while(i<10){
  X;
  i++;
}
```

Figure 2.10: A while loop and for loop operating in the same manner

and more flexible management of data. However if we constrain ourselves to comparing expressiveness solely on the basis of the computational power of the language, then both languages have the same expressive power. Both TMs and C are Turing Complete so this mode of comparison is not as helpful. We need to expand the definition to accommodate the distinctions above.

2.5.1 Formalisations

Elgot and Robinson [23] spared a paragraph to muse over the comparison of programming languages by implementing them with RASP machines which would result in a fully defined set of semantics to use as a baseline for language comparison. Landin first considered the question of *what* we could compare in a language [52]. He began to classify some programming constructs as essential and some as "syntactic sugar".

Figure 2.10 considers the **for** loop versus the **while** loop. Either of these looping constructs can be discarded without any effect on the computational power of the language. A similar example for higher order functional languages is the **let** construct which is a binding of a value to some variable in some expression and is equivalent to a function call.

In logic, Kleene identified the notion of eliminable constructs [46]. Coupled with the informal idea of a 'core' language [90, 75], Troelstra [93] defined the idea of a *conservative extension* S' of a formal system S as a superset of the logical expressions of S drawn from a richer set of operators. This extension allows S' to express more formulae and theorems than S, but if we were to restrict the expressions of S' to use only operators of S, then we would have exactly the formulae and expressions of S.

An extension may add computational power such that an extension S' com-

putes strictly more functions than the original language S. It may also be termed a *definitional extension* if there exists a mapping $\phi : S' \mapsto S$ which maps all expressions from the language of S' to that of S. A definitional extension does not increase the power of the formal system, since every expression in S' using the new operators can be expressed by S with its base set of operators.

2.5.2 Formalising Expressiveness

Felleisen has put substantial effort into expanding the above into a formal framework [25]. He starts by equating formal systems to programming languages and defining reciprocal definitions for conservative extensions and restrictions of programming languages. The following formulation is taken from [25].

Definition 4 (Programming Language)**.** *A programming language L consists of:*

- *a set of L-phrases, which is a set of terms freely generated from a grammar. The components of a phrase are from set of function symbols $F_1, F_2, \ldots$ with arities $a_1, a_2, \ldots$;*
- *a set of L-programs which is a non-empty recursive subset of L-phrases;*
- *a semantics $eval_L$ which is a predicate on the set of L-programs. If $eval_L(P)$ holds for some program P, then P terminates.*

Definition 5 (Conservative Extension/Restriction)**.** *A language L' is a conservative extension of L if:*

- *the functions of L are a proper subset of those of L', with the difference being $\{F_1, F_2, \ldots\}$;*
- *the sets of L-phrases and L-programs are proper subsets of their L' counterparts where there are no phrases or programs that contain the extra L' functions $\{F_1, F_2, \ldots\}$;*
- *$eval_L$ is a proper subset of $eval_{L'}$ and for all L-programs P, $eval_L(P)$ holds if and only if $eval_{L'}(P)$ holds.*

The converse is a conservative restriction.

Complementing the work of Kleene, for any extension to a Turing Complete language L, the extra functions introduced in L' can be expressed by the basic functions of L. These are known as eliminable constructs.

Definition 6 (Eliminable Constructs)**.** *Let L' be a conservative extension to L where the functions of are defined as $L' = L \cup \{F_1, \ldots, F_n\}$. The extra operators $F_1, \ldots, F_n$ are eliminable if there exists a mapping ϕ from L'-phrases to L-phrases such that:*

- *$\phi(p)$ is an L-program for all L'-programs p;*
- *$\phi(F(a_1, \ldots, a_n)) = F(\phi(a_1), \ldots, \phi(a_n))$ for all operators F of L (ϕ is homomorphic in L);*
- *$eval_{L'}(p)$ holds if and only if $eval_L(\phi(p))$ holds for all L'-programs p.*

It can also be said that L can express the facilities of L'. Finding which constructs are eliminable is achieved by showing operational equivalence between L-phrases. Felleisen defines a program context as an L-phrase or program which has a 'slot' in which we insert the L-phrase to be tested. Two L-phrases, x and y can be shown to be equivalent if and only if for every program context C, $eval_L(C(x)) = eval_L(C(y))$.

These program contexts can be thought of as individual tests, or as satisfying assignments in a proof. If two programs give identical results for each context (or satisfy a proof), then we can be sure that the two programs compute the same function.

The above definitions capture the intuitive notion of expressivity. However Felleisen wishes to impose a stricter definition where the mapping ϕ preserves program structure.

Definition 7 (Macro Eliminability)**.** *As in definition 6 above, L' is a conservative extension to L. The extra functions of L', $\{F_1, \ldots, F_n\}$ are macro eliminable if they are eliminable and the mapping ϕ fulfil the extra constraint:*

- *for each a-ary function $F \in \{F_1, \ldots, F_n\}$, there exists an a-ary syntactic abstraction A over L such that $\phi(F(e_1, \ldots, e_a)) = A(\phi(e_1), \ldots, \phi(e_a))$*

Macro expressibility defines the intuition that we would have by introducing an ADD function to the RASP. The RASP can express addition using JGZ, INC and DEC amongst others, so ϕ would swap out cases of the addition function with the appropriate L-phrase to satisfy the syntactic abstraction A. Macro

expressibility has theorems for contexts and operational equivalences as above. Section 7.3.1 discusses how the work of this thesis can be viewed in the context of this framework.

2.5.3 The Conciseness Conjecture

Felleisen concludes by asking if a language L is Turing Complete, what is the advantage of programming in an extended language L'? The advantage of the extra constructs of L' is to save programmer effort. As the size of an L-program increases, a pattern of L-phrases emerge where we frequently use these phrases to emulate the functionality of a more expressive language.

For example, addition in the RASP is a relatively large, if uncomplicated procedure. A program that uses a lot of addition would have a single instance of the procedure, and would call it when necessary. Calling a procedure in the RASP is a process of fixing values and return locations in the procedure body, then jumping to the beginning. This has a distinct structure of the kind that Felleisen discusses. A more expressive language with an addition function removes the need for these structures.

Felleisen articulates the *Conciseness Conjecture* where sensible use of the additional functions in more expressive languages results in fewer "programming patterns" than the equivalent programs in less expressive languages. This informal conjecture is a link between the ideas of elegance and expressiveness.

2.6 Conclusion

After reviewing the literature, it is concluded that Felleisens Conciseness Conjecture (Section 2.5.3) is a useful statement of the question which is investigated by the work herein. We discuss different metrics of information such as Software Science (Section 2.2.3) and Kolmogorov-Chaitin complexity (Sections 2.2.1 and 2.2.2). Due to reservations over the theories underlying Software Science, the characters/bytes metric of Shannon et al. will be adopted.

Felleisen has studied matters relating to the expressiveness of programming languages (Section 2.5.2), and has sketched a formal framework. A language is

at least as expressive as another if the former can express all the faculties of the latter, within the parameters of Felleisens expressivity framework.

Expressivity in Felleisens framework is tied to the notion of conservative extensions. Such extensions will contain more information in the semantic of the extended language than in the base language. This tentatively suggests that there is a connection between the expressivity of semantics, and their size. In an ideal case, we can imagine that this is true, there may exist a counterexample however.

The hypotheses in Section 1.3 make very general statements as to the relationship between the programs and semantics. In light of the literature here, it would be beneficial to revise these to take into account some notion of elegance and expressivity. This shall be done in Section 3.1.

Chapter 3

Preliminaries

This chapter revisits the hypotheses to refine them according to the literature surveyed and lays out the measures and methodologies for the primary investigation. It includes a discussion of the metrics we adopt, semantic representations of the TM, RASPs, SKI combinators, and the λ-calculus. Also presented are the formats of the semantics and programs which we measure in order to determine their levels of information.

3.1 Hypotheses Revisited

We revisit the hypotheses originally stated in Section 1.3 in the light of the context provided by the literature. Chaitin's formulation of elegance is concerned with finding the shortest program to produce output o. For every possible output o and language l, the elegant program definition covers only programs which when run with no input, output o.

Chaitin's elegance is of little use for the 'practical' programs which we wish to measure. Our programs compute some function given an input. The output is thus based on that input. However it is not unreasonable to expect that Chaitin's definition can be extended to include such practical programs.

3.1.1 Blums Axioms

Blums axioms [6] define measures of computational complexity. An abstract measure of the performance of a model of computation (e.g. number of steps,

memory used) is a complexity measure if it satisfies his axioms:

Definition 8 (Blums axioms for measures of performance)**.** *For any model of computation M, there exists a Gödel numbering ϕ which enumerates all machines of M such that for any $i \in \mathbb{N}$, $\phi_i(x)$ is a machine running with the input x.*

Let Φ denote an ordered subset of the machines of model M. Φ is a sequence of performance measure functions for ϕ if and only if:

- *$\phi_i(x)$ is defined $\leftrightarrow$ $\Phi_i(x)$ is defined*
- *There exists a function R such that:*

$$R(i, x, y) = \begin{cases} 1 & \text{if } \Phi_i(x) = y \\ 0 & \text{if not} \end{cases}$$

So ϕ is a sequence of all possible functions, while Φ is the sequence of halting functions. An input x has a unique Φ because the halting behaviour of some functions change depending on input.

Two canonical examples of *Blum complexity measures* are space and time. Using time as a measure, $\Phi_i(x)$ runs the (halting) function $\phi_i(x)$ and returns the number of steps that it took (for a sensible definition of "step"). The function $R(i, x, y)$ takes the number of the function to execute i, an input x, and a guess at step count y. It returns 1 if the guess was correct and 0 otherwise.

Blum goes on to define the *speed-up theorem* [5] which states: There exists a function f with the property that for every index i for f, there exists an index j for f such that:

$$\Phi_i(n) > \Phi_j(n)^{\Phi_j(n)}$$

Which is to say that in any ordering of partial recursive functions there exists a function where the *Blum complexity measure* (a measure of complexity that fulfils Blums axioms) for that function *can be improved to an exponential degree.*

It seems natural that we can extend the definition of Chaitin's elegance to include programs which calculate a specific function. For any function f and language l, a program p is elegant if p is written in l, there is no smaller program

written in l which performs the function of p, and:

$$\forall(x \mapsto y) \in f : p(x) \mapsto y$$

This is encouraging as it implies that for very simple functions, there may conceivably exist programs with a size below the undecidability threshold which we can be assured are elegant. However, if we were to include programs which take input, then the input size also has to be considered when determining if a program is elegant or not. A measurement of a program taking account the size of the programs input makes it a Blum complexity measure.

Chaitin's proof (Section 2.2.2) determines that elegance is undecidable for functions over a certain size. The proof below asserts the existence of functions where no elegant characterisation can be found for infinitely many inputs.

There is a subtle difference in the nature of the programs discussed in each proof. Chaitin's original proof concerns his formulation of elegant programs. These programs are very constrained in that they return a specific output when run.

The programs referred to in this new proof are more general in that their output is conditional on their input. While Chaitin's elegant programs are constant functions, these possibly elegant programs are not necessarily constant. Extending elegance to include these functions requires a new proof of the uncomputability of elegance for them.

Theorem 2 (Undecidability of Elegance)**.** *Deciding the elegance of program to compute a non-constant function f is uncomputable.*

Proof. This new proof proceeds by showing that Φ is a Blum complexity measure. Given the ordering ϕ where function $\phi_i(n)$ is a function to compute f with input n, $\Phi_i(n) = k$ is a function which determines the size of the program i and its input n. The function $\Phi_i(n)$ is defined if and only if $\phi_i(n)$ is defined as you cannot work out the information required to compute a non-halting function, which satisfies the first condition of the axiom.

The second condition is satisfied by the existence of R such that $R(i, x, y) = 1$ if $\Phi_i(x) = y$ and 0 if not. It returns 1 if $f(x)$ can be calculated in exactly y

characters.

Since information is a Blum complexity measure, the speed-up theorem applies. This means that in the ordering ϕ there exists a function f which, for any program ϕ_i, there exists a program ϕ_j where the information required to compute $f(n)$ follows:

$$\Phi_i(n) > \Phi_j(n)^{\Phi_j(n)}$$

for almost all n. This implies that there is no singular elegant program for computing the function f, concluding the proof. □

The problem here is down to input encoding. As a concrete example, say there exist two TMs which perform addition, where M_1 uses a unary encoding for its input, and M_2 uses binary. TM M_1 is exactly the unary addition machine in Section 4.2.1, and one can imagine that M_2 is slightly larger by (say) c characters:

$$size(M_1) + c = size(M_2)$$

Considering only the size of the program, as in the case of Chaitin's elegance, we could say that M_1 is more elegant than M_2. However, when size of inputs are considered, the information complexity of M_1 with an input i will be lower than the information complexity of M_2 with i

$$\begin{aligned} size(M_1) + i &< size(M_2) + log_2(i) \quad &: i < log_2(i) + c \\ size(M_1) + i &= size(M_2) + log_2(i) \quad &: i = log_2(i) + c \\ size(M_1) + i &> size(M_2) + log_2(i) \quad &: i > log_2(i) + c \end{aligned}$$

In the infinite limit, the growth rate of the input encoding is what asymptotically determines the elegance of a given function in some language. Unfortunately, it seems that the amount of information required to calculate a function f is a consequence of how elegantly one can encode the inputs of f. Section 6.6 gives another concrete example of this encoding phenomenon with the universal TMs.

3.1.2 The Semantic Information and Total Information Hypotheses

Explicitly invoking elegance as a necessary attribute of the programs which the hypotheses range over is folly. The undecidability results mean that there can be no formal assurance of the elegance of the programs measured.

A similar case is present with input sizes. For each model and program the realisation with the slowest input growth rate is the most elegant for infinitely many inputs. This reduces the problem of elegance to one of finding a method which produces the most elegant encoding of the inputs.

These problems pull focus away from the central question: How does the amount of information in the semantics affect the amount of information required to define a program? In the interest of fair comparisons, it is important to define notions of how small we can reasonably expect programs to be, and the effort expended on the encodings of program inputs.

Consider the breadth of possible encodings for some piece of data d. Depending on how large the alphabet for language l is, there is a sliding scale of the density of the possible encoding $e_l(d)$:

Definition 9 (Natural, Sparse, Dense Encodings)**.** *An input encoding $e(d)$ is* natural *if there is an approximately 1:1 ratio between the tokens of the unencoded input and tokens of the encoded input. Where $n > 1$, a sparser encoding has a 1:n ratio between the unencoded and encoded inputs (many encoded tokens to represent one unencoded token). A denser encoding has an n:1 ratio the unencoded and encoded inputs (one encoded token to many unencoded tokens).*

The exact nature of a *token* depends on the language of the input of the models. For instance, a token for the TM would be a single symbol. Tokens in the RASP are single numbers of k characters. A token for the SKI would be a single combinator, and tokens for the λ-calculus may be single terms such as individual numerals, or structural terms like PAIR, NIL, etc.

Natural encodings are approximately a 1:1 ratio of encoded to unencoded input because the alphabets in question may not permit an exact 1:1 relationship. There is a sliding scale of how natural the encodings are and those with ratios

closest to the 1:1 relationship are the "most natural".

Every model has an encoding method which can be deemed natural relative to its own input language, but it may not be considered natural relative to the language of another model. These encodings imply rates of input size growth and are examined more thoroughly in Section 6.5. The set of programs which are used to evaluate the hypotheses operate over natural input encodings.

Informally, programs are written to be as "elegant as possible" while admitting natural encodings of data as inputs. To differentiate these from elegant programs, we call them *succinct*.

The Semantic Information (SI) hypothesis states that a model with more semantic information will produce more elegant (now succinct) programs than a model with less semantic information. Considering the extreme cases of a 3^{rd} generation language (Java, Haskell) versus assembler, we can imagine that this holds. But a more nuanced example which does not conform can be constructed as follows.

Consider a conservative extension to the RASP; RASPX. RASPX has an extra instruction, LOOP. The LOOP instruction decrements the PC so that a RASPX machine encountering LOOP immediately enters an infinite loop. As a conservative extension, RASPX has a larger set of semantics, but no program can execute the LOOP function and terminate. This is a direct counterexample to our hypothesis, so we need to make it more specific.

A program p *utilises* some semantic information i if p invokes some operator defined in the semantics which depends directly or indirectly on i:

Hypothesis 1 (Semantic Information)**.** *For two Turing Complete models; if model A has more semantic information (larger semantics) than model B, the average size of succinct programs (where at least one program utilises the extra semantic information) written for model A will be lower than the average for model B.*

We should consider the 'scope' of this hypothesis. The selection of models in this investigation captures the following:

- Extensions to a model

- Comparisons across models in the same paradigm
- Comparisons across paradigms

Hypothesis 1 is very strong because it makes a general statement concerning information over the entire space (all three scopes) of models and programs. While the models of computation presented in Chapter 2 are all different, some of them share features with each other beyond their Turing completeness. This allows us to split this strong hypothesis into sub-hypotheses such that the strong hypothesis is satisfied iff the three sub-hypotheses all hold.

The RASPs all share a significant portion of their semantics. The semantic rules which guide their evaluation in the form of the fetch-execute cycle are identical, with portions of the instruction set distinguishing the models from each other. These models are said to be in the same *family*. While each model has unique instructions which effect different changes on the state and contents of the memory, the rules which govern the structure remain constant (e.g. the fetch-execute cycle, bounded size and contents, arbitrarily rewritable and executable memory locations).

Models which share some aspects with each other, but not as far as directly sharing evaluation methods, can be classified in the same paradigm. In this thesis there is the imperative paradigm, occupied by the RASPs and TM, and the functional paradigm which contains SKI and λ-calculus. The RASP and TM have a global state and their underlying structure is a linear array of numbers/symbols. The λ-calculus and SKI both use graph reduction for evaluation (Section 3.4.3) and have no state. Figure 3.1 shows the models grouped into families and paradigms.

We propose three weaker hypotheses which range over the scopes of family, paradigm and across paradigms. This approach will allow us to apply the SI hypothesis and discover where the hypothesis holds, even if the strong hypothesis does not hold in general. "A programs" are defined as succinct programs written for model A.

Hypothesis 1a (Semantic Information within family)**.** *For two Turing Complete models A and B in the same family. If A has more semantic information than B, the average size of A programs will be lower than the average for B programs.*

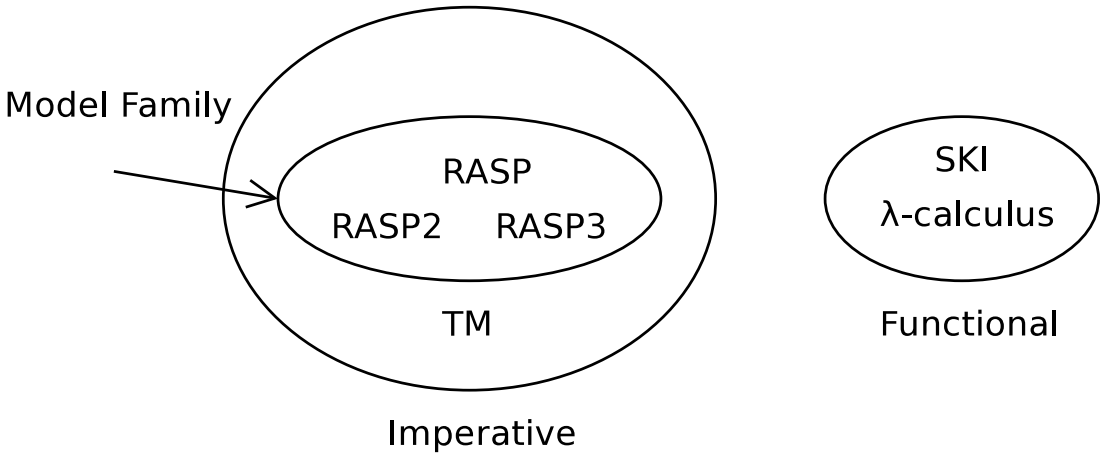


Figure 3.1: Paradigmal relationships

Hypothesis 1b (Semantic Information within paradigm)**.** *For two Turing Complete models A and B in the same paradigm. If A has more semantic information than B, the average size of A programs will be lower than the average for B programs.*

Hypothesis 1c (Semantic Information across paradigms)**.** *For two Turing Complete models A and B in different paradigms. If A has more semantic information than B, the average size of A programs will be lower than the average for B programs.*

The sizes of the semantics are stated in Section 3.5. Knowledge of these sizes and of the above sub-hypotheses, we can predict what would happen if the hypotheses are correct:

Prediction 1.1 (Program Sizes: RASP)**.** *The semantic sizes of the three RASP models (measured in characters, Section 3.3.1) follow the relation* $RASP < RASP2 < RASP3$*. It is predicted that the average succinct program sizes follow the relation* $RASP3 < RASP2 < RASP$*.*

Prediction 1.2 (RASP vs TM)**.** *The RASP semantics are larger than those of the TM. It is predicted that succinct RASP programs are smaller than succinct TM programs on average.*

Prediction 1.3 (λ-calculus vs SKI)**.** *The λ-calculus semantics are larger than the SKI semantics. It is predicted that succinct λ-calculus programs are smaller than succinct SKI programs on average.*

Prediction 1.4 (Across Paradigms)**.** *If model A of paradigm X has larger semantics than model B of paradigm Y, it is predicted that succinct program in model A are smaller than succinct program in model B on average.*

Prediction 1.1 relates to Hypothesis 1a. Predictions 1.2 and 1.3 support Hypothesis 1b, and prediction 1.4 supports 1c.

The Total Information (TI) hypothesis conjectures that as complexity of programs that we measure increases, the average TI of more complex models will eventually decrease to below that of simpler models. We again reformulate the hypothesis to include the necessary stipulation of succinct programs.

The statement of "Complex models" recalls Section 2.6 where it is tentatively established that there is a connection between the expressivity of a model and the size of its semantics. If this connection is well founded, we will observe that the more expressive models which produce smaller programs will have larger semantics.

The complexity of a function can be defined in many ways. Intuitively division is a more complex function than addition and a universal machine is more complex than division. Actually classifying these functions hierarchically is a surprisingly thorny proposition. One approach is time and space complexity where the complexity function is determined by the number of steps or tape cells required for computation relative to the size of the input.

This characterisation feels unsatisfactory (especially in the context of Blum's speed-up theorem). One alternative is to rely on the arithmetical hierarchy [46, 76], which classifies functions on their halting and output behaviour. While the arithmetical hierarchy separates addition and division from universal machines, there is too little nuance to differentiate between the addition and division functions.

Another alternative is to provide a definition in terms of elegant programs. A function a is more complex than function b in some language l if the elegant program to calculate a is smaller than the elegant program to calculate b. This makes sense because we believe that function deemed "more complex" would have a higher minimum requirement of information. This intuition is not objective though, as some models may be inherently suited towards some calculations rather

than others. Any elegant comparison of the complexity of a function is made relative to the language l.

The notion of the "complexity" of a function is based on intuition, computability, and computational complexity. There is no definitive ranking of functions according to their complexity, so we have to rely on this notion to guide us. When this thesis discusses the complexity of a function, it refers to the size of the succinct program to represent the function.

Hypothesis 2 (Total Information)**.** *For two Turing Complete models X and Y, where X has more semantic information than Y; As the size and complexity of a program increases, the average total information (TI) of a succinct implementation in X will decrease relative to the total information of a succinct implementation in Y.*

To illustrate this hypothesis, consider the RASP family. For simple functions (say arithmetic), we predict that the TI for the RASP machine be lower than the TI of the RASP2 or RASP3. This is because the reduction in program size for the RASP2/3 does not outweigh the extra information in the semantics of the RASP2 and RASP3. However as the tested functions increase in complexity (say the universal machines), we expect to see the TI averages for the RASP2 and RASP3 drop relative to the TI averages for the RASP. With a sufficiently large and diverse set of functions containing programs which utilise the extra semantic information of the RASP2 and RASP3, we should see the TI follow the relation RASP3<RASP2<RASP.

This reformulation of the total information hypothesis is also strong, not unlike the semantic information hypothesis above. We can again split this into three sub-hypotheses with predictions for each analogous to the structure of the SI hypothesis above:

Hypothesis 2a (Total Information within family)**.** *For two Turing Complete models A and B, where A and B are in the same family and A has larger semantics; as a program grows in size and complexity, the average TI to realise the program succinctly in A will decrease relative to the average TI to realise the program succinctly in B.*

Hypothesis 2b (Total Information within paradigm). *For two Turing Complete models A and B, where A is in the same paradigm as B and has larger semantics; as a program grows in size and complexity, the average TI to realise the program succinctly in A will decrease relative to the average TI to realise the program succinctly in B.*

Hypothesis 2c (Total Information across paradigms). *For two Turing Complete models A and B, where A is in a different paradigm from B and has larger semantics; as a program grows in size and complexity, the average TI to realise the program succinctly in A will reduce relative to the average TI to realise the program succinctly in B.*

Again using the information from Section 3.5, we make a variety of predictions of what will happen if the sub-hypotheses above hold:

Prediction 2.1 (Total Information: RASPs). *As the size and complexity of a set of programs increases, it is predicted that the average TI of succinct implementations of the programs in the RASP3 will reduce relative to the TI of the RASP2 which in turn will reduce relative to the TI of the RASP.*

Prediction 2.2 (Total Information: RASP vs TM). *As the size and complexity of a set of programs increases, it is predicted that the average TI of succinct implementations of the programs in the RASP will reduce relative to the TI of succinct implementations in the TM.*

Prediction 2.3 (Total Information: λ-calculus vs SKI). *As the size and complexity of a set of programs increases, it is predicted that the average TI of succinct implementations of the programs in the λ-calculus will reduce relative to the average TI of succinct implementations in the SKI calculus.*

Prediction 2.4 (Total Information: Across paradigms). *If model A of paradigm X has larger semantics than model B of paradigm Y; as the size and complexity of a set of programs increases, it is predicted that the average TI of succinct implementations of the programs in model A will reduce relative to than the average TI of succinct implementations in model B.*

3.1.3 The Semantic Circuit and Total Circuit Hypotheses

In this thesis, we also translate the semantics for the RASP and Turing machines into the VHSIC Hardware Description Language (VHDL). This is then compiled down to a series of electronic components of a Field Programmable Gate Array (FPGA), and the number of components required to implement the various machines are counted. A circuit A is said to be *larger* than circuit B if the combined total of Look-up tables, slice registers, and flip-flops (Chapter 5) in A is higher than the total for B. We hypothesise that the more semantic information in a model, the larger the circuit to execute the semantics:

Hypothesis 3 (Semantic Circuit sizes)**.** *Consider two models A and B. If model A has larger semantics than model B, the FPGA circuit which implements the semantics of A will be larger than the FPGA circuit for B.*

Hypothesis 3a (Semantic Circuit sizes within family)**.** *For two models A and B in the same family. If A has larger semantics than B, then the circuit which implements the semantics of A will be larger than the circuit to realise B.*

Hypothesis 3b (Semantic Circuit sizes within paradigm)**.** *For two models A and B in the same paradigm. If A has larger semantics than B, then the circuit which implements the semantics of A will be larger than the circuit to realise B.*

Prediction 3.1 (RASP semantics order)**.** *The three RASP models have semantic sizes measured according to the relation* $RASP < RASP2 < RASP3$ *(Section 3.5). It is predicted that the circuit sizes follow this relation.*

Prediction 3.2 (RASP vs TM)**.** *The RASP has larger semantics than the TM, therefore the circuit for the TM semantics is predicted to be smaller than the circuit for the RASP semantics.*

Predictions 3.1 and 3.2 support sub-hypotheses 3a and 3b respectively. Similar to the TI hypothesis, we have a Total Circuit (TC) size hypothesis which attempts to predict sizes of the total implementation (components for program + components for semantics) of the RASP and TM. The programs which are mapped to FPGA circuits will be the same programs as those which are used to evaluate Hypotheses 1 and 2 above.

Hypothesis 4 (Total Circuit sizes). *For two models A and B, where the circuit implementation of the semantics of A is larger than the circuit for the semantics of B; as a function grows in complexity, the average total implementation size of a succinct realisation of the function in model A will reduce relative to the average for model B.*

Hypothesis 4a (Total Circuit sizes within family). *For two models A and B in the same family; if the semantics of A are larger than the semantics of B, then as a program grows in size and complexity, the average total implementation size of the program in model A will reduce relative to the average for model B.*

Hypothesis 4b (Total Circuit sizes within paradigm). *For two models A and B in the same paradigm; if the semantics of A are larger than the semantics of B, then as a program grows in size and complexity, the average total implementation size of the program in model A will reduce relative to the average in model B.*

Prediction 4.1 (RASP total circuit size). *As the size and complexity of a program increases, it is predicted that that the average total implementation size for the RASP3 will reduce relative to the total implementation size for the RASP2 which, in turn, will also reduce relative to that of the RASP.*

Prediction 4.2 (RASP vs TM). *As the size and complexity of a program increases, it is predicted that the average total implementation size of the RASP will reduce relative to the average total implementation size of the TM.*

3.1.4 Hypotheses Summary

The hypotheses and corresponding predictions are be summarised below:

1. Strong SI hypothesis
   - 1a. SI within family hypothesis
     - 1.1. Program Sizes (RASP) prediction
   - 1b. SI within paradigm hypothesis
     - 1.2. SI RASP vs TM prediction
     - 1.3. λ-calculus vs SKI prediction

1c. SI across paradigms hypothesis

1.4. Across paradigms prediction

2. Strong TI hypothesis

2a. TI within family hypothesis

2.1. TI for RASPs

2b. TI within paradigm hypothesis

2.2. TI RASP vs TM

2.3. TI λ-calculus vs SKI

2c. TI across paradigms hypothesis

2.4. TI across paradigms prediction

3. Strong SC hypothesis

3a. SC within family hypothesis

3.1. SC for RASPs

3b. SC within paradigm hypothesis

3.2. SC RASP vs TM

4. Strong TC hypothesis

4a. TC within family hypothesis

4.1. TC for RASPs

4b. TC within paradigm hypothesis

4.2. TC RASP vs TM

3.2 Comparison Metrics

There are two prime candidates for information comparison metrics; bytes and characters. Both have their advantages and disadvantages.

The characters which most programming languages use to express commands (the *basic execution* character set) are represented as 7 bit ASCII [44, 43]. Since the basic execution character set is all that is needed to write programs, the handling of characters outwith the set are typically a function of the compiler and assorted programming tools.

Our models also draw exclusively from 7 bit ASCII, save the λ-calculus which requires 'λ's. The semantics additionally use logical predicates $\forall$, $\exists$ as well as the connectives; $\wedge$, $\vee$ and $\implies$.

The predicates and connectives represent more complex ideas than a numeral or single letter so it seems appropriate to assign more bytes (under the UTF-8 scheme [92] it is two bytes each) such characters. In this way we acknowledge that $\forall$ contains more information than a numeral.

Character sets are defined not on the information required to represent an idea, but rather the frequency with which a character is used in computer applications. The addition and subtraction operators are also more complex ideas than a single numeral, but are represented in ASCII as one byte. Do we add a byte to all occurrences of $+$ and $-$ to make our comparison fair?

If we do this we start creating our own character set. So the only way our measurements would be demonstrable is if we measured them on a computer implementing our character set. Even if we did accept that we should use a single byte for add and subtract, and 2 bytes for other functions, the measurements we make are still wholly dependent on the standards implemented by the machine on which we measure. Our measurements could conceivably change from one machine to the next.

The use of characters as a metric is established by Solomonoff [87, 88], Kolmogorov [47] and Chaitin [9]. Character metrics are independent of the reference machine and are solely dependent on the input format of the model which is specified by the semantics. This is more suited to our needs so it will be the adopted metric for the rest of this investigation.

3.3 Formats

Irrespective of the metric choice, the aim is to write programs and semantics in a way to economise on the amount of information which is supplied. The first and foremost method to minimise this information is in the choice of algorithm used to compute the functions, favouring brevity over any time or (utilised) space concerns. But how the programs and semantics are themselves presented should

also be considered.

3.3.1 Semantics

As the formalism from which everything is measured, a SOS can be encoded in whichever way is convenient and it is assumed that an 'SOS machine' can interpret this encoding and translate it to the correct corresponding SOS rules for execution. To do this, common functionality is split out and in-lined into the appropriate rules. Reverse Polish Notation is also employed to shorten the expressions by removing the brackets which denote function application.

Łukasiewics [56] developed Polish ('prefix') notation for sentential logic and we adopt the reversed notation here to remove the brackets on function calls. Reverse Polish notation (RPN or 'postfix' notation) is a mathematical representation which typographically arranges functions after their parameters [35]. As an example, the expression $(3 - 4) \times 5$ (remembering the order of operations) is $3\,4\ -5\times$.

This expression is executed using a stack. First, the values three and then four are pushed onto the stack. When the subtraction operator is read, the top two elements of the stack are popped (since subtraction is a binary operator) the operation is applied and the result is pushed back on top of the stack. The intermediate expression is $-1\ 5\times$, and with the -1 already on the stack, the 5 is pushed, then both are popped to be multiplied together and the result (-5) is pushed back on top of the stack.

The advantage of Polish notation is that it obviates the need for bracketed expressions. Specific examples of its usage are given in Section 3.4.

3.3.2 Turing Machines

A Turing machine is a collection of quintuples $\langle st_{old}, sy_{old}, st_{new}, sy_{new}, dir\rangle$ which denote: the current state, the current symbol on the tape, the new state, the new symbol, and the direction in which to move the head. Figure 3.2 shows the Turing machine for addition. The symbol table for this TM consists of 5 lines of 9 characters each (45). The tape (101) is two unary numbers separated by a single symbol '0', which we define in the symbol table as a blank.

```
1,1,1,1,R
1,0,2,1,R
2,1,2,1,R
2,0,3,0,L
3,1,0,0,R
101
```

Figure 3.2: The 'raw' Turing machine for addition with an input of 1+1

Our convention is that a TM will start over the leftmost symbol on the tape unless there is a caret (^) in which case the head will be over the symbol to the right of it. For example, the tape 1^011 will start the machine with the head over the '0'.

3.3.3 RASP machines

An n-bit RASP machine is a $2^n - 3$ size array of naturals. This is represented and counted as a comma separated list of numbers. For instance the program LOAD 1;LOAD 2;HALT would be represented as the sequence $3, 1, 3, 2, 0$.

A caveat for the RASP machine is that the displayed array is *exactly* $2^n - 3$ in length. For all programs that are less than $2^n - 3$ instructions long, the extra room is 'padded out' with HALT instructions.

3.3.4 λ-calculus

A term in the λ calculus is structured as follows; λs are not grouped, so an expression with multiple λs would be of the form $\lambda x.\lambda y.e$. The expression is parsed in a left associative manner, so brackets are used for disambiguation. An expression $(((\lambda x.x)y)z)$ is written $(\lambda x.x)yz$ without any loss of meaning.

We measure λ terms by their expressions as above. For instance, the number of characters in the term ONE $(\lambda f.\lambda x.f\ x)$ is 9, including the space to separate the f and x variables.

We can compress complex λ functions by pushing repeated terms into abstractions. To illustrate we begin with a term ready to be applied, say to linearly search a list (Section 4.3.2):

$$\text{SEARCH} \equiv Y(\lambda a.\lambda b.\lambda c.\text{NULL } c \text{ ONE } (\text{EQ}(\text{HEAD } c)b)\text{FALSE}(\text{SUCC}(a\ b(\text{TAIL } c))))$$

HEAD and TAIL are the expressions ($\lambda p.p$ TRUE) and ($\lambda p.p$ FALSE) respectively, so they are substituted into the main term:

$$Y(\lambda a.\lambda b.\lambda c.\text{NULL } c \text{ ONE } (\text{EQ}((\lambda p.p \text{ TRUE})c)b)\text{FALSE}$$
$$(\text{SUCC}(a\ b((\lambda p.p \text{ FALSE})c))))$$

EQ tests for the equality of two numbers, returning TRUE if equal and FALSE otherwise, and this can again be substituted into the main term. NULL is also replaced with its corresponding expression:

$$Y(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.\text{FALSE}))c \text{ ONE } (((\lambda m.\lambda n.n \text{ PRED } m(\lambda x.\text{FALSE})\ldots$$
$$\text{TRUE}(m \text{ PRED } n(\lambda x.\text{FALSE})\text{TRUE})(n \text{ PRED } m(\lambda x.\text{FALSE})\ldots$$
$$\text{TRUE})))((\lambda p.p \text{ TRUE})c)b)\text{FALSE}(\text{SUCC}(a\ b((\lambda p.p \text{ FALSE})c))))$$

With these names fully substituted with their corresponding terms, there are three occurrences of PRED, six occurrences of FALSE, and four occurrences of TRUE. Since abstraction in the λ calculus enables argument duplication and placement wherever it is desired in the body of an expression, repeated occurrences can be abstracted out. First, PRED is abstracted by binding a new variable k and applying that binding to PRED:

$$(\lambda k.Y(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.\text{FALSE}))c \text{ ONE } (((\lambda m.\lambda n.n\ k\ m(\lambda x.\text{FALSE})\ldots$$
$$\text{TRUE}(m\ k\ n(\lambda x.\text{FALSE})\text{TRUE})(n\ k\ m(\lambda x.\text{FALSE})\ldots$$
$$\text{TRUE})))((\lambda p.p \text{ TRUE})c)b)\text{FALSE}(\text{SUCC}(a\ b((\lambda p.p \text{ FALSE})c)))))\text{PRED}$$

Then the same is done for TRUE (t) and FALSE (g):

$$(\lambda g.\lambda t.\lambda k.Y(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.g))c \text{ ONE}$$
$$(((\lambda m.\lambda n.n\ k\ m(\lambda x.g)t(m\ k\ n(\lambda x.g)t)(nkm(\lambda x.g)t)))((\lambda p.p\ t)c)b)$$
$$g(\text{SUCC}(a\ b((\lambda p.p\ g)c)))))\text{FALSE TRUE PRED}$$

Abstracting out some term from an expression entails adding three characters to the start of the expression and one character per occurrence in the body. In exchange, we can remove all but one of the occurrences of the term which is moved to the end of the expression.

This method of reducing the size of expressions requires that we make some pre-reductions when applying this expression in order to obtain. This involves more computation overhead in the classic time/space trade-off, but we do not care about run times. The measured λ programs have all had this compression method applied to them where possible.

3.3.5 SKI combinators

A term in the SKI combinator calculus is expressed as a string of S,K,I characters as well as the left and right parentheses. Unlike the λ-calculus, SKI terms do not require spaces. For example, the term for two is $S(S(KS)K)I$ which is 10 characters long.

Much like how the λ calculus has α and η conversion to transform superficially different terms into a common simple term, we can structurally decompose SKI calculus expressions into equivalent and shorter terms.

For the Church numerals, we can alternatively represent any non prime number as the product of f factors. This trick multiplicatively combines the factorisation into a 'full' numeral when something is applied to it. The generalised form is thus:

$$\begin{aligned} 4 &= S(K\ \text{TWO})\text{TWO} \\ 8 &= S(K(S(K\ \text{TWO})\ \text{TWO}))\ \text{TWO} \\ 16 &= S(K(S(K(S(K\ \text{TWO})\ \text{TWO}))\ \text{TWO}))\ \text{TWO} \\ n &= S(K^{f-1})\text{factor}^f \end{aligned}$$

Comparing the factorised form of 4 to the (SUC^n ZERO) form saves 4 characters:

SUCC(SUCC(SUCC(SUCC ZERO)))	$S(K\ \text{TWO})\text{TWO}$
$S(S(KS)K)(S(S(KS)K)(S(S(KS)K)I))$	$S(K(S(S(KS)K)I))(S(S(KS)K)I)$

The application of functions to the factorised numeral reduces (with more steps) to the correct and expected form, for example:

$$
\begin{aligned}
S(K\ \text{TWO})\text{TWO}fx &\equiv \ldots \\
&\Rightarrow_S K(S(S(KS)K)I)f(S(S(KS)K)If)x \\
&\Rightarrow_K S(S(KS)K)I(S(S(KS)K)If)x \\
&\Rightarrow_S \ldots \\
&\Rightarrow Kf(I(S(S(KS)K)If)x)(If(I(S(S(KS)K)If)x)) \\
&\Rightarrow_K f(If(I(S(S(KS)K)If)x)) \\
&\Rightarrow_I \ldots \\
&\Rightarrow f(f(Kfx(Ifx))) \\
&\Rightarrow_K f(f(f(Ifx))) \\
&\Rightarrow_I f(f(f(fx)))
\end{aligned}
$$

When representing a number as a product of its factors, we wish to use more factors of smaller numbers rather than less factors of larger numbers. The reason for this is that to add another factor the overhead is: $S(K\ldots)$ of 4 characters whereas the distance between $n > 1$ and SUCC n is 11 characters. If we cannot directly factor a number, such as with a prime, then we factor a non-prime neighbour and apply SUCC to it.

Unlike the λ-calculus, abstraction in SKI is information intensive as each level of nesting in a SKI expression requires combinators to 'push' a passed expression down to where it should be. The strategy of maximal abstraction outlined above for the λ calculus is detrimental to the size of the resulting SKI expression. We therefore convert λ expressions to SKI via bracket abstraction without performing the extra abstraction detailed in Section 3.3.4, preferring instead to normalise as much of the expression as possible before conversion.

3.4 Semantics

Our models of computation transform their inputs into outputs by following the rules of their semantics. If a program is a description of what is to be done, the semantics are *how* it is done. The semantics of a model combine the aspects of a model "understanding" the input program (parsing) and performing the functions

of the model (evaluation).

The semantics for each model manipulate discrete structures for each term. This is the *internal representation* of the input. The program formats above are presented in an *external representation* which may not necessarily directly reflect the internal representation.

The external representations of the SKI and λ-calculus do not directly translate into the internal representation, so we require semantics which perform lexical parsing via pattern matching. To provide an even-handed analysis, we also define parsers for the RASP and TM which have very similar internal and external representations. The full semantics for each model in the RPN notation are presented in Appendix D.

3.4.1 Turing Machines

There are multiple ways to formally define Turing machines:

$$\langle Q, \delta, \Sigma, \Gamma, q_0, q_a, q_r \rangle \tag{3.1}$$

$$\langle Q, \delta, \Gamma, \gamma, q_0, q_h \rangle \tag{3.2}$$

$$\langle Q, \delta, \Sigma, \Gamma, q_0 \rangle \tag{3.3}$$

where Q is the set of states, Σ which is the input alphabet, Γ is the tape alphabet (which symbols can be read from or written to the tape), δ is the transition function $Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$, $q_0 \in Q$ is the initial state, $q_a \in Q$ and $q_r \in Q$ are accepting and rejecting states respectively, q_h is the halt state, and γ is the blank symbol.

Definitions 3.1, 3.2, and 3.3 are from [86, 78, 32] respectively. Further checks of sources [74, 3, 39, 50, 15, 49] show that the TM is broadly defined as the above with minor variances. Each definition varies in the details, but all are equivalent in power.

We can combine parts of these definitions with our conventions to produce a definition for the TM which is different from those above, but is still Turing complete. Our conventions are 1.) Each TM starts in state 1, and 2.) A TM halts if it transists to state 0 *OR* there is not a transition in δ for the current

$$
\begin{gathered}
st : Q \\
sy : \Gamma \\
h : \mathbb{Z} \\
d : \{L, R\} \\
T : \mathbb{Z} \mapsto \Gamma \\
\delta : Q \times \Gamma \mapsto Q \times \Gamma \times d \\
P_\delta : (\Gamma \cup Q \cup d \cup \{,\})^+ \mapsto \delta \\
P_T, P_{NT} : (\Gamma^+ \cup \{\hat{}\}) \times \mathbb{Z} \mapsto T
\end{gathered}
$$

Figure 3.3: Type definitions for the variables and functions of the TM

state/symbol pair.

To define our own machines, we need a set of states and a set of transition functions: Q and δ. We also need a tape alphabet Γ, but we would like to permit the use of the blank symbol on the input tape so we exclude Σ, opting instead to explicitly state the blank symbol itself as γ. Our starting state is always going to be 1, so individual machine definitions do not need to specify it. Similarly, we can define the halt state as a state with no exiting transitions. We wind up with a definition of a TM conforming to our convention as:

$$\langle Q, \delta, \Gamma, \gamma \rangle$$

We now proceed to translate this definition into Structured Operational Semantics.

Every TM has a tape T, the symbol table δ, the current state st and a head position h. T is a unary function which takes an integer and returns the symbol at that position on the tape. The symbol $T(0)$ is defined as either the leftmost symbol of the input, or immediately to the right of the caret ($\hat{}$) in a TM definition. Our initial tape function is T_0.

The symbol table $\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{L, R\}$ is a function which takes a state and symbol pair and returns a triple of state, symbol and shift direction. The type definitions for the TM are in Figure 3.3.

Before we execute the TM, we first have to populate δ and T_0. The 'raw' TM is an expression $e \in (\Gamma \cup Q \cup d \cup \{,\})^+$ where $+$ is "One or more" analogous to $*$ which is the Kleene Closure [39]. The symbol table parsing rules supplied by the

$$\frac{e \implies st, sy, st', sy', d\ e'}{P_\delta(e) \implies \{\langle st, sy\rangle \mapsto \langle st', sy', d\rangle\} \cup P_\delta(e')}$$

(a) Parsing a rule into δ

$$\frac{}{P_\delta(e) \implies \{\}}$$

(b) Default rule

Figure 3.4: Parsing a raw symbol table e into the internal representation δ

$$\frac{\begin{array}{c} f \implies f_1 \hat{}\, g f_2 \\ g \in \Gamma \end{array}}{P_T(f, 0) = P_{NT}(f_1, -1) \cup \{0 \mapsto g\} \cup P_T(f_2, 1)}$$

(a) Finding $\hat{}$, if it exists

$$\frac{\begin{array}{c} f \implies g f_1 \\ g \in \Gamma \end{array}}{P_T(f, n) = \{n \mapsto g\} \cup P_T(f_1, n+1)}$$

(b) Parsing symbols after the $\hat{}$

$$\frac{}{P_T(f, n) = \{\}}$$

(c) No symbol to parse after

$$\frac{\begin{array}{c} f \implies f_1 g \\ g \in \Gamma \end{array}}{P_{NT}(f, n) = \{n \mapsto g\} \cup P_{NT}(f_1, n-1)}$$

(d) Parsing symbols before the $\hat{}$

$$\frac{}{P_{NT}(f, n) = \{\}}$$

(e) No symbol to parse before

Figure 3.5: Parsing a raw tape into the internal representation T

function P_δ are shown in Figure 3.4.

Similarly the 'raw' tape is an expression $f \in \Gamma^+ \cup \{\hat{}\}$. The function P_T parses f into the initial tape T_0 and is shown in Figure 3.5. The functions δ and T are constructed recursively by the union of each mapping of input to output. The initial state of a TM ready to be executed is therefore:

$$st_0 = 1$$
$$h_0 = 0$$
$$T_0 = P_T(f)$$
$$\delta = P_\delta(e)$$

The current state, head position and tape all change during the evaluation of the machine while none of the TM execution rules change δ. The function $E : Q \times (\mathbb{Z} \mapsto \Gamma) \times \mathbb{Z} \mapsto (\mathbb{Z} \mapsto \Gamma)$ executes a TM:

$$\frac{\begin{array}{c}T(h) = sy \\ \delta(st, sy) = \langle st', sy', d\rangle \\ d = L \\ T'(h) = sy' \\ h' = h - 1\end{array}}{E(st, T, h) \implies E(st', T', h')}$$

(a) Left shift

$$\frac{\begin{array}{c}T(h) = sy \\ \delta(st, sy) = \langle st', sy', d\rangle \\ d = R \\ T'(h) = sy' \\ h' = h + 1\end{array}}{E(st, T, h) \implies E(st', T', h')}$$

(b) Right shift

$$\frac{\begin{array}{c}T(h) = sy \\ \delta(st, sy) \neq \langle st', sy', d\rangle\end{array}}{E(st, T, h) \implies T}$$

(c) Halting

Figure 3.6: The rules for executing the TM; left shift, right shift, and halt

$$T_{end} = E(st_0, T_0, h_0)$$

The Turing machine consists of three rules; a rule for shifting left, one for shifting right, and one for no defined state and symbol pair. Figure 3.6 shows the rules for running a TM. The machine halts when there is not a defined state and symbol pair in δ. As described earlier, this is a transition to state 0, but this convention is not enforced by the semantics, any state without a transition for the current symbol will do.

To minimise the size of these semantic rules, we can in-line the T functions into the δ function. Doing this eliminates the need for the sy variable which saves us more characters. We can also in-line the $d = R/L$ lines too, but have to keep the d variable for the third rule unless we do R/L variations for that too. The shift right rule is now:

$$\frac{\delta(st, T(h)) = \langle st', T'(h), R\rangle}{E(st, T, h) \implies E(st', T', h + 1)}$$

More methods to reduce the size are to remove the 'primed' variables and redefine st to just s. If we define $i = h + 1$, $t : Q$ and $U : \mathbb{Z} \mapsto \Gamma$ we can reduce all identifiers to single characters:

$$\frac{\delta(s, T(h)) = \langle t, U(h), R\rangle}{E(s, T, i) \implies E(t, U, h + 1)}$$

Using RPN, we can remove the brackets for function calls transforming the line $\delta(s, T(h)) = \langle t, U(h), R\rangle$ into the less readable $shT\delta\langle thUR\rangle =$ which saves us 6 characters The TM semantics transformed in this way total 335 characters in size.

3.4.2 RASP Machines

The definition of a RASP as presented by Elgot and Robinson (Section 2.3.1.2) provides a framework for the abstract operation of the machine, but is very general. There are a few examples of instructions that could be defined (such as the machines of Cook [16] and Hartmanis [36]), but the details of a machine are generally left up to the designer.

Due to the extensible nature of the RASP family presented herein, the semantics have been split into *model semantics* for the semantics of parsing an the F-E cycle, and *language semantics* which describe the operation of the instructions. This distinction is made because the RASP2 and RASP3 (Sections 3.4.2.1 and 3.4.2.2) iterations on the RASP where the instructions which are executed have changed, but the underlying fetch-execute cycle remains constant.

A RASP machine is a pair $\langle S, X\rangle$ of a machine $S \in K_o$ and an output vector X. The registers of S are numbered from 0, and registers 0, 1, and 2 are the PC, IR and ACC respectively. The vector X is written to by the OUT command and is initially empty. For an n-bit machine, there is a set $G = \{0 \ldots 2^n - 1\}$ of the possible integers representable by the machine. There is also a set $I \subset G$ which represents the non-halting instructions of the machine.

The RASP machines for the primary investigation in this thesis will have a fixed instruction set mapping of $\{0 \mapsto HALT, 1 \mapsto INC, 2 \mapsto DEC, 3 \mapsto LOAD, 4 \mapsto STO, 5 \mapsto OUT, 6 \mapsto JGZ, 7 \mapsto CPY\}$. The mapping is enforced by the semantics, but changes to the mappings affect the total number of steps a machine can make before halting. Appendix A investigates how the properties of RASPs change when the instruction set mapping changes.

The type definitions for the RASP are shown in Figure 3.7. To aid the understanding of the semantics, we also define mappings for the addresses PC, IR, and ACC to the natural numbers and do the same for the instructions.

$$
\begin{array}{ll}
S : \mathbb{N} \mapsto \mathbb{N} & INC = 1 \\
X : \mathbb{N} & DEC = 2 \\
G : \{0 \ldots 2^n - 1\} & LOAD = 3 \\
I \subseteq G & STO = 4 \\
\# : S \mapsto \mathbb{N} & JGZ = 5 \\
A : S \times X \mapsto (S \times X) & OUT = 6 \\
P : (G \cup \{,\})^+ \times \mathbb{N} \mapsto S & CPY = 7 \\
E : S \times X \mapsto S \times X & HALT = 0 \\
PC\_INC(S) = mod(S(PC) + 1, \#S) & PC = 0 \\
S_0 = \{0 \mapsto 3, 1 \mapsto 0, 2 \mapsto 0\} & IR = 1 \\
X_0 = \{\} & ACC = 2
\end{array}
$$

Figure 3.7: Definitions required for the RASP.

$$
\frac{\begin{array}{c} e \implies g, e_1 \\ g \in G \end{array}}{P(e, n) \implies \{n \mapsto e\} \cup P(e_1, n + 1)}
$$

(a) Parsing a natural number out of e

$$
\frac{}{P(e, n) \implies \{\}}
$$

(b) Default rule.

Figure 3.8: Parsing the external representation e

The initial machine and output vector are S_0 and X_0. S_0 is primed with the initial values of the PC IR and ACC (3,0,0), and the external representation of the RASP to be executed is $e \in (G \cup \{,\})^+$ which is a $2^n - 3$ sequence of integers. The function P parses the machine into our internal representation (Figure 3.8). This readies the RASP for evaluation by the function E:

$$\langle S_{final}, X_{final} \rangle = E(S_0 \cup P(e, 3), X_0)$$

Figure 3.9 shows the two rules of the RASP model semantics. If the instruction under the program counter is in I, then that corresponding instruction is applied to the machine S. If it isn't, the number is copied to the IR and the machine stops. If a numeral is indeed a RASP operation, the function A applies what is in the IR of S' to S' and X.

The language semantics for the RASP are 10 rules for the 7 non halting

$$\frac{\begin{array}{c} S(S(PC)) \in I \\ S'(PC) = PC\_INC(S) \\ S'(IR) = S(S(PC)) \\ \langle S'', X' \rangle = A(S', X) \end{array}}{E(S, X) \implies E(S'', X')} \qquad \frac{\begin{array}{c} S(S(PC)) \notin I \\ S'(IR) = S(S(PC)) \end{array}}{E(S, X) \implies \langle S', X \rangle}$$

Figure 3.9: The rules for the F-E cycle of the RASP

$$\frac{\begin{array}{c} S(IR) = INC \\ S'(ACC) = mod(S(ACC) + 1, \#S) \end{array}}{A(S, X) \implies \langle S', X \rangle} \qquad \frac{\begin{array}{c} S(IR) = DEC \\ S'(ACC) = mod(S(ACC) - 1, \#S) \end{array}}{A(S, X) \implies \langle S', X \rangle}$$

(a) The INC instruction

(b) The DEC instruction

Figure 3.10: The semantics for INC and DEC

instructions. Figure 3.10 shows the semantics for the INC and DEC instructions. Figure 3.11 displays the rules for the LOAD, OUT and CPY instructions. These instructions have a single semantic rule, and those that require a parameter load it into the IR and call the PC_INC function again to move the PC to the next instruction.

Figures 3.12 and 3.13 show the semantic rules for the STO and JGZ instructions. STO requires three rules to handle special cases. One case is that of the PC where storing the contents of the ACC to the PC constitutes a jump with a post-STO increment. The second case deals with storing the ACC in the IR, which means that the IR equal to the ACC, rather than the destination address. The third case is the general case for addresses > 1. The two rules for JGZ define the cases for jumping and not jumping.

The semantics for INC are reduced to a succinct form through first substituting back the integers for PC, INC, IR etc. We define additional terms for S and X to prevent the need for primed variants and replace the modulo function with the commonly used infix symbol %. The intermediate semantics are:

$$\frac{\begin{array}{c} S(0) = 1 \\ K(2) = (S(2) + 1)\%\#S \end{array}}{A(S, X) \implies \langle S', X \rangle}$$

Using RPN again, we can convert the lines into a more concise form. The

$$\frac{\begin{array}{c}S(IR) = LOAD \\ S'(IR) = S'(ACC) = S(S(PC)) \\ S'(PC) = PC\_INC(S)\end{array}}{A(S,X) \implies \langle S', X\rangle}$$

(a) The LOAD instruction

$$\frac{\begin{array}{c}S(IR) = OUT \\ X' = X \cup \{S(ACC)\}\end{array}}{A(S,X) \implies \langle S, X'\rangle}$$

(b) The OUT instruction

$$\frac{\begin{array}{c}S(IR) = CPY \\ S'(IR) = S(S(PC)) \\ S'(ACC) = S(S'(IR)) \\ S'(PC) = PC\_INC(S)\end{array}}{A(S,X) \implies \langle S', X\rangle}$$

(c) The CPY instruction.

Figure 3.11: The semantics for LOAD, OUT, and CPY

$$\frac{\begin{array}{c}S(IR) = STO \\ S'(IR) = S(S(PC)) = 0 \\ S'(PC) = S(ACC) \\ S''(IR) = 0 \\ S''(PC) = PC\_INC(S')\end{array}}{A(S,X) \implies \langle S'', X\rangle}$$

(a) Storing the PC

$$\frac{\begin{array}{c}S(IR) = STO \\ S(S(PC)) = 1 \\ S'(IR) = S(ACC) \\ S'(PC) = PC\_INC(S)\end{array}}{A(S,X) \implies \langle S', X\rangle}$$

(b) Storing in the IR

$$\frac{\begin{array}{c}S(IR) = STO \\ S'(IR) = S(S(PC)) \\ S'(IR) > 1 \\ S'(S'(IR)) = S(ACC) \\ S'(PC) = PC\_INC(S)\end{array}}{A(S,X) \implies \langle S', X\rangle}$$

(c) Storing elsewhere

Figure 3.12: The semantics for storing in the PC, IR, and elsewhere

$$\frac{\begin{array}{c}S(IR) = JGZ \\ S'(IR) = S(S(PC)) \\ S(ACC) = 0 \\ S'(PC) = PC\_INC(S)\end{array}}{A(S,X) \implies \langle S', X\rangle}$$

(a) JGZ when $S(ACC) = 0$

$$\frac{\begin{array}{c}S(IR) = JGZ \\ S(ACC) > 0 \\ S'(IR) = S'(PC) = S(S(PC))\end{array}}{A(S,X) \implies \langle S', X\rangle}$$

(b) JGZ when $S(ACC) > 0$

Figure 3.13: The JGZ instruction

$$\frac{\begin{array}{c} S(IR) = ADD \\ S'(IR) = S(S(PC)) \\ S'(ACC) = mod(S(ACC) + S'(IR), \#S) \\ S'(PC) = PC\_INC(S) \end{array}}{A(S, X) \implies \langle S', X \rangle}$$

(a) The ADD instruction

$$\frac{\begin{array}{c} S(IR) = SUB \\ S'(IR) = S(S(PC)) \\ S'(ACC) = mod(S(ACC) - S'(IR), \#S) \\ S'(PC) = PC\_INC(S) \end{array}}{A(S, X) \implies \langle S', X \rangle}$$

(b) The SUB instruction

Figure 3.14: The ADD and SUB instructions for the RASP2.

line $K(2) = (S(2) + 1)\%\#S$ becomes $2K2S1 + S\#\% =$. The semantics in this concise form total 228 characters for the model semantics and 328 characters for the language semantics. The full RPN expressions of the semantics are stated in Appendix D.

3.4.2.1 RASP2

The RASP2 uses the same model semantics and largely the same language semantics as the basic RASP. The difference lies the removal of the INC and DEC rules and replacing them with ADD and SUB. Figure 3.14 shows the ADD and SUB instructions.

These semantic rules are reduced according to the procedure laid out above and the RASP2 semantics are measured as 228 characters for the model semantics – the same as for the RASP – and 357 characters for the language semantics.

3.4.2.2 RASP3

As with the RASP2, the RASP3 semantics have their own ADD and SUB instructions presented in Figure 3.15. The RASP3 semantics have sizes of 228 and 359.

$$\frac{\begin{array}{c} S(IR) = ADD \\ S'(IR) = S(S(PC)) \\ S'(ACC) = mod(S(ACC) + S(S'(IR)), \#S) \\ S'(PC) = PC\_INC(S) \end{array}}{A(S, X) \implies \langle S', X \rangle}$$

(a) The ADD instruction

$$\frac{\begin{array}{c} S(IR) = SUB \\ S'(IR) = S(S(PC)) \\ S'(ACC) = mod(S(ACC) - S(S'(IR)), \#S) \\ S'(PC) = PC\_INC(S) \end{array}}{A(S, X) \implies \langle S', X \rangle}$$

(b) The SUB instruction

Figure 3.15: The ADD and SUB instructions for the RASP3.

3.4.3 λ-calculus

Unlike the variance in the RASP and TM definitions, the λ-calculus tends to have a constant definition in the literature [12, 46, 25]. At its core, the reduction and conversion rules β, α, η do not change. Rather, the variation arises from the reduction strategy (i.e. normal or applicative order). A λ term E is constructed from the grammar:

$$E := \lambda v.E | (E\ E) | v$$
$$v \in \{a \dots z\}^+$$

As explained in Section 2.3.2.1, the three main rules of the λ-calculus are β reduction, α conversion and η conversion. 'Execution' of a term is via the substitution mechanism β reduction, while α and η conversion are used to tidy, find equalities between terms, and resolve ambiguities.

Traditional semantics of the λ-calculus assume that a reader/interpreter of the semantics can substitute expressions in situ, expanding or contracting the original expression as desired. But this property of expanding or contracting expressions is quite abstract and can be problematic to implement from a mechanical perspective. As the RASP and TM semantics above are represented at a resolution where we manipulate individual symbols/numbers/discrete structures,

it behoves us to represent the λ-calculus in a manner where we also manipulate such structures.

We can represent a term as a TM tape, one character per cell of the tape as normal. When a substitution is made, we erase the symbol which is to be replaced and repeatedly shuffle the rest of the term to the right in order to make a space large enough. We then copy the term in, repeat the process for any more variables, then erase the term from the right, the abstraction at the far left and close up the brackets.

This is a poorly disguised TM. Furthermore, this 'string evaluation' method is tedious to specify, and we suspect that it would take many semantic rules to explain the process, not including the parsing and renaming rules.

We observe that the bracketed nature of λ expressions allows us to represent them as trees. If we do this, evaluation becomes a case of shuffling sub-trees around until the expression is in normal form, if a normal form exists. This method of representation and evaluation is called *Graph Reduction* [102, 68].

Figure 3.16 shows how we could parse the expression $(\lambda a.\lambda b.b\ a)(\lambda x.x)$. Parsing begins by recognising the application of $(\lambda a.\lambda b.b\ a)$ to $(\lambda x.x)$ This forms an "APP" node which signifies an application. The right side has an abstraction ("ABS") over x and the single variable. The left side parses two abstractions, then parses the application of a to b. While it is not explicitly shown here, the application rule matches the expression from the right hand side. So if we had a third expression (say X), the first match would be rule (e) with $e_1(X)$ and would form an APP node with X on the right and the structure of 3.16 on the left.

So how do we parse an expression into this tree? The external representation is assumed to be a λ expression with unique variable names. Brackets are included only for disambiguation and expressions are left associative. The tree nodes are defined as T:

$$T = \{z, T_L, T_R\}$$
$$z = ABS|APP|v$$
$$v \in \{a \ldots z\}^+ \setminus \{\emptyset\}$$

An ABS node denotes an abstraction, APP an application, and v a variable. The variables v are drawn from a dictionary formed by the Kleene closure over

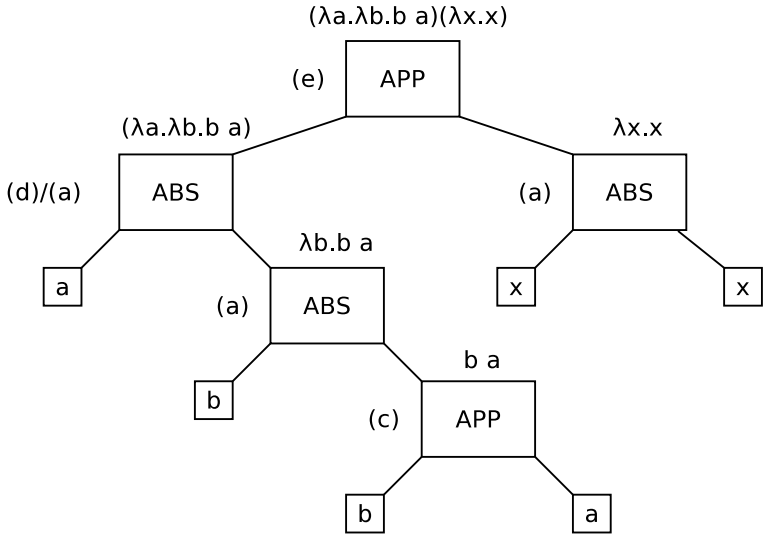


Figure 3.16: The parsing of a λ expression. Leaf nodes are formed by application of rule 3.17b.

the alphabet, excluding the empty string. Figure 3.17 shows the five rules to construct a tree from a λ expression, $T_{root} = parse(e)$.

The parsing pattern matches from the *right*, rather than from the left. This is because a LHS parsed expression will derive a right associative tree.

The resulting tree structure with the root T_{root} enables the recursive evaluation of any given λ term. In the traditional semantics, a substitution is represented by the notation $M[x/F]$. Colloquially, we say that all free occurrences of the variable x in the expression M are replaced by the expression F. If F is a variable itself, we must ensure that the name is not bound in M prior to substitution. If F is bound, then we first rename it before we substitute it in.

We define a function E to evaluate from T_{root}. The function detects where a reduction can be made, checks if there are any name conflicts with the variables, renames if necessary, and substitutes the sub-expression on the right into the sub-expression on the left.

Figure 3.18 shows the rules for β reducing an expression. The dot syntax (.) denotes an indirection which references a an element of a tree node. For example $T.T_L.z$ is a reference to the value of z in the left child of the node T.

Evaluation proceeds from the root. If a node T is an APP node and the node directly to its left, T_L is an ABS node then all occurrences of nodes named with the variable $T.T_L.T_L.z$ in the branch $T.T_L.T_R$ are replaced with $T.T_R$ (Figure

$$\frac{e \implies \lambda v.e_1}{parse(e) \implies \{ABS, parse(v)parse(e_1)\}}$$

(a) Parsing an abstraction

$$\frac{e \implies v}{parse(e) \implies \{v, \emptyset, \emptyset\}}$$

(b) Parsing a variable

$$\frac{e \implies e_1 v}{parse(e) \implies \{APP, parse(e_1), parse(v)\}}$$

(c) Applying an expression to a variable

$$\frac{e \implies (e_1)}{parse(e) \implies parse(e_1)}$$

(d) Stripping parentheses

$$\frac{e \implies e_1(e_2)}{parse(e) \implies \{APP, parse(e_1), parse(e_2)\}}$$

(e) Applying an expression to another

Figure 3.17: Rules for parsing a λ expression into a tree

$$\frac{\begin{array}{c} T.z = APP \\ T.T_L.z = ABS \\ T.T_R.z \notin B_v(T.T_L.T_R) \end{array}}{E(T) \implies S(T.T_L.T_R, T.T_R, T.T_L.T_L.z); E(T_{root})}$$

(a) Applying a substitution where the name of the RHS is not bound on the LHS

$$\frac{\begin{array}{c} T.z = APP \\ T.T_L.z = ABS \\ B_T = B_v(T.T_L.T_R) \\ T.T_R.z \in B_T \\ z' \notin B_T \end{array}}{E(T) \implies S(R_n(T.T_L.T_R, z', T.T_R.z), T.T_R, T.T_L.T_L.z); E(T_{root})}$$

(b) Applying a substitution where the name of the RHS is bound on the LHS

$$\frac{}{E(T) \implies \{T.z, E(T.T_L), E(T.T_R)\}}$$

(c) Moving down the tree

$$\frac{T = \emptyset}{E(T) \implies \emptyset}$$

(d) Terminating evaluation at the leaves

Figure 3.18: Determining where a substitution should be made

$$\frac{T.z = ABS}{B_v(T) \implies \{T.T_L.z\} \cup B_v(T.T_R)} \qquad \frac{T.z = APP}{B_v(T) \implies B_v(T.T_L) \cup B_v(T.T_R)}$$

(a) Adding a bound variable to the set (b) Recursing down the tree

$$\frac{}{B_v(T) \implies \emptyset}$$

(c) Default rule terminating the function

Figure 3.19: The function to determine the bound variables of a sub-expression

3.22a).

The function $S(a, b, c)$ is the substitution function. Given a branch of the tree to substitute into a, an expression to substitute b, and the variable which we want to be substituted c, we traverse the tree checking to see if the leaf nodes have the same value for z as c. If they are, we replace that leaf node with a copy of the expression b (Figure 3.22b). If the variable c is rebound at some point in the tree (i.e. is to the left of an ABS node) then the substitution is terminated. Once a substitution has finished, the new tree is re-evaluated from the root until no more substitutions can be made.

If b is itself a variable, we have to check that the name of b is not bound in the sub-expression. Consider the expression $(\lambda x.(\lambda f.\lambda x.f(fx))x)$. We reduce this expression by substituting the rightmost x for the bound variable f in the inner expression. If we do this without any renaming the expression will become $(\lambda x.(\lambda x.x(xx)))$. The two substituted xs are now bound by the inner abstraction. This is called *variable capture*.

To avoid this, we obtain a list of the bound variables of the sub-expression into which we are substituting (Figure 3.19). If b is not in this list, we substitute as normal (Figure 3.18a). If it is, we rename the variables in the sub-expression to something other than b (Figures 3.18b and 3.20) before substitution.

This method of evaluation aims for full evaluation via normal order reduction. The term $(\lambda a.\lambda b.ba)(\lambda x.x)$ will reduce to the normal form $(\lambda b.b(\lambda x.x))$ where the evaluation will halt.

It has been a conscious choice to reduce a term to full normal form rather than *weak head normal form* (WHNF). Where normal form is an expression with

$$\frac{T.z = k}{R_n(T, v', k) \implies \{v', \emptyset, \emptyset\}} \qquad \frac{T = \emptyset}{R_n(T, v', k) \implies \emptyset}$$

(a) Rule for renaming a variable (b) Rule for terminating at the leaves

$$\frac{}{R_n(T, v', k) \implies \{T.z, R_n(T.L, v', k), R_n(T.R, v', k)\}}$$

(c) Default rule for moving down the tree

Figure 3.20: The renaming rules

$$\frac{}{S(T, T_P, j) \implies \{T.z, S(T_L, T_P, j), S(T_L, T_P, j)\}}$$

(a) Moving down the tree

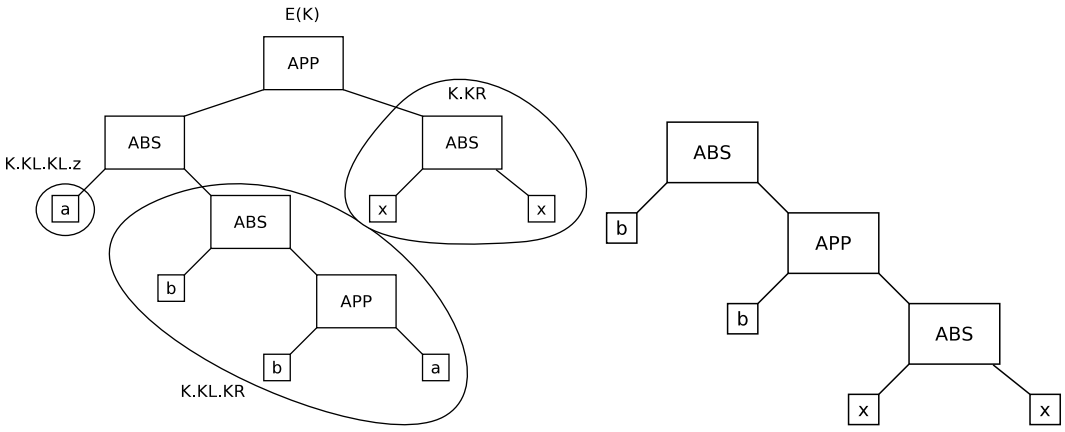
$$\frac{T.z = j}{S(T, T_P, j) \implies T_P} \qquad \frac{T = \emptyset}{S(T, T_P, j) \implies \emptyset}$$

(b) Replacing the node T with T_P (c) Terminating substitution at the leaves

$$\frac{\begin{array}{c} T.z = ABS \\ T.T_L.z = j \end{array}}{S(T, T_P, j) \implies T}$$

(d) Terminating a substitution when encountering a re-binding of the variable j

Figure 3.21: The substitution rules to replace bound variables with another expression.



(a) Before substitution. Applying rule 3.18a (b) After substitution

Figure 3.22: Application and substitution

no more reductions, an expression in WHNF is one with no reduction for the leftmost abstraction. There may be redexes in sub-expressions, but the WHNF strategy reduces *only* the leftmost outermost redex.

Adopting a WHNF strategy can reduce the number of semantic rules in the semantics. If we take it as a convention that all bound and free variables have unique names, we can reduce a term to WHNF [68] without the need for renaming. Consider the expression $(\lambda t.tt)(\lambda f.\lambda x.fx)$ with all initially unique variables. A single reduction step will produce $(\lambda f.\lambda x.fx)(\lambda f.\lambda x.fx)$ and another will produce the WHNF $(\lambda x.(\lambda f.\lambda x.fx)x)$. At this point a variable name has been duplicated, but the term is still unambiguous as to which variables are bound by each abstraction.

If we want a full normal form, we can continue to reduce the expression by substituting the rightmost x bound by the leftmost abstraction into the sub-expression for f producing $(\lambda x.(\lambda x.xx))$. This is variable capture, and shows that enforcing unique variable names in the initial term is not sufficient enough to prevent such variable capture. At the time of substitution, the machine has to check if there are unique

These semantics which strictly reduce to normal form do not confer extra computational power over WHNF, but the extra rules relax the convention of variable uniqueness. This in turn means that we are not constricted to ≤ 26 unique single symbol bindings before needing to add more symbols to the variable names.

The λ-calculus semantics are markedly different from the semantics of the RASPs and TM. The semantics focus on evaluation in the form of graph reduction and eschew semantic rules for a particular expressions. The λ expressions discussed thus far: ONE, PAIR, SUCC, etc. have no special rules as far as the semantics are concerned. These semantics have no "language semantics" component as the the RASPs do. The λ-calculus semantics are 515 characters in size.

3.4.4 SKI combinator calculus

The SKI formalism revolves around the three combinators S, K, and I. We can represent any computable term in this formalism [17, 79]. Expressions are struc-

$$\frac{e \implies (e_1)}{P(e) \implies P(e_1)}$$

(a) Stripping brackets

$$\frac{e \implies e_1(e_2)}{P(e) \implies \{A, (P(e_1), P(e_2)\}}$$

(b) Application of an expression to another

$$\frac{e \implies e_1 z}{P(e) \implies \{A, P(e_1), P(z)\}}$$

(c) Application to a variable/combinator

$$\frac{e \implies z}{P(e) \implies \{z, \emptyset, \emptyset\}}$$

(d) Parsing a variable or combinator

Figure 3.23: The parsing rules for SKI

tured similarly to the λ-calculus, and can be therefore be parsed into a tree and evaluated using graph reduction [99]. The evaluation of a SKI term is again via normal order.

A SKI term E is generated from the grammar:

$$E := (EE)|z$$
$$z := S|K|I$$

where E is a non terminal symbol, and S, K, I are terminal symbols.

Like a λ-calculus expression, we parse E into a tree structure T similar to our λ-calculus tree structure above:

$$T = \{z, T_L, T_R\}$$
$$z = S|K|I|A$$

The parsing proceeds similarly to the λ-calculus minus the rules for parsing an abstraction. Figure 3.23 shows these rules.

The parsing of the SKI expression $S(KI)I(KII)$ is shown in Figure 3.24. As with the λ semantics, application is matched from the right hand side of the expression. Each leaf node in a SKI tree is a combinator or variable.

Evaluation of SKI terms requires that we look ahead for combinators and expressions because a combinator will not evaluate if it does not have enough arguments (e.g $SII \equiv SII$). Figure 3.25 shows the reduction rules for S, K and I.

To evaluate the identity function from a node T, we check to see that the left branch is an I (Figure 3.26). The reduction returns the right branch of T. As with the λ semantics, we re-evaluate from the root of the tree T_{root} after each

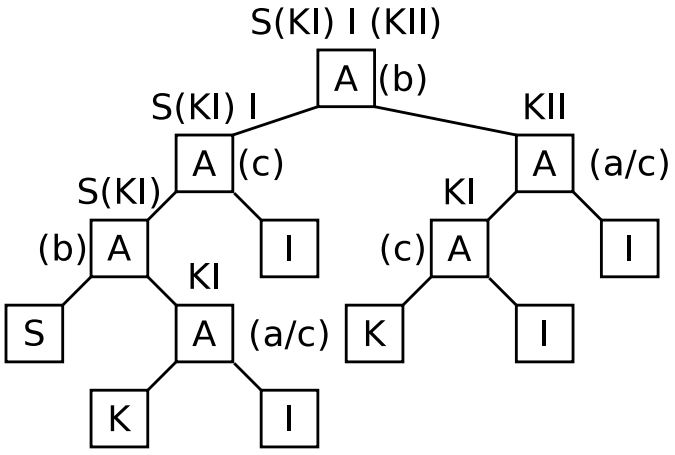


Figure 3.24: The tree of a parsed SKI expression

substitution.

The evaluation of K requires that the leftmost branch terminates with a K. The K combinator ignores its second argument and returns the first (Figure 3.27).

The S combinator requires 3 arguments. The leftmost branch three levels down should be an S and it should have 3 expressions to the right of each application node on each level. The result of this reduction is a tree which applies e_1 to e_3 (j), e_2 to e_3 (g), and j to g.

As a form of hybrid between the singular focus on graph reduction (λ-calculus) and semantic rules for particular instructions (RASPs). The SKI semantics evaluate expressions in a graph reduction manner, but the particular reduction is informed by the combinator read. The semantics for the SKI are the smallest at 291 characters.

3.5 Semantic Sizes

Measuring the semantics of our models yields Table 3.1. The Turing machine is the simplest imperative model, and an abstract machine to interpret and run a TM is consequently small. The RASP Figures are split into model+language semantics so that the difference in their instruction sets can be quickly seen.

RASP	RASP2	RASP3	TM	SKI	λ-calculus
228+328	228+357	228+359	335	291	515

Table 3.1: The semantic sizes for the models

$$\frac{T.v = A}{R(T) \implies \{A, R(T_L), R(T_R)\}}$$

(a) Moving down the tree

$$\frac{\begin{array}{c} T.z = A \\ T.T_L.z = I \end{array}}{R(T) \implies T.T_R; R(T_{root})}$$

(b) The I rule

$$\frac{\begin{array}{c} T.z = A \\ T.T_L.T_L.z = K \end{array}}{R(T) \implies T.T_L.T_R; R(T_{root})}$$

(c) The K rule

$$\frac{T = \emptyset}{R(T) \implies T}$$

(d) The terminating rule

$$\frac{\begin{array}{c} T.z = A \\ T.T_L.T_L.T_L.v = S \\ T.T_L.T_L.T_R = e_1 \\ T.T_L.T_R = e_2 \\ T.T_R = e_3 \end{array}}{R(T) \implies \{A, \{A, e_1, e_3\}, \{A, e_2, e_3\}\}; R(T_{root})}$$

(e) The S rule

Figure 3.25: SKI reduction rules

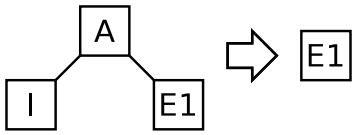


Figure 3.26: I reduction

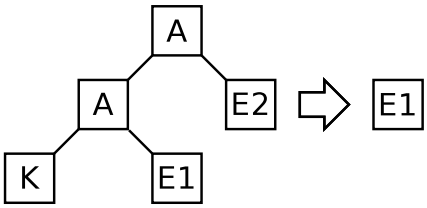


Figure 3.27: K reduction

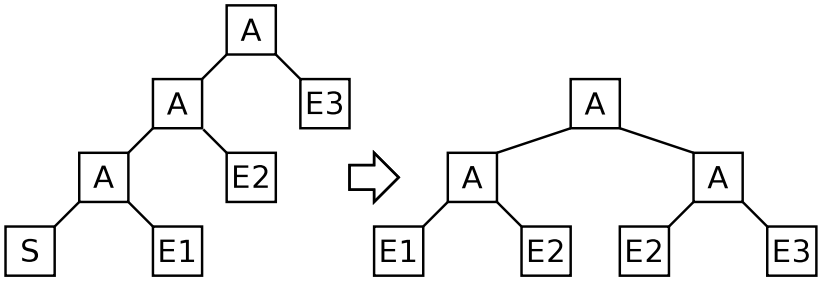


Figure 3.28: S reduction

Interestingly, the semantics for the λ-calculus are more comparable in size to the RASP rather than the traditional comparison to the Turing machine. In the next chapter we shall see what effect this has on program size and Chapter 6 will discuss how the comparative sizes of these semantics relate to the sizes of programs.

Chapter 4

Arithmetic, List and Universal Programs

This chapter covers the implementation and measurement of programs which have been selected to benchmark the models. The concepts of primitive and partial recursion are introduced, the functions listed, and realisations of these functions explained in each model.

To strive for an equitable comparison, the programs featured to compute these function are both succinct and operate over natural encodings of the function input. There are programs which compute a function using less program information, but use sparser input encodings. Section 6.6 gives an example of such a machine, and Section 6.5 details the growth rates of natural inputs for each program and model.

For the sake of brevity, not all all functions are explained in depth for each individual model – the RASPs and λ-calculus/SKI are often grouped as they use the same algorithm. The full programs for each model and function are presented in Appendix B.

4.1 Primitive and Partial Recursion

The definition of the primitive recursive (PR) functions starts with the natural number 0, the successor function, the projection function and induction [63].

The successor function adds 1 to a natural number n, thus obtaining the next

number in the sequence:

$$s(n) = n + 1$$

From that we can derive the predecessor function, which given $n + 1$ returns n:

$$p(n) = \begin{cases} 0 & : z(n) = 1 \\ x & : n = (s(x)) \end{cases}$$

The predecessor function requires a test for zero z:

$$z(n) = \begin{cases} 1 & : n = 0 \\ 0 & : n > 0 \end{cases}$$

The composition and projection functions pack and unpack tuples of variables. The base form of the PR functions has a restriction on the number of variables which a function can operate over. While a function can operate over any number of *constants*, PR induction can only be performed on a single variable. So if T is a PR function then the definition of $T(1, f)$ is permissible, but $T(x, f)$ (where x and f are two natural numbers changed by T) is not.

However it seems appropriate that if variable x of T is the result of another PR function L, then the inductive definition of x is 'handled' by the definition of L. Intuitively, the composition of PR functions should also result in a PR function. Kleene [46] treats this matter in a formal manner, explaining the role of composition and projection. In the function definitions which follow, we shall be using standard mathematical notation rather than 'strict PR' formulations which make use of composition and projection.

All primitive recursive functions are *total*. That is they are defined on all inputs in their domain. There exists total functions which are not primitive recursive however [7].

The partial recursive functions are defined with the inclusion of the μ operator. Also known as the *minimisation*, or *unbounded search* operator, μ is used to search for the smallest natural number which satisfies some function. Where the PR functions recurses *downwards* towards zero, μ recurses *upwards* and may never return a result. Say there was a TM R, and we want to find out the number of steps R will make before halting: $n = \mu(R)$. The minimisation operator μ is

paired with a UTM and runs R a step at a time until R reaches some defined halting state. However R could loop forever in which case μ will never return a value [63, 20, 46].

The functions which form the comparison set are a mixture of primitive and partial recursive function. The set of primitive recursive functions include *arithmetic operations*: addition, subtraction, equality, multiplication, division, and exponentiation. And *operations on lists*: list membership, linear search, reversal via constructing a new list, reversal via swapping elements in place, and bubble sorting. The partial recursive functions are the universal Turing and universal RASP machines.

This function set aims to represent a reasonable spread of operations such that a wide range of arbitrary programs makes use of one or more of these functions. Many of the implementations are drawn from the literature, especially implementations of the arithmetic functions and UTM in the TM and λ-calculus. The list reversal and search functions in the λ-calculus have also been drawn from the literature. The other functions have been hand constructed and continuously refined by the author.

The arithmetic functions are hierarchical in nature where the functions on level n make use of the functions on level $n-1$. These arithmetic functions operate over pairs of data, while the list functions operate over a finite list of contiguous data and demonstrate several common functions like search and sort. The two reversal functions highlight how differences in the intensionality of two programs to compute the same function affects the program information. Where possible, the definitions and programs presented here are drawn from the literature.

4.2 The Arithmetic Functions

The arithmetic functions are a hierarchy defined over the natural numbers. The base functions are the successor and predecessor functions which are defined above. Each subsequent level in the hierarchy is defined by multiple application of the functions in the levels below. Addition is iterated successor, multiplication is iterated addition, and so on. These functions are all primitive recursive and

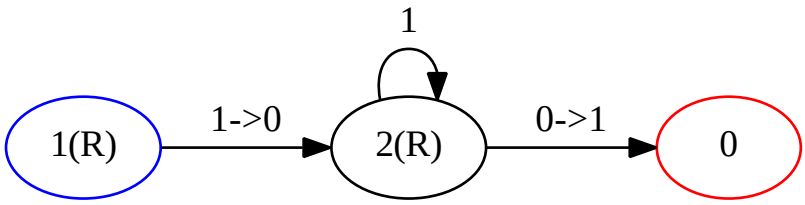


Figure 4.1: The state diagram for the addition TM

are therefore guaranteed to halt.

Each function here is detailed and the programs/expressions in all of the models are described. Many of the RASPs and SKI calculus programs behave similarly to other RASPs or λ-calculus expressions and so may not be detailed to avoid needless repetition.

4.2.1 Addition

The definition of the function add is:

$$add(x, y) = \begin{cases} y & : x = 0 \\ add(p(x), s(y)) & : x \neq 0 \end{cases}$$

4.2.1.1 Turing Machine

Figure 4.1 shows a state diagram of the machine. The TM starts in state 1, and follows the edges of the transitions. If a transition is labelled with a single symbol, the TM will write that symbol back. Transitions of the form $x \to y$ will overwrite x with y. The direction that the machine will shift is annotated as 'L' or 'R' on the states.

The initial tape for the addition Turing machine contains the numbers x and y inscribed in unary with a single space between them. The head of the machine begins over the far left symbol of x. It replaces this symbol with a blank and shifts right until it reaches the space between x and y. Once this space has been found, the TM fills it in and halts.

Instr	Data	I Label	D Label
LOAD	3	:addStart	;x
JGZ	'adding		
HALT			
DEC		:adding	
STO	'x		
LOAD	4		;y
INC			
STO	'y		
LOAD	1		
JGZ	'addStart		

Figure 4.2: The RASP program for addition.

Instr	Data
LOAD	x
ADD	y

Figure 4.3: RASP2 adding x and y.

Instr	Data	I Label
LOAD	x	
ADD	'label	
y		:label

Figure 4.4: RASP3 adding x and y.

4.2.1.2 RASP

The RASP performs addition by looping over x, decrementing it and incrementing y until x is zero before halting. Figure 4.2 adds the numbers 3 and 4 together to produce 7.

4.2.1.3 RASP2/3

The RASP2 and RASP3 semantics have pre-defined ADD and SUB instructions so all that they have to do is invoke these instructions. Tables 4.3 and 4.4 show very concise programs to add two numbers together.

4.2.1.4 λ-calculus

Addition in the λ-calculus exploits the higher order functionality of the Church numerals. Where SUCC $\equiv (\lambda n.\lambda f.\lambda x.f(nfx))$, addition is $\lambda x.\lambda y.x$ SUCC y. Figure 4.5 shows the reduction with the numbers 3 and 1.

4.2.1.5 SKI

The SKI expression for addition is very similar to the λ expression because the SKI expression is derived from λ expression via bracket abstraction (Section 2.3.2.2).

$$
\begin{aligned}
\text{ADD THREE ONE} &\Rightarrow^*_\beta \text{ THREE SUCC ONE} \\
&\equiv \lambda f.\lambda x.f(f(fx))\ \text{SUCC ONE} \\
&\Rightarrow^*_\beta \text{ SUCC(SUCC(SUCC ONE))} \\
&\Rightarrow_\beta \lambda f.\lambda x.f(\text{SUCC(SUCC ONE)}fx) \\
&\Rightarrow_\beta \lambda f.\lambda x.f((\lambda j.\lambda h.j(\text{SUCC ONE } jh))fx) \\
&\Rightarrow^*_\beta \lambda f.\lambda x.f(f(\text{SUCC ONE } fx)) \\
&\Rightarrow_\beta \lambda f.\lambda x.f(f((\lambda j.\lambda h.j(\text{ONE } jh))fx)) \\
&\Rightarrow^*_\beta \lambda f.\lambda x.f(f(f(\lambda a.\lambda b.ab)fx)) \\
&\Rightarrow_\beta \lambda f.\lambda x.f(f(f(fx))) \\
&\equiv \text{ FOUR}
\end{aligned}
$$

Figure 4.5: Addition of the Church numerals 3 and 1.

The successor function is defined as $S(S(KS)K)$ and prepends the expression to any natural number to create the successor. The full expression for addition is $SI(K(S(S(KS)K)))$ which operates exactly as the above λ expression.

4.2.2 Subtraction

The "proper" form of subtraction returns $x - y$ if $x >= y$; otherwise it returns zero:

$$
sub(x, y) = \begin{cases} x & : y = 0 \\ 0 & : x = 0 \\ sub(p(x), p(y)) & : y \neq 0 \wedge x \neq 0 \end{cases}
$$

4.2.2.1 TM

The initial tape of the TM is arranged with x followed by y in unary, separated by a single blank symbol. The TM traverses to the far right side of y and replaces the rightmost '1' with a blank. It then moves to the far left and replaces the leftmost '1' from x.

If the machine encounters two consecutive blanks when moving right, it halts immediately since y has been depleted. If it encounters consecutive blanks when moving left, x has been depleted, so it shifts right again and erases the rest of y before halting.

Instr	Data	I Label	D Label
LOAD	y	:subStart	;y
JGZ	'subbing		
HALT			
SUB	1	:subbing	
STO	'y		
LOAD	x		;x
JGZ	'subbing2		
HALT			
SUB	1	:subbing2	
STO	'x		
LOAD	1		
JGZ	'subStart		

Figure 4.6: RASP2 properly subtracting x and y

4.2.2.2 RASP

Subtracting y from x in the RASP involves repeatedly decrementing both values until one of them reaches zero. The program to do this is almost exactly the same as the subtraction program in Figure 4.6, with the exception that the "SUB 1" instructions are replaced with "DEC".

4.2.2.3 RASP 2/3

The SUB functions for the RASP2 and 3 do not conform to the rules of proper subtraction because they pay no heed to the underflow of registers. This means that SUBbing y from x directly will not return 0 in the event of $y > x$, which makes the SUB instruction unsuitable for the task of proper subtraction.

The basis of subtraction is to decrement x and y in turn until one of them reaches zero. Figure 4.6 shows the RASP2 program to do this. It is not hard to define an analogous machine in the RASP3. The lack of a DEC instruction for the RASP2 and 3 means that the decrementing of x and y requires two instructions rather than just one.

Before the decrement y, it is tested for zero. If y is zero the program halts, otherwise it is decremented and x is tested for zero. If x is greater than zero, the machine decrements it and loops to decrement y again. The result of the subtraction is held in the register for x.

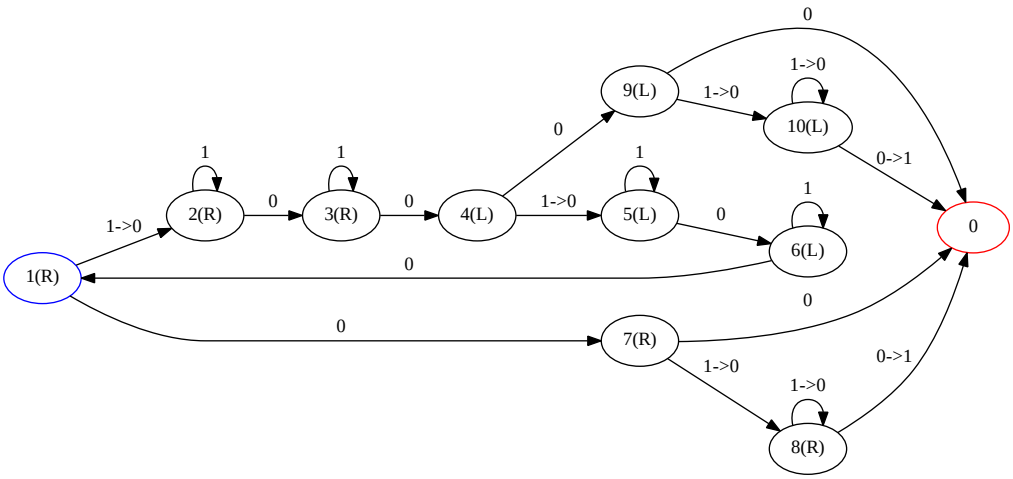


Figure 4.7: The TM to calculate equality of x and y.

4.2.2.4 λ-calculus and SKI

The PRED function for the SKI and λ-calculus has the same definition as the predecessor function $p()$. So any application of ZERO to PRED will result in ZERO as a matter of course. This means that any y can be subtracted from a smaller x using PRED and the result will be zero. The SUB expression is therefore:

$$\text{SUB} \equiv (\lambda a.\lambda b.b \text{ PRED } a)$$

which is evaluated much like the expression for addition above.

4.2.3 Equality

Equality on the naturals recursively decrements x and y until one or both reach zero. A return value of 1 (true) is returned if they are both zero, and 0 (false) is returned if they are not both zero at the same time:

$$eq(x,y) = \begin{cases} 1 & : x = 0 \wedge y = 0 \\ 0 & : (x = 0 \wedge y \neq 0) \vee (x \neq 0 \wedge y = 0) \\ eq(p(x), p(y)) & : \text{otherwise} \end{cases}$$

4.2.3.1 TM

The Turing Machine to compute equality begins with the numbers x and y inscribed on a tape in unary with a single blank space between them and the head

Instr	Data	I Label	D Label
LOAD	6		;num1
SUB	6		;num2
JGZ	'out		
HALT			
LOAD	1	:out	

(a) The RASP2 program for 6 =? 6

Instr	Data	I Label	D Label
LOAD	6		;num1
SUB	'num2		
JGZ	'out		
HALT			
LOAD	1	:out	
HALT			
5	:num2		

(b) The RASP3 program for 6 =? 5

Figure 4.8: RASP2/3 programs for equality

over the far left of x. The machine (Figure 4.7) begins by removing the far left digits of x and the far right digits of y one at a time to preserve the space in-between x and y.

If the machine removes a digit from x and finds there are no more digits in y, it moves back over x eliminating the remaining digits before halting. If the machine finds that there are no more digits in x, it moves across to y. If there are digits in y, it removes them and halts with a blank tape. If there are no digits in x and y, it changes a 0 to a 1 and halts.

4.2.3.2 RASP Machines

In the above equation, two numbers are equal if they are both zero after the same number of predecessor operations. The RASP repeatedly decrements x and y until x is zero. At that point y is checked for zero. If it is, the two numbers are equal, 1 is loaded into the ACC and the machine halts. If not, zero is loaded and the machine halts.

The RASP2 and 3 just subtract y from x. If the answer is 0, the machines halt with a 1 in the ACC. If not, they halt with zero (Figure 4.8).

4.2.3.3 λ-calculus and SKI

Rather than outputting the numerals 1 and 0, the λ-calculus and SKI use the terms TRUE and FALSE (Section 2.3.2.1) respectively. The LEQ expression tests if one number is less than or equal to another. The EQ expression is a conjunction of LEQ x y and LEQ y x. It tests if m is less than or equal to n and then if n is less than or equal to m. If both expressions are true, then $m = n$:

$$\begin{aligned} \text{LEQ} &\equiv \lambda m.\lambda n.n \text{ PRED } m(\lambda x.\text{FALSE})\text{TRUE} \\ \text{EQ} &\equiv (\lambda m.\lambda n.\text{AND}(\text{LEQ } m\ n)(\text{LEQ } n\ m)) \end{aligned}$$

4.2.4 Multiplication

Multiplication is iterated addition:

$$mul(x,y) = \begin{cases} 0 & : x = 0 \vee y = 0 \\ add(x, mul(x, p(y)) & : x \neq 0 \wedge y \neq 0 \end{cases}$$

4.2.4.1 TM

Multiplication in the TM uses a tape of x and y written in unary with a single space between them like the other programs seen thus far. It first removes the leftmost digit of x and makes a copy of y on the right hand side of the tape, leaving a gap of a single blank between y and its copy.

Once a copy has been made, the TM removes another digit from x and copies y again, placing it next to the previous copy. This continues until all of x is depleted, at which point the machine moves right to erase y before halting with $x \times y$ on the tape.

4.2.4.2 RASP Machines

Multiplication of two numbers in the RASP is repeated addition. The multiplier (y) is initially tested for zero. If it is zero, the machine halts. The machine tests the multiplicand (x) for zero and then decrements it, storing the new multiplicand.

A copy is made of the multiplier and the copy is added to a "runningTotal" register which is initialised as zero. The program loops and and continues until the value for x is 0. The result of the program is held in the "runningTotal" register and holds the value of $(x \times y)\%2^n$ (Figure 4.9) where n is the number of RASP bits. The RASP2 and RASP3 use the same looping mechanism, but use their respective ADD functions to increase "runningTotal".

Instr	Data	I Label	D Label
CPY	'multiplier		
JGZ	'return		
HALT			
LOAD	5	:return	;multiplicand
JGZ	'mul_start		
HALT			
DEC		:mul_start	
STO	'multiplicand		
LOAD	5		;multiplier
STO	'tmp		
LOAD	0	:loop	;tmp
JGZ	'add		
LOAD	1		
JGZ	'return		
DEC		:add	
STO	'tmp		
LOAD	0		;runningTotal
INC			
STO	'runningTotal		
LOAD	1		
JGZ	'loop		

Figure 4.9: The RASP program to multiply 5 and 5

4.2.4.3 λ-calculus

Unlike the RASP and TM, multiplication in the λ-calculus is not iterated addition which is a deviation from the definition above. Rather than iteration, the expression $(\lambda m.\lambda n.\lambda f.m(n\ f))$ combines two Church numerals m and n by creating m copies of $(n\ f)$. In these expressions, n is applied to the free variable f and the resulting expressions are applied to each other.

The intermediate step of applying the f ensures that the subsequent applications of the numerals to each other would be substituted for the second argument

$$
\begin{aligned}
\text{MULT TWO TWO} &\equiv (\lambda m.\lambda n.\lambda f.m(n\ f))\ \text{TWO TWO} \\
&\Rightarrow_\beta (\lambda n.\lambda f.\text{TWO}(n\ f))\text{TWO} \\
&\Rightarrow_\beta \lambda f.\text{TWO}(\text{TWO}\ f) \\
&\Rightarrow^*_\beta \lambda f.\lambda x.(\lambda a.\lambda b.a(a\ b)f)((\lambda a.\lambda b.a(a\ b)f)x) \\
&\Rightarrow_\beta \lambda f.\lambda x.(\lambda b.f(f\ b)((\lambda a.\lambda b.a(a\ b)f)x)) \\
&\Rightarrow_\beta \lambda f.\lambda x.f(f((\lambda a.\lambda b.a(a\ b)f)x)) \\
&\Rightarrow_\beta \lambda f.\lambda x.f(f(\lambda b.f(f\ b)x)) \\
&\Rightarrow_\beta \lambda f.\lambda x.f(f(f(f\ x))) \\
&\equiv \text{FOUR}
\end{aligned}
$$

4.2.4.4 SKI

The SKI term for multiplication is striking in its simplicity and is the shortest term of all the functions: $S(KS)K$. Multiplication works by creating a new number through applying a multiplier to a multiplicand so that we get x copies of y. The term prevents the application of x to y by means of the leading S and K which hold the term in normal form until something can be applied to the new number.

$$
\begin{aligned}
\text{MULT TWO THREE} &\equiv S(KS)K(S(S(KS)K)I)(S(S(KS)K)(S(S(KS)K)I)) \\
&\Rightarrow_S KS(S(S(KS)K)I)(K(S(S(KS)K)I))(S(S(KS)K) \\
&\quad (S(S(KS)K)I)) \\
&\Rightarrow_K S(K(S(S(KS)K)I))(S(S(KS)K)(S(S(KS)K)I)) \\
&\Rightarrow_S S(K\ \text{TWO})\text{THREE}
\end{aligned}
$$

This expression for six is shorter than the expression for six obtained by repeatedly finding the successor of zero. This behaviour inspired the factorisation method described in Section 3.3.5.

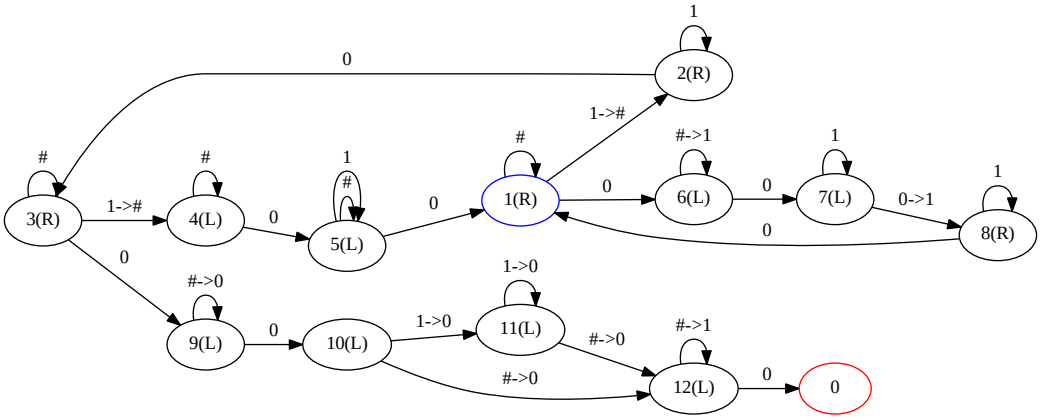


Figure 4.10: The TM to divide x by y

4.2.5 Division

Integer division returns a pair of a quotient and a remainder. The divisor is repeatedly subtracted until $x < y$. The number of times this is accomplished is counted, and the remainder is whatever is left of x after this repeated subtraction of y:

$$div(x, y) = \langle quot(x, y), rem(x, y) \rangle$$

$$quot(x, y) = \begin{cases} 0 & : x < y \\ 0 & : y = 0 \\ s(quot(sub(x, y), y)) & : \text{otherwise} \end{cases}$$

$$rem(x, y) = \begin{cases} 0 & : y = 0 \\ sub(y, x) & : x < y \\ rem(sub(x, y), y) & : \text{otherwise} \end{cases}$$

4.2.5.1 TM

TM division starts with y followed by x on the tape separated by a blank (note the swapping of the two numbers). The machine first tries to mark y symbols of x. If it can do this (i.e $y \leq x$) then it moves to the left of y and prints a '1'. It then repeats the process until there are no more symbols left in x to mark.

If y divides x perfectly, then both x and y are eliminated from the list to leave the quotient. If it does not, then the machine eliminates x and the remaining unmarked y symbols to leave the quotient and remainder on the tape separated by a '0' (Figure 4.10).

Instr	Data	I Label	D Label
LOAD	y	:start	;y
JGZ	'divStart		
HALT			
STO	'tmp	:divStart	
LOAD	x		;x
STO	'remainder		
LOAD	0	:loop	;tmp
JGZ	'sub		
LOAD	1		
JGZ	'return		
DEC		:sub	
STO	'tmp		
CPY	'x		
JGZ	'nl		
HALT			
DEC		:nl	
STO	'x		
LOAD	1		
JGZ	'loop		
LOAD	0	:return	;quotient
INC			
STO	'quotient		
JGZ	'start		
0		:remainder	

Figure 4.11: RASP2 dividing x by y.

4.2.5.2 RASP Machines

Figure 4.11 shows the RASP machine to perform integer division. The RASP first checks that y isn't zero. It then copies the value x to the remainder register and attempts to subtract y from x. If it succeeds, the quotient value is incremented and the program jumps back to the start. If it cannot fully subtract y from x, the program halts immediately and the quotient and remainder can be found in the memory at the labelled locations.

The RASP2 and 3 operate almost exactly as the RASP does. Since the SUB instruction does not conform to the rules of proper subtraction, the machine can not know if $x < y$ through directly subtracting. Therefore the machines have to use "SUB 1" and cannot take advantage of their potential.

4.2.5.3 λ-calculus and SKI

Division in the λ-calculus and SKI is the first recursive function in the set of arithmetic functions defined by means of the Y combinator:

$$\mathrm{Y}(\lambda g.\lambda q.\lambda a.\lambda b.\mathrm{LT}a\ b(\mathrm{PAIR}\ q\ a)(g(\mathrm{SUCC}\ q)(\mathrm{SUB}\ a\ b)b))\mathrm{ZERO}$$

The initial ZERO is the quotient of the division. If a (x) is less than b (y) this quotient is returned paired with x.Each recursive call tests if $x < y$. If not, the function is called again with an incremented quotient and $x - y$ as the new value for x.

4.2.6 Exponentiation

Exponentiation is repeated application of the multiplication function:

$$exp(x, y) = \begin{cases} 1 & : y = 0 \\ mult(x, exp(x, p(y))) & : y \neq 0 \end{cases}$$

4.2.6.1 TM

The TM is initialised with a tape of y, x, and f which is a single 1. Each term is separated by a single space. The TM checks off one of the digits of y and proceeds to multiply x by f to create a new number to the right of f.

Once the multiplication has been completed, the current f is erased and the result of the multiplication; $x \times f$ assumes the role of f. The machine continues by erasing another digit of y and repeating the process with x and the new f. This proceeds until there are no more digits in y at which time the machine halts. The output tape contains x and f (which is the results of x^y) with one or more blank symbols between them.

4.2.6.2 RASP Machines

RASP exponentiation is a loop added to the multiplication program. The exponent is initially checked for zero. If it is, the machine halts and the return value defaults to 1. Otherwise, the power is decremented and the current total (f) is

multiplied by x.

Once this is done, the program jumps to the start of the program, tests and decrements the power, continuing until the power is 0. For the RASP2 and 3, exponentiation is multiplication inside another loop and is written as expected.

4.2.6.3 λ-calculus and SKI

The λ-calculus and SKI again leverage the higher order functionality of the Church numerals. Exponentiation applies one Church numeral to another. In the case of x^y, x is applied to y:

$$
\begin{aligned}
\text{EXP x y} &\equiv (\lambda a.\lambda b.ba)\text{TWO THREE} \\
&\Rightarrow^*_\beta \text{THREE TWO} \\
&\Rightarrow_\beta \lambda x.\text{TWO(TWO(TWO } x)) \\
&\Rightarrow_\beta \lambda x.\lambda f.\text{TWO(TWO } x(\text{TWO(TWO } x)f) \\
&\Rightarrow_\beta \lambda x.\lambda f.(\lambda a.\text{TWO } x(\text{TWO } xa))((\lambda a.\text{TWO } x(\text{TWO } xa))x) \\
&\Rightarrow_\beta \lambda x.\lambda f.\text{TWO } x(\text{TWO } x((\lambda a.\text{TWO } x(\text{TWO } xa))x)) \\
&\Rightarrow_\beta \lambda x.\lambda f.(\lambda a.x(xa))((\lambda a.x(xa))((\lambda a.(\lambda b.x(xb))((\lambda b.x(xb))x))x)) \\
&\Rightarrow_\beta \lambda x.\lambda f.x(x((\lambda a.x(xa))((\lambda a.(\lambda b.x(xb))((\lambda b.x(xb))x))x))) \\
&\Rightarrow_\beta \lambda x.\lambda f.x(x(x((\lambda a.(\lambda b.x(xb))((\lambda b.x(xb))x))x)))) \\
&\Rightarrow_\beta \lambda x.\lambda f.x(x(x(x((\lambda a.x(xa))((\lambda a.x(xa))x))))) \\
&\Rightarrow_\beta \lambda x.\lambda f.x(x(x(x(x((\lambda a.x(xa))x)))))) \\
&\Rightarrow_\beta \lambda x.\lambda f.x(x(x(x(x(x(xf))))))
\end{aligned}
$$

The EXP function could be defined as the identity and computed as $(\lambda x.x)yx$. However a function constructed in this manner only requires a single argument and *if* two were supplied, both were Church numerals, and happened to be supplied in the correct order, only then will the "correct answer" be calculated. This behaviour is more an accidental side effect of the identity function and evaluation method given the correct conditions than any kind of calculated construction.

The SKI expression is very similar. Given two numerals A and B:

$$\begin{aligned} \text{EXP A B} &\equiv S(K(SI))KAB \\ &\Rightarrow_S K(SI)A(KA)B \\ &\Rightarrow_K SI(KA)B \\ &\Rightarrow_S IB(KAB) \\ &\Rightarrow_I B(KAB) \\ &\Rightarrow_K BA \end{aligned}$$

4.3 Functions on a List

As opposed to the arithmetic functions above which operate on two discrete pieces of data, the list functions operate on a list structure. For our purposes, a list is a structure of zero or more elements which are connected in a linear fashion. Lists are often delimited to separate elements (like in the TM) and may have end markers (SKI and λ-calculus; NIL).

Common recursive definitions making use of lists use four base functions. The 'head' function returns the first member of a list, the 'tail' function returns the list without the first element, and '[]' is the empty list. Like the arithmetic functions, the list functions are primitive recursive.

4.3.1 List Membership

The list membership function returns true if an element is in the list and false otherwise. It can be defined thus:

$$mem(x, list) = \begin{cases} true & : eq(x, head(list)) \\ false & : mem(x, []) \\ mem(x, tail(list)) & : \text{otherwise} \end{cases}$$

4.3.1.1 TM

The list on the tape for the membership TM is a sequence of binary numbers separated '*' symbols and bookended by the end list symbol 'E'. The target to be searched for is prepended by a 'T', and the symbol to the left of it is 0 if the

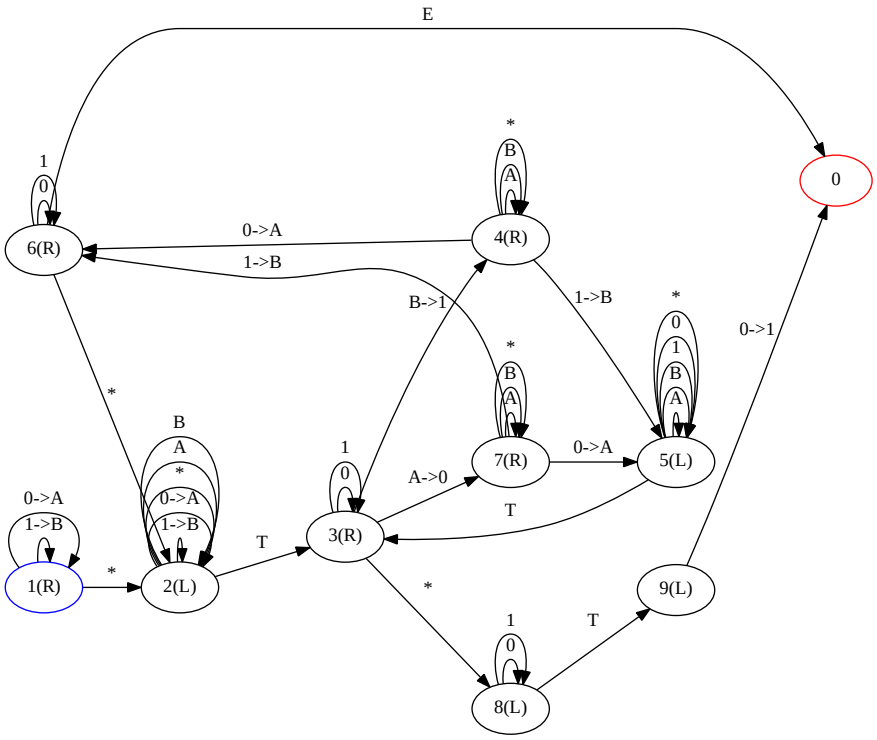


Figure 4.12: The TM to decide membership of a list.

target number has not been found and 1 if it has. An initial tape is of the form:

$$0T\langle x\rangle * \langle data1\rangle * \langle data2\rangle * \ldots E$$

Figure 4.12 shows the state machine for the membership TM. The TM searches for the target, x by marking off a character in the target, shifting to the current data range being checked and attempting to mark off the same character in the same position. If it can, the machine continues to try and mark off all the characters in the target. If the current data doesn't match the target, the machine marks off all the data in the range, resets the target and tries again.

The 'found value' is at the far left of the tape, after the 'T'. The machine halts with 0 = false and 1 = true. If the machine does not find the target in the list before reaching the end of the list it halts. If the machine matches all of the symbols in the target with the symbols in one of the the data blocks, it moves back to the start and overwrites the 'found value' with a 1 before halting.

4.3.1.2 RASP Machines

A RASP list is defined at the end of the program memory and is a contiguous array of elements with one element per address. Labels are defined on the start and end addresses of the list so that the machine knows the size and bounds.

The RASP programs to determine membership start with the first element of the list, comparing it to the target. If the element is equal to the target, it loads a 1 into the ACC and halts. If not, the address to be compared is incremented and tested against the end of the list.

If the current address is still a part of the list, the machine loops and tests the element in the address against the target. If the address is past the end of the list, a 0 is loaded into the accumulator and the machine halts.

The RASP2 and 3 use their subtraction instructions to work out if the target is equal to the current element whereas the RASP has an equality function defined in the memory which it uses repeatedly.

4.3.1.3 λ-calculus and SKI

Lists in the λ-calculus and SKI are expressions made of of nested pairs terminated with the NIL expression:

$$(\text{PAIR } A(\text{PAIR } B(\text{PAIR } \ldots(\text{PAIR } Z \text{ NIL})\ldots)))$$

This function searches through a list of numbers for a specific one:

$$\text{MEM} \equiv \text{Y}(\lambda a.\lambda b.\lambda c.\text{NULL } b \text{ FALSE}(\text{EQ}(\text{HEAD } b)c \text{ TRUE}(a(\text{TAIL } b)c)))$$

This function initially tests the list to see if it is NIL. If it is, the end of the list has been reached and the target has not been found. FALSE is returned. If it is not NIL, the head of the list (b) is tested to see if it is equal to the target (c). If it is, then TRUE. If not, the function recurses to test the rest of the list.

4.3.2 Linear Search

The linear search of a list for an element x returns either the position of an element or the size of the list + 1:

$$search(x, list) = \begin{cases} 0 & : x = head(list) \\ 1 & : list = [] \\ s(search(x, tail(list))) & : x \neq head(list) \wedge list \neq [] \end{cases}$$

4.3.2.1 TM

The TM tape of a searchable list is a set of $\langle address, data \rangle$ pairs. Each pair is structured as: $\#address * data\#$ where the '#' separates the pairs and '$*$' is an internal delimiter. The tape of this machine is structured as:

$$E\langle ReturnAddress\rangle T\langle target\rangle\#\langle addr1\rangle * \langle data1\rangle\# \ldots E$$

Initially, the "ReturnAddress" portion of the tape is empty, and the "target" portion contains the data which the list is to be searched for.

To locate the target, the TM searches the list as in the membership TM. If the current in $datax$ is the target, the machine copies the address of that location to the "ReturnAddress" between the 'E' and 'T' symbols before halting. If the TM reaches the far right of the list without finding the target, it returns to the return address and replaces the symbols with asterisks ($*$) to signify that the target is not a member of the list.

4.3.2.2 RASP Machines

The linear search RASP machines operate as the membership RASPs except that they halt with the address of the found element in the accumulator. If the list does not contain the target, the RASP increments the final address of the list and halts with it in the accumulator.

$Y(S(K(S(K(S(S(KS)(S(K(S(K(S(K(SS(K(K(KI)))))(S(S(\text{NULL})\ldots$
$(K\ \text{ONE})))))(S(S(K\ \text{EQUAL}))(\text{HEAD}))))\text{ZERO}))))\ldots$
$(S(K(S(K(\text{SUCC})))))))(S(K(S(K(SS(K(\text{TAIL}))))K))))$

Figure 4.13: The SKI term with only the abstraction combinators shown.

4.3.2.3 λ-calculus and SKI

The abstract λ-calculus/SKI term to search a list is:

$$\begin{aligned}\text{SEARCH T L} \equiv\ & \text{(NULL L ONE (EQ (HEAD L) T) ZERO} \\ & \text{(SUCC SEARCH T (TAIL L)))}\end{aligned}$$

In the SKI, the recursive SEARCH call is afforded by the use of the Y combinator which is $SSK(S(K(SS(S(SSK))))K)$. For the copy of SEARCH, and those of L and T, a series of S and K combinators draw the L and T arguments into the body of the function.

Figure 4.13 shows the term with all of the combinators to move terms into the expression. This overhead is typical of SKI terms that have been obtained through bracket abstraction; a term can blow-up in size through the number and occurrences of abstracted values.

The expression first tests if it is the last element of the list – which is NIL. If it is, the expression returns ONE. If the current element is the target, the expression returns ZERO. If the current element in not NIL and is not the same as the target, the expression returns the successor of a recursive call to itself. The expression successively increments until it finds the target or end of the list to either return the position of the target, or the size of the list+1.

4.3.3 Reversing a List

Functionally reversing a list involves building a new list from the old one. Each recursive call adds a new outer element until the end of the input list is reached.

$$rev(l) = revh(l, []) = \begin{cases} revh(tail(l), pair(head(l), x)) & : revh(l, x) \\ x & : revh([], x) \end{cases}$$

4.3.3.1 TM

The list structure for this TM consists of binary words separated by the symbol '$*$', bookended at the left with the 'E' symbol, and the right with '#'. The machine starts at the far right side of the tape with the head positioned over the '#' symbol.

It operates by moving left until it reaches an asterisk. The number to the right of the asterisk is copied to the left hand side of the '#'. Once the number has been copied, it is delimited with a '$' symbol and the process repeats. When the TM encounters the 'E' at the far left of the list, it copies the number to the far right of the new list and halts. The TM halts with the initial list to left of the '#', and the reversed list to the right (Figure 4.14).

4.3.3.2 RASP Machines

A RASP machine to reverse a list is initialised with the program at the beginning of the memory, and the list to be reversed at the end. The machine will finish with a new list appended to the end of the memory. In light of this, it is beneficial to make sure that the machine is initialised with enough free memory to hold a new list without overwriting previous data.

Figure 4.15 shows the RASP machine. The location to start writing the new list is first obtained by loading the address of the end of the list and incrementing twice as to create a gap between the new and old list. The program proceeds by copying the value at the end of the old list to the first value in the new list.

After each copy the old list pointer is compared to the start of the list to see if they are equal. If they are, the machine halts. If not, the new list pointer is incremented, the old list pointer is decremented and another copy is made.

4.3.3.3 λ-calculus and SKI

Reversal of a list in the SKI and λ-calculus recurses through an input list and builds an output list from those elements:

$$\text{REV} \equiv \text{Y}(\lambda g.\lambda a.\lambda l.\text{NULL } l \; a(g(\text{PAIR}(\text{HEAD } l)a)(\text{TAIL } l)))\text{NIL}$$

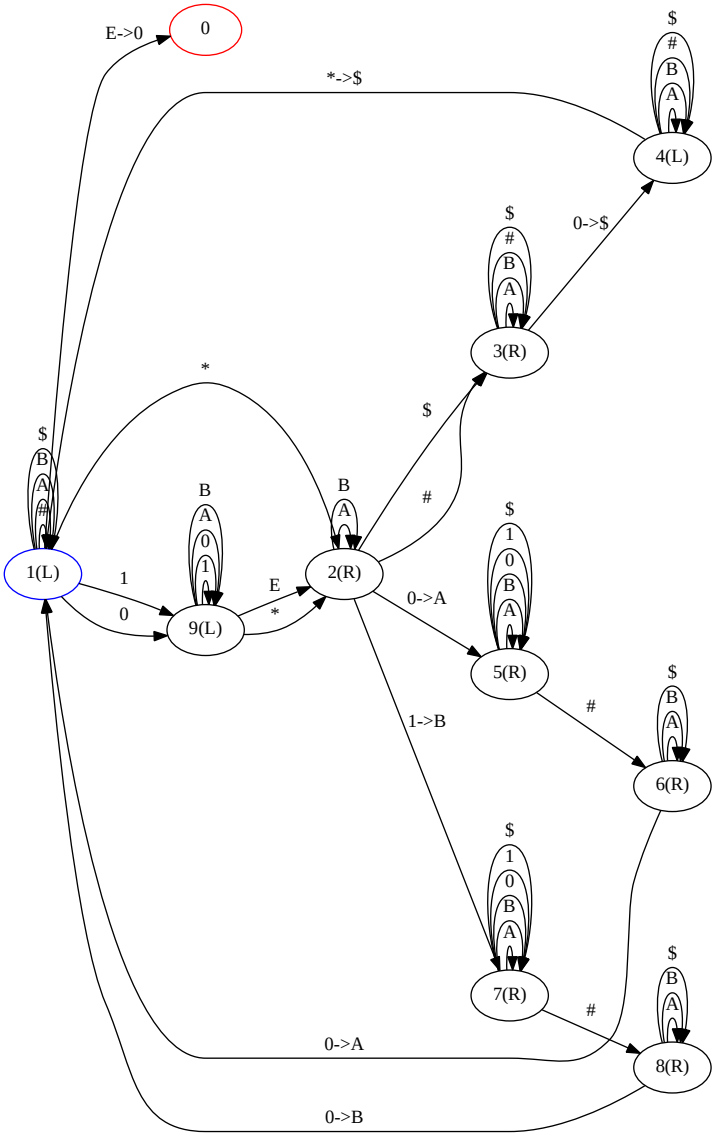


Figure 4.14: TM to reverse a list

Instr	**Data**	**I Label**	**D Label**
LOAD	'listEnd		
STO	'cpyPointer		
INC			
INC			
STO	'writePointer		
LOAD	0	:main	;writePointer
STO	'writeSTO		
LOAD	0		;cpyPointer
STO	'cpyLOC		
CPY	0		;cpyLOC
STO	0		;writeSTO
CPY	'writePointer		
INC			
STO	'writePointer		
CPY	'cpyPointer		
STO	'tmp1		
LOAD	'listStart		
STO	'tmp2		
LOAD	0	:loop	;tmp1
DEC			
STO	'tmp1		
LOAD	0		;tmp2
DEC			
STO	'tmp2		
JGZ	'loop		
CPY	'tmp1		
JGZ	'decWritePointer		
HALT			
CPY	'cpyPointer	:decWritePointer	
DEC			
STO	'cpyPointer		
JGZ	'main		
0		:listStart	
10		:listEnd	

Figure 4.15: The RASP machine to reverse a list by creating a new list.

If the input list in not NIL, the expression makes a recursive call with the tail of the input list and a pair of the head of the input list with the current construction of the output list. The NIL term at the end of the expression is the initial output list which gets paired up with the elements of the input list. Once the expression finds the NIL term at the end of the input list, it returns the currently constructed output list.

4.3.4 Statefully Reversing a List

Statefully reversing a list mutates the input list by swapping the elements, rather than recursively traversing the input list to create a new one as above. We maintain two pointers to the list, x and y, initialised to the first and last elements. At each step, if $x < y$ then the elements are swapped, and x is incremented while y is decremented.

$$
\begin{aligned}
stateRev(list) &= stateRevh(list, 0, p(length(list))) \\
stateRevh(list, x, y) &= \begin{cases} stateRevh(swap(tail(list), xy), s(x), p(y)) & : x < y \\ list & : x \geq y \end{cases} \\
length(l) &= \begin{cases} 0 & : length([]) \\ s(length(tail(l)) & : \text{otherwise} \end{cases} \\
nth(x, l) &= \begin{cases} head(l) & : nth(0, l) \\ nth(p(x), l) & : \text{otherwise} \end{cases} \\
swap(x, y, l) &= substitute(x, nth(i, l); substitute(i, nth(x, l), l)) \\
substitute(x, i, l) &= \begin{cases} pair(i, tail(l)) & : x = 0 \\ pair(head(l), substitute(p(x), i, tail(l))) & : \text{otherwise} \end{cases}
\end{aligned}
$$

4.3.4.1 TM

The TM tape to reverse a list statefully is an 'E' bounded, '$*$' delimited list of binary numbers:

$$E * \langle data1 \rangle * \langle data2 * data3 * \ldots E$$

The machine operates by copying the first element to empty space at the far right of the tape. The head then moves to the right hand side and finds the first number which has not been moved. It copies this number into the previously

vacated space and then moves the first number into the newly vacated space.

If there are an odd number of elements in the list, upon encountering the final element it copies the contents to the far right. It will then detect that there is no matching element to replace the first element with, and so it copies the value back to its original place before halting.

4.3.4.2 RASP Machines

The RASP machine to statefully reverse a list maintains two pointers. One is initialised to the first element of the list, and the other is initialised to the last element. The program proceeds by switching the two elements, incrementing the first pointer, and decrementing the second one.

After this, the machine compares the two pointers. If the front pointer is a memory address lower than the rear, it loops again to swap the next pair of elements. If the value of the front pointer is greater than or equal to the rear, then the two pointers are either pointing at the same element, or have crossed. In either of these cases, the machine halts.

4.3.4.3 λ-calculus and SKI

The stateful reverse is a complicated operation which the λ-calculus and SKI are not at all suited to:

$$\lambda x.(Y(\lambda a.\lambda b.\lambda c.\lambda d.\text{LT } b\ c(a(\text{SUCC } b)(\text{PRED } c)(\text{SWAP } b\ c\ d))d))$$
$$\text{ZERO}(\text{PRED } (\text{LENGTH } x))x$$

where LENGTH obtains the length of a list and SWAP switches the positions of two elements in a list. The expression operates on the list by maintaining pointers to the beginning and end of the list to swap the elements in a pairwise fashion.

It first obtains the length of the list, tests to see if the front pointer is lower than the rear one, and swaps the values if this is the case. It recurses on the list and increments the front pointer, while decrementing the rear one.

This proceeds until the front pointer is greater than or equal to the rear pointer, signifying that they are either pointing to the same element (the list has an odd number of elements) or that they have crossed each other (the list has

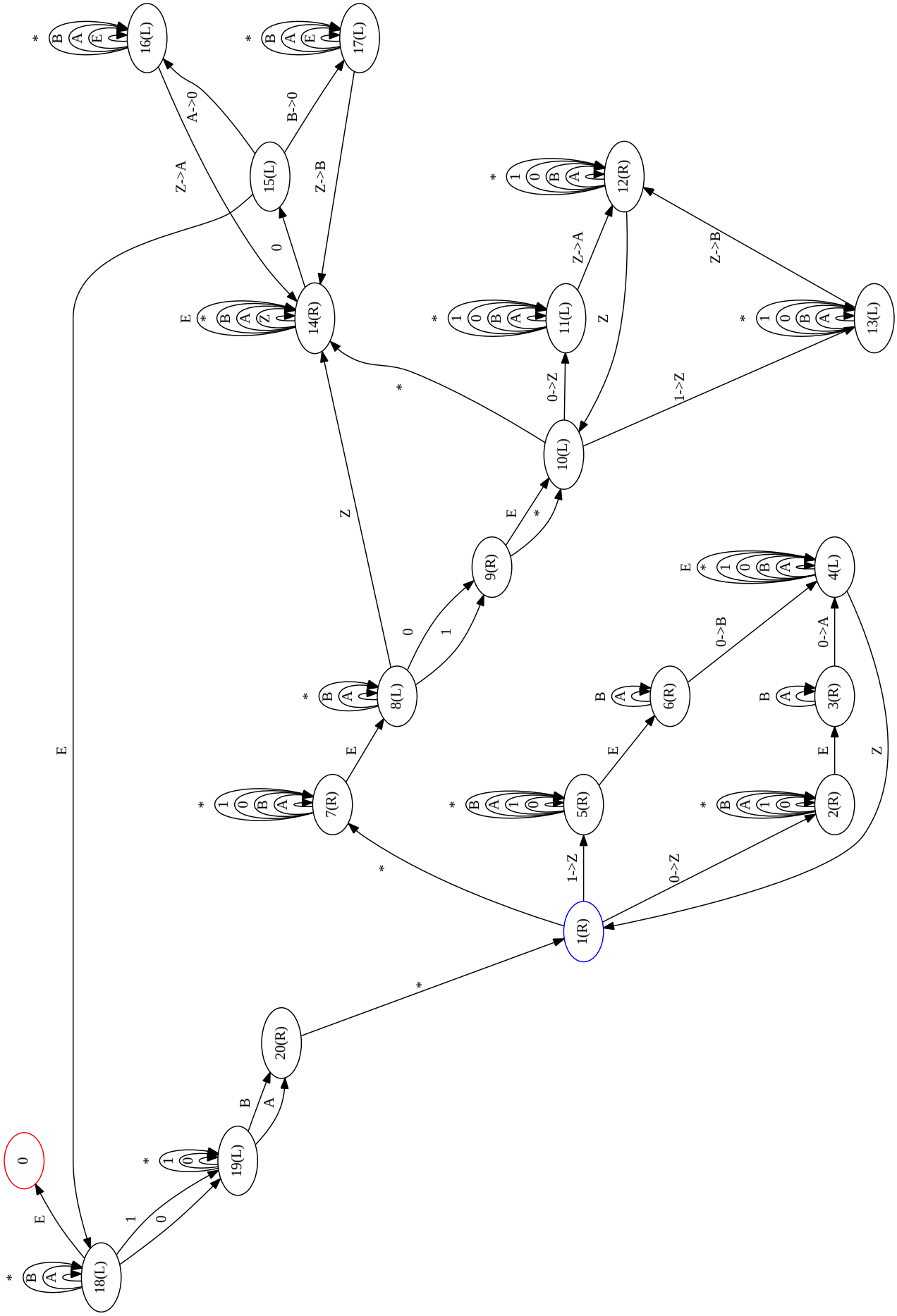


Figure 4.16: Stateful reversal TM

an even number of elements). The function halts with the list and its reversed elements.

The complexity of the stateful reverse is mostly in the SWAP and SUBST functions:

$$\begin{aligned}\text{SWAP} &\equiv \lambda a.\lambda b.\lambda c.\text{SUBST } a(\text{NTH } bc)(\text{SUBST } b(\text{NTH } bc)c)\\ \text{SUBST} &\equiv Y(\lambda a.\lambda b.\lambda c.\lambda d.\text{ISZERO } b(\text{PAIR } c(\text{TAIL } d))\\ &\qquad (\text{PAIR}(\text{HEAD}d)(a(\text{PRED } b)c(\text{TAIL } d))))\end{aligned}$$

SUBST recurses through the list until it finds the location it requires, it then substitutes the current list member with the new list member. SWAP applies SUBST twice to the list to swap both members of the list.

4.3.5 Bubble Sort

The bubble sort algorithm commences by comparing the value at the start of the list v with its neighbour on the right n. If the value is greater than its neighbour, the two values are swapped. It continues by comparing v to its new neighbour n_1, swapping as appropriate until it reaches the end of the list, or a neighbour is greater than v.

Once a value has been 'bubbled' to its appropriate position, the algorithm goes back to the start of the list and bubbles up another value. If the algorithm compares each value to its neighbours without making a swap, the list is sorted and the program terminates.

$$sort(list) = sorth(list, false, 0, 1)$$

$$sorth(l, f, x, y) = \begin{cases} sorth(swap(l, x, y), T, s(x), s(y)) & : y \leq p(len(l)) \\ & \wedge nth(x, l) > nth(y, l) \\ sorth(l, f, s(x), s(y)) & : y \leq p(len(l)) \\ sorth(l, F, 0, 1) & : y > p(len(l)) \wedge f = T \\ l & : y > p(len(l)) \wedge f = F \end{cases}$$

4.3.5.1 TM

The tape of the TM to perform the bubble sort is again a '∗' delimited list of binary numbers. The tape is bookended on the left and right using '#' symbols. The machine first marks the left hand delimiter of the element which is being bubbled. It then compares the current numeral to the one on its right. If the current numeral is greater than its neighbour the machine swaps the numerals in the style of the stateful reverse. The marker is moved one element to the right and the cycle repeats.

If an element is not greater than its neighbour, it is in position and the machine skips over the element to sort its neighbour to the right. If an element being considered is at the far right of the list, the machine traverses to the far left of the list to restart the process. A single symbol past the left hand marker of the tape indicates whether a swap has been made in each left-to-right transversal. If the machine completes a full left to right traversal without a swap being made, the list is sorted and the machine halts.

4.3.5.2 RASP Machines

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer1		
ADD 1			
STO	'pointer2		
LOAD	0		
STO	'flag		
LOAD	0	:cmpPointers	;pointer1
STO	'p1ref		
CPY	0		;p1ref
STO	'cmp1		
LOAD	0		;pointer2
STO	'p2ref		
CPY	0		;p2ref

Instr	**Data**	**I Label**	**D Label**
STO	'cmp2		
LOAD	'incPointers		
STO	'cmpOther		
STO	'equal1		
LOAD	'swap		
STO	'cmp1Greater		
LOAD	0	:cmpStart	;cmp2
SUB	1		
STO	'cmp2		
JGZ	'cmp1dec		
CPY	'cmp1		
SUB	1		
JGZ	0		;cmp1Greater
LOAD	1		
JGZ	0		;equal1
LOAD	0	:cmp1dec	;cmp1
SUB	1		
STO	'cmp1		
JGZ	'cmpStart		
LOAD	1		
JGZ	0		;cmpOther
CPY	'pointer1	:incPointers	
ADD	1		
STO	'pointer1		
CPY	'pointer2		
STO	'p2sub		
LOAD	'listend		
SUB	0		;p2sub
JGZ	'returnToInc		
LOAD	0		;flag
JGZ	'start		

Instr	Data	I Label	D Label
HALT			
CPY	'pointer2	:returnToInc	
ADD	1		
STO	'pointer2		
JGZ	'cmpPointers		
CPY	'pointer2	:swap	
STO	'p2SwpRef		
STO	'p2WriteRef		
CPY	0		;p2SwpRef
STO	'swp		
CPY	'pointer1		
STO	'p1SwpRef		
STO	'p1WriteRef		
CPY	0		;p1SwpRef
STO	0		;p2WriteRef
LOAD	0		;swp
STO	0		;p1WriteRef
LOAD	1		
STO	'flag		
JGZ	'incPointers		
7		:listStart	
3		:listend	

Table 4.1: The RASP2 bubble sort

The RASP machines maintain two pointers: v and $n = v + 1$. The pointer v is initialised to the start of the list, and n is the next element.The machine compares the value in register v with the value in n. If $M[v]$ is greater than $M[n]$, the machine swaps the values and switches a flag to indicate that a swap has been made.

Both pointers are incremented, and the swaps continue until n is pointing to the last element in the list. At this point the machine checks to see if a swap has

been made in this transversal. If a swap has not been made, the machine halts with a sorted list.

If a swap has occurred, the machine resets v, n and the flag to their initial values and loops until it can traverse the list without making a swap. Table 4.1 shows the RASP2 implementation.

4.3.5.3 λ-calculus and SKI

The bubble sort expression in the λ-calculus and SKI is:

$Y(\lambda a.\lambda b.\lambda c.\lambda d.\lambda e.$LEQ d(PRED(LEN e))(LT(NTH d e)(NTH c e)(a TRUE
(SUCC c)(SUCC d)(($\lambda a.\lambda b.\lambda c.$SUBST a(NTH b c)(SUBST b(NTH a c)c))c d e))
(a b(SUCC c)(SUCC d)e))(b(a FALSE ZERO ONE e)e))FALSE ZERO ONE

The five parameters to this expression are: the expression itself for recursive calls (a), the swap flag (b), the pointer v (c), the pointer $n = v + 1$ (d), and the list to be sorted (e). If n is less than the predecessor of the length of the list (recalling that these lists are terminated with a NIL element), the elements at positions v and n are compared. If a swap is required, the elements are swapped and a recursive call is made with incremented pointers and the swap variable as TRUE.

If n points at the end of the list and there has been a swap ($b \equiv$ TRUE), a recursive call is made with the pointers reset and the swap variable as FALSE: (b(a FALSE ZERO ONE e)e). Otherwise, the current (sorted) list is returned. Elements are swapped via the SUBST expression explained previously.

4.4 Universal Machines

This thesis considers only the "direct simulation" machines. These are machines that actually simulate machines in some suitable encoding. For example, there are numerous choices for which UTM to use. Neary [65] has demonstrated direct simulation machines of: (3,11) which is 3 states and 11 tuples with 32 tuples, (6,6) with 32 tuples, (5,7) with 33 tuples, (7,5) with 33 tuples, and (8,4) with 30 tuples. The obvious choice for a concise UTM is the (8,4) machine, but the

encoding of the input is very convoluted (Section 6.6), thus the intentionality of Neary's machine does not match well with the intentionality of the UTM realisations in the RASPs, λ-calculus and SKI. So another machine is considered which is both a direct simulation UTM, and has a more natural input encoding.

4.4.1 Universal Turing Machines

4.4.1.1 TM

The UTM adopted is the direct simulation TM from Minsky [63]. The initial tape of the UTM is arranged as $[w][st_1][sy][M]$ which is a right unbounded tape, with the current state, the current symbol under the head, and the symbol table following respectively. The symbol table is arranged in quintuples of $st_x, sy_x, st_y, sy_y, D$.The states are binary numbers, symbols are either 1 or 0, and the direction D is either 0 or 1 to indicate a left or right shift.

The symbol table is terminated with the symbol Y, and the tape is of the form:

$$\ldots 00000M000Y\langle st_1\rangle\langle sy \text{ under } M\rangle X\langle st_1, sy_1, st_p, sy_p, D\rangle X \ldots X \ldots Y0$$

The symbol M on the tape is the simulated head of the machine. The space between the first Y and the first X from the left contains the current state and symbol pair which is used to search the symbol table for the correct tuple. The algorithm of the machine operates by searching the start of each tuple in the symbol table for the state and symbol combination held between the first 'Y' and 'X' from the left. This is a search to find the tuple which corresponds to the current state and current symbol. If a tuple matching these is not found, then the machine halts.

Once a matching tuple has been found, the new state is copied into the space between 'Y' and 'X', the simulated tape head is replaced by the new symbol, the head is moved left or right, and the new current symbol is printed next to the new current state. Figure 4.17 shows the TM.

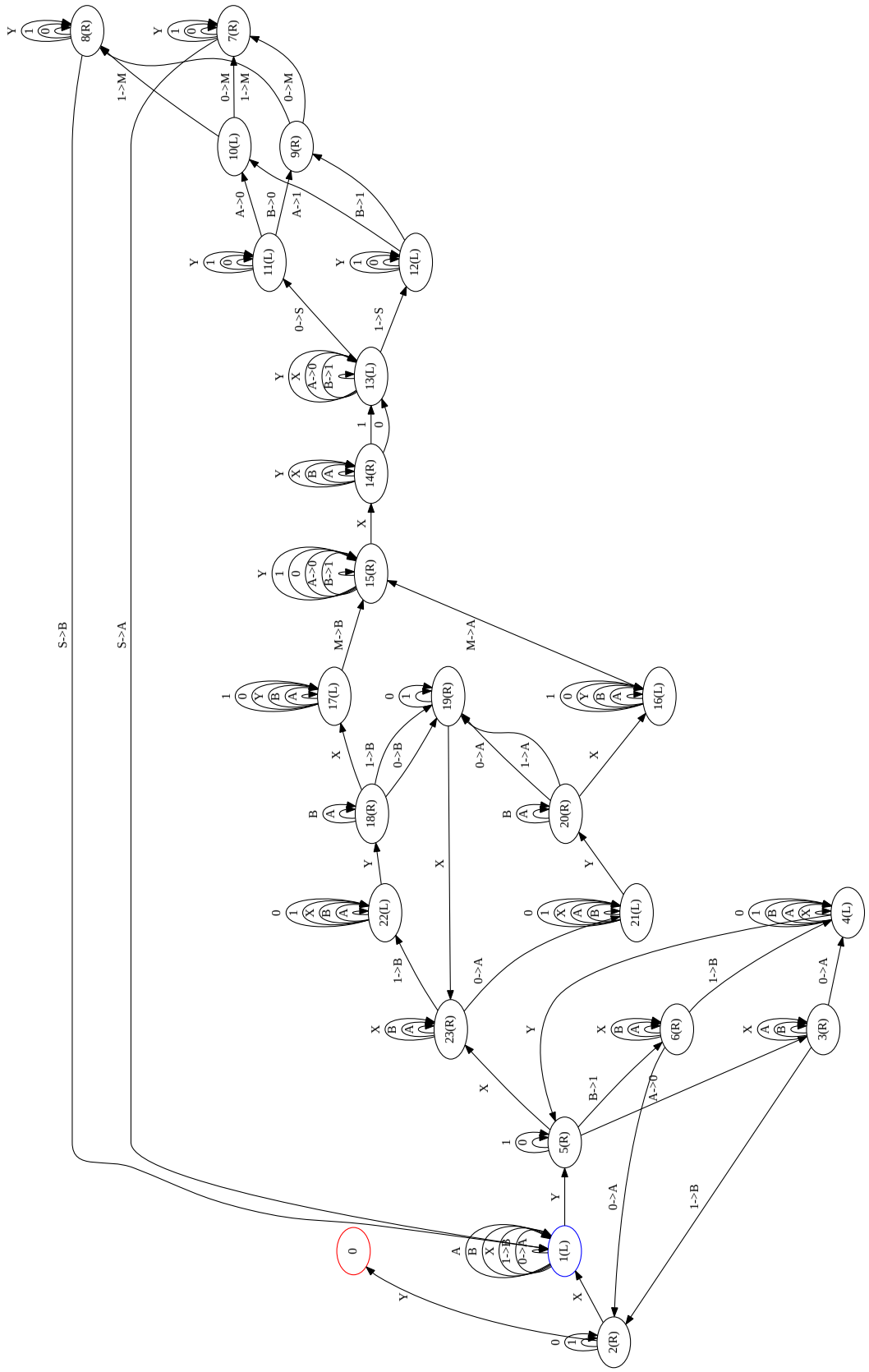


Figure 4.17: Minsky's UTM

4.4.1.2 RASPs

The TM simulator in RASP simulates an arbitrary (m, n) Turing machine, subject to the limitations of the size of the RASP memory. The machine is organised with the program at the start of the memory, followed by the symbol table of the machine, and finally a right-infinite tape structure at the end. The machine maintains variables such as the current head position and current state. The initial head position is defined as the far left of the tape and the initial state is 1.

The symbol table format for the RASP is of the form:

$$\ldots, \langle S_o \rangle, \langle Sy_o \rangle, \langle S_n \rangle, \langle Sy_n \rangle, \langle D \rangle, \langle S_i \rangle, \ldots, 0 \ldots$$

which is the state and symbol read, followed by the new state, new symbol and direction. The final tuple in the table is followed by a single zero. The tape of the TM then extends from the end of the symbol table to the end of the memory. The machine maintains a label to the start of the tape, and a variable of where the read/write head is.

Evaluation of a TM symbol table and tape, copies the current state and symbol under the head to a searching routine. This routine traverses the symbol table linearly until either both the symbol and state are found, of the end of the table is reached.

If the end of the table is reached, the machine halts, otherwise it replaces the current state with the new state, writes the new symbol to the tape over the old symbol, and either increases the head position variable for a right shift, or decreases it for a left shift.

Searching for the correct tuple in the symbol table involves using an equality function to test that the current state and current symbol are equal to the tuple state and symbol. If they are, variables for the new state, new symbol, direction, and search success are written to and the search jumps back to the main loop.

If the state or symbol do not match the current tuple, the machine either adds 5 or 4 respectively to find the next tuple in the table. If the machine tries to compare the current state to zero then it has reached the end of the symbol table and halts.

The machine executes the simulated TM until its current state and symbol pair is not in the symbol table, or it transits to state 0.

4.4.1.3 λ-calculus and SKI

The TM tape is a list of numbers, each number represents a symbol. The symbol table is a list of 5-tuples in the form $st_x, sy_x, st_y, sy_y, D$. This is a list of 5 element lists:

$$\begin{aligned}
\text{TAPE} &\equiv (\text{PAIR ONE}(\text{PAIR ONE}(\text{PAIR ZERO}(\text{PAIR ONE}(\ldots \text{NIL})))))\\
\text{SYTABLE} &\equiv \text{PAIR}(\text{PAIR ONE}(\text{PAIR ZERO}(\text{PAIR ONE}(\text{PAIR ONE}\\
&\quad (\text{PAIR ONE NIL})))))(\ldots \text{NIL})(\text{PAIR}(\text{PAIR}\ldots \text{NIL})))
\end{aligned}$$

The term to evaluate a TM symbol table and tape requires four parameter; the current state, the current head position, the symbol table and the tape:

$$\begin{aligned}
&\text{Y}(\lambda a.\lambda s.\lambda h.\lambda ta.\lambda tp.\text{NULL}(\text{TABLES } s(\text{NTH } h\ tp)ta)tp\\
&(a(\text{HEAD}(\text{TABLES } s(\text{NTH } h\ tp)ta))(\text{ISZERO}(\text{HEAD}(\text{TAIL}(\text{TAIL}\\
&(\text{TABLES } s(\text{NTH } h\ tp)ta))))(\text{PRED } h)(\text{SUCC } h))ta(\text{SUBST } h(\text{HEAD}\\
&(\text{TAIL}(\text{TABLES } s(\text{NTH } h\ tp)ta)))tp)))
\end{aligned}$$

A search is performed on the symbol table for the current state and current symbol (extracted from the element at the head position of the tape) pair. Failure to find this pair results in the return of the tape as evaluation ends.

Once the tuple to match the current state and symbol have been found, a recursive call is made where the current state is replaced, the tape at element h is replaced with the new symbol, and the head position is either decremented if the fifth element of the tuple is ZERO, and incremented otherwise. The function to search through the table is:

$$\begin{aligned}
\text{TABLES} &\equiv \text{Y}(\lambda a.\lambda st.\lambda sy.\lambda tab.\text{NOT}(\text{NULL } tab)(\text{AND}\\
&\quad (\text{EQ } st(\text{HEAD}(\text{HEAD } tab)))(\text{EQ } sy(\text{HEAD}(\text{TAIL}(\text{HEAD } tab))))\\
&\quad (\text{TAIL}(\text{TAIL}(\text{HEAD } tab)))(a\ st\ sy(tailtab)))\text{NIL})
\end{aligned}$$

This expression searches the table by testing the passed in state and symbol against the first two elements of the current tuple. If these match, a triple of the

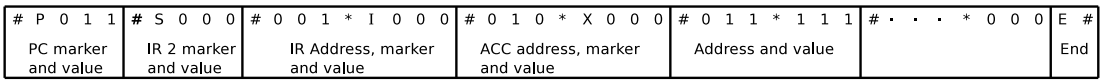


Figure 4.18: A 3 bit RASP arranged on a TM tape.

next state, symbol and direction is returned. No match prompts a recursive call with the tail of the table. If the function does not find a matching tuple in the table, it returns NIL which prompts the expression to halt.

4.4.2 Universal RASP Machines

The Universal RASP (URASP) can simulate an arbitrary RASP machine. As with the UTM, all of the universal RASPs are direct simulation machines.

4.4.2.1 URASP in TM

Consider a 3 bit RASP machine. The machine is initially expressed on a TM tape as depicted in figure 4.18. The memory of the machine is bounded by the PC marker (#P) at the far left and the end marker (E#) at the far right. There are also four letters which mark the three usual registers (P,I, and X) in the machine and the one secondary IR (S).

With the exception of the P and S registers, the memory of the machine is laid in (address, data) pairs: $\#\langle address\rangle * \langle data\rangle\#$. For the IR and ACC, there are the characters 'I' and 'X' which act as markers to reduce the required number of states in the machine. Both address and data are expressed as little endian binary numbers.

Algorithm 2 shows the how the TM operates the fetch-execute cycle. The machine starts with the head positioned on the second # from the left (bold in the above diagram). From there, it attempts to pattern match the value in the PC (011) with the addresses in the machine. If it succeeds, the corresponding data value is copied into the first and second 'I' and 'S' instruction registers. If the pattern matching fails, then the PC must be pointing at itself and therefore the 'P block' is copied to the 'S' and 'I' blocks.

Once the copy has been made, the RHS bit of 'S' is tested. There are four instructions which take a parameter and four that do not.

```
while not halted do
    Find address in P;
    if address not found then
        Copy P to S;
    end
    else
        Copy data in P's address to S;
    end
    Copy data in S to I;
    if Least significant bit of S is 1 then
        Increment P;
        Find address in P;
        if address not found then
            Copy P to I;
        end
        else
            Copy data in P's address to I;
        end
    end
    Decode and Execute S;
end
```

Algorithm 2: The Fetch Execute cycle of the RASP in TM

- 000: OUT
- 001: LOAD
- 010: HALT
- 011: STO
- 100: INC
- 101: JGZ
- 110: DEC
- 111: CPY

If the least significant bit is a 0, the rest of the instruction is decoded and executed. If the first bit is a 1, the PC is incremented and another search happens. Once this is done, the data is copied to the 'I block' only. The instruction is decoded from the value in 'S' and executed. These instructions affect the memory layout of the machine to the degrees described in Section 2.3.1.2.

There are several repeated functions in the operation of the fetch execute cycle. Finding addresses, copying data from one register to another, and housekeeping operations like resetting the tape can be performed more than once per cycle. To facilitate reuse of such functions, each time the TM performs a task in Algorithm 2 it enters a switching state which prints or reads a symbol immediately to the left of '#P'. The symbol informs the machine which task it is to complete next in the fetch-execute cycle.

All of the RASP instructions, except for OUT make changes which affect only

Instr	Data	I Label
DEC		:IncrementInstruction
JGZ	'DecrementInstruction	
CPY	'ACC_P	
INC		
STO	'x	
...	...	...
DEC		:DecrementInstruction
JGZ	'LoadInstruction	
CPY	'ACC_P	
JGZ	'dc	
...	...	...

Figure 4.19: Decoder of the universal RASP

the machine. The TM executes an occurrence of OUT by copying the contents of the 'X' block (ACC) to the far right hand side of the tape, past the 'E#', separating occurrences with a '*'.

4.4.2.2 RASPs

The universal RASP machine simulates the execution of another RASP via performing the fetch-execute cycle. The URASP keeps track of the locations of the simulated PC, IR, and ACC as well as the size of the the machine and an 'offset' which is the memory address of the PC of the simulated machine.

Execution of the fetch execute cycle involves adding the offset to the contents of the PC and using that to copy the contents of the addressed register to the IR. The IR is decoded by repeatedly decrementing the number contained in the simulated IR until it equals zero. After each decrement a test is made for zero and if the number is zero, the corresponding instruction is executed (Figure 4.19). Otherwise the machine decrements and retests. If the IR instruction is zero, or the instruction in the simulated IR is not in the range 0–7, then the machine halts.

Once the correct instruction has been found, the machine uses the offset to enact the effects of the instruction against the memory of the simulated machine as described in Section 2.3.1.2. If the executed instruction is not a HALT, the simulator increments the PC of the simulated machine and jumps back to fetch and execute the next instruction.

The total size of the machine is known to the simulator. When any increments or decrements take place, the simulator checks that the change in the register will not over- or underflow. If it will, the register is set to either zero or the maximum permissible value.

4.4.2.3 λ-calculus and SKI

RASP machines are represented in the λ-calculus and SKI as a pair of; a list of 2^n elements to represent the machine, and an initially empty list to represent the output vector. The element at position x of the machine list holds the contents of register x.

The expression to evaluate the RASP machine is of the form:

$$\mathrm{Y}(\lambda a.\lambda m.\lambda o.\langle\mathrm{INC}\rangle(\langle\mathrm{DEC}\rangle(\langle\mathrm{LOAD}\rangle(\langle\mathrm{STO}\rangle(\ldots(\langle\mathrm{HALT}\rangle)\ldots)))))$$

The sub-expressions compare the numeral in the fetched machine to ONE to SEVEN and execute the relevant instruction according to the numeral in memory. The sub-expressions for the INC, DEC, and LOAD instruction are as follows:

$$\begin{aligned}
\mathrm{INC} &\equiv \mathrm{EQ}(\mathrm{NTH\ ONE}(\mathrm{FET}\ m))\mathrm{ONE}(a(\mathrm{INCA\ ZERO}(\mathrm{INCA\ TWO}(\mathrm{FET}\ m)))o)\\
\mathrm{DEC} &\equiv \mathrm{EQ}(\mathrm{NTH\ ONE}(\mathrm{FET}\ m))\mathrm{TWO}(a(\mathrm{INCA\ ZERO}(\mathrm{DEC}(\mathrm{FET}\ m)))o)\\
\mathrm{LOAD} &\equiv \mathrm{EQ}(\mathrm{NTH\ ONE}(\mathrm{FET}\ m))\mathrm{THREE}(a(\mathrm{INCA\ ZERO}(\mathrm{LOAD}(\mathrm{FET}\ m)))o)
\end{aligned}$$

These are all structurally similar. The FET expression copies the value in the register pointed to by the contents of register zero into register one. It is this value which is decoded via comparison with a suitable numeral. If the numerals are not equal, the simulator compares it with the next numeral in the list, up to seven. A numeral larger than that is not a non-halting instruction, so the simulator will halt by returning a pair of the current machine and the OUT vector.

Once it has been determined which instruction to execute, a recursive call (via the Y combinator and the variable a) is made with the machine which has had a fetch, the instruction, and a PC increment applied to it. The INCA function increments the value of the specified address modulo the machine size. The specific functions for fetching, incrementing and executing RASP instructions

are:

$$
\begin{aligned}
\text{FET} &\equiv \lambda m.\text{SUBST ONE(NTH(NTH ZERO } m)m)m \\
\text{INCA} &\equiv \lambda a.\lambda m.(\text{EQ(PRED(LENGTH } m))(\text{NTH } a\ m)) \\
&\quad (\text{SUBST } a \text{ ZERO } m)(\text{SUBST } a(\text{SUCC(NTH } a\ m))m) \\
\text{DEC} &\equiv \lambda a.\lambda m.(\text{EQ(NTH } a\ m)\text{ZERO})(\text{SUBST } a \\
&\quad (\text{PRED(LENGTH } m))m)(\text{SUBST } a(\text{PRED(NTH } a\ m))m) \\
\text{LOAD} &\equiv \lambda m.\text{SUBST TWO(NTH ONE(FET(INCA} \\
&\quad \text{ZERO } m)))(\text{FET(INCA ZERO } m)) \\
\text{STO} &\equiv \lambda m.\text{SUBST(NTH ONE(FET(INCA ZERO } m))) \\
&\quad (\text{NTH TWO } m)(\text{FET(INCA ZERO } m)) \\
\text{CPY} &\equiv \lambda m.\text{SUBST TWO(NTH(NTH ONE(FET(INCA} \\
&\quad \text{ZERO } m)))(\text{FET(INCA ZERO } m)))(\text{FET(INCA ZERO } m)) \\
\text{OUT} &\equiv \lambda m.\lambda o.(\text{PAIR(NTH TWO } m)o) \\
\text{JGZ} &\equiv \lambda m.(\text{EQ(NTH TWO(FET(INCA ZERO } m))))\text{ZERO}) \\
&\quad (\text{FET(INCA ZERO } m))(\text{DEC ZERO(SUBST ZERO} \\
&\quad (\text{NTH ONE(FET(INCA ZERO } m)))(\text{FET(INCA ZERO } m))))
\end{aligned}
$$

The INCA function increments the value of the specified address modulo the machine size. Passing the expression ZERO as a parameter increments the PC of the machine, and passing TWO increments the ACC.

4.5 Results

Table 4.2 presents the number of characters required to implement the above functions in each model. On first glance, the RASP2 and RASP3 appear to require less characters than the RASP, which requires less than the TM on average. Figure 4.20 plots the information amounts.

The character counts for the imperative models follow a somewhat smooth curve (the equality function notwithstanding) as the perceived complexity of measured functions increases. In contrast, the λ-calculus and SKI character counts exhibit no such curve. Additive functions, where the input numerals are combined together, are much smaller in comparison to the subtractive functions:

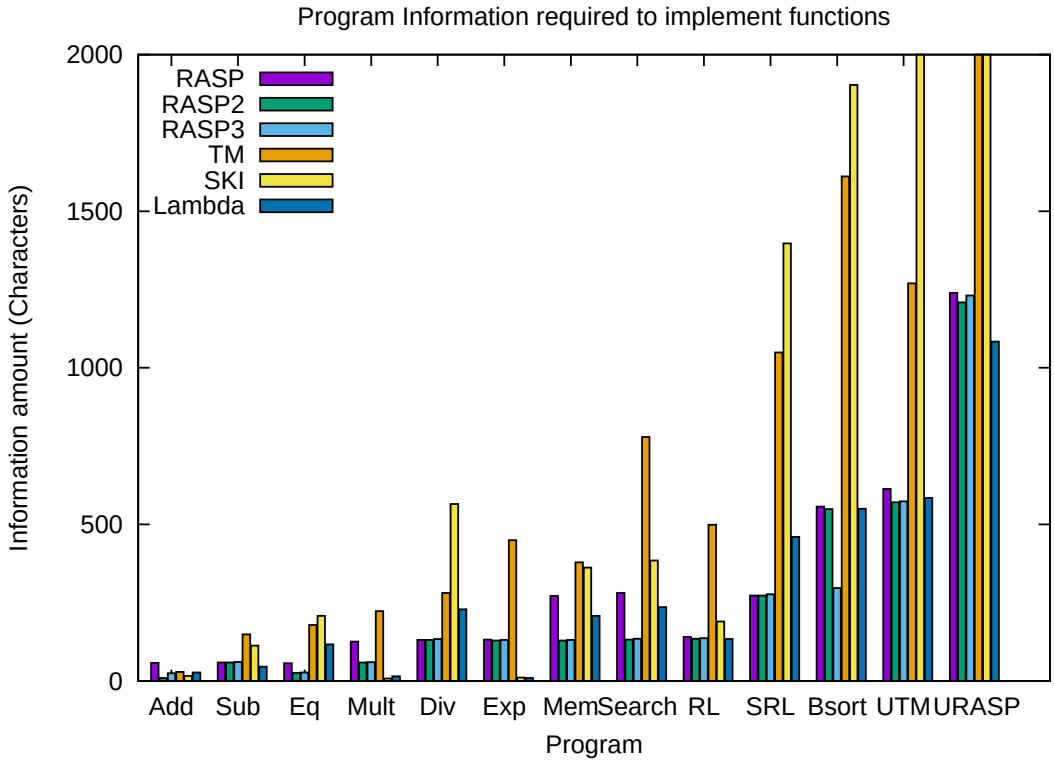


Figure 4.20: Program information to implement the functions

subtraction, equality, and division.

This is due to the higher order functionality of the Church numerals. As functions, the numerals (or parts thereof) can be applied to each other directly to create larger numbers, which is exhibited by the multiplication and exponentiation functions. Subtractive functions operate by recursively decrementing numerals, much like the RASPs. However, decrementing a numeral in the λ-calculus and SKI is a much more program information costly operation than in the RASPs which have defined semantics and this causes a schism between the measurements of the additive functions and those of the subtractive ones.

4.6 Conclusion

This chapter has presented the technical details of the programs which are measured in each of the models. The functions which the programs are written for can be separated into three classes: Arithmetic (Section 4.2), List (Section 4.3), and Universal (Section 4.4). The arithmetic and list functions are primitive recursive and the universal functions are partial recursive (Section 4.1).

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
Addition	58	9	25	29	16	27
Subtraction	59	59	61	149	113	46
Equality	57	26	27	179	208	117
Multiplication	126	59	60	223	8	15
Division	131	131	134	281	565	229
Exponentiation	132	129	131	450	11	9
List Membership	271	129	131	379	362	208
Linear Search	281	132	135	779	385	236
List Reversal	140	135	137	499	190	134
Stateful List Rev	273	273	277	1049	1397	460
Bubble Sort	557	549	297	1611	1903	550
Universal TM	613	571	574	1270	2593	584
Universal RASP	1239	1209	1231	14414	9554	1084
Semantics Size	556	585	587	335	291	515

Table 4.2: Number of characters to implement each program

Most of these explanations of the programs in this chapter have been fairly abstract to facilitate understanding. The measurements in Table 4.2 are taken of the programs in the format described in Section 3.3. The full collection of programs in the formats measured above are presented in Appendix B. Chapter 6 analyses the measurements to confirm or contradict the hypotheses stated in Chapter 3.

One aspect of the programs in this investigation which has not been hitherto discussed is that of functional equivalence. With the exception of the λ-calculus and SKI, assuming that the bracket abstraction algorithm is correct, we cannot be currently assured that the different realisations of each function are all extensionally equivalent. This equivalence is important for any formal assertion of the nature of the relationships.

Such formal statements are not provided in this thesis, and there is no assertion that these programs are equivalent. Deriving such equivalences are high on the list of further work and essential to any effort which seeks to generalise these results. Section 7.3.2 considers how equivalences can be drawn between the programs here via induction over encoding functions.

Chapter 5

Circuit Information

In this chapter, we detail the design and implementation of the RASP and TM on a Field Programmable Gate Array (FPGA). Where the SOS is a "mathematical baseline", the FPGA implementations act as a physical baseline and we can equate the required information by measuring the circuit sizes.

5.1 Infinite Regress

Using operational semantics as a baseline from which to measure the information in our models is an approximation.

When we think of the total information in a system, we consider some axiomatic ideal from from which we build the theorems used to construct models of computation. We can view operational semantics as a baseline axiomatic system.

Taking such a baseline makes the assumption that all of the axioms (the natural numbers, sets, universal and existential quantifiers) are required by every model to some degree. This assumption effectively sets the information content of each model to $a + m$, where a is the information of the axioms and m is the information of the model definition. However, not all of our models use the same axioms.

We implicitly use the natural numbers, set membership, set indirection and logical connectives among others. Some of these are used by all of the models, such as set membership, but some are not. The TM and RASP models implicitly use the natural numbers, but the SKI and λ calculi do not require them. Similarly,

the TM and RASP do not use indirection of subsets, whereas SKI and λ calculus make heavy use of it to graph reduce expressions.

In this thesis, we largely accept that these inaccuracies are inherent in our implementation (much in the same way that we accept that we cannot obtain elegant programs). But we can explore how to mitigate or even eliminate these inaccuracies.

First, we could keep SOS as a baseline and use it to formalise itself. SOS can be thought of as a highly abstract Turing complete programming language, so we could use it to write a universal machine for SOS.

On the surface, this is an attractive proposition. It defines those SOS structures and operations (as mentioned above) which we use implicitly. And we can attribute some value for their information content. This value can be added to the information figures for the models depending on how the models use the operations.

Implementing our SOS baseline in SOS still requires implied information though. It is impossible to use a model of computation A to implement another model B without using some implicit information from A. Adding another model C to implement A merely changes the origin of the implied and undefined information. Rather than it coming from A, it now stems from C.

Using other models to implement C leads to a spiralling infinite regress of implementation where we keep on reimplementing our baseline formalism in the hope that we reduce the amount of implied information. In reality, we are just pushing the origin of the implied information back to the 'first' formalism in the chain.

Gödel built his meta mathematical constructs from pure mathematics [28, 64]. Elgot and Robinson initially specified the RASP using first order logic [23]. We could follow in these examples by building own formalism, constructed from the basic axioms of set theory and logic, to describe our models.

Starting from these axioms, we could systematically define the underlying concepts for each model such as natural numbers and therefore determine the information content of concept. A formalism constructed as such gives us finer control over what information is implied in the definitions of our models. This

then gives us a more accurate account of the total information. Implementing this is high on the list of future work, and is explored further in Section 7.3.5.

This rest of this chapter deals with implementation through reducing the models to a physical baseline. We describe the semantics of our models in the language of FPGA components and connections. These components are subsequently defined by transistors, clocks and small sections of RAM.

5.2 Background

VHSIC Hardware Description Language (VHDL) is a strongly typed hardware description language developed in the 1980s in collaboration with the US Department of Defence as a method of documenting the behaviour of Application Specific Integrated Circuits (ASICs). The language was specified, implemented, and standardised in the period of 1986 to 1988 [1]. As with most languages, it has been expanded and re-standardised over the years, resulting in 5 other versions of the language up to 2008 with VHDL 4.0 [2].

Though originally designed to describe ASICs, VHDL, along with other hardware description languages like Verilog HDL [14], has been adopted as one of the primary tools for specifying the behaviour of FPGAs. Indeed, any language which can accurately encapsulate the operations of a given piece of circuitry can be used for either purpose.

Programmable logic is a small section of the semiconductor market and addresses the need for integrated circuits (ICs) that can be reprogrammed as a requirement or for application where a small number of ICs are needed. Programmable logic is faster than software running on a general purpose machine, but is also much cheaper than designing and fabricating ASICs which often require clean rooms and so forth for production. An FPGA board treads the line between speed and affordability, providing a programmable fabric and often external IO, sometimes with a supplementary general purpose CPU to provide a hardware/-software interconnect. Such devices are known as *System on a Chip* [106].

5.2.1 Architecture and Components

An FPGA is essentially a 'configurable chip'. Rather than converting an HDL specification into something akin to assembly code – as how a regular PC processor would operate – the specification is "synthesised" into a Register Transfer Logic (RTL, [33]) diagram. This diagram expresses the high level HDL logic as an electronics diagram, with components like gates, flip-flops, multiplexers and so forth.

An FPGA is split into blocks and slices (depending on the terminology of the manufacturer). These blocks/slices have transistors arranged in discrete structures (such as the above gates, flip-flops etc). At configuration time, the configuration tool for the board "maps" the RTL gates to a component or set of components in a slice or block, and activates routes between them so that signals can be transferred between these mapped components.

This results in a chip that physically performs the task specified by the HDL and RTL, though it may not necessarily have any resemblance to the schematic, as the components in the FPGA may need to be constructed as the lowest common denominator in order to provide the most usability. For instance a RAM 'block' may be constructed by many flip-flops across multiple blocks/slices rather than having all of the flip-flops physically close together.

5.2.1.1 Zedboard

In this thesis, we use the Zedboard[1], an FPGA board aimed at hobbyists and education. It features the Xilinx Zynq-7000 SoC which sports a Xilinx series 7 programmable logic fabric along with an ARM cortex-A9 processor [106].

The series 7 PL fabric [105] consists of Configurable Logic Blocks (CLBs). Each CLB is split into two slices, where each slice contains 4 look up tables (LUTs), 8 flip flops (FF), 3 multiplexers (MUX), and a 4 bit carry chain which can be combined with other chains to implement arithmetic.

Each LUT in a slice can accept up to six bits to implement arbitrary functions. The LUTs in a slice can be combined using MUXs to produce functions up to 7 and 8 bits wide. LUTs can also be chained with LUTs in other slices to implement

[1] `http://www.zedboard.org`

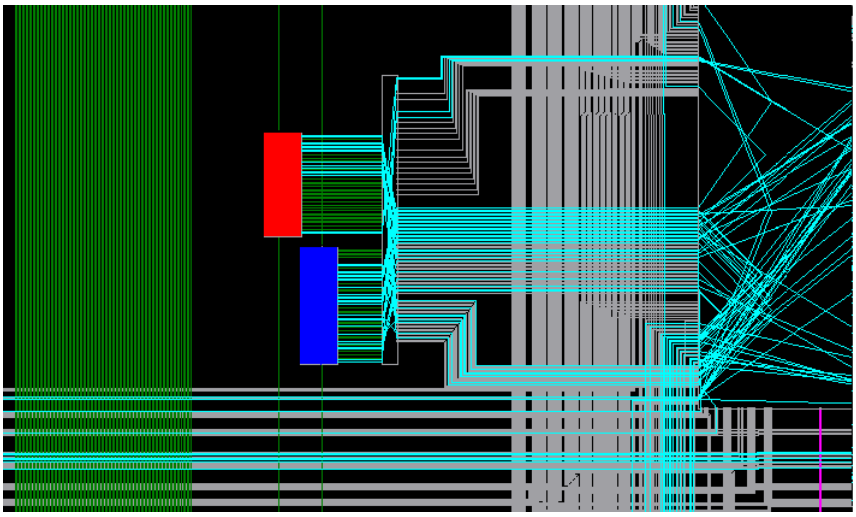


Figure 5.1: A diagrammatic view of two mapped slices (red and blue boxes). The Cyan connections are those which will be used by the FPGA when it executes. Grey are not mapped and green connections are component I/O.

functions with more than 8 bits.

For storage, each slice has 8 elements (collectively known as "slice registers"), The registers can be paired with an LUT to create up to 4 flip flops, with each flip flop able to be either edge or level sensitive. These flip flops can be chained with those in other slices to create larger volatile memories.

A specialised slice type: SLICEM, contains components for distributed memories and shift registers. The distributed memory elements can be combined with LUTs to form a 256 bit RAM element, which can naturally be combined with other slices. The majority of slices on the FPGA are SLICEL, which do not have these types of memory elements.

The FPGA also contains a number of 36K block RAMs. The RAMs can be decomposed into 2×18K, 4×9K, 9×4K and so on down to 72×512B. The Zynq-7020 contains 106,400 slice registers, 53,200 LUTs, and 140 36K block RAMs for a total of 13,300 slices and 6650 CLBs.

5.3 Implementations

Broadly, the TM and RASP in VDHL are both composed of 3 components:

- Control – The state machine and tape read/write/shift mechanics for the TM, and fetch-decode-execute mechanics for the RASP.

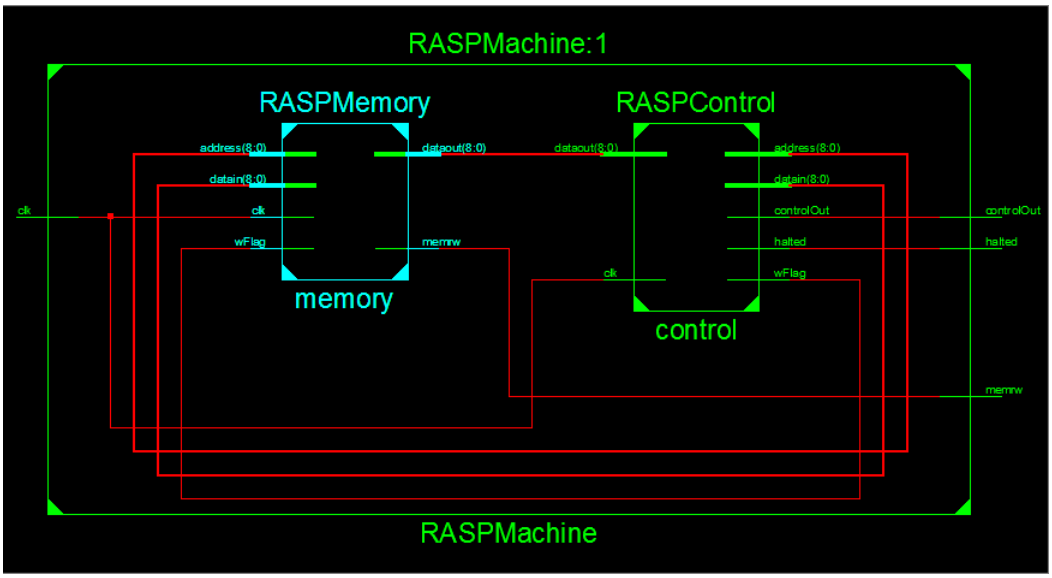


Figure 5.2: The top level RTL schematic of the RASP machine.

- Memory – The tape for the TM, and the RAM for the RASP.
- Machine – Links both the memory and control modules together.

Figures 5.2 and 5.3 are top level RTL diagrams of the RASP and Turing machines. The components are clocked by an oscillator present on the board which coordinates the memory and control components. The control performs some action when the clock ticks up to 1 (also known as *rising edge*) and the memory does something when the clock ticks to 0 (*falling edge*).

The memories for the machines operate in the same manner. They are binary arrays of a fixed size which are written to and read from depending on the flag and access values in the control state. Figure 5.4 shows the RTL schematic, here utilising a block RAM, for the memory component. The TM flavour of the component is very much the same.

Each block in both machines also contain output signals. The memory component has a read/write signal which goes high if the memory is being written to and low if it is read from. The control component both has an output signal (for the OUT command) and a halted signal which goes high once the machine is deemed to have halted. In practice, these signals are wired up to LEDs on the Zedboard.

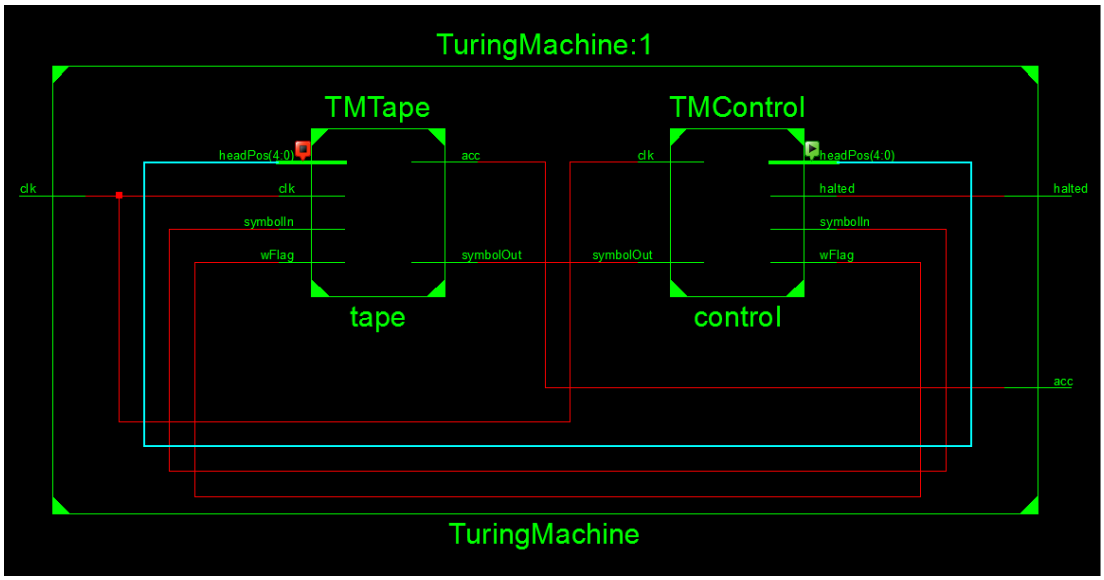


Figure 5.3: The top level RTL schematic of the Turing machine.

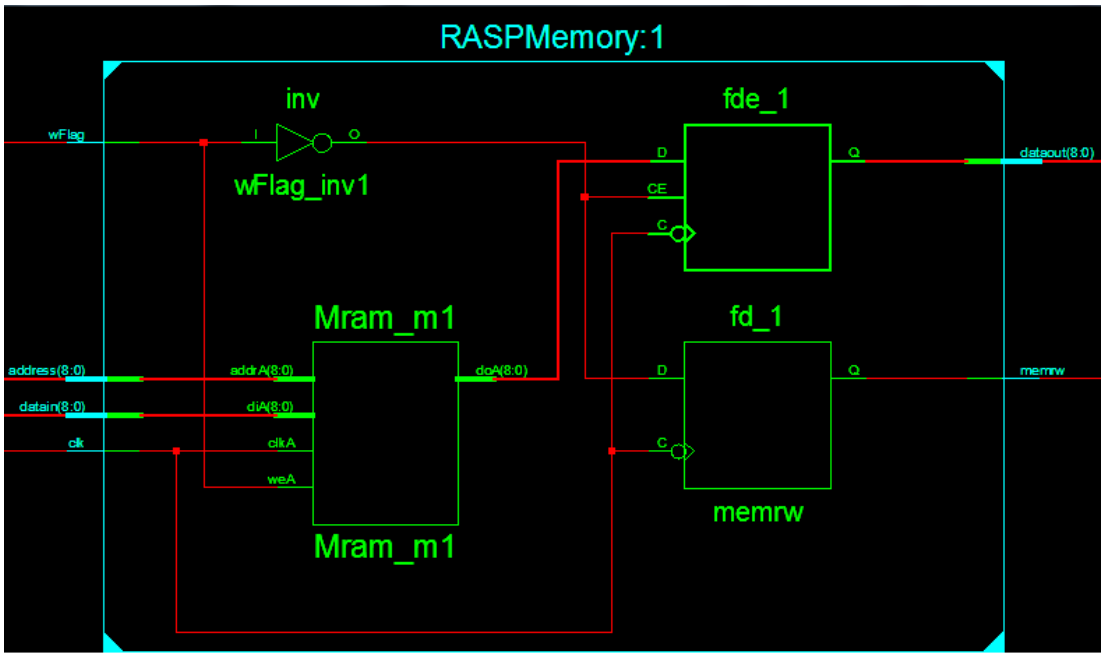


Figure 5.4: The RTL schematic of the RASP memory.

```
if rising_edge(clk) then
  case fetchCounter is                    --Start outer fetch
    ...
   when "010" =>                           --Write S(S(PC)) to S(IR)
     address <= "001";
     datain <= dataout;
     wFlag <= '1';
     currentInstr <= dataout;
     fetchCounter := fetchCounter+1;
   when "011" =>
     case currentInstr is
       when "000" =>                        -- HALT code
         case executeCounter is
           ...
         end case
       when "001" =>                        -- INC code
         case executeCounter is
           ...
         end case
       ...
     end case
    when "101" =>
       ...                                -- Increment PC
     fetchCounter := "000";               -- reset to "000"
  end case
end if
```

Figure 5.5: The VHDL skeleton for the RASP control

5.3.1 RASP

It is in the control component where we see a distinction between models. The control is written as a finite state machine. In the RASP, there is a *fetch counter* and an *execute counter*. Recalling the FE cycle, the fetch counter steps the machine through the reads and writes which move the current instruction in the pointed to memory into the IR. After the execute counter decodes and executes the instruction, the fetch counter increments the PC and resets itself to 0, so that the process can start over in the next clock cycle.

Once an instruction has been fetched into the IR, the execute counter takes over and steps the machine though the actions required to successfully execute the current instruction. Once the instruction has been executed, the execute counter increments the instruction counter and resets itself.

Figure 5.5 shows the truncated code of the state machine of the RASP and Figure 5.6 depicts the gates of the RASP control in their entirety.

5.3.2 TM

In contrast with the RASP, the Turing machine uses a single counter to read the tape, search the symbol table, and write the new symbol to the tape. As with the RASP simulation of the TM, the symbol table search is more information intensive than the TM SOS would suggest. Figure 5.7 shows the controlling state machine for the TM. There exists VHDL primitives for looping over finite data structures which are used in the search function.

Figure 5.8 shows the RTL diagram for the addition TM. The area surrounded by the dark blue square is mainly state information which informs the control what should be done. Additionally the controller for the block outputs are contained here. The cyan lines are the output of the flip flop which holds the counter.

The symbol table for the TM is packed into the control component as ROM. This is reflected in the RTL by the pattern and connections of AND gates, XOR gates and MUXes (yellow box in Figure 5.8). These pathways are activated when the control needs to read from the symbol table.

Since AND and XOR gates do not actually exist on the FPGA, there is a disconnect between the logical (RTL) mapping and the physical (technological) mapping performed by the VHDL compiler. Since we desire that the minimal amount of area is used, the FPGA mapping algorithm endeavours to reduce the number of utilised LUTs as much as possible. It therefore packs the symbol table into another RAM block configured for read only behaviour.

The rest of the logic in the technology schematic is implemented by LUT+FF pairings. Figure 5.9 shows a small section of the technology schematic for the addition TM. Both RAM18 blocks for the tape and symbol table are present and we can see a handful of the LUT and FFs utilised in the implementation.

Figure 5.6: The RTL schematic of the RASP control. The memory schematic is in the top left for scale.

```
if rising_edge(clk) then
 case counter is
   when "000" =>
     ...                  -- Retrieve Symbol
   when "001" =>
     ...                  -- If the state is 0, stop
   when "010" =>
     for i in symbolTable'RANGE loop
       if symbolTable(i).stateR = currentState and
                symbolTable(i).symbolR = symbolOut then
         ...              -- Loop over symbol table
         ...              -- for state/symbol pair
       end if;
     end loop;
     counter <= counter +1;
   when "011" =>
     if found = '1' then
       ...      -- Write new symbol to tape
     else
       ...      -- Set state to 0
     end if;
   when "100" =>
     wFlag <= '0';
     if(symbolTable(var).dir = '1') then
       hPos <= hPos + 1;          -- Right
    else
       hPos <= hPos - 1;          -- Left
    end if;
      counter <= "000";
   when others =>
end case;
```

Figure 5.7: The VHDL skeleton for the TM control.

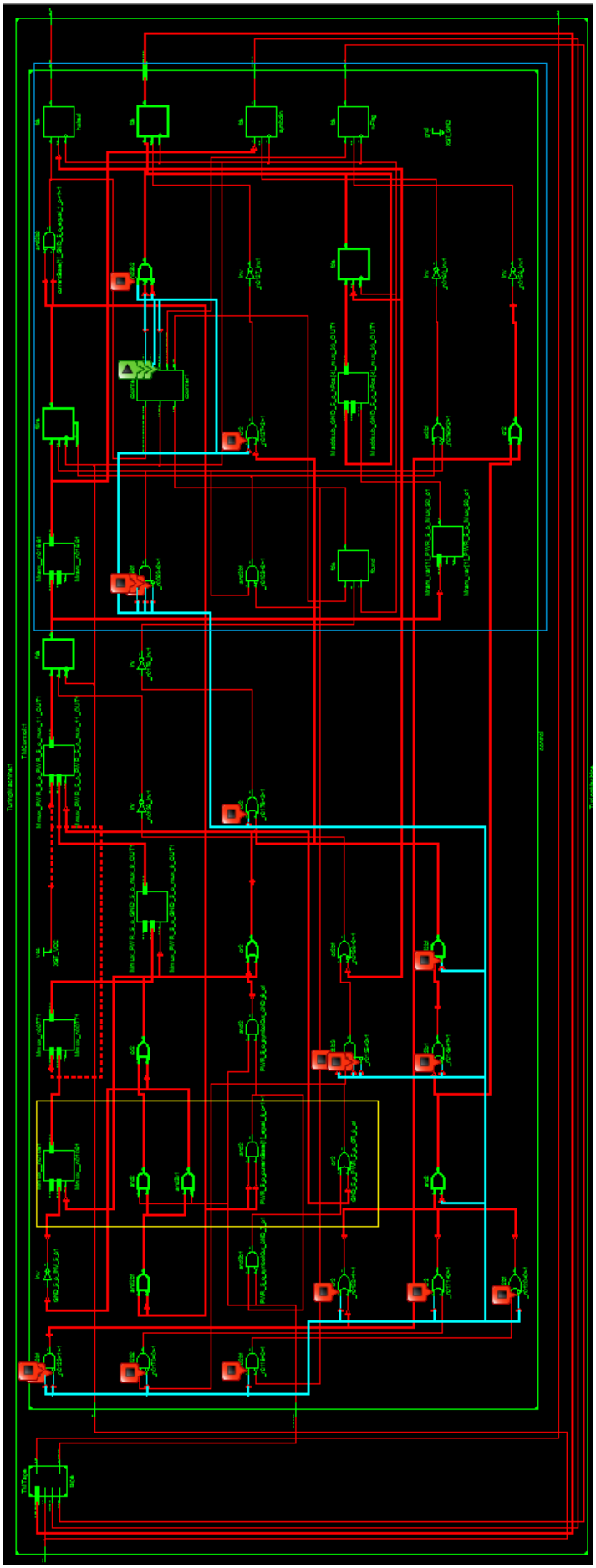


Figure 5.8: The RTL schematic of the TM control. The dark blue square contains the typically constant structures of the TM. The cyan connections are the output of the counter flip flop.

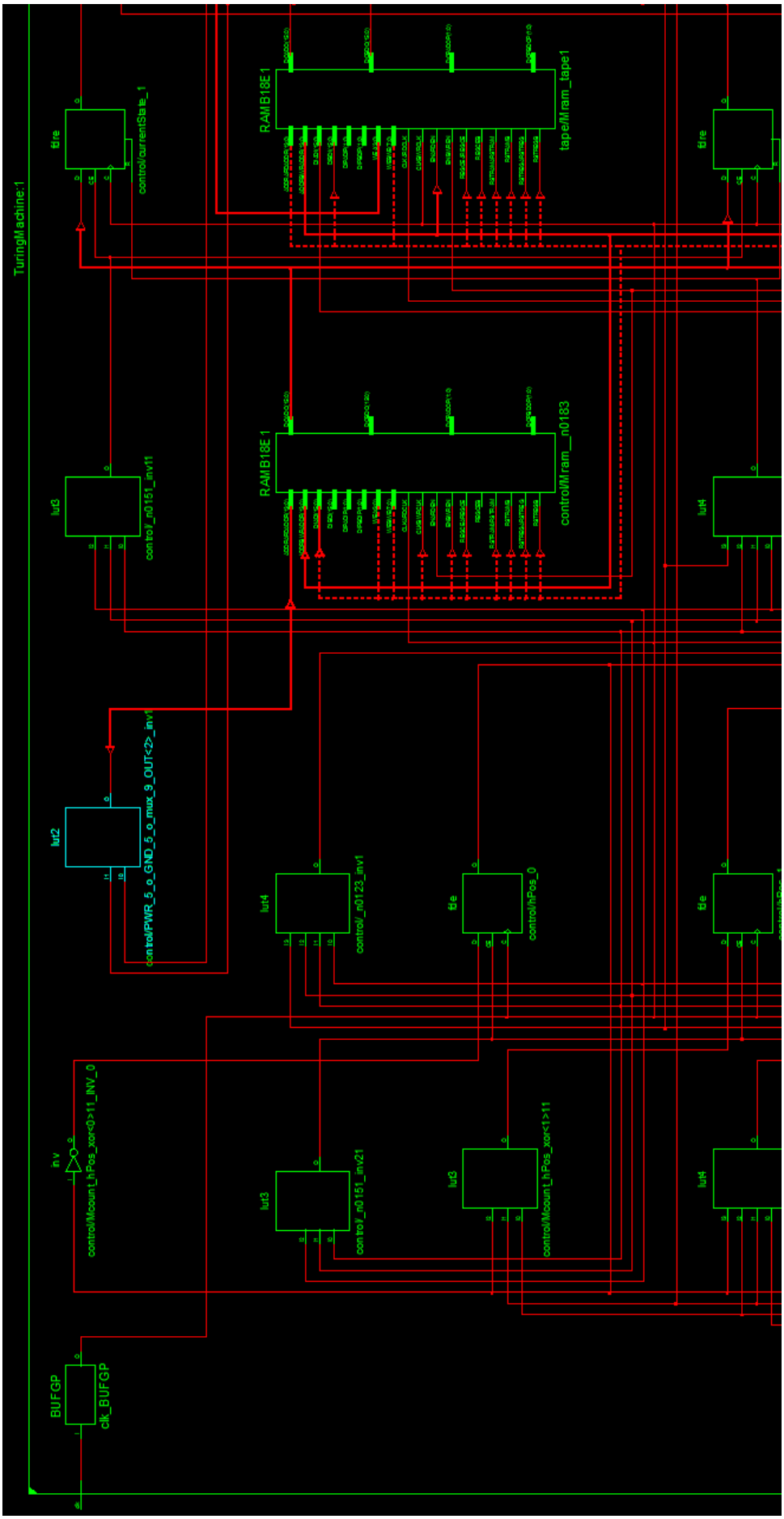


Figure 5.9: A part of the technology schematic of the addition TM with an input tape.

	Slice Reg	LUTs	FFs	RAMB18
Addition	28	66	28	3
Subtraction	28	66	28	3
Equality	28	66	28	3
Multiplication	32	74	32	3
Division	32	74	32	3
Exponentiation	32	74	32	3
List Membership	37	81	37	2
Linear Search	37	81	37	2
Reverse List	32	74	32	3
Stateful Rev List	37	81	37	2
Bubble Sort	41	90	41	2
Universal TM	41	89	41	2
Universal RASP	46	92	45	2

Table 5.1: Components for RASP implementations

5.4 Results

Each program for the RASPs and TM were translated into VHDL (Appendix C), compiled and mapped to the Zedboard. The compiler option specified a minimal area strategy, with maximal logic optimisation and compression. This strategy attempts to minimise the amount of LUTs required to implement the logic of the machines, sometimes preferring to pack logic into block RAMs.

This compilation was made from a 'program only' perspective, therefore the tape for the TM was minimal in size (1 cell). Complicated inputs for the RASP (lists) were also truncated and the number of bits selected so that the entirety of the program fits in memory, excluding any inputs. The VHDL programs described in this chapter which produce the data here are shown in full in Appendix C.

Tables 5.1, 5.2, 5.3, and 5.4 show the raw figures and geometric means of the mapping results. We analyse this data with respect to the SOS and program counts in Chapter 6, but we briefly comment on the data here.

We first notice that the figures for the RASP machines are 'stepped'. Which is to say that if two separate programs require the same number of bits, then the recorded FPGA utilisation figures are *exactly* the same.

The RAMB18 numbers for the RASP machines are initially puzzling. Our intuition is that we would only require the one block of RAM, to hold our program, but for some machines three blocks are utilised and some other have two. One of

	Slice Reg	LUTs	FFs	RAMB18
Addition	21	51	21	3
Subtraction	28	70	28	3
Equality	24	60	24	3
Multiplication	28	70	28	3
Division	32	79	32	3
Exponentiation	32	79	32	3
List Membership	32	79	32	3
Linear Search	32	79	32	3
Reverse List	32	79	32	3
Stateful Rev List	37	86	37	2
Bubble Sort	41	96	41	2
Universal TM	41	96	41	2
Universal RASP	45	108	45	2

Table 5.2: Components for RASP2 implementation

	Slice Reg	LUTs	FFs	RAMB18
Addition	25	70	25	3
Subtraction	29	78	29	3
Equality	25	70	25	3
Multiplication	29	78	29	3
Division	33	91	33	3
Exponentiation	33	91	33	3
List Membership	33	91	33	3
Linear Search	33	91	33	3
Reverse List	33	91	33	3
Stateful Rev List	38	102	38	2
Bubble Sort	38	102	38	2
Universal TM	42	112	42	2
Universal RASP	46	123	46	2

Table 5.3: Components for RASP3 implementations

	Slice Reg	LUTs	FFs	RAMB18	Tuples
Addition	14	13	14	1	3
Subtraction	15	13	15	1	15
Equality	16	16	16	1	18
Multiplication	19	20	19	1	22
Division	19	22	19	1	27
Exponentiation	20	30	20	1	41
List Membership	22	44	22	1	38
Linear Search	22	49	22	1	73
Reverse List	23	32	23	1	50
Stateful Rev List	23	80	23	1	94
Bubble Sort	24	150	24	1	140
Universal TM	23	195	23	1	113
Universal RASP	19	1019	18	1	1111

Table 5.4: Components for TM implementations

the extra block RAMs is to hold state information for the control, but what of the third one?

On inspection of the technological schematics, machines with a third RAM wire the output of this RAM directly to the controlOut signal which is triggered by the OUT command. It is not known why this happens, but hypothesise that it is an artefact resulting from the heavy optimisation options. The TM also has at least one case where the optimiser provides a undesirable result which can be improved by relaxing the options.

Because the symbol table for the TM is part of the control, utilisation results for the TM programs vary from one to the next. With the exception of the list membership program, the utilisation figures tend to follow the number of tuples involved in the program. This is not a smooth trend though, as the gap of ten tuples between the Addition and Subtraction yields less of a difference than the gap between the equality and multiplication programs which is only four tuples.

Further experimentation has revealed that the optimiser attempts to combine tuples and even trims away ones that are deemed ‘constant’. The optimiser was given a symbol table of two states, both of which did the exact same thing. The optimiser threw a warning and said that the second state would be trimmed.

It stands to reason then that the optimiser algorithm tries to combine as many signals as possible into common LUTs and FF pairs to reduce space. However the optimiser can lock itself into a non-optimal route and can cause problems

as in the case of the Universal RASP. A strict area optimisation strategy vastly exaggerates the required number of LUTs (> 2000) required by the universal RASP, whereas a more balanced one yields 1025. Since we are unfortunately not privy to the optimisation algorithms inner workings, we cannot entirely be sure what it does to inflate the LUT requirement.

Without the work of constructing individual gates themselves, we are reliant on the optimiser to deliver us a near-optimal circuit. However the above examples highlight that the results may not be perfect, and so we should take these FPGA numbers as estimates much like the figures from the previous chapter.

That said, a hardware realisation at this level is a time effective solution to the infinite regress problem, and it provides another set of results with which to compare against our hand constructed semantics and programs as a sanity check.

Chapter 6

Analysis

This chapter collates the data from the previous three chapters and provides an analysis. It analyses and then compares the models. There is data which either supports or contradicts the hypotheses and analysis of this evidence is performed relative to the hypotheses. The revised hypotheses postulated at the beginning of Chapter 3 are resolved starting in Section 6.2.8.

Section 6.1 overviews the trends in the program and semantic size measurements from Table 4.2 in Chapter 4. It reviews the data in discrete sets of the arithmetic, list, and universal functions.

Section 6.2 pairs the models (i.e. RASP and TM or RASP and SKI) and examines how the relative information contents of the semantics and programs for those models conform to the hypotheses.

Section 6.2.8 uses the comparisons made in Section 6.2 to resolve the Semantic Information (SI), and Total Information (TI) hypotheses (Section 3.1.2).

The FPGA measurements from Tables 5.1–5.4 in Chapter 5 are analysed in Section 6.3. These analyses are used to evaluate the veracity of the Semantic Circuit (SC) and Total Circuit (TC) hypotheses (Section 3.1.3).

Section 6.4 in the second half of this chapter makes further observations on the data which do not influence the outcome of the hypothesis evaluation. Section 6.5 compares the input encodings for the programs in each model. It also gives a concrete example of how the size of a program can change in relation to the density of the encoding system as introduced in Sections 3.1.1 and 3.1.2.

$$A = \frac{1}{n}\sum_{i=1}^{n} a_i \qquad G = \left(\prod_{i=1}^{n} g_i\right)^{\frac{1}{n}}$$

(a) Calculating the arithmetic mean A (b) Calculating the Geometric mean G

Figure 6.1: The formulae for calculating the arithmetic and geometric means

6.1 Overall Trends

This section provides general comments on how the information contents of the programs relate to one another. The programs are grouped into sets and their Program Information (PI), and Total Information (TI = PI + Semantics size) amounts are compared across models. The sets include the arithmetic (AR) functions, the List (L) functions, arithmetic and list (AR+L), and the arithmetic, list, and universal functions (All).

We compute the arithmetic and geometric means for the PI and TI of each grouping by using the standard formulae in Figure 6.1. The difference between two arithmetic means is an indicator of the absolute difference of characters between the sets of data. The difference in geometric means is more of an indicator of the ratios between datasets implemented in different models.

We use both means as evidence to resolve the hypotheses and often the means are in agreement; if the arithmetic mean for one model is lower than the arithmetic mean for another, then the geometric mean should also be lower. Interestingly this is not always the case. As evidenced by the AR means in Table 6.1 which show that the arithmetic means for the SKI and λ-calculus are larger than those of the RASP2 and RASP3, but their geometric means are lower. As discussed in Section 6.2.8, these geometric ratios appear to indicate if a model has an aptitude for representing the specific set in a more more succinct manner.

6.1.1 Arithmetic

Table 6.1 shows all of the program, semantics, and mean sizes for the arithmetic functions. The imperative models (the RASPs and TM) steadily grow in the amount of information required to express the addition function up to the exponentiation function. This growth is expected as the functions increase in

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
Addition	58	9	25	29	16	27
Subtraction	59	59	61	149	113	46
Equality	57	26	27	179	208	117
Multiplication	126	59	60	223	8	15
Division	131	131	134	281	565	229
Exponentiation	132	129	131	450	11	9
Semantics Size	556	585	587	335	291	515
AR PI Mean	93.83	68.83	73	218.50	153.83	73.83
AR TI Mean	649.83	653.83	660	553.50	444.83	588.83
AR PI Geo Mean	86.71	48.95	59.27	167.15	51.52	40.62
AR TI Geo Mean	648.84	652.18	658.53	538.68	410.48	584.09

Table 6.1: The program and semantic sizes of the arithmetic functions for each model.

complexity and involve more nested loops.

On the other hand the functional models (SKI and λ-calculus) have large subtractive functions (subtraction, division, and equality), but comparatively small combinative functions (addition, multiplication, exponentiation). The reason for this is to do with how the λ-calculus and SKI represent numerals. The higher order functionality of the Church numerals enables very succinct combinative functions. For example, the exponentiation function directly applies one numeral to another.

RASP numerals are defined as naturals and the INC and DEC instructions are defined to operate over these in the semantics. The SKI and λ-calculus so not have such defined structures and operators in their semantics, which results in the numerals and operations such as decrementation needing to be defined in each expression which wants to use them.

Section 2.3.2.1 describes why the λ-calculus PRED function is larger than SUCC. In requiring a "program level" definition for PRED, expressions which use it are inflated in size compared to expressions which do not. If numerals and SUCC/PRED were defined in the semantics of the λ-calculus and SKI, it would be expected that the (PI) of the functions would normalise to look something more like the RASP figures.

The means show that the PI for the expressive models (RASPs and the λ-calculus) is lower than for the less expressive models. However TI of the less

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
List Membership	271	129	131	379	362	208
Linear Search	281	132	135	779	385	236
Reverse List	140	135	137	499	190	134
Stateful Rev List	273	273	277	1049	1397	460
Bubble Sort	557	549	297	1611	1903	550
Semantics Size	556	585	587	335	291	515
L PI Mean	304.4	243.6	195.4	863.4	847.4	317.6
L TI Mean	860.4	828.6	782.4	1198.4	1138.4	832.6
L PI Geo Mean	276.67	202.98	181.93	757.23	588.18	278.13
L TI Geo Mean	850.31	814.57	778.72	1123.07	953.07	817.85

Table 6.2: Program and semantic sizes of the list functions for each model

expressive models is overall lower than that of the more expressive ones. For these arithmetic functions, it appears that the extra information in the semantics of the RASPs and λ-calculus outweighs the average information saving for their programs. The implementations of the division and exponentiation functions in the TM require more TI than their RASP and λ-calculus contemporaries. This is also true for the SKI division TI.

6.1.2 List

Table 6.2 shows the sizes and means of the programs and semantics for the list functions. The data for this function set is more homogeneous across the models in comparison to the arithmetic function sizes. Here the difference in size from one function to the next is roughly correlative across all models.

Sections 4.3.3 and 4.3.4 imply that reversal of a list by building a new list is a simpler function than reversal by swapping elements in place. The PIs here support that implication as there is a jump in the required amount of information for all of the models.

The means for these functions show that the more expressive models have now have a lower PI and TI amounts than the less expressive models. The RASP3 has the lowest PI and TI of all of the models and has the largest semantics. The TM has the highest PI and TI despite having larger semantics than the SKI calculus.

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
Universal TM	613	571	574	1270	2593	584
Universal RASP	1239	1209	1231	14414	9554	1084
Semantics Size	556	585	587	335	291	515

Table 6.3: Program sizes of the universal functions for each model

1 Strong Semantic Information hypothesis

1a. SI within family. **For:** 6.2.1
1b. SI within paradigm. **For:** 6.2.2, 6.2.3
1c. SI across paradigms. **For:** 6.2.4, 6.2.7 **Against:** 6.2.5, 6.2.6

2. Strong Total Information hypothesis

2a. TI within family. **For:** 6.2.1
2b. TI within paradigm. **For:** 6.2.2, 6.2.3
2c. TI across paradigms. **For:** 6.2.4, 6.2.7 **Against:** 6.2.5, 6.2.6

Figure 6.2: Hypotheses and evidence for each

6.1.3 Universal

Table 6.3 shows the sizes of the universal RASP and Turing machines for each model and their semantics. The data shows that models with larger semantics ($>$ 500) require roughly double the amount of information to represent the URASP compared with representing the UTM. In contrast, less expressive models require significantly more information. This is evidence that there is a fundamental difference between the expressive models and less expressive models in how they manage the memory structures of the TM and RASP. This topic is covered in further detail in Section 6.4.

6.2 Grouped Analysis

This section groups the models so that relations between them can be observed and evidence can be gathered to confirm or refute the SI and TI hypotheses. Figure 6.2 list confirming and contradicting evidence up front. The SI and TI hypotheses are defined in Section 3.1.2 and are recapped here.

Hypothesis 1: The Semantic Information (SI) hypothesis states that: "For two Turing Complete models; if model A has more semantic information (larger semantics) than model B, the average size of succinct programs (where at least

1 Strong Semantic Information hypothesis
  1a. SI within family hypothesis
    1.1. Program Sizes (RASP) prediction.
  1b. SI within paradigm hypothesis
    1.2. SI RASP vs TM prediction
    1.3. λ-calculus vs SKI prediction
  1c. SI across paradigms hypothesis.
    1.4. Across paradigms prediction
2. Strong Total Information hypothesis
  2a. TI within family hypothesis
    2.1. TI for RASPs
  2b. TI within paradigm hypothesis
    2.2. TI RASP vs TM
    2.3. TI λ-calculus vs SKI
  2c. TI across paradigms hypothesis
    2.4. TI across paradigms prediction

Figure 6.3: Breakdown of the Strong SI and TI hypotheses

one program utilises the extra semantic information) written for model A will be lower than the average for model B." (Section 3.1.2). This 'strong' hypothesis is broken down into three sub-hypotheses which state the above relation for models for the same family (1a), models in the same paradigm (1b), and models in different paradigms(1c).

Hypothesis 2: The Total Information (TI) hypothesis states that: "For two Turing Complete models X and Y, where X has more semantic information than Y; As the size and complexity of a program increases, the average total information (TI) of a succinct implementation in X will decrease relative to the total information of a succinct implementation in Y." (Section 3.1.2). Again, there are set of sub-hypotheses to cover the paradigmal possibilities (2a, 2b, and 2c). Figure 6.3 presents the hierarchy of hypotheses and the predicted nature of the relationships. Section 3.1.2 gives the exact wordings of the sub-hypotheses and predictions.

Tables 6.4, 6.5, and 6.6 show: all of the size measurements for the programs and semantics of all the models, the arithmetic means of the groupings, and the geometric means of the groupings. These tables shall all be referred to throughout

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
Addition	58	9	25	29	16	27
Subtraction	59	59	61	149	113	46
Equality	57	26	27	179	208	117
Multiplication	126	59	60	223	8	15
Division	131	131	134	281	565	229
Exponentiation	132	129	131	450	11	9
List Membership	271	129	131	379	362	208
Linear Search	281	132	135	779	385	236
Reverse List	140	135	137	499	190	134
Stateful Rev List	273	273	277	1049	1397	460
Bubble Sort	557	549	297	1611	1903	550
Universal TM	613	571	574	1270	2593	584
Universal RASP	1239	1209	1231	14414	9554	1084
Semantics Size	556	585	587	335	291	515

Table 6.4: The combined program and semantic sizes for each model

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
AR PI	93.83	68.83	73	218.50	153.83	73.83
AR TI	649.83	653.83	660	553.50	444.83	588.83
L PI	304.4	243.6	195.4	863.4	847.4	317.6
L TI	860.4	828.6	782.4	1198.4	1138.4	832.6
AR + L PI	189.55	148.27	128.64	511.64	468.91	184.64
AR + L TI	745.55	733.27	715.64	846.64	759.91	699.64
All PI	302.85	262.38	247.69	1639.38	1331.15	284.54
All TI	858.85	847.38	834.69	1974.38	1622.15	799.54

Table 6.5: The arithmetic means of the program groupings

the analysis.

6.2.1 RASP Machines

The RASP machines are a family of models. They have a common core of model semantics which share a number of functions. They each differ in how they modify the value in their accumulator: RASP uses INC and DEC, RASP2 has a direct ADD x and SUB x, and RASP3 has an indirect ADD x and SUB x.

The RASP machines are relevant in the resolution of SI/TI within family sub-hypotheses. The vanilla RASP machine has the smallest semantics, followed by the RASP2, and then the RASP3 (Table 6.4). By the SI and TI within family hypotheses, it is therefore expected that the instruction counts (Table 6.7), character counts (Table 6.4), and means (Tables 6.5–6.6) follow the trend

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
AR PI	86.71	48.95	59.27	167.15	51.52	40.62
AR TI	648.84	652.18	658.53	538.68	410.48	584.09
L PI	276.67	202.98	181.93	757.23	588.18	278.13
L TI	850.31	814.57	778.72	1123.07	953.07	817.85
AR+L PI	146.93	93.44	98.68	332.16	155.84	97.39
AR+L TI	733.70	721.54	710.74	752.26	601.98	680.66
All PI	193.22	130.78	137.21	492.16	265.52	134.54
All TI	814.65	802.48	793.38	1002.53	841.93	754.17

Table 6.6: The geometric means of the program groupings

Program	RASP	RASP2	RASP3
Addition	17	4	6
Subtraction	18	22	22
Equality	19	9	11
Multiplication	32	24	24
Division	42	45	45
Exponentiation	51	43	40
List Membership	71	34	31
Linear Search	87	36	35
New List Rev	57	45	43
In Place Rev	73	78	77
Bubble Sort	131	127	123
Universal TM	200	148	137
Universal RASP	313	292	283
Arithmetic Mean	85.46	69.76	67.56
Geometric Mean	57.99	40.79	41.47

Table 6.7: Registers used by the various RASP programs

where the RASP3 counts grow slower than the RASP2, which in turn grow slower than the RASP counts.

RASP machine sizes grow according to the value 2^n, where n is the number of bits that the machine can hold in each register. The size of the machine's memory and maximum natural number which can be represented is therefore 2^n for an n-bit machine. A program fits into the memory if there is at least one register available to fit each instruction/datum in the program starting from register 3. Unused registers are padded with the HALT instruction (0) and can be, in principle, utilised by the program for storage, but the program at initialisation does not directly write to or read from the registers.

Table 6.7 shows the number of utilised registers for each program in each RASP machine. For the arithmetic functions, the RASP2 uses fewer registers on

average than the RASP3 and RASP. However for the list functions, the RASP3 requires fewer registers on average than the RASP2. This trend continues for the universal functions. On average, the RASP3 requires fewer registers than the RASP2, which requires fewer registers than the RASP. This data fits Prediction 1.1 (Figure 6.3) where the model with the most SI requires the least number of registers/instructions.

Referencing the RASP columns of Table 6.5. The arithmetic means of the program groupings show the RASP2 with the overall lowest PI for the arithmetic functions, the RASP with the overall lowest TI of the arithmetic functions, and the RASP3 with the overall lowest PIs and TI for every other group. The RASP3 rankings for L, AR+L, and All is closely followed by RASP2, and then followed by the RASP.

The geometric means (RASP columns, Table 6.6) show the PI of the RASP2 as the lowest for all sets excluding the L set. The RASP3 PI is the overall lowest for the L set and the TI is the overall lowest for every set except the AR TI set. The RASP has the lowest AR TI for the arithmetic and geometric means.

The arithmetic and geometric mean data fits Predictions 1.1 and 2.1. These state that the RASP3 will eventually have the lowest average PI and TI respectively. The TI of the RASP is the lowest of the three models for the arithmetic function grouping, but as the set of tested functions grows, the RASP3 becomes the model with the lowest TI.

With the exception of the PI geometric means for each category (PI rows, Table 6.6), which show the RASP2 using less PI than the RASP3, these expectations have been met and the data is in favour of confirming sub-hypotheses 1a and 2a (Figure 6.3).

With the exception of the above geometric PI measure, Prediction 1.1 has been fulfilled by the "All PI" row of Table 6.5 showing RASP3 with the lowest PI of the RASPs. The utilised register average of Table 6.7 also substantiates this. The contrary geometric mean figures show the RASP2 as having the least utilised registers in Table 6.7, and lowest PI in Table 6.6. This carries less weight in our minds as the geometric mean is weighted very heavily towards the shorter arithmetic functions. The RASP3 requires fewer characters to implement the

functions of Table 6.4 than the RASP2, (3249 vs 3439).

Prediction 2.1 has also been fulfilled by as the average TI of the RASP3 is the lowest of all of the RASPs, and the TI of the RASP is the greatest. This relationship holds for both the arithmetic and geometric means.

This analysis concludes that the data is consistent with predictions 1.1 and 2.1, and therefore we confirm sub-hypotheses 1a and 2a; SI/TI within family.

6.2.2 RASP vs TM

Comparisons of the RASP and TM models seeks evidence for the SI/TI within paradigm sub-hypotheses (hypotheses 1b and 2b) fully stated in Section 3.1.2. To paraphrase; the SI within paradigm hypothesis predicts that there is an inverse size relationship between semantics size and program size for models of the same paradigm. The TI within paradigm hypothesis states that as a program or programs grows in size and complexity, the average TI (SI+PI) of an expressive model implementing these programs reduces relative to the average TI of a less expressive model in the same paradigm.

This section compares the RASPs and TM to gather evidence for the imperative paradigm. Section 6.2.3 also gathers evidence for these hypotheses, but in the functional paradigm using the SKI and λ-calculus.

The Turing machine semantics are smaller than the semantics of the RASP machines. We therefore expect to see (Predictions 1.2, 2.2) that the TM produces larger program on average than the RASP. We also expect that for some of the simpler programs, the TI of the TM is lower than that of the RASPs, but as the set of programs grows the TI of the RASPs drops to below that of the TM.

The program sizes (RASP and TM columns, Table 6.4 show that the average program size for the TM is larger than those for the RASP. The only exception to this is the addition program. The means in Tables 6.5 and 6.6 substantiate this with the PI rows. The average PI of the TM in every category is higher than that of the RASPs. This data supports the SI within paradigm sub-hypothesis (1b).

Turning attention to the TI within paradigm sub-hypothesis, we consider the TI means of Tables 6.5 and 6.6. The TI means for the TM implementing the AR

functions is lower than the TI means of the RASPs. However as more functions are introduced: L, AR+L, and All; the TIs of the RASPs end up lower than the TIs of the TMs. This is substantiating evidence for the TI within paradigm sub-hypothesis as it satisfies Prediction 2.2.

This analysis is consistent with our Predictions 1.2 and 2.2, which support the SI/TI within paradigm sub-hypotheses. The SI/TI within paradigm sub-hypotheses appear to be confirmed with respect to the RASP and TM.

6.2.3 SKI vs λ-calculus

Like the RASP vs TM comparison above in Section 6.2.2, this analysis aims to find evidence supporting, or contradicting, the SI/TI within family sub-hypotheses (Section 3.1.2, hypotheses 1b and 2b. If these hypotheses are correct, the relationship between the SKI and λ-calculus information sizes will broadly mirror the observed relationship between the TM and RASP.

The SKI semantics are smaller than those of the λ-calculus so it is expected that the average size of SKI programs is larger than that of the λ-calculus (by the SI within paradigm hypothesis). It is also expected that for some of the simpler programs, the TI of the SKI is lower than that of the λ-calculus, but as the set of programs grows the TI of the λ-calculus drops to below that of the SKI.

Like the resolution of the SI hypothesis with the RASP and TMs, the mean program sizes from Tables 6.5 and 6.6 (SKI and λ-calculus columns) show the PI means of the λ-calculus to be lower than that of the SKI. The measurements from Table 6.4 substantiate this, with the multiplication function as the only exception. The SI within paradigm sub-hypothesis (1b) is therefore supported by this data.

Evidence for the TI within paradigm sub-hypothesis can be found in the mean Tables 6.5 and 6.6. For the arithmetic means (Table 6.5), the TI figures for the AR set shows that the SKI is lower than that of the λ-calculus, but as other sets get introduced, the TI of the λ-calculus returns to below that of the SKI.

This is almost a mirroring of the results of the RASP and TM comparisons. However, the geometric TI means of Table 6.6 show the mean SKI TI diverging from the λ-calculus at a slower rate. The RASP and TM diverged after the AR

set, but the SKI and λ-calculus diverge after the AR+L set of functions.

The SKI and λ-calculus program sizes are highly correlated, especially considering that the SKI programs are derived from the λ-calculus via bracket abstraction (Section 2.3.2.2). Therefore it makes sense that it takes more programs to show a separation in program size for the SKI/λ-calculus than for the RASP and TM which are not derived from one another.

The arithmetic and geometric means therefore support the TI within paradigm sub-hypothesis (hypothesis 2b, Section 3.1.2). Along with the analysis of the program size means, both the SI and TI sub-hypotheses are supported by the data of the SKI and λ-calculus. Both the evidence for this analysis, and the RASP/TM analysis (Section 6.2.2) are briefly reiterated in Section 6.2.8 where the SI/TI within paradigm hypotheses are resolved.

6.2.4 RASP vs SKI

The RASP vs SKI analysis produces evidence for the SI/TI across paradigms sub-hypothesis (hypotheses 1c and 2c). The SI sub-hypothesis states that there is an inverse relationship between the size of the semantics and the average size of programs which holds when two models from different paradigms are compared (Section 3.1.2).

Table 6.4 (RASP and SKI columns) shows that the SKI calculus has a smaller set of semantics than any of the RASP machines. It also shows that the SKI programs for the combinative AR functions (addition, multiplication, exponentiation) are smaller than any of the RASP programs. The higher-order functionality of the Church numerals allows the SKI (and λ-calculus) to produce very concise combinative AR functions.

As a result of this, the geometric "AR PI" mean (Table 6.6) favours the SKI over the RASPs. The "L PI" geometric mean for the SKI is much larger than that of the RASP, and this extra information pushes the means in favour of the RASP machines. The "AR+L PI" geometric means for the RASP is lower than the corresponding mean for the SKI. The gap widens when the universal machines are introduced to the test set. The arithmetic means (Table 6.6) are not as influence by the small combinative functions as the geometric mean, so

they show the RASPs have less PI than the SKI in all program sets.

This evidence conforms to the SI within paradigm hypothesis, and is in line with prediction 1.4 (Section 3.1.2) because the larger RASP semantics result in smaller programs on average compared to the SKI.

The TI across paradigms sub-hypothesis (2c) states that as a program or programs grows in size and complexity, the average TI (SI+PI) of an expressive model implementing these programs reduces relative to the average TI of a less expressive model in a different paradigm (Section 3.1.2).

The arithmetic TI means in Table 6.5 show that the SKI has a lower TI than the RASPs for the AR functions. As more functions are introduced however, the TI of the RASPs drops to below the TI of the SKI. It takes longer for the geometric means to diverge (RASP and SKI columns, Table 6.6). The "AR TI" and "AR+L TI" means show that the SKI requires less TI on average than the RASPs. Including the universal machines also

The data from this analysis supports the SI/TI across paradigms hypotheses. For these hypotheses to be confirmed though, analysis has to be made of the RASP vs λ-calculus (Section 6.2.5), TM vs SKI (Section 6.2.6), and TM vs λ-calculus (Section 6.2.7).

6.2.5 RASP vs λ-calculus

The RASP vs λ-calculus analysis produces evidence for the SI/TI across paradigms sub-hypothesis (hypotheses 1c and 2c). The SI sub-hypothesis states that there is an inverse relationship between the size of the semantics and the average size of programs which holds when two models from different paradigms are compared (Section 3.1.2).

The RASP machines all have larger semantics than the λ-calculus (Table 6.4) so if the SI hypothesis were to hold, it is expected that the programs in the RASPs are smaller on average compared to those in the λ-calculus. As with the SKI, the λ-calculus has small combinative arithmetic functions, and large subtractive functions.

The RASP and λ-calculus columns of Table 6.5 show that the λ-calculus uses less PI for the AR functions, than the RASP and RASP3 but more than the

	RASP	λ-calculus	Difference
AR PI	93.83	73.83	20
AR+L PI	189.55	184.64	4.9
All PI	302.85	284.54	18.30

Table 6.8: Difference between RASP PI arithmetic means and the λ-calculus means

RASP2. For the "AR + L PI" function set, the RASP2 and RASP3 sets use less PI than the λ-calculus. Adding the universal functions ranks the RASPs and λ-calculus in terms of required PI as: RASP3 < RASP2 < λ-calculus < RASP.

This analysis contradicts the SI across paradigms hypothesis, which is interesting considering that the evidence for the previous hypotheses is confirmatory. The vanilla RASP has more semantic information than the λ-calculus, so by Prediction 1.4 (Section 3.1.2) we expect to see that the λ-calculus requires more PI than the RASP. This is not the case. And from Table 6.8 we can see that the gap between the PIs shrinks from AR to AR+L, but widens when the universal functions are included. The relationship between the PIs of the RASP and λ-calculus are too complex to be simply characterised by the SI within paradigms hypothesis.

The TI across paradigms sub-hypothesis (2c) states that as a program or programs grows in size and complexity, the average TI (SI+PI) of an expressive model implementing these programs reduces relative to the average TI of a less expressive model in a different paradigm (Section 3.1.2).

Because the λ-calculus has smaller semantics, Prediction 2.4 (Section 3.1.2) sets out the expectation of the RASPs requiring less TI to represent all of the functions. From Tables 6.5 and 6.6, this is not the case at all. The TI measurements of the λ-calculus implementations are consistently lower than any of the RASP measurements.

We conclude that the SI and TI across paradigms hypotheses (1c and 2c) with respect to the RASPs and λ-calculus cannot be confirmed. The data here does not conform to the prediction that the λ-calculus will have a higher mean PI and TI than the RASPs. Indeed, the difference between the PI and TI of the models fluctuates as more sets of programs are compared, with no clear relationship which can be explained to fit the hypothesis. This is discussed more in Section

6.2.8.

6.2.6 TM vs SKI

The TM vs SKI analysis produces evidence for the SI/TI across paradigms sub-hypothesis (hypotheses 1c and 2c). The SI sub-hypothesis states that there is an inverse relationship between the size of the semantics and the average size of programs which holds when two models from different paradigms are compared (Section 3.1.2).

The TM has more semantic information than the SKI, (Table 6.4) so it is expected that the TM will require less PI on average than the SKI to compute the functions.

Tables 6.5 and 6.6 show that the mean PI measurements for the SKI are exclusively lower than the PI measurements of the TM. These measurements lend no evidence to the SI across paradigms hypotheses. Indeed, this data contradicts the hypothesis, much like the data from the RASP and λ-calculus comparison in Section 6.2.5.

The TI across paradigms sub-hypothesis (2c) states that as a program or programs grows in size and complexity, the average TI (SI+PI) of an expressive model implementing these programs reduces relative to the average TI of a less expressive model in a different paradigm (Section 3.1.2).

Again, Tables 6.5 and 6.6 demonstrate that the TI of the SKI is lower than the TI of the TM for both arithmetic and geometric means in all program sets. The analysis here of the SKI measurements against the TM measurements contradict the SI/TI across paradigms hypotheses (1c and 2c). This is very similar to the examination of the λ-calculus and RASP in Section 6.2.5

6.2.7 TM vs λ-calculus

The final comparison which we draw in this part of the analysis is between the TM and λ-calculus. This analysis serves to find evidence for the SI/TI across paradigms hypothesis (hypotheses 1c and 2c).

The TM semantics are smaller than the λ-calculus semantics (Table 6.4), so it is expected, by the SI across paradigms hypothesis, that the average size of

programs in the λ-calculus is lower than the average size of programs in the TM. The means in Tables 6.5 and 6.6 show that the PIs of the λ-calculus functions are lower than the PIs of the TMs in all function sets. This behaviour fits with prediction 1.4, much like the RASP and TM comparison in Section 6.2.2.

The TI across paradigms sub-hypothesis (2c) states that as a program or programs grows in size and complexity, the average TI (SI+PI) of an expressive model implementing these programs reduces relative to the average TI of a less expressive model in a different paradigm (Section 3.1.2).

The λ-calculus has larger semantics of than the TM, so the TI arithmetic and geometric means (Tables 6.5 and 6.6) of the AR function set show that the TM requires less TI than the λ-calculus. As the function sets expand, the TI required for the λ-calculus reduces relative to the TI required for the TM.

Prediction 2.4 is also satisfied by this behaviour. The λ-calculus and TM comparison produces evidence with supports both of the SI/TI across paradigms hypotheses.

6.2.8 The SI and TI Hypotheses

Figure 6.2 lists the evidence gathered for each sub-hypothesis and the section where that evidence is found. The semantic information (SI) hypothesis predicts (Predictions 1.1 - 1.4) that if two models have differing semantic sizes, the model with more semantic information will require less information to implement succinct programs on average compared to the model with less semantic information. At least one of the programs should utilise the extra operators afforded by the larger semantics in order to see the benefit (Section 3.1.2).

This PI data fulfils Predictions 1.1, and 1.2 and 1.3, therefore Sub-hypotheses 1a (family) and 1b (within paradigm) are confirmed. The RASP data shows that over the whole set of compared functions, the RASP3 uses less information on average than the RASP2 and RASP. The RASP3 has the largest semantics, while the RASP has the smallest (Section 6.2.1).

The within paradigm hypothesis is supported by the comparison of the size of λ-calculus expressions versus the size of SKI expressions (Section 6.2.3). In the imperative paradigm, the average TM PI versus the average RASP PI shows

that the TM programs are typically larger than the RASP ones (Section 6.2.2).

The third sub-hypothesis, hypothesis 1c requires that four comparisons are made: RASP and SKI, RASP and λ-calculus, TM and SKI, and TM and λ-calculus. Unlike the other two sub-hypotheses, the comparison across paradigms reveals evidence contrary to the hypothesis and does not satisfy the prediction entirely.

Over this test set, an imperative model compared with a functional one with approximately the same amount of Semantic Information will show that the functional model has a lower average PI than the imperative model.

The RASP and λ-calculus comparisons show that the λ-calculus requires less PI than the RASPs, despite the fact that the RASPs have more SI (Section 6.2.5). Similarly, the TM has more SI than the SKI, but the SKI still has smaller programs on average (Section 6.2.6).

The Strong Semantic Information hypothesis is not confirmed. The within family and within paradigm hypotheses have evidence enough to confirm them. The across paradigms hypothesis has evidence for it, but more importantly, has strong evidence against it.

The Total Information (TI) hypothesis predicts (Predictions 2.1 - 2.4) that as the size and complexity of a program, or programs, increases; the TI (SI + PI) of succinct implementations of the programs in a model which is more expressive will reduce relative to the TI of the implementations in a model which is less expressive (Section 3.1.2).

Much like the SI hypothesis, the TI hypothesis has support from the within family, and within paradigm hypotheses (2a and 2b). The RASP semantic sizes are ordered as RASP<RASP2<RASP3. When the entirety of the program set is considered, the TI sizes of the RASPs are RASP3<RASP2<RASP which fits the prediction and confirms the within family hypothesis (Section 6.2.1).

The within paradigm hypothesis is supported by the evidence of the TM vs RASP and λ-calculus vs SKI comparisons. While the smaller models had a lower TI for the AR set of functions, as the set was augmented with the list, and then universal, functions, the TI shifted in favour of the larger models. Section 6.2.2 compared the RASP with the TM while Section 6.2.3 compared the λ-calculus

and SKI.

Generalising TI to across paradigms appears to fall into the same trouble as the SI corresponding hypothesis. Comparing two models of differing paradigms with roughly the same amount of SI will favour the functional model as the comparisons of the RASP and λ-calculus (Section 6.2.5), and SKI vs TM (Section 6.2.6) suggest.

Like the SI hypothesis, sub-hypotheses 2a and 2b are confirmed, while sub-hypothesis 2c is not. The strong TI hypothesis in this case cannot be confirmed. It is suspected that the simple metric of raw character distance between the semantics of models from differing paradigms is too naïve to capture the subtleties of their evaluation method. The evaluation method produces less of an impact on the information values for those models in the same model family or paradigm, compared to across paradigm comparisons where the evaluation method is much more relevant.

Returning to the geometric means, comparing the RASP2/3 means against the λ-calculus means in Tables 6.5 and 6.6, it can be seen that while the arithmetic PI means of the λ-calculus are always greater than those of the RASPs, the geometric means do not necessarily follow. This appears to stem from the PI required to represent the AR functions.

The λ-calculus uses much smaller expressions for the additive arithmetic functions in comparison to the RASPs due to the Church numerals and their combinatoric attributes. This results in a lower geometric mean for the AR functions, even though the arithmetic mean is higher (because of the relatively large subtractive AR functions). It would then be interesting to consider the geometric-arithmetic mean relationship as an indication of a models aptitude at representing a set of functions. In this case, the λ-calculus has an advantage in representing AR functions.

This indication is less clear however as the sets are combined. The AR+L and all sets also have lower geometric means despite the L set and universal sets alone having no notable deviation in this geometric-arithmetic relationship.

3. Semantic Circuit hypothesis
   - 3a. SC within family hypothesis
     - 3.1. SC for RASPs.
   - 3b. SC within paradigm hypothesis
     - 3.2. SC RASP vs TM
4. Total Circuit hypothesis
   - 4a. TC within family hypothesis
     - 4.1. TC for RASPs.
   - 4b. TC within paradigm hypothesis
     - 4.2. TC RASP vs TM

Figure 6.4: Breakdown of the FPGA hypotheses

6.3 FPGA Analysis

This section provides an analysis of the FPGA measurements with respect to evaluating the Semantic Circuit (SC) size and Total Circuit (TC) size hypotheses. This section provides an overview of the measurements. Section 6.3.1 covers comparisons of the RASP machines to find evidence for the SC and TC hypotheses. Section 6.3.2 compares the RASP implementation to the TM implementation for more evidence. Section 6.3.3 uses the evidence of the aforementioned sections to evaluate the hypotheses.

Figure 6.4 breaks down the SC and TC hypotheses. Like the SI and TI hypotheses, there are sub-hypotheses defined. Because only the RASPs and TMs are defined in the FPGA, there are no "across paradigms" hypotheses. The Semantic Circuit hypothesis states that there is a direct relationship between the SI and the size of the circuit to represent the semantics. Simply put, SI is proportional to SC.

The Total Circuit hypothesis is analogous to the TI hypothesis. It states that for two models A and B, where A has a larger semantic circuit than B. As the set of tested programs grows in size and complexity, the average total implementation size (number of FPGA components required to implement the semantics and program) for A will decrease relative to the average total implementation size for B.

In Chapter 5, the RASP and TM models were realised in VHDL and synthe-

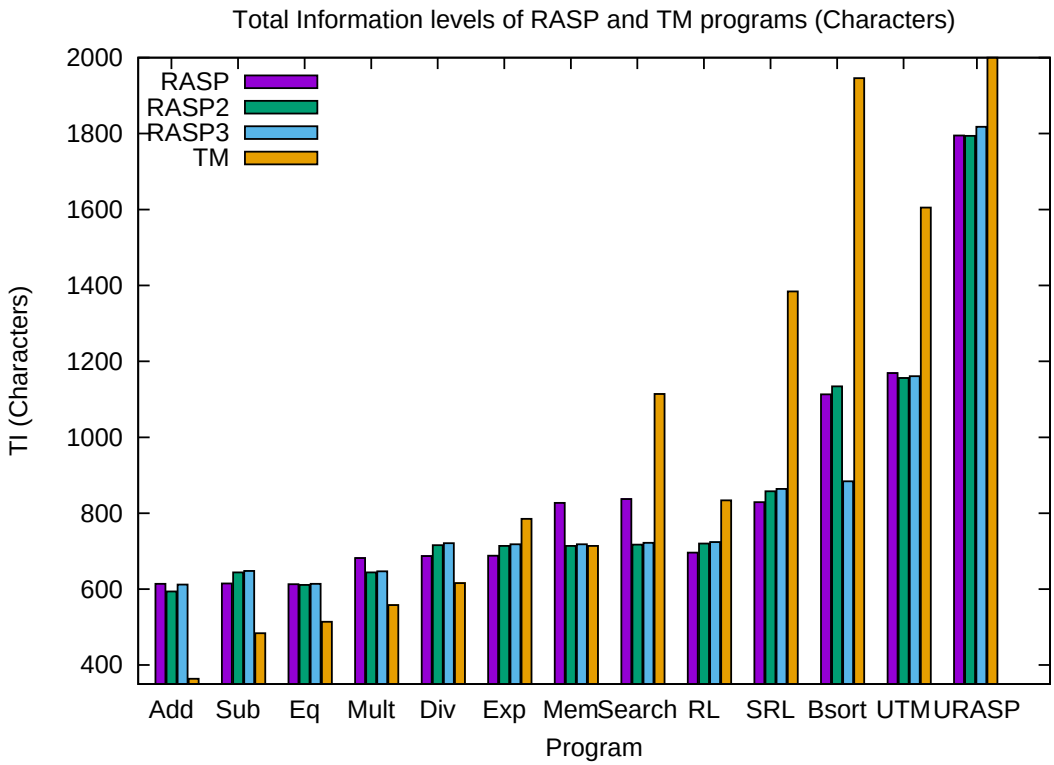


Figure 6.5: RASP and TM Total Information contents

sised down to registers, flip-flops (FFs) and look up tables (LUTs). Synthesis of VHDL to FPGA components not only converts programs to electronic components, but also the semantics of the model. In essence, an instance of the machine is constructed and loaded with the program and data ready to be executed.

If the number of required FPGA components can be used to predict the TI of programs in models, then it is expected that the component counts correlate with the TI figures of the programs/models. Figure 6.5 plots the TIs for the RASPs and TM from the figures presented previously in this chapter.

The Slice Registers (Table 6.9, Figure 6.6) are individual memory locations used by the models. Both the RASP and TM use registers (which are configured to be flip-flops) to store state information of the model. Various counters within the model keep track of which instructions are to be executed in each clock cycle, and these counters are stored in slice registers.

Furthermore, the RASPs store their programs in slice registers, the number of which depend on the memory size of the particular machine. The RASP plots in Figure 6.6 exhibits similarities in shape with the TI RASP plots of Figure 6.5. These similarities can be interpreted as; the number of slice registers used

	RASP	RASP2	RASP3	TM
Addition	28	21	25	14
Subtraction	28	28	29	15
Equality	28	24	25	16
Multiplication	32	28	29	19
Division	32	32	33	19
Exponentiation	32	32	33	20
List Membership	37	32	33	22
Linear Search	37	32	33	22
Reverse List	32	32	33	23
Stateful Rev List	37	37	38	23
Bubble Sort	41	41	38	24
Universal TM	41	41	42	23
Universal RASP	46	45	46	19

Table 6.9: Slice registers for programs and models on FPGAs

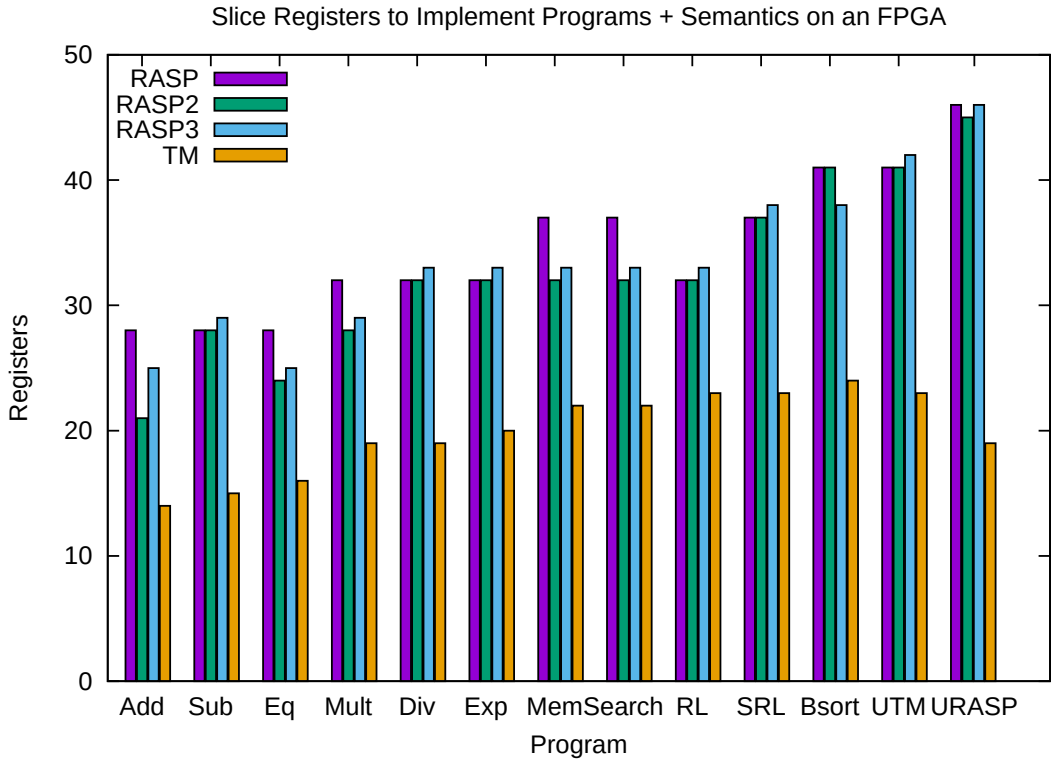


Figure 6.6: Slice registers for RASPs and TM

	RASP	RASP2	RASP3	TM
Addition	66	51	70	13
Subtraction	66	70	78	13
Equality	66	60	70	16
Multiplication	74	70	78	20
Division	74	79	91	22
Exponentiation	74	79	91	30
List Membership	81	79	91	44
Linear Search	81	79	91	49
Reverse List	74	79	91	32
Stateful Rev List	81	86	102	80
Bubble Sort	90	96	102	150
Universal TM	89	96	112	195
Universal RASP	92	108	123	1019

Table 6.10: LUTs for programs and models on FPGAs

	RASP	RASP2	RASP3	TM
Slice Registers	0.795	0.754	0.808	0.00
LUTs	0.706	0.742	0.807	0.980
Flip-Flops	0.776	0.754	0.808	-0.076

Table 6.11: The Pearson correlation coefficient of the TI vs the components

to implement a RASP program on an FPGA is an indicator of the amount of TI required to implement the program against the semantics. There is no similarities which can be observed between the TI of the TM and the number of slice registers used.

The number of LUTs required to implement the RASP and TM programs in the FPGA is presented in Table 6.10 and plotted in Figure 6.7. These figures correlate with the TI levels of the TM. This suggests to that, like the slice registers for RASPs, the number of LUTs is an indicator of the TI of a program written for a TM.

Table 6.11 shows the Pearson correlation coefficient between the TI figures and the various component counts. As we have noted above, the number of slice registers do not correlate at all with the TI counts of the TMs. However, the correlation coefficient of the number of LUTs in the TM implementation is 0.984 which is a very high correlation and suggests a causal link.

There is also a correlation between the TI of the RASP and the number of slice registers. This correlation decreases slightly for the RASP2, and increases

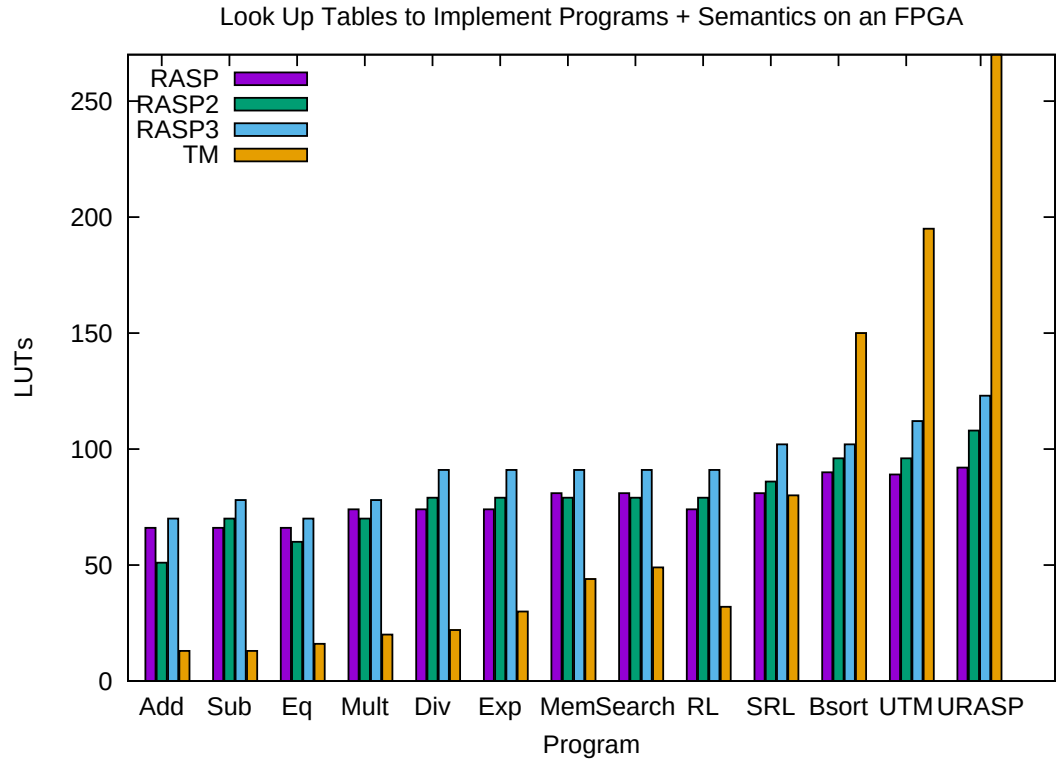


Figure 6.7: LUTs for RASPs and TM

again for the RASP3.

LUTs strongly correlate with the TI of TMs, but are not a perfect indicator. The TM to find the membership of a list is 38 tuples in size and 379 characters long. The list reversal TM is 50 tuples in size and 499 characters long. The bubble sort is 140 tuples/1611 characters and the universal machine is 113 tuples/1270 characters. The number of LUTs to implement the membership TM is 44, as opposed to 32 for the reversal TM. Similarly, it takes 150 LUTs to implement the bubble sort and 195 for the UTM. The number of components for each pairing is at odds with the number of tuples and characters required. If there were a direct correlation between the number of LUTs and number of tuples, then these relations would be switched.

The unknown variable in the FPGA compilation process is the optimisation stage. The optimiser is set up for a much compression as possible, and it is conceivable that the tuples for the bubble sort and reversal can be combined into a smaller overall package. New work focused on this question would bring insight as to why.

Despite the inconsistencies regarding the membership, reversal, bubble sort,

	RASP	RASP2	RASP3	TM
Addition	28	21	25	14
Subtraction	28	28	29	15
Equality	28	24	25	16
Multiplication	32	28	29	19
Division	32	32	33	19
Exponentiation	32	32	33	20
List Membership	37	32	33	22
Linear Search	37	32	33	22
Reverse List	32	32	33	23
Stateful Rev List	37	37	38	23
Bubble Sort	41	41	38	24
Universal TM	41	41	42	23
Universal RASP	46	45	46	18

Table 6.12: FFs for programs and models on FPGAs

and the UTM; the high correlation between the TI and LUT count strongly indicates that the TI of a TM implementation affects the corresponding LUT count of that implementation in a FPGA.

The number of LUTs in an implementation does not appear to directly link the RASP machines to their TI, but is useful when the RASPs are compared against each other later in this section.

The slice registers on the FPGA are versatile. They can be configured as and/or logics, latches, latch-thrus, or D-type flip-flops [105, 13]. With the exception of the universal RASP in the TM, slice registers in these implementations have been exclusively used to implement flip-flops. The table and plot for the flip-flops are very similar to the table and plot for the slice registers, so what has been said about the slice registers applies here. The FF counts are not an indicator of the TI of TM implementations, and have a correlation coefficient on par with the slice registers for the RASPs.

For this data set, the slice registers (Table 6.9) and flip-flop counts (Table 6.12) are almost identical. But if there was more variety in the configurations for the slice registers, then the number of flip-flops could be a better indicator of RASP program information as it corresponds to the size of the RASP memory and state memories. The absolute slice register count would be a better indicator of TI as it covers not only the program size and state memories, but also the ancillary logics and latches that a slice register can be used for.

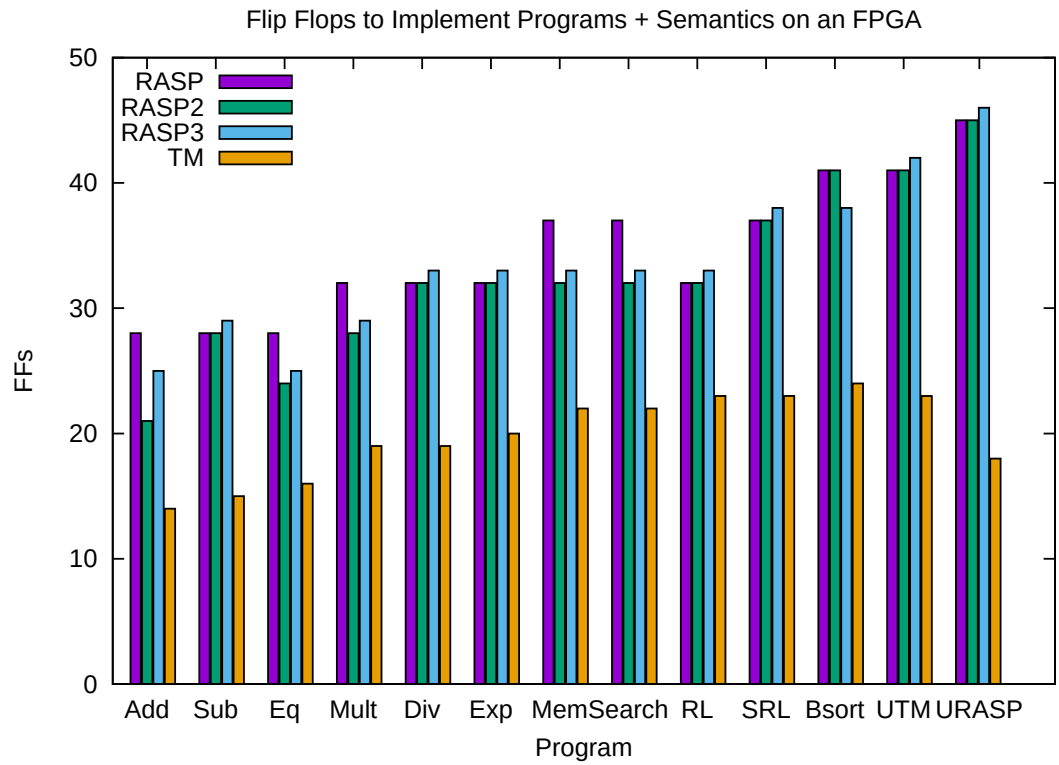


Figure 6.8: FFs for RASPs and TM

To properly evaluate the Semantic Circuit (SC) hypothesis, the components to implement the programs and data are required to be separated from the control units of the RASP and TM implementations. The RASP FPGA implementations have highly coupled control units and memory; each semantic rule holds numerous pre- and post-conditions on the state of the memory. Since the state machine for the RASPs also performs switches on the data in memory, the memory has to be able to hold at least eight values for the eight instructions of the machine. Furthermore, any value in the memory could be an address, so the memory must be addressable by eight distinct values.

This inherent dependency between data and memory size restricts us to a lower bound on memory size for RASPs at eight. Any lower and the machine either cannot address memory locations, or the synthesis tool optimises out parts of the RASP state machine that cannot be run because the required instruction cannot be held in memory.

The compromise is a flat comparison of the three RASP machines with memories of size eight. The FPGA FPGA utilisation report provided by the compiler shows the number of components to implement the control module of the models.

	Slice Registers	LUTs	FFs
RASP	21	48	21
RASP2	21	50	21
RASP3	22	63	22
TM	10	7	10

Table 6.13: Components to implement semantics

Table 6.13 displays the number of components required to implement the minimal state machines (and memories) of the models. The RASPs were all measured with an empty memory of size 8, and the TM had a single tape cell and a single tuple in the symbol table.

6.3.1 RASPs on FPGAs

The Semantic Circuit hypothesis (SC, hypothesis 3) states: "Consider two models A and B. If model A has larger semantics than model B, the FPGA circuit which realises the semantics of A will be larger than the FPGA circuit for B." (Section 3.1.3). In essence, as the semantics get more expressive, more LUTs, flip-flops, and slice registers are required to represent the semantics in hardware.

The semantics of the vanilla RASP are smaller than the semantics of the RASP2, which in turn are smaller than those of the RASP3. The data in Tables 6.9 – 6.12 is consistent with prediction 3.2, and supports sub-hypotheses with respect to the SC within family (3a), and SC within paradigm (3b).

The slice registers/flip-flop counts (Tables 6.9 and 6.12) show that the RASP and RASP2 are equal in size, with the RASP3 only requiring one extra slice register.

The LUT counts in Table 6.10 show that the RASP2 semantics are larger than the RASP semantics while the RASP3 semantics are larger than the other two. This falls into line with what would be expected given the relationship of the SOS sizes. Because the LUTs primarily implement random logic and slice registers are typically purposed for state variables/memories, there is more of an inclination to weigh the LUT count over the register count with respect to the rules of the semantics. Prediction 3.2 is therefore satisfied, and sub-hypothesis SC within family (3a) is confirmed.

The Total Circuit Size hypothesis (TC, hypothesis 4) states: "For two models

	RASP		RASP2		RASP3	
	Slice R	LUTs	Slice R	LUTs	Slice R	LUTs
Arith Mean Arithmetic	30	70	27.5	68.17	29	79.67
Geo Mean Arithmetic	29.93	69.89	27.2	67.38	28.81	79.2
Arith Mean List	36.80	81.40	34.8	83.80	35	95.4
Geo Mean List	36.69	81.24	34.62	83.55	34.92	95.25
Arith Mean Arithmetic + List	33.09	75.18	30.82	75.27	31.73	86.82
Geo Mean Arithmetic + List	32.83	74.84	30.35	74.30	31.44	86.13
Arith Mean All	34.69	77.54	32.69	79.38	33.62	91.54
Geo Mean All	34.28	77.06	32.01	77.99	33.11	90.33

Table 6.14: Arithmetic and geometric means of RASPs on FPGA

A and B, where the circuit implementation of the semantics of A is larger than the circuit for the semantics of B. As a function grows in complexity, the average total implementation size of a succinct realisation of the function in model A will reduce relative to the average for model B." (Section 3.1.3). The RASP specific hypothesis is the TC within family hypothesis 4a and prediction 4.1 sets out what we expect to observe.

Table 6.14 shows the arithmetic and geometric means of the RASP programs. Unlike the TI Tables 6.5 and 6.6, there is no trend in number of LUTs or slice registers which shows the RASP3 requiring less components on average than the RASP or RASP2. Where considering all functions, the TI of the RASPs conformed to the relation: RASP3<RASP2<RASP, the TC of the FPGA realisations for all functions is: RASP<RASP2<RASP3 for the LUTs, and RASP2<RASP3<RASP for the slice registers. This evidence contravenes the TC within family sub-hypothesis. The reduction in average slice registers provides an indication of smaller programs for the RASP2 and RASP3 relative to the RASP, but the LUT relationship remains consistent.

The plots of slice registers and LUTs shed some light on why this is the case. The slice registers for the programs in Figure 6.6 show the RASP3 and RASP2 following roughly the same plot. The exceptions are the addition function, where RASP2 uses less memory than the RASP3, and the bubble sort, where RASP3 uses less. The RASP2/3 plots are below the RASP plot when the RASP2/3 use less memory than the RASP, otherwise they use slightly more.

The LUTs for the machines (Figure 6.7) also show the RASP2 and 3 following

the same plot, again where the RASP3 has an overhead on top of the RASP2. The bubble sort, where the RASP3 has a smaller program than the other RASPs, is slightly reduced but not to the extent where the hypothesis would be considered confirmed.

The RASP data here is strong evidence for the confirmation of the SC hypothesis (Hypothesis 3) via the within family sub-hypothesis. The analysis also finds evidence which contradicts the TC hypothesis. The RASP3 having a larger semantic circuit does not imply that the total number of components required for programs will be lower than the components required for the RASP2 and RASP implementations. This evidence contravenes both the within family and within paradigm hypotheses (4a and 4b).

6.3.2 RASP vs TM

Contrasting the data of the TM against that of the RASPs. If the SC hypothesis were to hold, we would expect that can find evidence which is predicted by 3.2, which states that since the TM semantics are smaller than the RASP semantics, the TM semantic circuit will be smaller also. Table 6.13 shows that the TM uses less slice registers, LUTs, and FFs to represent the semantics. This satisfies prediction 3.2 and supports the SC within paradigm sub-hypothesis.

The implementations and character-wise measurements of the various programs in TM with respect to the RASP measurements (Table 6.4) show that, excepting addition, the TM programs are larger than any of the RASPS. If the TC hypothesis holds, then it is expected that the mean number of components to implement the

The abstract implementations the models in SOS and their associated programs show the TI of the TM growing rapidly relative to the RASP machines. With the exception of the addition function, the TI of the TM is greater than that of the RASPs.

In contrast, the number of components to implement the TMs on the FPGA is much lower than than of the RASPs. With the exception of the number of LUTs required to implement the bubble sort, UTM, and URASP, the TM values are always lower than the RASP component numbers. The TC within paradigm sub-

3. Strong Semantic Circuit hypothesis
   3a. SC within family hypothesis **For:** 6.3.1
   3b. SC within paradigm hypothesis **For:** 6.3.2
4. Strong Total Circuit hypothesis
   4a. TC within family hypothesis **Against:** 6.3.1
   4b. TC within paradigm hypothesis **Against:** 6.3.2

Figure 6.9: FPGA hypotheses and evidence for each

hypothesis 4b, like the TC within family sub-hypothesis 4a, cannot be confirmed by this data.

6.3.3 The SC and TC Hypotheses

Figure 6.9 lists the evidence gathered for each sub-hypothesis and the section where that evidence is found. The Semantic Circuit (SC) Hypothesis is concerned with the FPGA realisations of the semantics and programs of the RASP and TMs. The hypothesis states that if model A has more semantic information (as measured by the size of the SOS implementation) than model B, then the FPGA circuit which implements the semantics of model A will be larger than the circuit to implement the semantics of model B.

This hypothesis is verifiable using the semantics sizes taken from Table 6.13. For the RASP and Turing machines, the SOS sizes of the semantics follow the relation: TM<RASP<RASP2<RASP3 (Table 6.4), and this relation is mirrored in the semantic circuit sizes. The LUTs largely implement the state machines of the control units, while slice registers are dedicated to state information and the memories of the machines. From examining the table, the number of slice registers and LUTs show that the TM has the smallest circuit size (Section 6.3.2), followed by the RASP, RASP2, and then RASP3 with the largest (Section 6.3.1).

These observations satisfy the within family (3a) and within paradigm (3b) sub-hypotheses in order to confirm the SC hypothesis.

The TC hypothesis is analogous to the TI hypothesis. The Total Circuit hypothesis predicts that as the size and complexity of a program, or programs, increases the total circuit size of a succinct implementation of the program(s) in an expressive model will reduce relative to the implementations in a less expressive

model.

This hypothesis is not confirmed at all. Within the RASP family (Section 6.3.1), there is no indication of the average number of LUTs or slice registers reducing relative to the RASP2. The RASP relation sits at RASP<RASP2<RASP3 for average number of LUTs, and RASP2<RASP<RASP3 for slice registers (Table 6.14).

Comparing the TM TC size to the RASP TC size (Section 6.3.2) shows that the total circuit sizes for the RASP tend to be much lower than the total circuit sizes for the RASPs. Only the bubble sort and universal RASP programs in the TM require more LUTs than the corresponding RASP programs. As a result, the TC hypothesis cannot be confirmed.

It should be considered *why* the TC hypothesis cannot be confirmed for two models in the same paradigm as the TI hypothesis. The abstract realisations of the semantics of the models are isolated relative to the programs which are measured. Once the author of a semantics is satisfied that the semantics are correct, they are bundled with programs of all sizes to measure and obtain the TI.

It is clearly practical to do so. A semantics has no regard for size bounds. If size were to be regarded, a different semantics would be required for each program unless the programs happened to be the same size as some other. Rather, structures in the semantics are defined via types – which are sets which can be bounded or unbounded in size. For instance, the memory of a RASP is defined as a size 2^n list of numbers, with each number between 0 and $2^n - 1$. The type of the memory structure is $\mathbb{N}$ which denotes the natural numbers. The exponent n is also a natural number, so the RASP model permits memories of size 2^0 up to an arbitrarily large value of n without the need to change the semantics because set theory permits infinite sets.

The real world is unfortunately not as flexible. The semantics for the FPGAs are defined with fixed sizes for the RASP memory, TM symbol table, or TM tape so that the compiler can allocate the appropriate level of resources to represent these memories or structures. Furthermore, the rules have a less 'functional' implementation in the FPGA semantics and therefore require the use of temporary

	RASP		RASP2		RASP3		TM	
	Slice R	LUTs	Slice R	LUTs	Slice R	LUTs	Slice R	LUTs
Addition	7	18	0	1	3	7	4	6
Subtraction	7	18	7	20	7	15	5	6
Equality	7	18	3	10	3	7	6	9
Multiplication	11	26	7	20	7	15	9	13
Division	11	26	11	29	11	28	9	15
Exponentiation	11	26	11	29	11	28	10	23
List Membership	16	33	11	29	11	28	12	37
Linear Search	16	33	11	29	11	28	12	42
Reverse List	11	26	11	29	11	28	13	25
Stateful Rev List	16	33	16	36	16	39	13	73
Bubble Sort	20	42	20	46	16	39	14	143
Universal TM	20	41	20	46	20	49	13	188
Universal RASP	25	44	24	58	24	60	9	1012

Table 6.15: Components for programs only on FPGAs

variables which also have to grow in size to correctly store intermediate values of the execution.

This creates an overhead in the FPGA realisations where the size of the semantics increases proportionally to the size of the program being executed. Assuming that the semantics sizes in the FPGA realisations are fixed according to Table 6.13, the number of semantic components can be subtracted from the TC component values to obtain the program information analogue for the FPGAs in Table 6.15.

The overhead of the semantic growth is rolled into the FPGA program informations. The list functions in this table show that the program information for the TM is often higher than that of the RASPs with respect to the number of LUTs, and very close to the RASPs when considering the slice registers. From this perspective, if the complexity of the functions were to smoothly grow, the eventual average TC of the TMs would become lower than that of the RASPs.

The RASP3 has a smaller implementation of the bubble sort than the other models and this is reflected in the LUT and slice register counts. This shows that the reduction in the number of required components for the RASP3 implementation can conceivably outweigh the extra components required for the semantic overhead. It is hypothesised that given more complex functions, if the RASP3 implementations were to keep reducing in size relative to the other RASPs as in-

dicated in Section 6.2.1, then the TC size with respect to the RASP2 and RASP implementations will drop as evidenced by the TI figures.

In conclusion, this analysis finds that the number of a specific component (LUTS for TM and slice registers for RASPs) is an indicator of the TI relative to the TI of other programs implemented in that model. Conversely, using said component counts to analyse TI measurements *across* models does not work. A reason for this is the growing overhead of the semantics implemented on the hardware. In the abstract semantics an infinite set can be designated for all programs to use, but in these concrete realisations, the sets must be bounded and have to grow according to the size of the program implemented.

6.4 Further Observations

This section discusses the dramatic increase in required information for the SKI and TM when representing the Universal functions opposed to the RASP and λ-calculus. It also considers how the use of parsing semantics affects the information measurements made.

6.4.1 Model Attributes

Figures 6.10 to 6.13 show plots of the geometric and arithmetic means of the PI's and TI's. The geometric plots show the normalising effect of the geometric mean process and bunches the models together.

The arithmetic mean plots are more interesting. The RASP machines are bunched together much like in the geometric mean graph, which is not surprising due to their operational similarity. But the λ-calculus is also grouped with the RASP machines. Furthermore, the SKI and TM plots are separated from the RASP and λ grouping, and are correlated together.

The SKI expressions are derived from the λ-calculus expressions via bracket abstraction (Section 2.3.2.2). The TM programs are not derived from, nor have any direct translation to the corresponding RASP program. Despite this, the RASP and TM figures show the same separation as from the SKI and λ-calculus. The TM and SKI numbers correlate very strongly with Pearson's r between 0.985

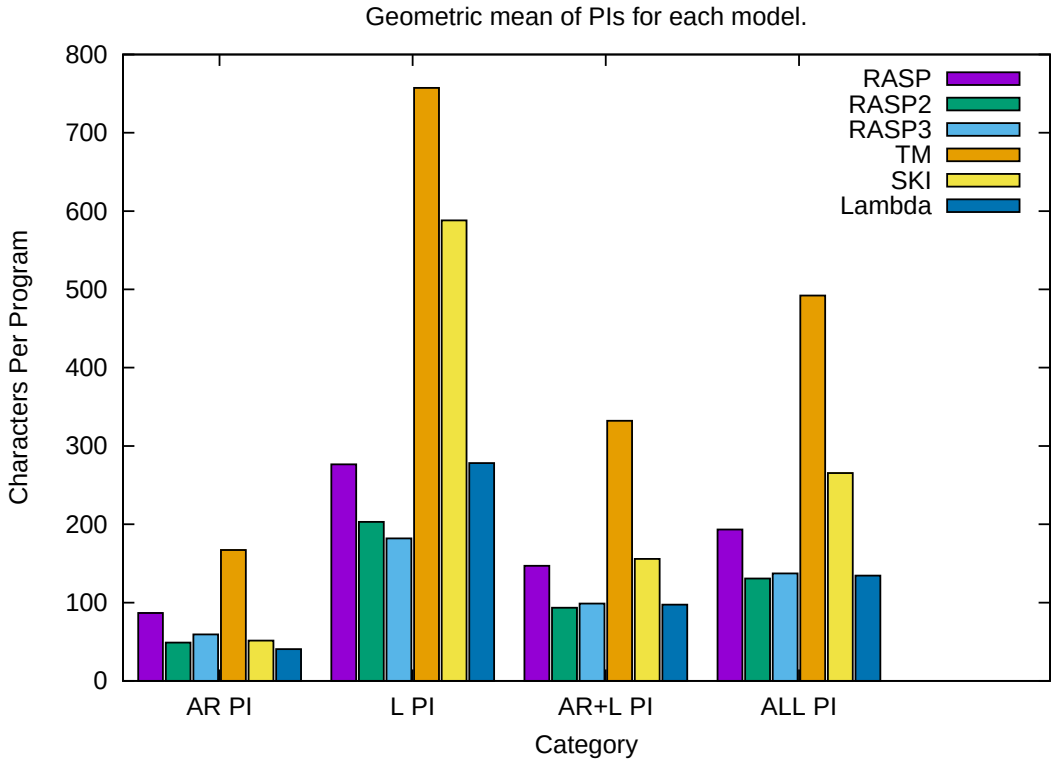


Figure 6.10: The PI geometric means from Table 6.6

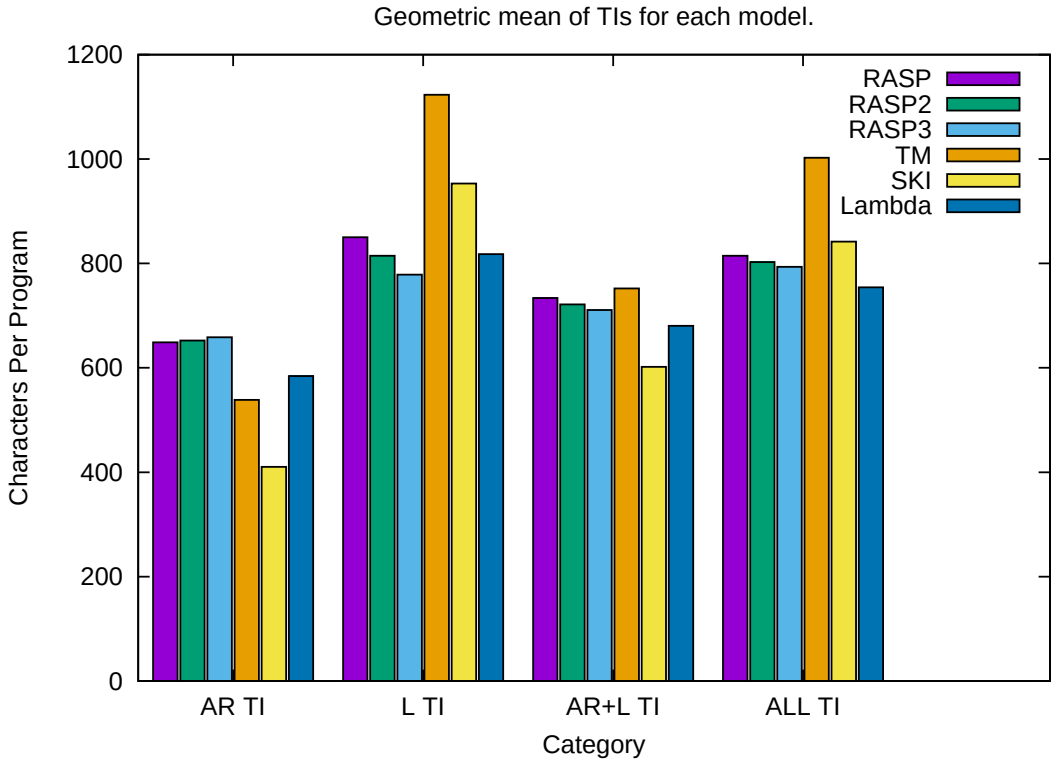


Figure 6.11: The TI geometric means from Table 6.6

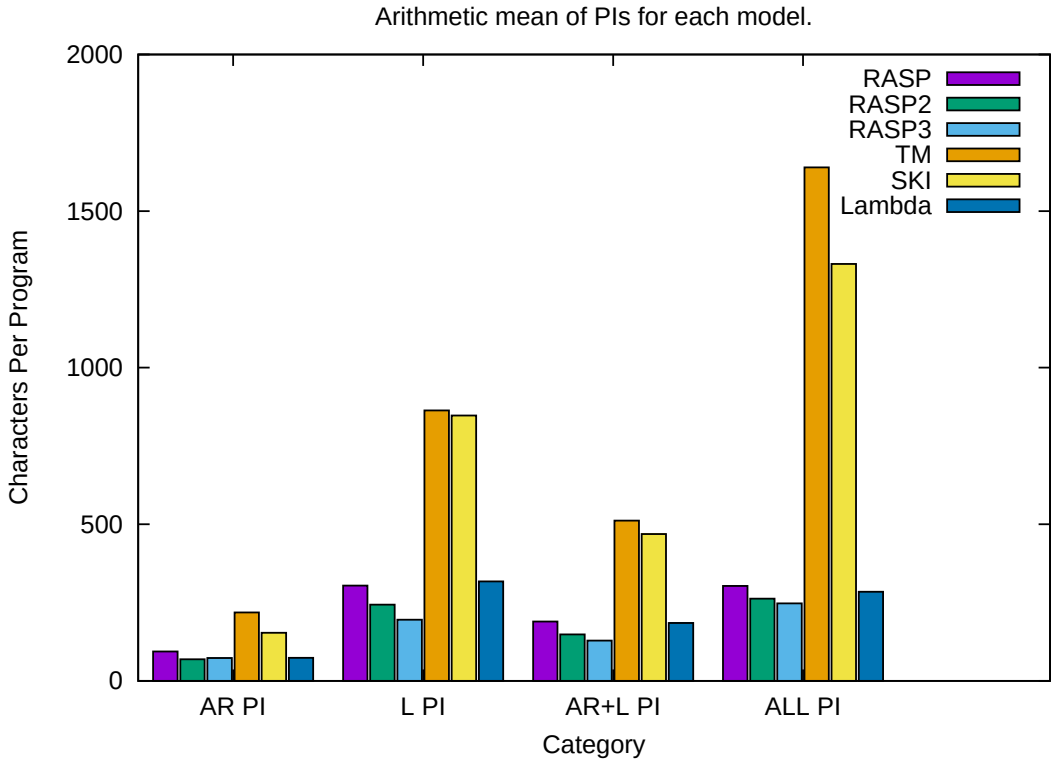


Figure 6.12: The PI arithmetic means from Table 6.5

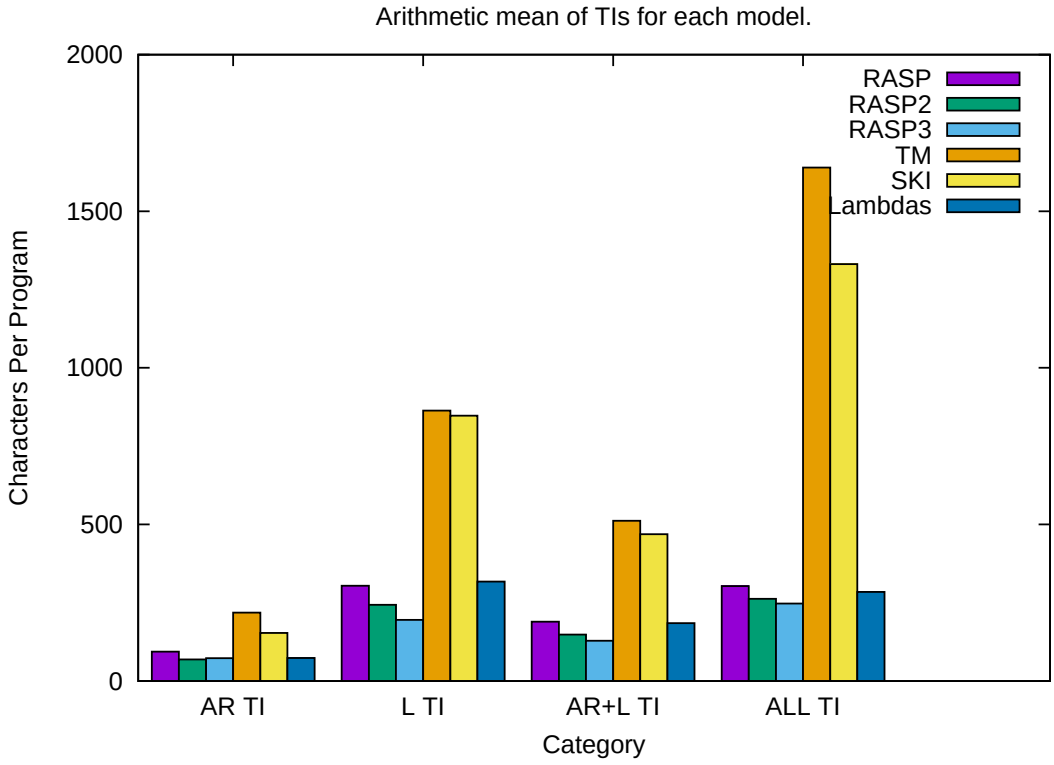


Figure 6.13: The TI arithmetic means from Table 6.5

	RASPs+λ	TM+SKI	Factor
AR	55.74	177.56	3.19
L	150.08	602.45	4.01
AR+L	143.42	537.08	3.74
U	327.20	6157.32	18.82
ALL	319.17	3233.37	10.13

Table 6.16: Standard deviations of the sample of RASPs+λ-calculus vs the TM+SKI.

and 0.99 for the above plotted datasets.

The separately correlated data points in Figures 6.12 and 6.13 are thought to be due to attributes of the models which do not affect the computational power of the models.

It is to be expected that the less expressive models will have some overhead in the representation of the program set. After all, the intuition of expressivity laid out in the introduction is supported by the data gathered. Table 6.16 shows the standard deviation of the sample for the two groups of data points. The deviations for the TM+SKI data points is about 3 times that of the RASP+λ points for the arithmetic functions 4 times for the list functions and 18 times for the universal functions.

Combining the program categories produces a factor of 3.74 for AR+L, and 10.13 for the entire set. This suggests that there is a difference between the data points for the universal functions which is above the norm shown by comparisons of the AR and L function sets.

The RASPs and λ-calculus both have some form of random access which merely speeds up memory access times. The RASPs have random access memory and the λ-calculus has variables which can be substituted using β reduction. TM and SKI do not. The TM has to sequentially shift the tape and the SKI has to repeatedly evaluate combinators at the far left hand side to move applied expressions into each other which the λ-calculus achieves though abstraction and substitution alone.

Adding more semantic operators for TMs or SKI which enable random access, such as a TM search which returns the first occurrence of a particular symbol to the right or left of the head position, is hypothesised to adjust the mean values

	RASP	RASP2	RASP3	TM	SKI	λ-Calc
Parsing Semantics	71	71	71	203	101	162
Evaluation Semantics	484	513	515	146	190	381
Ratio Parsing:Eval	0.15	0.14	0.14	1.39	0.53	0.43

Table 6.17: The ratio of parsing semantics to evaluation semantics for each model.

such that they converge to those of the RASPs and λ-calculus. Section 7.3.4 presents a hypothesis to guide investigation into this observed separate correlations of PI/TI and discusses the possibility of other separations precipitated by other model attributes.

6.4.2 Interpretation vs Evaluation Semantics

The comparisons which have been explained thus far have been made relative to the entirety of the semantics for each model. A program has been written in some external representation, converted into the internal representation using the parsing semantics, and evaluated with the evaluation semantics.

The parsing semantics do not add any computational power to the models. A different perspective could be gain through comparing only the size of the evaluation semantics of the models with the size of programs. Table 6.17 compares the size of the parsing semantics with the evaluation semantics. Note that the sum of the parsing and evaluation semantics is often greater than the presented sizes in Table 6.4 and in the rest of this thesis. This is because both the parsing and evaluation parts may share a function or definition which has to be defined for both when the semantics are split.

The external and internal representations of the RASP machines are very similar, so there is little overhead in parsing programs. The parser pattern matches natural numbers from the left hand side adding them to the mapping which makes up the initial memory of the program.

The SKI and λ-calculus have a more complicated parsing procedure which converts the linear external representation into the tree-like internal representation. The conversion procedure for both models is similar. The expression is pattern matched from the right hand side and the tree is recursively constructed from the root. In SKI, internal tree nodes denote applications with combinators as leaves.

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
AR PI	86.71	48.95	59.27	167.15	51.52	40.62
AR TI	576.72	579.98	586.35	341.12	301.19	448.82
L PI	276.67	202.98	181.93	757.23	588.18	278.13
L TI	777.44	741.38	706.54	919.60	832.81	681.07
AR+L PI	146.93	93.44	98.68	332.16	155.84	97.39
AR+L TI	660.57	648.46	638.21	535.40	478.20	542.50
All PI	193.22	130.78	137.21	492.16	265.52	134.54
All TI	739.42	727.23	718.52	743.90	690.48	612.10

Table 6.18: Geometric means of the program sets using evaluation semantics

	RASP	RASP2	RASP3	TM	SKI	λ-Calculus
AR PI	93.83	68.83	73	218.5	153.50	73.83
AR TI	577.83	581.83	588	364.5	343.50	454.83
L PI	304.4	243.6	195.4	863.4	847.4	317.6
L TI	788.4	756.6	710.4	1009.4	1037.4	698.6
AR+L PI	189.55	148.27	128.64	511.64	468.91	184.64
AR+L TI	672.82	661.27	643.64	657.64	658.91	565.64
All PI	302.85	262.38	247.69	1639.38	1331.15	284.54
All TI	786.85	775.38	762.69	1785.38	1521.15	665.54

Table 6.19: Arithmetic means of the program sets using evaluation semantics

The λ-calculus parses both applications and abstractions as internal nodes and uses variables for the leaves. As mentioned in Chapter 3, this transformation is to facilitate graph reduction where nodes are swapped when sub-expressions move around the term.

The TM parsing semantics presented here are larger than the evaluation semantics. In contrast to the RASPs and functional models, a TM definition is in two parts; a symbol table and a tape. Both of these have to be parsed and they are both done in a different manner. The symbol table is pattern matched for the discrete elements of the tuples which are combined into a mapping to create the symbol table. If the tape contains a caret (^) the symbol to the left is mapped to zero in the tape function and the rest of the function is filled in recursively left and right, which are mappings to negative and positive integers respectively. This necessitates the creating of multiple rules with specific functionalities which are difficult to generalise. If the ability for the TM to start at an arbitrary point on the tape were to be removed, three of the parsing rules could be removed.

Tables 6.18 and 6.19 show the means of the the program sets when parsers

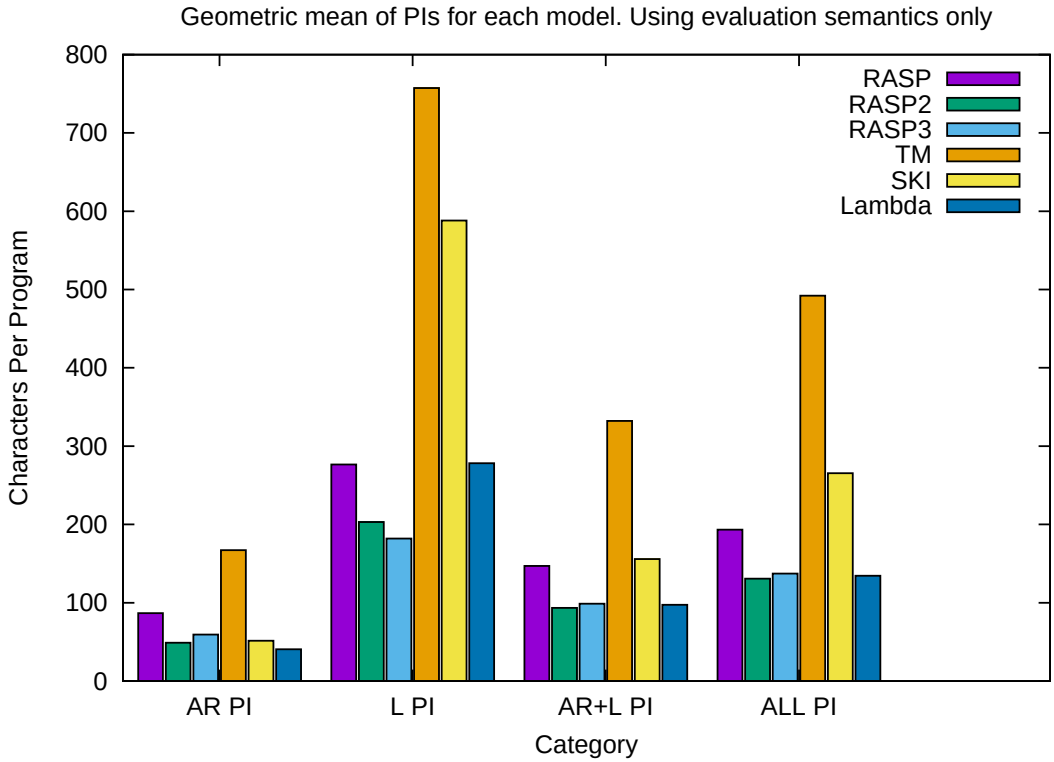


Figure 6.14: Plot of PI geometric means from Table 6.18.

are discounted. Removing the parsing semantics from consideration results in less homogeneity in the means between the models. Figures 6.14 to 6.17 show the plots of these means.

Comparing these plots to the arithmetic and geometric mean plots of the full semantics, there is not a dramatic difference. The arithmetic plot shows the TM and SKI closer together and the λ-calculus TI means trending downwards, further from the means of the RASPs. The apparent separation between SKI/TM and RASPs/λ-calculus is still observable which is encouraging in that it is not simply an artefact of the inclusion of parsers.

The geometric plot notably shows the smoothing of the TM curve and the eventual lowering of the geometric mean of all TM programs to below those of of the RASPs. The λ-calculus and SKI have the most and second most minimal information contents of all of the models under the geometric mean. This data further reinforces our assertion that hypothesis 2c is incorrect as the semantics of the functional models are now much smaller than the RASP and still maintain an overall lower TI.

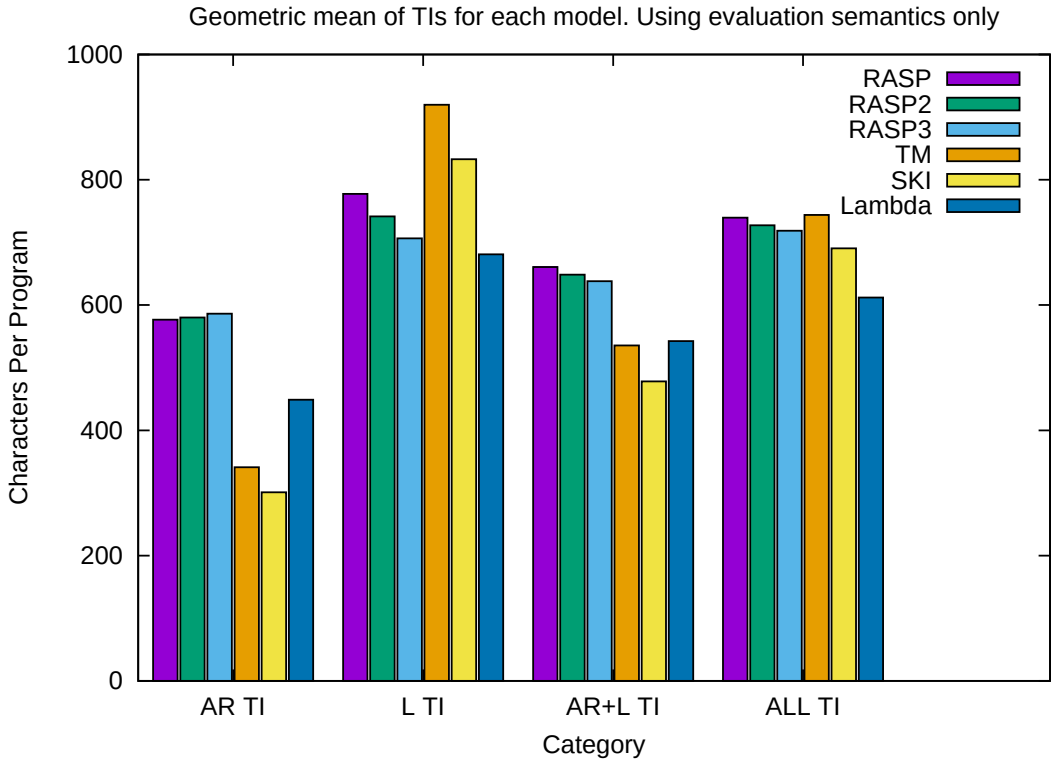


Figure 6.15: Plot of TI geometric means from Table 6.18.

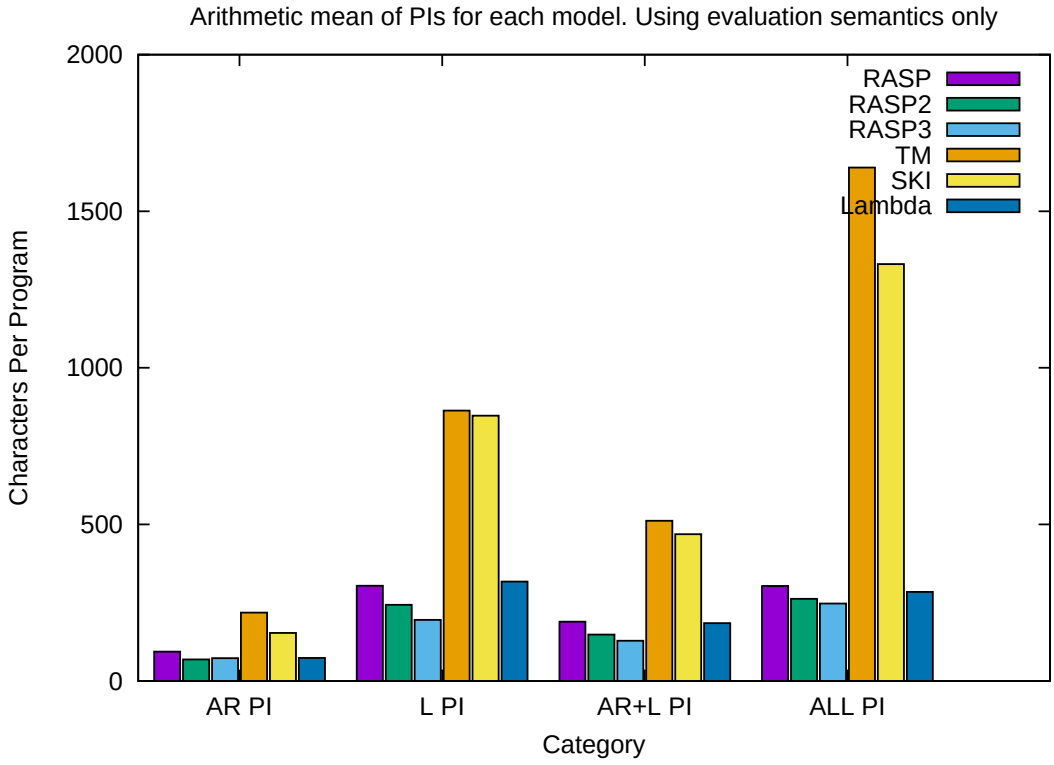


Figure 6.16: Plot of PI arithmetic means from Table 6.19.

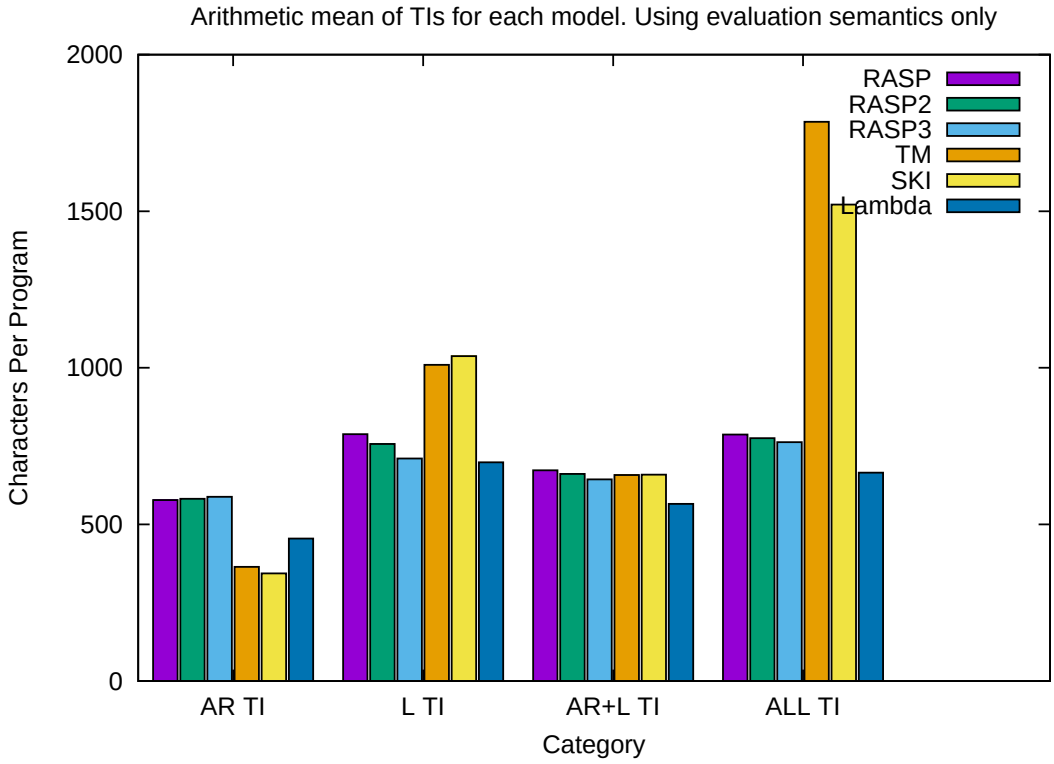


Figure 6.17: Plot of TI arithmetic means from Table 6.19.

6.5 Inputs

The measurements made and hypotheses evaluated thus far have considered only the size of the semantics and programs. Section 3.4 has made the case for the 'parsing semantics' to be included in the overall semantic sizes comparisons. In essence, the programs for these models are all commonly expressed in a linear fashion, while the structure of λ-calculus and SKI expressions which are actually evaluated may be very different. These expressions are linear, but their linearity belies their tree structure which is directly manipulated to evaluate the expressions via graph reduction (Section 3.4.3). Therefore there has to be some semantic rules to convert the linear external representation into the tree-like internal representation.

In a similar way, expressions and programs written for a model parse inputs from the external, into internal representations and evaluate them. Information for computation is hierarchical and regressive. Programs are bespoke semantics and models to compute specific functions. The most general of these functions are universal which have their own language/encoding for their inputs.

It is these program specific languages which are focused on now. The programs presented in Chapter 4 assumed natural encodings for the inputs and these encodings are measured directly in characters and asymptotic notation, Big O [86, 32], will be supplied for these.

6.5.1 RASPs

The floored logarithm to base x of n: $\lfloor log_x(n) + 1 \rfloor$ is a measure of the number of characters required to represent the base 10 number n in the base x numeral system. While the PI of the RASPs includes registers to hold the inputs for the program, the registers measured only hold single digits and are the minimal number of registers required to constitute an input (only two element lists for example).

The RASPs represent all of their inputs in base 10. Inputs are either discrete digits x and y, or a list of k elements with t as the largest number in the list. Furthermore, the PI of the RASPs grow as any of these variable grow in size. Recall that a 2^n length RASP can only hold a numeral from 0 to $2^n - 1$.

Assuming that all inputs for a 2^n RASP are numerals between 0 and $2^n - 1$, the arithmetic functions have two inputs x and y. The number of characters for these inputs is determined by the log rule:

$$\lfloor log_{10}(x) \rfloor + \lfloor log_{10}(y) \rfloor + 2 = \lfloor log_{10}(xy) \rfloor + 2$$

In big-O this is shortened to $O(log_{10}(xy))$ because the input size is dependent on both of the mutually independent variables x and y.

Lists in the RASP are a contiguous array of k registers. At least one register holds the numeral t, where t is the largest numeral in l. The list size is therefore bounded via the function:

$$k \times (\lfloor log_{10}(t) \rfloor + 1) \in O(k\ log_{10}(t))$$

The list membership and linear search functions also require a target value as input which could possibly be as large as t, which adds another $\lfloor log_{10}(t) \rfloor + 1$ characters.

The UTM is arranged as an encoded symbol table followed by a tape of symbols. The UTM in RASP can simulate a TM with s states and t symbols. A tape of k symbols requires $k \times (\lfloor log_{10}(t) \rfloor + 1) \in O(klog_{10}(t))$ characters. The symbol table is a list of $\langle S_o \rangle \langle Sy_o \rangle \langle S_n \rangle \langle Sy_n \rangle \langle D \rangle$ quintuples terminated with a 0 value (Section 4.4.1).

$$
\begin{aligned}
& s \times t(\lfloor log_{10}(s) \rfloor + \lfloor log_{10}(s) \rfloor + \lfloor log_{10}(t) \rfloor + \lfloor log_{10}(t) \rfloor + 4 + 1) + 1 \\
& = s \times t(\lfloor log_{10}(s^2 t^2) \rfloor + 5) + 1 \\
& \in O(s \times t \; log_{10}(s^2 t^2))
\end{aligned}
$$

Pairing the symbol table with the tape expression gives:

$$O(k \; log_{10}(t)) + O(s \times t) \in O(k \; log_{10}(t) + s \times t \times log_{10}(s^2 t^2))$$

Which is the final growth rate upper bound of TM expressions in the RASP UTM.

A RASP to be simulated by the universal machine is a list and grows according to the number of bits n for that machine. Again, there is a value t which is the largest figure in the simulated machine. The specific equation is similar to the list growth equation above, however k is replaced by the growth expression of 2^n:

$$2^n \times (\lfloor log_{10}(2^n - 1) \rfloor + 1) \in O(2^n \; log10(2^n - 1))$$

6.5.2 TM

The arithmetic functions of the UTM take unary inputs on their tape. Thereby, the number x requires x symbols to represent. For two variables, the growth rate is bounded by the sizes of both: $O(x + y)$.

Lists are a delimited array of binary numbers which come in two variants; $\#\langle addr * data \rangle \ldots$ and $\langle data1 \rangle * \langle data2 \rangle \ldots$ These lists hold binary numbers where t is the largest number in the list, and k is the number of elements. Both lists

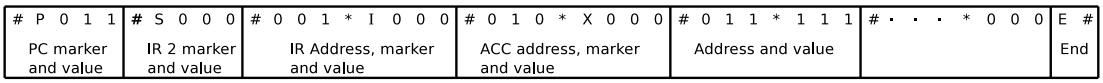


Figure 6.18: A 3 bit RASP arranged on a TM tape.

are terminated with a single symbol.

$$\begin{aligned} \#\langle addr * data\rangle \ldots &= k(log_2(k) + log_2(t) + 4) + 1 \in O(k\ log_2(k \times t)) \\ *\langle data1\rangle \ldots &= k(log_2(t) + 2) + 1 \in O(k\ log_2(t)) \end{aligned}$$

The linear search requires address/data pairs so the input size growth is bounded by $O(k\ log_2(k \times t))$. The other list functions require data only lists, so their input size growth is bounded by $O(k\ log_2(t))$

The UTM is covered in detail in Section 6.6. The universal RASP is represented on the tape as a list of $2^n - 1$ $\langle addr\rangle * \langle data\rangle$ pairs. The data for the PC has no address, and there is an additional IR which is used in the case of an instruction requiring a parameter (Figure 6.18).

For a size 2^n machine, each register is represented by two n-bit numbers. Each pair of numbers is prefixed and separated by a single symbol (#,*), two symbols end the memory and four of the n-bit numbers use a special symbol to indicate that they are registers used in the F-E cycle. Thus the number of characters to represent an n-bit RASP is:

$$\begin{aligned} 2^n(2(log_2(2^n) + 1) + 2) + 6 &= 2^n(2(n + 1) + 2) + 6 \\ &= 2^n(2n + 4) + 6 \\ &\in O(2^n) \end{aligned}$$

6.5.3 λ-Calculus

The magnitude of a Church numeral in the λ-calculus is the number of times the first argument is applied to the second. Aside from the numeral for zero, the number of characters to represent the Church numeral (n) is: $3n+8$. The numeral for 0 is 9 characters in size. For proper application, the numerals are externally bracketed. The numeral 3 is $(\lambda f.\lambda x.f(f(fx)))$. Arithmetic functions all have two numerals x and y as inputs, so the number of characters is $(3x + 8) + (3y + 8) \in O(x + y)$.

Lists in the λ calculus are lists of Church numerals. Each element of the list is a Church numeral paired with another list, or with the NIL expression. The NIL expression is 12 characters long and PAIR is 16, not counting the two external brackets which enclose the expression (PAIR p q). If k is the list length of the list and t is the size of the largest numeral, then the expression for the size of a list is bound by the expression:

$$18n + kt + 12 \in O(k \times t)$$

The membership and linear search expressions also require a single numeral, which could possibly be of size t, to search for; $O(t)$.

The UTM in the λ-calculus is a list of quintuples (5 element lists without a NIL terminator) for the symbol table, and a list of Church numerals for the tape. A quintuple consists of two numerals for states, two numerals for symbols and a numeral for direction. The largest state is s, largest symbol is t, and largest direction is ONE (11 characters). With s states and t symbols, the number of quintuples in the table is $s \times t$, and the size equation for the symbol table is:

$$\begin{aligned} s\ t(2(3s + 3t + 16) + 11 + 5(16 + 2)) + 12 &= s\ t(2(3s + 3t + 16) + 11 + 5 \times 18) + 12 \\ &= s \times t \times (2s + 2t + 101) + 12 \\ &\in O(s^2 \times t + s \times t^2) \end{aligned}$$

The tape is a list, so it conforms to the size equation for lists $O(k \times t)$, where k is the length of the tape, and t is as above. The upper bound of the entire input to the UTM in λ-calculus is $O(t(s^2 + t + k))$.

The universal RASP takes two inputs: a list of numerals of size 2^n, and an output vector which is to be populated by occurrences of the OUT instruction; which defaults to NIL. The numerals in the list can be have a maximum size of $2^n - 1$, so the numeral size is bounded by $3(2^n - 1) + 8$. Each RASP has a memory of 2^n, so there are 2^n occurrences of PAIR and a numeral, which one NIL to terminate the list. A RASP machine is bounded in terms of bits with the

equation:

$$\begin{aligned}&2^n(18 + 3(2^n - 1) + 8) + 2 \times 12 + 2\\&= 2^n(24 + 3(2^n - 1)) + 26\\&\in O(2^n)\end{aligned}$$

6.5.4 SKI

As has been the convention throughout the thesis, the SKI expression have been derived from the λ expression via bracket abstraction. It is therefore expected that the asymptotic growth of SKI inputs mirrors that of the λ-calculus. The specific size equations will be different however.

The number of characters required to represent a numeral $f > 2$ in SKI can be calculated as: $11f - 1$. The numeral for 0 is KI, 1 is I, and 2 is S(S(KS)K)I. Arithmetic operations over numerals x and y are thus $11(x + y) - 2 \in O(x + y)$

Lists are constructed pairwise and terminated with the SKI NIL expression. PAIR is 37 characters long, not counting the enclosing brackets. NIL is two characters in length. If k is the number of elements in a list, and t is the largest numeral, then an input for the list function is:

$$k(11t - 1 + 39) + 2 \in O(k \times t)$$

As with the other models, the SKI requires a further numeral as input for the list membership and linear search functions.

The UTM is a list of quintuples and a list of numerals for the symbol table and tape respectively. As with the λ-calculus, symbol table entries are tuples with five elements and no NIL terminator. There are two numerals for state (possibly state s), two numerals for symbols (possibly t), and a numeral for direction (either 0 or 1). Using s states, and t symbols the symbol table of a TM in SKI is sized as:

$$\begin{aligned}&st(4 \times 39 + 2(11s - 1) + 2(11t - 1)) + 12\\&= st(152 + 22(s + t)) + 12\\&\in O(ts^2 + s \times t^2)\end{aligned}$$

The tape of the UTM is a list of k elements and up to t symbols: $O(k \times t)$. The input size of the UTM is therefore bounded by $O(t(s^2 + t + k))$.

	RASPs	TM	SKI λ-Calculus
Arithmetic	$O(log_{10}(xy))$	$O(x+y)$	$O(x+y)$
List Membership	$O(k\ log_{10}(t))$	$O(k\ log_2(t))$	$O(kt)$
Linear Search		$O(k\ log_2(kt))$	
List Reversal		$O(k\ log_2(t))$	
Stateful List Rev		$O(k\ log_2(t))$	
Bubble Sort		$O(k\ log_2(t))$	
Universal TM	$O(k\ log_{10}(t) + stlog_{10}(s^2t^2))$	$O(s(log_2(s))^2 + k)$	$O(t(s^2 + st + k))$
Universal RASP	$O(2^n)$	$O(2^n)$	$O(2^n)$

Table 6.20: Big O notation of input size growth rates

The RASP machine encoded for the SKI is a list to represent the memory of the machine, and an initially empty vector for outputs. An n-bit machine has 2^n registers and each can hold a maximum number of $2^n - 1$:

$$2^n(39 + (11(2^n - 1) - 1)) + 4 \in O(2^n)$$

6.5.5 Comparison

Table 6.20 shows the big-O notations of the input growth rate. The variables x and y are numbers, k is the length of a list, t is the largest element of a list or number of symbols in a TM, s is the number of states in a TM, and n is the number of bits in a RASP machine.

These rates indicate the how the size of encoded input information changes depending on the size of unencoded inputs. It is useful to expose the advantages inherent to the encoding system of a model.

For example, the RASP uses the set of natural numbers in its semantics to evaluate machines because all of the RASP operations are defined over the set of natural numbers. This in turn makes makes the natural numbers (and the successor/predecessor operations) implicit information within the semantics of the RASP (nowhere are the naturals defined in the semantics).

Because the RASP operators are defined over the natural numbers, there is an injective mapping from the external representation to the internal representation. And because arrays of natural numbers are versatile enough to represent many different inputs, the encodings are consequently relatively succinct.

By virtue of the base 10 representation of natural numbers, the RASP has

overall the slowest growing inputs for the functions. The TM uses unary encoding for the arithmetic functions, and binary encodings for the other functions. Without another numerical base to use, the representations of the input Church numerals is linear throughout the entire set of functions.

Encodings for the URASP input grows at the same rate for all models, but that is not true for the UTM. The RASP and functional inputs grow in accordance to the number of states, symbols and the tape length. The TM however, is concerned only with the number of states and the length of the tape. While the RASP and functional UTM incarnations can simulate any arbitrary (s,t) TM, Minsky's UTM can only simulate (s,2) TMs.

This does not affect the computational power of the Minsky's UTM language relative to the languages of the RASP and function model UTMs, but may make it less expressive in that the TM to be simulated will have a more constricted input language.

As discussed earlier, the TI across paradigms sub-hypothesis (where TI = SI + PI) is contradicted by the fact that the λ-calculus and SKI have lower TIs to calculate the functions on average than the RASP and TM do respectively. Viewing the growth rates, it is possible that the definition of TI does not go far enough, in that it does not take the input size of functions into account. The input growth size indicates that after a sufficiently large input, the RASPs will have the lowest TI + input size for all models. This is little more than conjecture at this point but an interesting topic for future investigation.

The growth rates are for natural encodings, which are straightforward mappings from unencoded to encoded data. There exist programs which are strictly more (Chaitin) elegant than the programs measured, but have more complex encodings which grow faster. An example of this is the UTM by Neary.

6.6 The UTM

The contrast between two different universal machines is an informative example of how the encoding and information content of the input to a program influences the size of the program. Most notably for the TM, the elegance of the programs

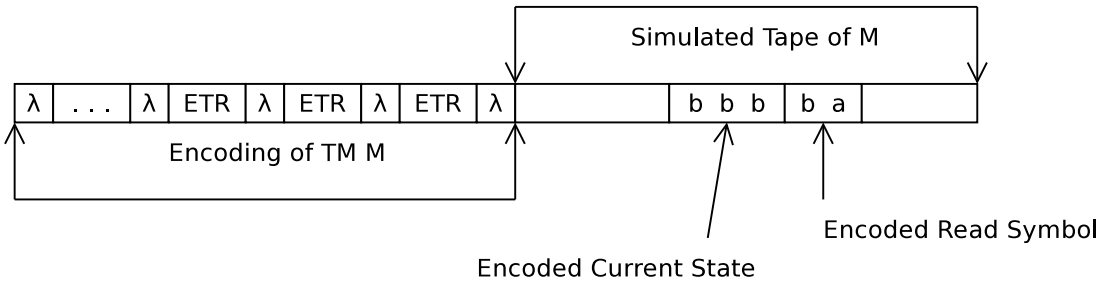


Figure 6.19: The tape of the UTM simulating a machine M. (From [65], pp 26)

can be influenced by the encoding scheme of their input. The intuition is that natural encodings of inputs require more program information to decode, whereas well constructed, larger, and more complex encoding schemes offload complexity from the program to the input.

Recalling the proof of the undecidability of elegant functions in Section 3.1, we are reminded that there exists at least one function where the amount of information required to specify the program+input can be improved for infinitely many inputs. The UTM may or may not be an example of such a function, but this example shows the extent that input encodings can have on program size.

6.6.1 Neary's UTM

Neary is the creator of the smallest currently known direct simulation UTM. His 8 state, 4 symbol machine is strongly universal, consists of 30 tuples, and can simulate 2 symbol Turing Machines. Traditional direct simulation UTMs encode a symbol table, and tape of a machine M. The simulator maintains pointers to which state the machine is currently in, and which position the head is at on the tape. This intuitive construction requires the head of the simulator to traverse the whole tape regularly.

Neary's machine stores the entire current state on the simulated tape, thereby using the state as a positional marker for the head. From an initial configuration with the symbol table represented as a collection of encoded transition rules (ETRs), and the state/symbol pair on the simulated tape (Figure 6.19), the machine operates in four cycles.

The first cycle scans the state and symbol pair on the tape. For each b in the pair, the machine ticks off a corresponding λ on the left. It does this until

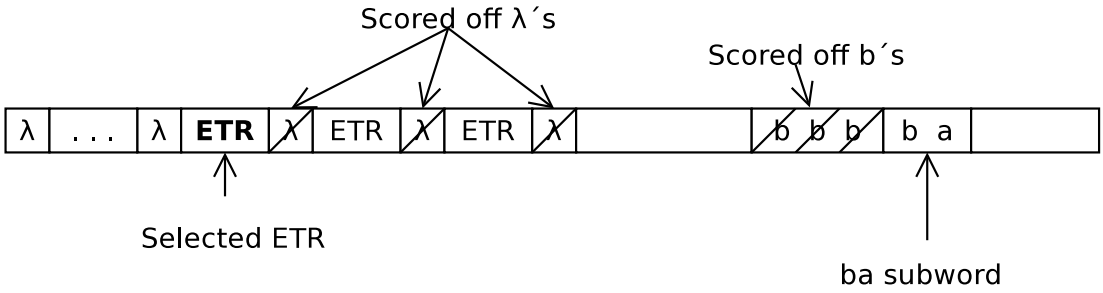


Figure 6.20: The UTM finding the relevant ETR (From [65], pp 26)

it reaches the word *ba*. In the example in Figure 6.20, the first three *b* symbols from the encoded state have stricken off the first three λ's from the right.

The second cycle, copies the relevant ETR over the current state and symbol pair on the tape. In this example, the ETRs are 5 symbols long and made up of *a* and *b* symbols. This cycle overwrites the ETR on the simulated tape with the selected ETR in the symbol table and initiates cycle three, which restores the tape of the UTM, unchecking the λ's and the symbols of the ETR that have been copied.

The behaviour of cycle four is dependent on whether the UTM has processed a left or right move. A right move executes a special ETR which incrementally shifts the ETR to the right. The symbols 0 and 1 on the tape are represented as the pairs *aa* and *ba*. A right shift would move the ETR from $ETR\ ba$ to $b\ ETR\ a$, to $ba\ ETR$.

Left shift ETRs are longer than the ETRs of the right shift. Since the copying of the new ETR is performed from the right hand size of the old state and symbol pair, the new left-shift ETR therefore protrudes over the space of the old ETR by two symbols to the left. This in effect shifts the tape relative to the ETR head and pushes the new head position to the right of the ETR where cycle 1 begins again.

Neary's machine has no specified halting state; rather it halts through the simulated machine trying to run off the left hand side of the tape. The techniques used in this UTM are simple in isolation. It exhibits simple searching for and copying of ETRs. The encoding of the symbol table as ETRs, belies the complexity of the simulation.

Neary has also produced a slightly larger (3,11) machine which operates, save

1,0,2,1,R
1,1,1,0,R
2,0,2,0,L
2,1,3,1,L
3,0,3,0,L
3,1,3,1,L

Figure 6.21: A (3,2) TM as the benchmark for testing input sizes

for very minor technical details, in the same manner as the (8,4) machine. These two machines have similar, but different encoding schemes. The (3,11) machine defines 31 tuples as opposed to the 30 of the (8,4) machine, so if the intuition of more tuples implying concise encodings is correct then it is expected that the expression for the TM in the (3,11) machine will be more concise than the (8,4) machine. Also in this comparison is the UTM from Minsky. This (23,8) machine uses many more tuples than Neary's machines, but has a much more natural expression of the tape and symbol table of the simulated TM.

6.6.2 Encodings

Consider the TM in Figure 6.21. This (3,2) machine will halt on Neary's UTM by running off the left hand side of the tape and is what shall be used for comparison of three UTMs.

The tape of Neary's UTM is initially arranged as a triple $\langle M\rangle\langle q_1\rangle\langle w\rangle$ of the encoding of the machine as Encoded Transition Rules (ETRs), an encoding of the initial state, and a right unbounded tape respectively. A tuple $t_{st,sy}$ is a a quintuple $t = \langle st_x, sy_x, sy_y, D, st_y\rangle$, where st_x is the original state, sy_x the original symbol, D is either R or L, and sy_y/st_y are the new symbol and state respectively[1]. Here $|Q|$ is the number of states and f is the symbol table itself. The encoding of M is as follows:

$$\begin{aligned}\langle M\rangle = \quad & \lambda\varepsilon(t_{|Q|,1})\lambda\varepsilon(t_{|Q|,0})\lambda\varepsilon(t_{|Q|,0})\lambda\varepsilon(t_{|Q|,1})\lambda\varepsilon'(f, t_{|Q|,0}) \\ & \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \ldots \\ & \lambda\varepsilon(t_{1,1})\lambda\varepsilon(t_{1,0})\lambda\varepsilon(t_{1,0})\lambda\varepsilon(t_{1,1})\lambda\varepsilon'(f, t_{1,0})\lambda e\end{aligned}$$

[1]Note that this form for tuples is from Neary and is used to make the reconciliation of his work easier. This notation will not be used in any other section.

The functions ε and ε' encode the specific tuples and depend on the particular UTM that the tuple is being encoded for. For the (3,11) UTM, the functions are:

$$\varepsilon_{3,11}(t) = \begin{cases} e^{a(t)}h^{b(t)} & \text{If } D = R, sy_y = 0 \\ he^{a(t)}h^{b(t)} & \text{If } D = R, sy_y = 1 \\ e^{a(t)-1}h^{b(t)}eee & \text{If } D = L, sy_y = 0 \\ e^{a(t)-1}h^{b(t)}ehe & \text{If } D = L, sy_y = 1 \end{cases}$$

$$\varepsilon'_{3,11}(f,t) = \begin{cases} e^{a(t^{R,x})-3}h^{b(t^{R,x})+2} & \text{If } \exists t^{R,x}, st_x \neq st_1 \\ \text{(Nothing)} & \text{If } \nexists t^{R,x}, st_x \neq st_1 \\ e^{5|Q|-3}h^4 & \text{If } st_x = st_1 \end{cases}$$

where $t^{R,x}$ is any transition rule that shifts right and transits to the current state from state x. The functions $a(t)$ and b(t) are defined by the equations:

$$a(t) = 5|Q| + 2 - b(t)$$

$$b(t) = 2 + \sum_{j=1}^{y} g(t,j,y)$$

where y is the state transitioned to by the tuple. Finally, the function $g(t,j)$ is defined:

$$g(t,j) = \begin{cases} 5 & \text{If } j < y \\ 3 & \text{If } D = L, j = y \\ 0 & \text{If } D = R, j = y \end{cases}$$

Functions $a()$, $b()$, and $g()$ are common to both of Neary's machines. Only the ε and ε' functions are different. The relevant functions for the (8,4) machine are as follows:

$$\varepsilon_{8,4}(t) = \begin{cases} bba(ab)^{a(t)b^{2(b(t))}}aa & \text{If } D = R, sy_y = 0 \\ aabbb(ab)^{a(t)-1}b^{2(b(t))}aa & \text{If } D = R, sy_y = 1 \\ a(ab)^{a(t)-1}b^{2(b(t))}(ab)^3aa & \text{If } D = L, sy_y = 0 \\ a(ab)^{a(t)-1}b^{2(b(t))}abbbabaa & \text{If } D = L, sy_y = 1 \end{cases}$$

ETR	T Rule	$t^{R,x}$	$b(t)$	$a(t)$	ε or ε'	Size
$\varepsilon'(f, t_{1,0}$	$q_1, 0, 1, R, q_2$	$q_1, 1, 0, R, q_1$	2+0=2	15	$e^{12}h^4$	16
$\varepsilon(t_{1,0}$	$q_1, 0, 1, R, q_2$		2+5+0=7	10	$he^{10}h^7$	18
$\varepsilon(t_{1,1}$	$q_1, 1, 0, R, q_1$		2+0=2	15	$e^{15}h^2$	17
$\varepsilon'(f, t_{2,0}$	$q_2, 0, 0, L, q_2$	$q_1, 0, 1, R, q_2$	2+5+0=7	10	e^7h^9	16
$\varepsilon(t_{2,0}$	$q_2, 0, 0, L, q_2$		2+5+3=10	7	$e^6h^{10}eee$	19
$\varepsilon(t_{2,1}$	$q_2, 1, 1, L, q_3$		2+5+5+3=15	2	$eh^{15}ehe$	19
$\varepsilon'(f, t_{3,0}$	$q_3, 0, 0, L, q_3$	(None)	null	null	(nothing)	0
$\varepsilon(t_{3,0}$	$q_3, 0, 0, L, q_3$		2+5+5+3=15	2	$eh^{15}eee$	19
$\varepsilon(t_{3,1}$	$q_3, 1, 1, L, q_3$		2+5+5+3=15	2	$eh^{15}ehe$	19

Table 6.21: Converting the benchmark to the format for Neary's (3,11) UTM (from [65] pp 30)

ETR	T Rule	$t^{R,x}$	$b(t)$	$a(t)$	ε or ε'	Size
$\varepsilon'(f, t_{1,0}$	$q_1, 0, 1, R, q_2$	$q_1, 1, 0, R, q_1$	2+0=2	15	$bba(ab)^{12}b^8aa$	37
$\varepsilon(t_{1,0}$	$q_1, 0, 1, R, q_2$		2+5+0=7	10	$aabbb(ab)^9b^{14}aa$	39
$\varepsilon(t_{1,1}$	$q_1, 1, 0, R, q_1$		2+0=2	15	$bba(ab)^{15}b^4aa$	39
$\varepsilon'(f, t_{2,0}$	$q_2, 0, 0, L, q_2$	$q_1, 0, 1, R, q_2$	2+5+0=7	10	$bba(ab)^7b^{18}aa$	37
$\varepsilon(t_{2,0}$	$q_2, 0, 0, L, q_2$		2+5+3=10	7	$a(ab)^6b^{10}(ab)^3aa$	41
$\varepsilon(t_{2,1}$	$q_2, 1, 1, L, q_3$		2+5+5+3=15	2	$a(ab)^1b^{30}abbbabaa$	41
$\varepsilon'(f, t_{3,0}$	$q_3, 0, 0, L, q_3$	(None)	null	null	a	1
$\varepsilon(t_{3,0}$	$q_3, 0, 0, L, q_3$		2+5+5+3=15	2	$a(ab)^1b^{30}(ab)^3aa$	41
$\varepsilon(t_{3,1}$	$q_3, 1, 1, L, q_3$		2+5+5+3=15	2	$a(ab)^1b^{30}abbbabaa$	41

Table 6.22: Converting the benchmark to the format for Neary's (8,4) UTM

$$\varepsilon'_{8,4}(f,t) = \begin{cases} bba(ab)^{a(t^{R,x})-3}b^{2(b(t^{R,x})+2)}aa & \text{If } \exists t^{R,x}, st_x \neq st_1 \\ a & \text{If } \nexists t^{R,x}, st_x \neq st_1 \\ bba(ab)^{5|Q|-3}b^8aa & \text{If } st_x = st_1 \end{cases}$$

These sets of equations encode the symbol table of the machine. Tables 6.21 and 6.22 present the working and results of encoding the test TM from Figure 6.21. The sixth column of the tables shows what will be on the tapes of the UTMs. The superscribed numerals next to potentially bracketed symbols indicate a repetition of those symbols. Each letter corresponds to a single symbol and the size of each conversion is given in characters.

In contrast to the Neary TMs, the initial tape of the Minsky UTM (Section 4.4.1) is arranged as $[w][st_1][sy][M]$. The symbol table is arranged in quintuples of $st_x, sy_x, st_y, sy_y, D$. The states are binary numbers, symbols are either 1 or 0, and the direction D is either 0 or 1 to indicate a left or right shift.

T Rule	Expression	Size
$q_1, 0, 1, R, q_2$	0101011	7
$q_1, 1, 0, R, q_1$	0110101	7
$q_2, 0, 0, L, q_2$	1001000	7
$q_2, 1, 1, L, q_3$	1011110	7
$q_3, 0, 0, L, q_3$	1101100	7
$q_3, 1, 1, L, q_3$	1111110	7

Table 6.23: Converting the benchmark to the format for Minsky's (23,7) UTM

Machine	Tuple size	State Enc	Other Overhead	UTM size	Prog + UTM
N(8,4)	559	17	17	299	892
N(3,11)	254	17	17	319	607
M(23,8)	40	4	9	1270	1323

Table 6.24: Information cost of setting up the test TM on the three UTMs

A tuple encoded for Minsky's simulation uses binary numbers for both states, and single symbols for the old symbol, new symbol and direction. The current state and symbol is stored elsewhere, necessitating another binary number and single symbol. There are a number of delimiters to include too.

Table 6.23 shows the tuples converted to their respective tape expressions. The conversion process of the Minsky UTM produces a tuple form which is much more in keeping with the original quintuples. Neary's conversion process leaves almost no easily discernible aspects of the original tuples. Without the tables and equations, it would be very difficult to derive the original tuples from this form.

The initial head position and state for Neary's UTMs ($\langle q_1 \rangle$) is an expression of length $(5|Q|)+2$. For both UTMs this is $a^{5|Q|}b^2$. Each symbol on the simulated tape is a pair of symbols on the UTM tape where $0 = aa$ and $1 = ba$. Each ETR is separated by the λ symbol and terminated by the sequence λe.

The overhead of symbols for Minsky's machine consists of the head position symbol M, the current state and symbol area between the first Y and first X from the left, the X symbol separating tuples, and the final $Y0$ at the far right which signifies the end of the symbol table. The simulated tape has a one to one correspondence with the UTM tape.

All of the UTMs simulate arbitrary (n,2) TMs. The measurements made measure the test TM implemented on the UTMs running with a blank tape. Neary's UTMs require that all tuples encoded via ε are represented twice in the

States	Minsky (23,8)	Neary (8,4)
2	37	318
3	53	626
4	86	1034
5	106	1542
6	126	2150

Table 6.25: Number of characters per symbol table

symbol table, and that the ε' tuples terminate each state. Table 6.24 tallies up the program information of a UTM set up to execute the test TM on a blank tape. The tuple sizes are measured as the total of expressions returned from $\varepsilon(')$ or the Minsky encoding to populate the symbol table. 'Other Overhead' symbols are delimiters and such.

There are 30 tuples in the (8,4) machine, 32 in the (3,11), and 113 in the (23,8) UTM. The data from the table shows that there is almost 1.5 times the tape information required to represent the test machine on the (8,4) UTM as opposed to the (3,11) UTM, which is two tuples larger. The (23,8) machine is much larger than the other two machines, but the representation of the test TM is very concise in comparison. For this example the TI (measured in this case as the size of the TM tuples and the encoding of the benchmark machines) of the Minsky machine is still larger than the TIs of the smaller machines.

6.6.3 Input Growth

If s is the number of states in the machine, the characters required to implement the symbol table for a Minsky-simulated machine is:

$$2s(2(\lfloor log_2(s) \rfloor + 1) + 4) + (\lfloor log_2(s) \rfloor + 1) + 3$$

The Minsky encoding is agnostic to the operations of the tuples. The Neary encoding however changes depending on the shifts and state transitions which take place. The encoding function ε' changes the sizes of the encoding depending on whether there is a right moving transition into the current state. If state x does not have a right moving transition entering it, then ε' for the (8,4) (like state 3 in Table 6.22) machine is a single character, rather than something larger.

The number of characters required to implement the benchmark machines in the Neary (8,4) and Minsky (23,8) UTMs are shown in Table 6.25. The growth of Nearys encoding here fits the recurrence relation: $a_{n+1} = a_n + 100(n+1) + 8$. Solving this relation gives an equation that indicates that the growth in encoding size is quadratic in the number of states:

$$a_n = 50n^2 + 58n + 2$$

The encoding for Minsky's machine grows slightly more than quasilineraly ($n\ log_2(n)$), but far less than quadratically (n^2). A compromise is reached with the function $n\ (log_2(n))^2$ which grows faster than the formula for Minsky's encoding. Neary's UTMs tape has two symbols per simulated symbol ($2k$), and Minsky's UTM uses only one (k).

Thus, the big O notations for the Minsky and Neary encoding functions are $O(s(log_2(s)^2) + k)$ and $O(n^2 + k)$ respectively. This data shows that although Neary's UTM is much smaller than the UTM of Minsky, the encoding function grows at a much higher rate. Solving the formulae for the symbol table sizes and adding in the TI of of Minsky's machine at 1271 and Neary's at 300 characters shows that the breakpoint between encodings occurs at 5 states. At simulating a 5 state TM, it is more information efficient to use the UTM of Minsky.

6.7 Conclusions

This chapter has analysed the data from Chapters 4 and 5, to evaluate the hypotheses. Figure 6.22 summarises the hypotheses and sub-hypotheses, lists sections with analyses which are for and against the hypotheses and states **(C)** if the hypothesis is confirmed, and **(NC)** if not.

What has been found is that the Strong Semantic Information and Strong Total Information hypotheses (Section 3.1.2) cannot be fully confirmed. While the number of characters as an information metric is predictive for the RASP family and between models of the same paradigm, the metric appears to fail to account for the differences between models of different paradigms.

The SI and TI hypotheses are consistent within the confines of model paradigms

1 Strong Semantic Information hypothesis **(NC)**
   1a. SI within family. **For:** 6.2.1 **(C)**
   1b. SI within paradigm. **For:** 6.2.2, 6.2.3 **(C)**
   1c. SI across paradigms. **For:** 6.2.4, 6.2.7 **Against:** 6.2.5, 6.2.6 **(NC)**
2. Strong Total Information hypothesis **(NC)**
   2a. TI within family. **For:** 6.2.1
   2b. TI within paradigm. **For:** 6.2.2, 6.2.3 **(C)**
   2c. TI across paradigms. **For:** 6.2.4, 6.2.7 **Against:** 6.2.5, 6.2.6 **(NC)**
3. Strong Semantic Circuit hypothesis **(C)**
   3a. SC within family hypothesis **For:** 6.3.1 **(C)**
   3b. SC within paradigm hypothesis **For:** 6.3.2 **(C)**
4. Strong Total Circuit hypothesis **(NC)**
   4a. TC within family hypothesis **Against:** 6.3.1 **(NC)**
   4b. TC within paradigm hypothesis **Against:** 6.3.2 **(NC)**

Figure 6.22: Hypotheses with evidence and confirmation status

(Sections 6.2.1 – 6.2.3). What separates the paradigms is their internal representation and method of evaluation. The RASP and TM are primarily based on arrays. The λ-calculus and SKI models have a graph based internal model and evaluation system. It is conjectured here that this difference between the models affects the data which is contrary to the SI and TI hypotheses. What is implied by the current results is that the functional models are more 'information efficient' on average in comparison to the imperative models.

There appears to be a large separation in the in the TI amounts required for the RAPS/λ-calculus opposed to the TI required for the TM/SKI (Section 6.4.1). While the RASPs and λ-calculus have the concept of random access/variables for the manipulation of data and structures, the TM and SKI access data in a sequential fashion. The TI required to implement the universal TM and universal RASP programs in the TM and SKI are highly correlated; the information amounts are much larger than the information amounts required for the RASP and λ-calculus implementations.

The FPGA implementations, in defiance of the abstract TI implementations, show that while there is a relationship between the number of times a particular component is used and the abstract TI of a program in a model, that relationship disappears when attempting to compare the TIs of different models (Section

6.3.3). In other words, if program A uses more LUTs on an FPGA than program B for the TM, we can be reasonably confident that A has a higher TI than B. However if A in the RASP uses more LUTs than B in the TM, we still cannot deduce the relative TIs between programs A and B. Cross model comparisons do not work.

Part of the reason for this is there is an overhead in the semantics incurred proportionally to the size of the programs. While the abstract semantics can easily define a number as a member of the natural numbers, the FPGA realisations require a concrete range. As the number grows, so must the number of components required to represent that number at the hardware level.

Despite the TI being a poor indicator of relative circuit sizes, there exists strong correlations between component counts and the information contents of programs. Table 6.11 shows that there is a very strong correlation between LUTs and the TI levels for TMs. The correlation for RASPs is not as strong, but shows a correlation of both LUTs and slice registers with the TIs.

Potential elegance has been sacrificed by the author in favour of natural expressions of program inputs (Section 3.1.1). Analysing the information growth rates of the models (Section 6.5) indicates that the growth rate of the TIs of RASP programs of this thesis, paired with the inputs is lower in the limit than the other models (Table 6.20). The TM follows the RASP due to its binary encoding. The functional models with linear encodings are the largest. This holds only for the specific models and programs in this thesis, but is worthy of further investigation.

A comparison between the machines of Neary and Minsky shows just how dramatic an effect input encoding schemes can have on the elegance of program sizes (Section 6.6). The input size for Neary's machines grows quadratically in relation to the number of states, while the input size for the Minsky UTM grows in an almost quasilinear fashion. Simulating a TM with 5 states requires less information for the Minsky TM than for Neary's (8,4) TM (Section 6.6.3, Table 6.24).

Chapter 7

Discussion and Conclusion

This chapter concludes the thesis. Section 7.1 recaps the aims, methodology, results, and contributions of the work. Section 7.2 is a discussion ranging from the role of type systems in programming languages to recreational programming. Each of these topics touches on an aspect of this investigation and are discussed in an informal manner. Finally, Section 7.3 covers possible further work arising from this investigation.

7.1 This Work

7.1.1 Aims

This work has been an empirical exploration of the intuition underlying the expressivity of models of computation and languages. The intuition is that more information in the semantics of model implies that the model is more expressive than a model with comparatively less information. That extra information in turn precipitates smaller programs in general.

More formally, the work has been an investigation into the relationship between the information content of the semantics of a model of computation, and the information content of programs written for that model. This is also known as the "Conciseness Conjecture" (Section 2.5).

The investigation was directed at resolving four hypotheses (Section 3.1): the Semantic Information (SI) hypothesis, the Total Information (TI) hypothesis, the Semantic Circuit (SC) hypothesis, and the Total Circuit (TC) hypothesis.

Consider two programs a and b which compute the same function and are programmed in computational models A and B respectively. The SI hypothesis states that if model A has larger semantics than B, then program a will be smaller in size than b on average. In essence, this asserts that there is an inverse relationship between semantic size and mean program size.

The TI hypothesis not only considers the size of the program, but also of the semantics. Consider programs a_1 and b_1 which calculate a mathematically trivial function such as addition, and programs a_2 and b_2 which calculate a more complex function such as a universal machine. If model A has significantly larger semantics than model B, then the Total Information (size of program + size of semantics) to calculate addition in model A may be higher than the Total Information to calculate addition in B: $sem(A) + a_1 > sem(B) + b_1$.

Considering the case of the more complicated function. The TI hypothesis states that with B having much smaller semantics than A, the program b_2 will be much larger in size than a_2. This difference in size of programs is larger than the difference in size of semantics and therefore $sem(A) + a_2 < sem(B) + b_2$.

The SC and TC hypotheses (Section 3.1.3) are in reference to Field Programmable Gate Arrays (Chapter 5). The SC hypothesis states that there is a direct relationship between the size of a models abstract semantics, and the size of a circuit which realises those semantics.

The TC hypothesis is an analogue of the TI hypothesis above. Models which larger semantic circuits will produce an overall smaller circuit implementing a complex function than a model with a simpler semantic circuit.

7.1.2 Method

To resolve these hypotheses, 6 models of computation are chosen. Models of computation can be separated into distinct groups based on their characteristics. Two of these groups: imperative and functional (Section 2.3) are represented here. The imperative models include the Turing Machine (Section 2.3.1.1) and a family of three Random Access Stored Program (RASP, Section 2.3.1.2) machines. The functional models include the λ-calculus (Section 2.3.2.1) and the SKI combinator calculus (Section 2.3.2.2).

These models have their methods of execution and internal data representations formalised in Structured Operational Semantics (SOS, Section 3.4) and thirteen programs are written for each model (Chapter 4). The programs encompass functions in the set of arithmetic, those from list processing, and the universal machines.

The sizes of the semantics and programs were measured by the number of characters it takes to write them (Section 3.2) as is traditional in information theory. As such, the programs were written to be as "elegant" (Section 2.2.2) as possible while utilising what could be called a "natural" input/output encoding (Section 3.1.2). As the most elegant program to calculate a function may not use a natural encoding, the programs and semantics measured are termed "succinct" (Section 3.1.2).

As we are interested in the total amount of information required to specify the program to compute functions, there are issues inherent in the approach of specifying the semantics of models in an unspecified formalism. Attempts to specify that formalism perpetuate such issues (Section 5.1). Thus the RASP and TM models are implemented in hardware using Field Programmable Gate Arrays (FPGAs, Chapter 5). The semantics and programs written for these models are compiled down to electronic components and the number of components are counted.

7.1.3 Results

The SI and TI hypotheses make general statements about how the information required to specify problems compares against models with different sizes of semantics. Given the variance of computational models tested in this investigation, the hypotheses were split into three sub-hypotheses each. These sub-hypotheses are: SI/TI within family (comparing the three RASP models), SI/TI within paradigm (comparing the TM with the RASPs, and the λ-calculus with SKI), and SI/TI across paradigms (comparing the TM with the λ-calculus/SKI and the RASPs with the λ-calculus/SKI). In doing this, an exhaustive comparison is made of the programs sizes of one model with the program sizes of another.

Chapter 6 provides the primary analysis of the measurements made to resolve

the four hypotheses (Section 6.7 and Figure 6.22):

- The SI/TI within family hypotheses are confirmed with respect to the data.
- The SI/TI within paradigm hypotheses are confirmed with respect to the data.
- The SI/TI across paradigms hypotheses are rejected with respect to the data.
- The Strong SI and TI hypotheses are not confirmed by the data.
- The SC hypothesis is confirmed by the data.
- The TI hypothesis is rejected by the data.

In general, there is evidence for Felleisen's Conciseness Conjecture. However the Total Information measure (semantics size + program size) used to gather this evidence cannot extend the conjecture to comparing models of computation across paradigms.

While TI seems suitable for comparing different models with the same evaluation methodology (i.e. imperative or graph reduction evaluation), it appears to be insufficient for heterogeneous comparisons of models. There appears to be subtle differences between the semantics which are not adequately conveyed by a simple character count (Section 6.2.8).

One of these subtleties could be in the implicit definition of operators in the semantics. Section 7.3.5 discusses this in detail, but it is seemingly an issue as to what is measured in the semantics and what is implied. For instance, the RASPs use the natural numbers without any definition of them, whereas the λ-calculus and SKI use no such numerical constructs.

Another lies in the definition of program inputs. A function is computable if there exists a program to solve any instance of that function. The program takes the instance as input, churns, and returns the solution. For any one function, if it is computable then there are an infinite number of programs to compute the function. This spectrum of programs may vary from clever to naïve, efficient to wasteful, small to large, and many other opposing adjectives.

The FPGA hypotheses assert that there is indeed a relationship between the size of the semantics represented in SOS, and the size of a circuit which represents the semantics. This confirms the SC hypothesis (Section 6.3.3). The TC hypoth-

esis cannot be confirmed however. Given that the TC is an FPGA analogue to the TI hypothesis, we would expect that the Total Circuit size of some program in models A and B would provide some insight into the relative expressiveness of the two models. It turns out that this cannot happen. Part of the reason for this is there is an overhead in the semantics incurred proportionally to the size of the programs. While the abstract semantics can easily define a number as a member of the natural numbers, the FPGA realisations require a concrete range. As the number grows, so must the number of components required to represent that number at the hardware level.

While the relative number of components in FPGA implementations of models is a poor indicator of the TI relationships between those models; there exists a correlation between the number of a specific component (the precise component is depended on the model), and the TI of the program implemented. In other words, given two FPGA programs a and a_1 written for the same model, if a_1 uses more of some component than a, then there is a reasonable certainty that the the abstract program a_1 will also be larger than the abstract program a.

The logic optimiser of the FPGA compilation software is an unknown variable in these comparisons. At compile time, the settings were tuned for maximum compression and it is currently unknown quite how the compiler optimises and packs the logic into registers and LUTs. Investigation into this could provide insight into why there is a correlation between component counts and TIs, but why the same component counts give no indication between the relative TIs of models.

7.1.3.1 Other Results

Aside from resolving the hypotheses for the chosen functions in the chosen models, the analysis has uncovered other results:

- There is evidence of a large jump in the required TI arising from sequential vs random access memories (Section 6.4.1).
- There is a relationship between the size of inputs and the TI of a program in a model (Section 6.5.5).

Through contrasting the UTM of Minsky with the UTMs of Neary (Section

6.6), it has been shown that two programs which compute the same function can vary dramatically in size by virtue of their input encoding. Programs with dense encoding systems of many symbols have relatively concise realisations of the input data, whereas programs with sparse encodings have larger inputs.

The current evidence shows that programs which dense input encodings are larger than programs with sparser encoding systems. It was shown that for up to TMs of size (5,2), Neary's (8,4) UTM has a lower TI + input encoding size than the Minsky UTM. However, for inputs of greater size, the Minsky machine requires less information for the TI + input. This suggests that over all inputs, programs with denser input encodings require less overall information.

Input encodings are not something which is addressed by the Conciseness Conjecture, or Chaitin's elegance. Indeed, if one were to also count the size of inputs as part of the total information, it would be found that there are functions which cannot have an optimal implementation for almost all inputs. These findings are consistent with Blum's speedup theorem which addresses this specifically (Section 3.1.1).

And, if one were to ignore input sizes and used Chaitin's elegant finder process to obtain a supposedly elegant program, it is only guaranteed that the found program is elegant relative to a specific input encoding.

The TI of the TM and SKI is much larger than the TIs of the RASP machines and λ-calculus (Section 6.4.1). The reason for this is suspected to be random access memories. The RASP can modify data in arbitrary registers via direct addressing. The λ-calculus abstraction mechanism reads and reorders inputs to the expression, precisely placing them via substitution without unduly influencing the structure of the rest of the expression not involved in the substitution.

In contrast, the sequential access of the TM tape requires that it uses at least one transition to shift left or right to access and modify data. Likewise, the SKI emulates the abstraction mechanism of the λ-calculus by using the S combinator to 'draw' inputs into terms, and the K combinator to eliminate unrequired duplicate terms. These attributes of the SKI and TM bloat their expressions and programs with 'memory access' terms which are not present in the RASP and λ-calculus counterparts.

7.2 Related Aspects

The effort of the investigation was directed towards being as broad and consistent as possible in the collection of data to resolve the hypotheses. The collection resulted in 6 models of computation and 13 programs. Furthermore, there were the FPGA implementations of the RASPs and TMs. This is a lot of data, but leaves the depth of the investigation, in particular the formalisation of the relationships, other minimal systems, alternative semantic representations, and other language features somewhat lacking.

Despite not being explicitly addressed in earlier chapters, there are arguments to be made which place these features of models and programming languages in the context of the SI, PI, and TI metrics explored.

7.2.1 Conservative Extensions

Felleisen's expressiveness as described in Chapter 2 is based on the concept of conservative extensions and restrictions. The idea is that a Turing Complete formal system A is more expressive than a Turing Complete system B if it can be shown that A is a conservative extension of B.

The RASP2 and 3 are not true conservative extensions of the original RASP. Rather, the respective ADD and SUB instructions have been added to the semantics, and the INC and DEC instructions removed. From Section 2.5:

Definition 2 (Conservative Extension/Restriction)**.** *A language L' is a conservative extension of L if:*

- *the functions of L are a proper subset of those of L', with the difference being $\{F_1, F_2, \ldots\}$;*
- *the sets of L-phrases and L-programs are proper subsets of their L' counterparts where there are no phrase or programs that contain the extra L' functions $\{F_1, F_2, \ldots\}$;*
- *$eval_L$ is a proper subset of $eval_{L'}$ and for all L-programs P, $eval_L(P)$ holds if and only if $eval_{L'}(P)$ holds.*

The converse is a conservative restriction.

The RASP2 and RASP3 do not fit this description. However by the data gathered, they are more expressive in that they require less information on average than the RASP to express programs. The author believes that for models with a similar evaluation method, the amount of semantic information is an indicator of relative expressiveness. Section 7.3 outlines work that can be done in this area to confirm or deny such a notion.

With the framework constructed in previous chapters, it is not hard to manufacture a RASP language that is a true conservative extension. The RASP2-1 and RASP3-1 are conservative extensions of both the vanilla RASP and their respective RASPx machines. In essence, these machines have INC and DEC instructions as well as ADD and SUB, and can use INC place of "ADD 1" which is sometimes necessary for RASP2/3 programs.

Predictions can be made as to the information levels of the two extensions. The semantic information of the extensions will be greater than that of the RASP2 and RASP3 owing to the INC and DEC instructions. It is also hypothesised that the program informations of the extensions will be the same, or less than the PI of the RASP and RASP3. The TI of the extensions will be initially greater than that of the RASP2 and RASP3; and, recalling the small amount of TI separating the RASP2 and RASP3, is unlikely that the extensions will have a lower TI than that of the smaller RASPs.

The RASP2-1 and RASP3-1 models have 10 instructions: the basic 8 from the vanilla RASP, and the ADD and SUB instructions from the RASP2 and RASP3 models respectively. This adds an extra 54 characters to the language semantics. The ADD and SUB instructions are mapped to the numbers 3 and 4, with the other instructions following on afterwards as in the definition of the RASP in Chapter 2.

Tables 7.1 and 7.2 show the characters of the implementations, and the number of instructions required. As expected the extensions facilitate smaller programs than the ordinary RASP2 and RASP3, but the difference is rather negligible. The extension is really only useful for replacing instructions such as "ADD/SUB 1" with the relevant INC or DEC, so it saves one instruction.

Where the difference is not negligible is in the RASP2 vs the RASP2-1 figures

	RASP2	RASP3	RASP2-1	RASP3-1
Addition	9	25	9	25
Subtraction	59	61	59	59
Equality	26	27	26	27
Multiplication	59	60	59	59
Division	131	134	131	131
Exponentiation	129	131	129	129
List Membership	129	131	129	130
Linear Search	132	135	132	134
Reverse List	135	137	135	134
Stateful Rev List	273	277	273	273
Bubble Sort	549	297	292	290
Universal TM	571	574	572	571
Universal RASP	1209	1231	1208	1205
Semantics Size	585	587	639	641

Table 7.1: Program and semantic sizes

	RASP2	RASP3	RASP2-1	RASP3-1
Addition	4	6	4	6
Subtraction	22	22	20	20
Equality	9	11	9	11
Multiplication	24	24	23	23
Division	45	45	42	42
Exponentiation	43	40	41	38
List Membership	34	31	33	30
Linear Search	36	35	35	33
New List Rev	45	43	43	39
In Place Rev	78	77	73	72
Bubble Sort	127	123	121	117
Universal TM	148	137	143	131
Universal RASP	292	283	280	270
Arithmetic Mean	69.76	67.56	66.69	64
Geometric Mean	40.79	41.47	39.11	39.43

Table 7.2: Registers used by the various RASP2/3 and their extensions

	RASP2	RASP3	RASP2-1	RASP3-1
Arith Mean All PI	262.38	247.69	242.62	243.62
Arith Mean All TI	847.38	834.69	881.62	884.62
Geo Mean All PI	130.78	137.21	124.59	135.21
Geo Mean All TI	802.48	793.38	842.69	846.30

Table 7.3: Means of the information levels of the implementations

for the bubble sort. The extension and relevant replacement of ADDs with INCs dropped the number of instructions below the lower 2^n-3 threshold which dictates RASP size, allowing for a smaller overall memory size. Table 7.2 shows that the RASP2-1 requires only 121 instructions, rather than 123 instructions like the RASP2.

The arithmetic and geometric means for all of the functions are in Table 7.3. As expected, the overall TIs of the extensions are larger than the non-extended RASPs. The savings on PI over the set of functions is lower than added SI. But as the set of tested functions increases in size, TIs of the RASP2-1 and 3-1 will increase slower than that of the RASP2 and 3.

The TI of the RASP2-1 is lower than the TI of the RASP3-1. This is because of the aforementioned drop in the size of the RASP2 machine for computing the bubble sort. This data contravenes hypothesis 2a: TI within model family. However Table 7.2 does show that the RASP3-1 requires less instructions than the RASP2-1 for the list and universal functions and that the low number of characters for the RASP2 implementing the addition function is largely the cause of the imbalance. It is not unreasonable to project that this imbalance is corrected as the set of tested functions grows.

The RASP2-1 and RASP3-1 are true conservative extensions of the RASP and RASP2 or RASP3. Felleisen's conciseness conjecture holds in this case as programs implemented in the extended models are on average smaller than those in the unextended models. This is also further evidence to the claim that SI = expressiveness for models with similar evaluation methods.

While they are a greater distance apart than the RASPs, the λ-calculus and SKI operate amongst similar principles. Indeed, recalling the mapping from SKI combinators to λ terms from Chapter 2 (Figure 7.1), the SKI can be mapped directly into the λ-calculus syntax.

$$
\begin{aligned}
I &\equiv (\lambda x.x) \\
K &\equiv (\lambda x.\lambda y.x) \\
S &\equiv (\lambda x.\lambda y.\lambda z.xz(yz))
\end{aligned}
$$

Figure 7.1: Combinator λ terms

	SKI	λ-Calculus	SKλ
Addition	16	27	16
Subtraction	113	46	46
Equality	208	117	177
Multiplication	8	15	8
Division	565	229	229
Exponentiation	11	9	9
List Membership	362	208	208
Linear Search	385	236	236
List Reversal	190	134	134
Stateful List Rev	1397	460	460
Bubble Sort	1903	550	550
Universal TM	2593	584	584
Universal RASP	9554	1084	1084
Semantics Size	291	515	600

Table 7.4: SKλ programs in comparison

The SKI language can be defined as a truncated version of the λ-calculus without arbitrary variables and abstractions. The only permissible abstractions are those within the S, K, and I combinators. As a result, the common language universe for the SKI and λ-calculus is very similar to the λ-calculus semantics.

SKλ is a conservative extension of both the SKI and λ-calculus. Retaining all of the abstraction, variable, and reduction rules of the λ-calculus, SKλ is augmented with named expressions, S, K, and I. At parsing time, these named expressions get transformed into their corresponding λ terms and parsed into the reduction tree.

Considering how we make use of named terms to explain λ expressions all throughout this thesis, especially in Chapter 4, we can immediately see how advantageous such a mechanism would be. Considering only the abbreviations S, K, and I, we can select the smaller of the SKI or λ-calculus as the SKλ program. Doing this yields Table 7.4.

There are likely other optimisations which can result in smaller expressions,

$$\frac{e \implies I(e_1)}{parse(e) \implies \{APP, parse(\lambda x.x) parse(e_1)\}}$$

(a) Parsing an I

$$\frac{e \implies K(e_1)}{parse(e) \implies \{APP, parse(\lambda x.\lambda y.x) parse(e_1)\}}$$

(b) Parsing a K

$$\frac{e \implies S(e_1)}{parse(e) \implies \{APP, parse(\lambda x.\lambda y.\lambda z.x\ z(y\ z)) parse(e_1)\}}$$

(c) Parsing an S

Figure 7.2: Extra parsing rules for SKλ

but this simple selection of the most concise expression of the two models demonstrates how the SKλ has lower averages than either the SKI or λ-calculus.

The semantics of SKλ primarily follow those of the λ-calculus, previously presented in Section 3.4.3. These semantics are augmented by a series of rules which substitute the correct expressions in for the combinators. Three new rules are required which are shown in Figure 7.2. In addition, the combinators have to be added to the syntax of the terms. The sizes of the new rules are added to the λ-calculus semantics to derive the semantics size at the bottom of Table 7.4.

As a conservative extension of the SKI and λ calculi, the SKλ language has larger semantics than either. It also produces smaller programs than either on average. Again, Felleisen's conciseness conjecture reinforced by this data.

7.2.2 Compilation

The focus of this thesis presents the models as *interpreted languages*. Essentially, the program universal SOS machine "runs" the SOS evaluation function ($E()$ in the case of the RASPs) step by step. These interpreted semantics are a constant size for all of the models. A RASP program which immediately halts has the same SI as the universal RASP machine, despite not using 7/8 of the instruction set.

More common in the programming language space is a *compiler*. A compiler

combines a program in language A with the semantics of A to produce a self contained package written in the language of the executing machine X, which is also known as the *target architecture*.

Defining a compiler for the RASP machines would perform a semantic fold where only the rules which correspond to instructions in the program would be packaged. Using the immediately halting program above, the model semantics of the RASP and rule for the HALT instruction would be all that was required to execute the program correctly. Discarding rules that the program will not execute results in a lower TI level than indicated in previous chapters.

In this way, compilers can reduce the TI of programs for models. Their efficacy of TI reduction is based on the size of the original semantics. A semantics containing many rules/instructions and a small program utilising only a handful of those rules has a large reduction in TI size. A model with comparatively small or "fully utilised" (where every rule is used in the execution of a program) semantics such as the TM or λ-calculus, would not achieve such a reduction in size.

The RASPs of this thesis are not an ideal testbed for a compiler. If a compiler includes only the rules where it is immediately evident that they will be executed, the resulting semantics will only contain the rules for the initial instructions. If an instruction is executed via self-modification which is not included in the bundle of semantics then the machine will halt, even if it is a valid instruction in the original semantics.

Future work addressing the compilation of programs to be run on the hypothetical target architecture X should use models which produce "static" (non-rewriting) programs. A convenient model to use would be the RAM model, which is not unlike the RASP and executes static programs.

7.2.3 Types

A type system is a restriction on the set of otherwise admissible programs. These programs are syntactically correct, and all assignments and function calls use either the correct types, or the incorrect types are properly *casted* into the correct type beforehand.

At a low level, programs and data are represented using very simple structures. Most, if not all, machines store and process information as binary numbers. Say that there was no type systems, and that each binary number referred uniquely to a piece of information. If A to Z were the binary numbers 00001-11010, then which binary numbers represent the numerals 1 to 26?

A type system provides context for how a particular piece of data should be evaluated. While the data is represented homogeneously at the lowest level, it should not be allowed that the letter "A" (ASCII value 01000001) can be added to the number 65 (also 01000001).

Say that one decided to eschew the traditional definitions of NIL and NULL (Section 2.3.2.1) for marking the end of and testing for the end of lists respectively. A more concise expression for NIL is just FALSE = $\lambda x.\lambda y.y$, which saves three characters per occurrence. A test for the new end of the list is NULL = $\lambda p.p(\lambda x.\text{NOT})$. Reuse of this common Boolean function for a very specific purpose breaks when a list of booleans is traversed. The test for NIL is a test for FALSE, which may possibly be in multiple positions in the list.

What is more disheartening is that the expression for FALSE is also the expression for ZERO. So not even lists of numbers are safe from this poor choice of representation. A typical type system is a pair of the term and a number which indicates the type of the term. Upon application of a function to arguments, the types of the arguments are compared to the expected types and if correct, the function is computed with the input and if not, the running program terminates.

The implementation of a type system in this manner is excessive for the set of functions examined here, but as the set grows and functions get more complicated, a type system is a relatively concise method to extend the applicability of expressions to multiple domains.

7.2.4 Semantic Schemes

The semantics of the models in this investigation were formalised as Structured Operational Semantics (SOS). The SOS notation is flexible enough to specify the semantics of the models in a reasonably concise and uniform fashion. The ability to specify the fine details of model operation resulted in a set of *small step*

operational semantics.

The SOS is also a model of computation. The elegance/succinctness of the model semantics is an indication of just how expressive the SOS model is. Since SOS is a model of computation, the semantics can be thought of as universal machines. There is little assurance that the information content of the semantics accurately reflects the expressivity of the models represented. While the information content measures broadly align with the intuition of model expressivity, corroborating measurements should be obtained by implementing universal machines of all models the model in every other model. Currently only the TM and vanilla RASP are implemented.

It is worth remembering that the assertions of model expressivity, relationships and the supporting measurements made by this thesis apply only to the models and notations explored here. There are numerous alternate notations and conservative extensions to models which may change the relationships.

DeBruijn indices are ostensibly a different notation for the λ-calculus [21]. Rather than variable names, λ abstractions are numbered starting from the innermost terms to the outermost. Bound variables are numerals which occur in the body of the expression. A numeral n is bound by the n^{th} λ from the innermost level. A variable n is bound if it is in the scope of at least n λ's.

As an example, consider the term:

$$(\lambda x.\lambda y.z\ (x(\lambda p.p\ x\ y)))(\lambda v.v\ k) \equiv (\lambda\lambda 4(3(\lambda 132)))(\lambda 12)$$

In the term, z and k are unbound in their parent expressions. Since there are three nested λ's in the expression, z is represented as '4' as to be out of scope. Likewise, there is a single λ in the expression on the right which binds the v, so the k is represented as '2'.

Reduction to normal form follows already established conventions:

$$
\begin{aligned}
&(\lambda x.\lambda y.z\ (x(\lambda p.p\ x\ y)))(\lambda v.v\ k) && \equiv (\lambda\lambda 4(3(\lambda 132)))(\lambda 12)\\
&\Rightarrow_\beta (\lambda y.z((\lambda v.v\ k)(\lambda p.p(\lambda v.v\ k)y))) && \equiv (\lambda 4((\lambda 15)(\lambda 1(\lambda 15)2)))\\
&\Rightarrow_\beta (\lambda y.z((\lambda p.p(\lambda v.v\ k)y)k)) && \equiv (\lambda 4((\lambda 1(\lambda 15)2)5))\\
&\Rightarrow_\beta (\lambda y.z(k(\lambda v.v\ k)y)) && \equiv (\lambda 3(4(\lambda 14)2))\\
&\Rightarrow_\beta (\lambda y.z(k(yk))) && \equiv (\lambda 2(3(13)))
\end{aligned}
$$

Given this behaviour, DeBruijn indices are a model for the λ-calculus, however the evaluation semantics are different. In the above reduction the variable z does not move, but it is renamed twice as the reduction proceeds. Similarly the variable k is renamed to be one above z. When a substitution is made under DeBruijn indices, there is a global renaming effort for the entire term to rename all variables according to the number of nested λs there are. This is as opposed to the familiar λ-calculus where renaming is done at a local level.

Expressions using DeBruijn indices are typically shorter than expressions using the syntax of the λ-calculus defined in this thesis. However the semantics of a DeBruijn model requires this global renaming and a notion of how to count in order to name variables. Expressions using DeBruijn indices therefore have to be evaluated on their own terms with their own semantic scheme. The semantics of DeBruijn's λ-calculus have not been made explicit and measured, but it is theorised that this global renaming behaviour requires larger semantics than the λ-calculus system exhibited throughout this investigation.

7.2.5 Related Minimalism

Elegance, or minimalism, in the size of programs is often a desirable property, in so far as achieving elegance does not adversely affect other measures of how good a program is; such as time/space efficiency or readability. This section briefly discusses systems which embrace minimalism to the fullest and the community of programmers which do the same.

7.2.5.1 Another minimal system

A *Turing tarpit* is a model of computation which can do everything, but is very hard to use. The term was coined by Perlis in [67]:

54. Beware of the Turing tar-pit in which everything is possible but nothing of interest is easy.

The SKI and TM are small models of computation, both with reasonably tarpitty qualities. Without judicious use of white space, it is near impossible to determine the function of a suitably large SKI term. And without a sketch of a state machine/sample tape to change it is difficult to determine the function of a TM.

Iota is a single combinator universal system [89]. The combinator is i where:

$$i \equiv (\lambda x.xSK)$$

where S and K are from the SKI combinator calculus. The system also uses an application operator; '*' such that $*FF = (FF)$ where F is an expression. The SKI combinators can be defined in Iota to demonstrate Turing completeness:

$$*i * i * i * ii = (F(F(F(FF)))) = S$$
$$*i * i * ii = (F(F(FF))) = K$$
$$*ii = (FF) = I$$

Iota is a syntactically inflexible extension to the SK combinator calculus (without the I). Though the syntax is small, the semantics are relatively large. The definition of i above implies the use of λ abstractions. Because including the semantics of the λ-calculus for a single occurrence of a λ abstraction is extremely wasteful, it is more prudent to define a new combinator $ix = xSK$. This sidesteps the requirement of λ abstractions in the semantic definition of the model.

What cannot be sidestepped is the requirement that the semantics of Iota use the internal representation and evaluation semantics of the S and K combinators. In addition, two new evaluation rules are required for the evaluation of $*ii = SK(SK)$ and $*ix = xSK$. The rule for I can be discarded from the original SKI semantics, along with the original parsing rules.

$$\frac{e \implies *e_1e_2}{P(e) \implies \{A, (P(e_1), P(e_2)\}}$$

(a) Application of an expression to another

$$\frac{e \implies i}{P(e) \implies \{i, \emptyset, \emptyset\}}$$

(b) Parsing an instruction

Figure 7.3: Parsing rules for Iota

$$\frac{\begin{array}{c} T.z = A \\ T.L.z = T.R.z = i \end{array}}{R(T) \implies \{A, \{A, \{A, S, K\}, \{A, S, K\}\}\}; R(T_{root})}$$

Figure 7.4: Applying i to itself

The parsing semantics for Iota are therefore are shown in Figure 7.3. Figure 7.4 shows the extra rule required to evaluate the Iota instruction. The size of the Iota semantics in the format described in Chapter 3 is 272 characters, which is slightly smaller than the semantics of the SKI at 291 characters.

Expressions in Iota are also very large. At present, most Iota expressions are derived from SKI, so large SKI combinations derived from λ-calculus expressions are made even larger through conversion of individual S, K and I combinators to their Iota counterparts.

A restricted syntax machine for the imperative paradigm also exists. The *Ultimate Reduced Instruction Set Computer* (URISC) model is Turing Complete using only a single instruction [59]. The exact nature of this instruction can vary, but one of the more studied models Subleq [60] uses an instruction which subtracts the contents of register A from the contents of register B, stores the result in B, then jumps if the result is less than or equal to zero.

7.2.5.2 Golfing

Code Golf is a recreational programming activity where a problem is presented and solutions are taken in either a specific language or a multitude of languages. The solutions are not only evaluated on their extensionality, but also their size. Golfers attempt to minimise their score by solving the problems with the fewest keystrokes possible.

Naturally there has been the development of domain specific languages for

code golf. A notable example is Golfscript, a stack language implemented in Ruby where common operations are mapped to single characters and overloaded such that function performed by an operator is dependent on the arguments supplied.

A Golfscript program (say $1. + 2 + 3 * 2+;$) is a list of literals. Individual numerals or characters are pushed onto the stack, operators like + and * pop the top two elements of the stack, add or multiply them, and then push the result onto the top. The (.) function duplicates the top element of a stack and pushes it; the (;) operator pops the top element. The program $1.+2+3*2+;$ is traversed from left to right. The 1 is pushed then duplicated, 1 and 1 are added to make 2, another 2 is pushed then 2 and 2 are added for 4, a 3 is pushed, 3 and 4 are multiplied to 12, 2 is pushed, 2 and 12 are added to make fourteen, finally the fourteen is popped from the stack.

There are operators for lists which can be concatenated with + ([1 2 1 3][4 5]+ $\mapsto$ [1 2 1 3 4 5]), blocks of code ({...}) and if, while, do, fold statements. Golfscript is Turing complete.

Golfscript can produce very concise programs, but the underlying semantics are quite large. While the internal representation as a vector of input symbols and a stack for processing is reasonably simple, in particular the overloading of operators necessitates a type system so that expressions are evaluated correctly. Golfscript would produce the smallest program for most, if not all, of the functions studied in this thesis, but the semantics would be larger.

7.3 Further Investigations

There is a considerable amount of further work arising from this investigation, from formalising what has been observed, to exploring the extent that input encoding affects the TI, to more accurate measurements by defining the implicitly used operators of the semantics all the way down to the axioms.

7.3.1 Formalism

The thesis results could be generalised and formalised as follows. Felleisen's definitions of expressiveness and language extensions are a good starting point. His notion of a *common language universe* is a conservative extension of two languages which he wishes to compare. This common language universe is used to define relative expressiveness.

Consider the languages L, L_0, and L_1 where L is a conservative extension of both L_0 and L_1. The language L_0 is said to be less expressive than L_1 with respect to L if L_0 can (macro-)express a subset of the operators of L where L_1 can (macro-)express the operations which L_0 can express as well as other operators of L.

There are some caveats to the "is expressible" statement. Felleisen defines expressibility in terms of a homomorphic (program structure retaining) translation ϕ. A language L is said to have the ability to express an operator $F(e_1, \ldots, e_a)$ if there exists ϕ such that $F(e_1, \ldots, e_a) \equiv \phi(F(e_1, \ldots, e_a))$ where $\equiv$ is operational equivalence.

While it is feasible for any Turing complete system to express the operations of any other, Felleisen imposes this restriction of a homomorphic mapping. That is, that the translation of a program using some operator does not require a global reorganisation of the rest of the program. Removing the original operator F and inserting the translation $\phi(F)$ should involve little disruption to the rest of the program.

Now consider the RASP2-1 with respect to the RASP and RASP2. RASP2-1 is L and the other two are L_0 and L_1 respectively. For the RASP2, the INC and DEC instructions of the RASP2-1 are eliminable as they are trivially equivalent to the instruction "ADD 1" and "SUB 1". Likewise, the ADD and SUB instructions of the RASP2-1 are eliminable with respect to the RASP as there exist RASP programs which are equivalent in function to the ADD and SUB instructions.

This symmetry in operational equivalence and macro-expressibility opens up an interesting edge case in Felleisen's framework. According to the framework, RASP2 and RASP have the same expressive power; despite the later conciseness conjecture positing that more expressive languages produce smaller programs

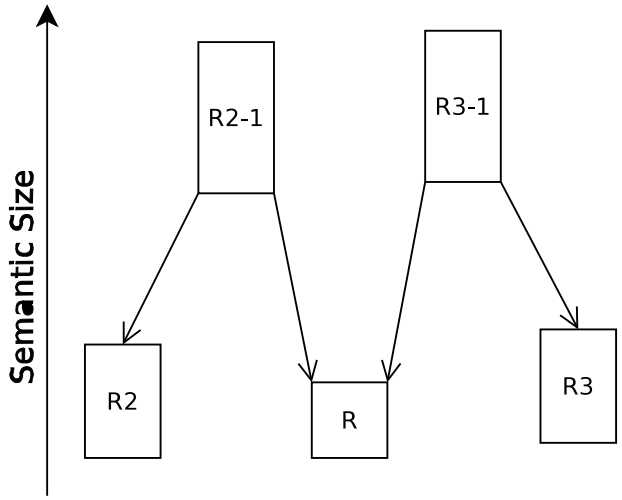


Figure 7.5: Conservative extensions of RASP machines by semantics size

relative to less expressive languages.

A notion of relative expressiveness could be defined to extend the notion of Felleisen's expressiveness paired with the size of the translation given by the mapping θ:

Definition 3 (Relative Expressiveness)**.** *Let L be a language and L_0, L_1 be conservative restrictions of L, where the set $\{F_1, \ldots, F_n\}$ is the set of operators not in L_0, and the set $\{A_1, \ldots, A_k\}$ is the set of operators not in L_1. If $n = k$ (both L_0 and L_1 do not define the same number of operators in L), then L_0 is more expressive than L_1 if:*

- *The operators $\{F_1, \ldots, F_n\}$ and $\{A_1, \ldots, A_k\}$ are (macro-)eliminable with respect to L_0 and L_1.*
- *The size of mapping ϕ_0 from $\{F_1, \ldots, F_n\}$ to L_0-phrases is smaller than the size of mapping ϕ_1 from $\{A_1, \ldots, A_k\}$ to L_1-phrases.*

This resolves the issue of apparent expressive equality of languages which have the same number of undefined operators in the common language universe. It may not be the correct approach however if RASP2 vs RASP3 is considered. Suppose the RASP4 combines the addition and subtraction functions of both RASPs for the functions ADDd/SUBd (direct) and ADDi/SUBi (indirect) in the way suggested in Section 3.4.2 when the RASP instructions of Hartmanis were discussed. The sets of eliminable functions: RASP4 \ RASP2 = {ADDi,SUBi}

CPY	'yval	
STO	'yreg	
LOAD	x	
ADD	0	;yreg
y		:yval

(a) RASP2 ADDi

CPY	'yval	
STO	'yreg	
LOAD		
SUB	0	;yreg
y		:yval

(b) RASP2 SUBi

LOAD	y	
STO	'tmp	
LOAD	x	
ADD	'tmp	
0		:tmp

(c) RASP3 ADDd

LOAD	y	
STO	'tmp	
LOAD	x	
SUB	'tmp	
0		:tmp

(d) RASP3 SUBd

Figure 7.6: Implementations of direct and indirect ADD/SUB

and RASP4 \ RASP3 = {ADDd,SUBd} are the same size, so it falls to the mappings θ_{R2} and θ_{R3} to tiebreak.

Figure 7.6 shows realisations of indirect and direct versions of ADD/SUB in the RASP2 and RASP3 respectively. The realisations are the same size. Each one requires nine registers. This thesis has maintained that the RASP3 is more expressive than the RASP2 by virtue of its larger semantics and conciser programs on average. If the definition for relative expressiveness holds, then the RASP2 and RASP3 are of the same expressive power.

Integrating the general trends of these comparisons into Felleisens framework would have to take these tiebreaker aspects into account, as well as why information-based cross-paradigm comparisons do not behave in the same manner as inter-paradigm comparisons.

7.3.2 Program Equivalences

Section 4.6 discusses the importance of establishing equivalence between two realisations of the same function before formal assertions are made. The work of this thesis has not shown that the implemented programs herein hold under extensional equivalence.

Though equivalence of programs in general is undecidable, equivalence of programs which compute the primitive recursive functions, barring erroneous occurrences of the μ operator (Section 4.1) *should* be computable.

There are multiple ways which extensional equivalence can be estimated, if not proven [29, 51, 69]. One approach involves induction over encoding functions.

Consider a general problem statement: "5+8", or "Search 5 in [1,2,4,5,3,6,7,10]". For each (program, model) pair there exists a pair of functions: an encoding function and a decoding function. The encoding function $enc_{x,y}(s)$ encodes the general statement $s \in S_y$ into a form suitable for evaluation with respect to program y in model x. Similarly, the $dec_{x,y}(q)$ decodes the result of an execution q according to the program y written in model x.

Suppose Y is the set of all functions, X is the set of all models, and S_y is the set of all valid statements which are inputs to function y. Two programs in models x and z which compute a given function y are extensionally equivalent if:

$$\forall s \in S_y : dec_{x,y}(sem_x(prog_{x,y}(enc_{x,y}(s)))) = dec_{z,y}(sem_z(prog_{z,y}(enc_{z,y}(s))))$$

where sem_x are the semantics which execute a program written in x, and $prog_{x,y}$ is a program written in x which computes the function y.

The abstraction afforded by the existence of *enc* and *dec* places the inner workings of the semantics and program into a black box, facilitating the use of induction to show equivalence.

7.3.3 Input Sizes

Sections 3.1.1 and 6.5 have discussed the effect of input encoding on program size. A renewed investigation would aim to fully explore the extent of how input encoding effects the TI of a model and function.

The density of encodings has an influence on the size of the programs. For example, returning to the UTMs of Section 6.5, a relatively natural encoding of the external tuple of the TM ($\langle st_o, sy_o, st_n, sy_n, D \rangle$) uses single symbols to represent the read and written symbols of each tuple and a single symbol for the direction. The nominally base 10 numerals denoting the states are encoded in binary, and tuples are delimited with a single symbol. The machine to utilise this encoding has a large number of state and symbols with many potential tuples: (23,8) with 184.

Reducing the size of the alphabet of the encoding results in a much smaller machine of (8,4) with only 32 potential tuples, however the encoding of the input is sparser and much more complex, increasing in size by nearly 14 times. The (3,11) of 33 potential tuples results in a denser encoding of just under half of the encoding for the (8,4) machine.

7.3.3.1 No Free Lunch and Invariants

It is suspected that a variant of the "No Free Lunch" (NFL, [103]) theorem applies to the relationship of information between semantics programs and their inputs. Both folklore and Felleisen hypothesise that small semantics beget large programs and vice versa. The existence of Neary's UTMs show that there can be concise programs with concise semantics relative to other models. However inputs for such programs are large. Similarly, a model with a very concise program and also concise input should have a large set of semantics.

The NFL theorem states that any two search algorithms are equivalent when their performance is averaged over all possible problems. If an algorithm is particularly good at searching over some arrangement of data, then it will be equally bad at searching some other arrangement:

Conjecture 1 (NFL for Information)**.** *Let P be an elegant program such that there exists no smaller program to calculate the function of P, which uses the same encoding function e for the input.*

Any reduction in the size of P would necessarily require an increase in the size of the semantics for the model of P (i.e. more instructions), or a new encoding function g such that:

$$\forall x : e(x) < g(x)$$

Consider an elegant semantics and an elegant program. A reduction in the semantics via elimination of some rule which is used by the program will increase the size of the program. To further reduce an elegant program will precipitate an increase in the semantics and maybe the input encoding. An elegant program cannot decrease in size without the introduction of new operators via the semantics.

The formulation of the NFL hypothesis suggests that for every function there exists some minimal amount of information which is distributed over the semantics, program, and input encoding for some model.

Conjecture 2 (Information Invariance)**.** *For all model and computable function pairs, there exists an information invariant i and overhead c. The value $i + c$ is distributed over the semantics, program, and encoding function. The program and encoding function are optimal when c is minimised, and that any further reduction of information in the semantics, program, or encoding function will correspond to a rise in information in the other two.*

7.3.4 Model Attributes

The radical difference in internal representation (array vs graph) and in evaluation method (sequential vs graph reduction) is believed to cause the disconnect between the TIs of the imperative and functional paradigms. It may be that some operators of the SOS formalism which are used in one paradigm but not the other contribute a large amount of computational power. Section 7.3.5 discusses how this could be accounted for.

Irrespective of the paradigm, there is a dramatic difference in the TI between the models with large semantics and the models with small semantics. This occurs most notably in the representations of the universal RASP and universal TM. The TI for the SKI and TM representations increase drastically when implementing these programs opposed to the RASP and λ-calculus implementations.

This is hypothesised to be precipitated by the difference in memory models between the less expressive models and the more expressive ones. The less expressive models use sequential access/reduction while the more expressive models have random access and arbitrary substitution:

Conjecture 3 (Model Attributes)**.** *The difference observed between the information contents of the TM/SKI and RASPs/λ-calculus is caused by the existence (or lack thereof) of random access memory structures in the models.*

There may exist more of these "jumps" in required TI. A non-deterministic model of computation is a model which leverages probability in order to compute.

Such a model has a valid program if there exists at least one valid computation path which returns the correct output. Non-deterministic varieties of all models of computation exist. There are non-deterministic TMs, reduction strategies, RAM and RASP machines [86, 32].

Because the machine can make a choice as to which computation path to execute, decisions which would ordinarily be highly specified need not be. This leads to a saving in the number of instructions, and thus information, needed to specify the decision paths of the machine.

Conjecture 4 (More Model Attributes)**.** *There exist other model attributes which precipitate a large difference in the required TI for programs similar to what has been observed in this thesis. It is conjectured that models with such attributes would not require as much TI as the models without.*

7.3.5 Symbol Grounding

Those schooled in logic and mathematics are familiar with the meaning of symbols like '$+$', '$\forall$', '$\times$', and '$\exists$'. They have been taught the functionality of what these symbols represent and know how and when to apply these functions to situations, and when not to.

Searle's famous *gedankenexperiment*, The Chinese Room [81], was written as an indictment against the proponents of Strong AI[1]. Searle asserted that the manipulation of symbols by some fixed set of instructions could be mistaken as consciousness when it is merely the following of instructions. The arguments for and against this position here will not be discussed here, but Searle's paper raises the question: at what point in a computational system are meanings ascribed to the symbols which make up the language of the system? This is known as the *symbol grounding problem* [27, 91].

Chapter 5 discusses the problem of infinite regress. Attempting to ground the functions of Structured Operational Semantics in some other expressive formalism begs the question of how *that* formalism is grounded. In this thesis a solution to the problem was formulated by grounding the models in FPGAs, but there is

[1] A philosophical position which states that there exists a computer program which embodies the attributes of consciousness/cognition.

another possible solution.

The semantics of the models in SOS presented herein are approximations. The functions of SOS have been taken as a baseline, but some models may use different aspects of SOS than others. The RASP and TM use numerals (natural numbers for the RASP and integers for the TM) and the SKI and λ-calculus use set indirection to reason about sub-trees.

The SKI and λ-calculus models do not require numerals for their operation; similarly RASP and TM do not require set indirection. However, a flat baseline like SOS would account for both. A semantic system could be devised where the operations of the semantic system are derived from the base axioms of a formalism such as First Order Logic, Zermelo-Frankel set theory, or Russell's type theory.

Not all of these systems are self contained however. The existential and universal quantifiers are a part of First Order Logic, but required for set theory. Furthermore, some operators cannot be defined in a lower system. The existential and universal quantifiers are axiomatic in their system. Such concepts will be *elementary definitions and axioms*. If FOL and ZFC were to be used, a few of these elementary definitions would include:

- Sets
- Variables
- Set Membership
- Existential/Universal Quantifiers (pick one)
- Zero

These definitions form the baseline, as concepts so basic such that there is no mathematical expression to define them. The constrained notation for mathematics is inadequate to define such concepts so natural language must be employed.

A logic constructed as such allows the information content of each logical construction built upon these axioms to be tracked. A semantic system as expressive as SOS, based on this axiomatic foundation would have an information value for its operators. Thereby any semantics which use an operator pays the information "price".

Formulating the semantics in this system would give a much higher resolution view of the information content of the semantics. It may then be possible to

perform experiments where a root semantics is modified multiple ways and the effect of such changes are measured across the set of test programs. This would also facilitate study of hybridised languages, for example which contain both imperative and functional subsets, and can judge if a 'best of both worlds' language provides benefits to mean program size.

7.3.6 Other Work

Imperative/Functional Comparisons It is clear with the resolution of the SI/TI across paradigms hypotheses that the relative expressivity of models across paradigms cannot be determined by TI alone. It is suspected that this is due to the vast difference in evaluation methodologies. This is not confirmed, so further work into investigating the information link across paradigms may confirm it.

Alternate Semantics Section 7.2.4 argues the notion that the measurements are relative only to the very specific representations and evaluation methodology as defined in the semantics. What is not known is if the hypotheses hold true for other models and semantic schemes. Further work here would be in the implementation of the models described here for other semantic systems and evaluating the hypotheses for these.

Real Applications While the FPGA realisations of the models do not provide useful data on the relative TI of models, it does provide an indication of the TI of programs in a singular model. This information could be generalised to the cross compilation of language subsets (such as C) to FPGAs. Measuring the TI of a C implementation may give an approximation of the size of the resulting circuit.

Bibliography

[1] IEEE standard VHDL language reference manual. *IEEE Std 1076-1987*, 1988.

[2] IEEE standard VHDL language reference manual. *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)*, pages c1–626, Jan 2009.

[3] M. A. Arbib. Brains, machines, and mathematics, 1964.

[4] G. Beekmans, M. Burgess, N. Coulson, et al. Linux from scratch. `http://www.linuxfromscratch.org`, 1999.

[5] M. Blum. A machine-independent theory of the complexity of recursive functions. *J. ACM*, 14(2):322–336, 1967.

[6] M. Blum. On the size of machines. *Information and Control*, 11(3):257–265, 1967.

[7] C. Calude, S. Marcus, and I. Tevy. The first example of a recursive function which is not primitive recursive. *Historia Mathematica*, 6(4):380 – 384, 1979.

[8] E. Cantú-Paz. A survey of parallel genetic algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10, 1998.

[9] G. Chaitin. *The Unknowable*. Springer, 1999.

[10] G. J. Chaitin. *The Limits of Mathematics : A Course on Information Theory and the Limits of Formal Reasoning (Discrete Mathematics and Theoretical Computer Science)*. Springer, Oct. 2002.

[11] A. Church. A set of postulates for the foundation of logic. *Annals of Mathematics*, 33(2):pp. 346–366, 1932.

[12] A. Church. An unsolvable problem of elementary number theory. *American Journal of Mathematics*, 58(2):345–363, April 1936.

[13] M. D. Ciletti. *Modeling, Synthesis, and Rapid Prototyping with the Verilog HDL*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1999.

[14] D. Coelho. *The VHDL Handbook*. Springer US, 2012.

[15] D. I. A. Cohen. *Introduction to computer theory*. Wiley, New York, 1986. Includes index.

[16] S. A. Cook and R. A. Reckhow. Time-bounded random access machines. In *Proceedings of the fourth annual ACM symposium on Theory of computing*, STOC '72, pages 73–80, New York, NY, USA, 1972. ACM.

[17] H. Curry, R. Feys, and W. Craig. *Combinatory Logic*. Number v. 1 in Studies in logic and the foundations of mathematics. North-Holland, 1968.

[18] J. Davidson and G. Michaelson. Brute force is not ignorance. In *Informal Proceedings of Computability in Europe*, CiE, 2013.

[19] J. Davidson and G. Michaelson. Elegance, meanings and machines. *Computability*, 2015.

[20] M. Davis. *Computability & Unsolvability*. Dover, 1958.

[21] N. de Bruijn. Lambda calculus notation with nameless dummies, a tool for automatic formula manipulation, with application to the church-rosser theorem. *Indagationes Mathematicae (Proceedings)*, 75(5):381 – 392, 1972.

[22] S. K. Debray, K. P. Coogan, and G. M. Townsend. On the semantics of self-unpacking malware code. Technical report, University of Arizona, 2008.

[23] C. C. Elgot and A. Robinson. Random-access stored-program machines, an approach to programming languages. *J. ACM*, 11(4):365–399, 1964.

[24] Euclid. *Elements*. Alexandria, 300BC.

[25] M. Felleisen. On the expressive power of programming languages. In *Science of Computer Programming*, pages 134–151. Springer-Verlag, 1990.

[26] R. W. Floyd. Assigning meanings to programs. *Proceedings of Symposium on Applied Mathematics*, 19:19–32, 1967.

[27] G. Frege. Sense and reference. *Philosophical Review*, 57(3):209–230, 1948.

[28] K. Gödel. On formally undecidable proposition in principa mathematica and related systems. *Monatshefte für Mathematik*, 149(1), 1931.

[29] B. Godlin and O. Strichman. Regression verification: proving the equivalence of similar programs. *Softw. Test., Verif. Reliab.*, 23(3):241–258, 2013.

[30] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[31] J. J. Gray. *The Hilbert Challenge.* Oxford University Press, 2003.

[32] J. Gruska. *Foundations of Computing.* International Thomson Computer Press, 1997.

[33] L. J. Hafer and A. C. Parker. Register-transfer level digital design automation: The allocation process. In *Proceedings of the 15th Design Automation Conference*, DAC '78, pages 213–219, Piscataway, NJ, USA, 1978. IEEE Press.

[34] M. H. Halstead. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Inc., New York, NY, USA, 1977.

[35] C. L. Hamblin. Translation to and from polish notation. *The Computer Journal*, 5(3):210–213, 1962.

[36] J. Hartmanis. Computational complexity of random access stored program machines. Technical report, Cornell University, 1970.

[37] D. Hilbert and M. W. Newton. Mathematical problems. *Bulletin of the American Mathimatical Society*, 33(4), 1927.

[38] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, Oct. 1969.

[39] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[40] P. Hudak, J. Hughes, S. Peyton Jones, and P. Wadler. A history of haskell: Being lazy with class. In *Proceedings of the Third ACM SIGPLAN Conference on History of Programming Languages*, HOPL III, pages 12–1–12–55, New York, NY, USA, 2007. ACM.

[41] D. Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, Sept 1952.

[42] J. Hughes. Why functional programming matters. *Comput. J.*, 32(2):98–107, Apr. 1989.

[43] International Standards Organisation. ISO C standard 1999. Technical report, 1999. ISO/IEC 9899:1999 draft.

[44] International Standards Organisation. *ISO/IEC 14882:2011 Information technology – Programming languages – C++*. International Organization for Standardization, Geneva, Switzerland, Feb. 2012.

[45] B. W. Kernighan. *The C Programming Language*. Prentice Hall Professional Technical Reference, 2nd edition, 1988.

[46] S. C. Kleene. *Introduction to Metamathematics*. Bibl. Matematica. North-Holland, Amsterdam, 1952.

[47] A. N. Kolmogorov. On tables of random numbers. *Theor. Comput. Sci.*, 207(2):387–395, Nov. 1998.

[48] J. R. Koza. *Genetic Programming: On the Programming of Computers by Means of Natural Selection*. MIT Press, Cambridge, MA, USA, 1992.

[49] D. C. Kozen. *Automata and Computability*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1st edition, 1997.

[50] E. V. Krishnamurthy. *Introductory Theory of Computer Science*. Springer-Verlag, 1985.

[51] S. Kundu, Z. Tatlock, and S. Lerner. Proving optimizations correct using parameterized program equivalence. *SIGPLAN Not.*, 44(6):327–337, June 2009.

[52] P. J. Landin. The next 700 programming languages. *Commun. ACM*, 9(3):157–166, Mar. 1966.

[53] R. E. Lewand. *Cryptological Mathematics*. Mathematical Association of America, Washington, DC, USA, 1st edition, 2000.

[54] S. Lin and T. Rado. Computer studies of turing machine problems. *J. ACM*, 12(2):196–212, Apr. 1965.

[55] P. Lucas and K. Walk. On the formal description of pl/i. *Annual Review in Automatic Programming*, 6:105–182, 1969.

[56] J. Łukasiewicz. *Selected Works*. Amsterdam,North-Holland Pub. Co., 1970.

[57] A. A. Markov. The theory of algorithms. *Russian Academy of Science*, 1954.

[58] H. Marxen and J. Buntrock. Attacking Busy Beaver 5. *Bulletin of the European Association for Theoretical Computer Science*, 40, 1990.

[59] F. Mavaddat and B. Parhami. *URISC: the Ultimate Reduced Instruction Set Computer*. Research report. University of Waterloo, Faculty of Mathematics, 1987.

[60] O. Mazonka and A. Kolodin. A simple multi-processor computer based on subleq. *CoRR*, abs/1106.2593, 2011.

[61] T. J. McCabe. A complexity measure. In *Proceedings of the 2Nd International Conference on Software Engineering*, ICSE '76, pages 407–, Los Alamitos, CA, USA, 1976. IEEE Computer Society Press.

[62] P. Michel. The busy beaver competition: a historical survey, 2012.

[63] M. L. Minsky. *Computation: Finite and Infinite Machines.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1967.

[64] E. Nagel and J. Newmann. *Gödels Proof.* NYU Press, 1967.

[65] T. Neary. *Small universal Turing machines.* PhD thesis, NUI Maynooth, 2007.

[66] T. Neary and D. Woods. Four small universal turing machines. *Fundam. Inform.*, 91(1):123–144, 2009.

[67] A. J. Perlis. Special feature: Epigrams on programming. *SIGPLAN Not.*, 17(9):7–13, Sept. 1982.

[68] S. L. Peyton Jones. *The Implementation of Functional Programming Languages (Prentice-Hall International Series in Computer Science).* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[69] A. M. Pitts. Operational semantics and program equivalence. In *Applied Semantics, International Summer School, APPSEM 2000, Caminha, Portugal, September 9-15, 2000, Advanced Lectures*, pages 378–412, London, UK, UK, 2002. Springer-Verlag.

[70] G. D. Plotkin. Structural approach to operational semantics. Technical report, Aarhus University, 1981.

[71] G. D. Plotkin. The origins of structural operational semantics. *The Journal of Logic and Algebraic Programming*, 60:3–15, 2004.

[72] E. L. Post. Formal Reductions of the General Combinatorial Decision Problem. *American Journal of Mathematics*, 65(20):197–215, 1943.

[73] T. Rado. On non-computable functions. *The Bell System Technical Journal*, 41(3):877–884, 1962.

[74] V. Rayward-Smith. *A First Course in Computability.* Blackwell Scientific Publications, 1986.

[75] J. C. Reynolds. Algol-like languages, volume 1. chapter The Essence of ALGOL, pages 67–88. Birkhauser Boston Inc., Cambridge, MA, USA, 1997.

[76] H. Rogers, Jr. *Theory of Recursive Functions and Effective Computability.* MIT Press, Cambridge, MA, USA, 1987.

[77] Y. Rogozhin. Small universal turing machines. *Theoretical Computer Science*, 168(2):215 – 240, 1996.

[78] J. E. Savage. *Models of Computation: Exploring the Power of Computing.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1997.

[79] M. Schönfinkel. über die bausteine der mathematischen logik. *Mathematische Annalen*, 92(3-4):305–316, 1924.

[80] D. S. Scott. Outline of a Mathematical Theory of Computation. Technical Report PRG–2, Oxford, England, November 1970.

[81] J. R. Searle. Minds, brains, and programs. *Behavioral and Brain Sciences*, 3:417–424, 1980.

[82] J. P. Seldin. The Logic of Curry and Church. 2006.

[83] C. E. Shannon and W. Weaver. *A Mathematical Theory of Communication.* University of Illinois Press, Champaign, IL, USA, 1963.

[84] V. Y. Shen, S. D. Conte, and H. E. Dunsmore. Software science revisited: A critical analysis of the theory and its empirical support. *IEEE Trans. Softw. Eng.*, 9(2):155–165, Mar. 1983.

[85] M. J. Shepperd and D. C. Ince. A critique of three metrics. *Journal of Systems and Software*, 26:197–210, 1994.

[86] M. Sipser. *Introduction to the Theory of Computation.* International Thomson Publishing, 1st edition, 1996.

[87] R. Solomonoff. A formal theory of inductive inference. part i. *Information and Control*, 7(1):1 – 22, 1964.

[88] R. Solomonoff. A formal theory of inductive inference. part ii. *Information and Control*, 7(2):224 – 254, 1964.

[89] M. Stay. Very simple chaitin machines for concrete AIT. *Computing Research Repository*, 2005.

[90] G. L. Steele and G. J. Sussman. Lambda: The ultimate imperative. Technical report, Cambridge, MA, USA, 1976.

[91] M. Taddeo and L. Floridi. Solving the symbol grounding problem: a critical review of fifteen years of research. *Journal of Experimental and Theoretical Artificial Intelligence*, 17:419–445, 2005.

[92] The Unicode Consortium. The unicode standard 6.0.0. Technical report, The Unicode Consortium, 2011.

[93] A. S. Troelstra. *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*. New York,Springer, 1973.

[94] J. Tromp. Binary lambda calculus and combinatory logic. 2014.

[95] A. M. Turing. On Computable Numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, 2(42):230–265, 1936.

[96] A. M. Turing. Computability and λ-definability. *The Journal of Symbolic Logic*, 2(4):pp. 153–163, 1937.

[97] A. M. Turing. Systems of Logic Based on Ordinals. *Proceedings of The London Mathematical Society*, s2-45:161–228, 1939.

[98] D. A. Turner. Another algorithm for bracket abstraction. *The Journal of Symbolic Logic*, 44(2):pp. 267–270, 1979.

[99] D. A. Turner. A new implementation technique for applicative languages. *Software: Practice and Experience*, 9(1):31–49, 1979.

[100] D. A. Turner. The semantic elegance of applicative languages. In *Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture*, FPCA '81, pages 85–92, New York, NY, USA, 1981. ACM.

[101] J. van Heijenoort. *From Frege to Gödel: A Source Book in Mathematical Logic, 1987-1931*. Harvard University Press, 1967.

[102] C. Wadsworth. *Semantics and Pragmatics of the Lambda-calculus*. University of Oxford, 1971.

[103] D. H. Wolpert and W. G. Macready. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation*, 1(1):67–82, 1997.

[104] E. V. Wright. *Gadsby*. Wetzel Pbublishing Co., 1939.

[105] Xilinx Inc. 7 series FPGA configurable logic block. Technical report, Xilinx Inc., 2014.

[106] Xilinx Inc. Zynq-7000 all programmable SoC overview. Technical report, Xilinx Inc., 2014.

[107] Y. Zhang. Bounded gaps between primes. *The Annals of Mathematics*, 2014.

Appendix A

The Busy Beaver Problem

The 'Busy Beaver game' was first formulated by Radó [73] in order to showcase an example of a simple undecidable problem. The game is a competition amongst Turing machine programmers to find the Turing machine of a certain number of states which, when started on a blank tape, writes the most symbols to the tape before halting.

More formally, Radó defined the game using n state, 2 symbol machines and there was a different category for each n. A current 'champion' machine is a pair (M, s) where M is the machine and s is number of steps before halting. Checking the champion then became a trivial task of running M for s steps and counting the number of 1's on the tape to ensure correctness.

Brady generalised the game to include k symbols [62] which introduced a new set of classes for machines to fall into. A busy beaver champion (M, s) fits into the class $\Sigma(n, k)$ when M has n states and k symbols.

There is a championship for the number of steps a machine will make as well as for the number of non-blank symbols on the tape, because a champion of symbols will not necessarily be a champion stepper and vice versa. The class analogous to $\Sigma(n, k)$, $S(n, k)$ is the class for champion steppers.

A.1 Turing Machine Busy Beavers

Soon after the definition of the busy beaver game. Lin and Radó [54] performed an exhaustive search of the classes (2,2) and (3,2). The size of the machine space is as follows:

$$((n + 1) \times 2k)^{nk}$$

where n and k are as defined above. This results in around 17 million machines for the (3,2) class, but normalisation techniques filter out machines that, immediately halt or do not print a 1 as their first action. This filtering reduces the number of possible champions to 82,944, which were tested for halting behaviour.

Trivial non-halting machines were filtered out and the non-trivial ones were executed by hand to determine their operation. As the authors note, there were no machines so complicated as to make it impossible to assert halting behaviours by hand. They concluded that $S(2, 2) = 6$, $\Sigma(2, 2) = 4$, $S(3, 2) = 21$, and $\Sigma(3, 2) = 6$.

At this time, 4 classes of busy beaver machines have had confirmed S and Σ

Date	Discoverer(s)	Bounds
1963	Radó, Lin	$S(2,2) = 6$, $\Sigma(2,2) = 4$ $S(3,2) = 21$, $\Sigma(3,2) = 6$
1964	Brady	$S(4,2) = 107$, $\Sigma(4,2) = 13$
February 1990	Marxen, Buntrock	$S(5,2) \geq 47,176,870$, $\Sigma(5,2) \geq 4098$
February 2005	T. and S. Ligocki	$S(2,4) \geq 40,737$, $\Sigma(2,4) \geq 3,932,964$
November 2007	T. and S. Ligocki	$S(3,3) \geq 119,112,334,170,342,540$, $\Sigma(3,3) \geq 374,676,383$ $S(2,5) > 1.9 \times 10^{704}$,$\Sigma(2,5) > 1.7 \times 10^{352}$
December 2007	T. and S. Ligocki	$S(3,4) > 5.2 \times 10^{13036}$, $\Sigma(3,4) > 3.7 \times 10^{6518}$
January 2008	T. and S. Ligocki	$S(4,3) > 1 \times 10^{14072}$, $\Sigma(4,3) > 1.3 \times 10^{7936}$ $S(2,6) > 2.4 \times 10^{9866}$, $\Sigma(2,6) > 1.9 \times 10^{4933}$
June 2010	Kropitz	$S(6,2) > 7.4 \times 10^{36534}$, $\Sigma(6,2) > 3.4 \times 10^{18267}$

Table A.1: Currently known lower bounds of the explored classes (2012 [62]).

scores with machines to match: BB(1,2), BB(2,2) BB(3,2) and BB(4,2). Marxen and Buntrock [58] have established lower bounds for the class (5,2) at $S(5,2) \geq 47,176,870$ and $\Sigma(5,2) \geq 4098$.

The father and son team of Terry and Shawn Ligocki have made progress in exploring the space of machines with more than 2 symbols by using simulated annealing techniques to obtain high scoring machines [62]. They currently hold the record for many of these classes.

Table A.1, by way of Michel [62] shows the current records for a few of the classes as of June 2012.

A.2 RASP Busy Beavers

A busy beaver variant for the RASP machine can be defined though the execution of the 'OUT' instruction. For a class of n-bit machines $\Sigma(n)$ is the competition for the number of times the 'OUT' command is executed, while $S(n)$ is the competition for the number of fetch-execute cycles performed.

The mapping of instructions to naturals in all RASP definitions (including the one presented earlier in Section 2.3.1.2) are arbitrary. There is no real reason for INC to be mapped to 1 and CPY to be mapped to 7. This isn't such a problem in the literature concerned with runtimes [16, 36] but in the investigation of machines with maximal output, we want to be thorough in considering all of the possibilities.

To facilitate this, we extend the RASP model as to admit an arbitrary mapping of naturals to instructions. We constrain the range to 2^n so that a machine cannot map an instruction to a natural that the machine cannot represent. Similarly, the mapping is injective. An entrant into the competition $BBR(n)$ is thus a pair $R(p,i)$ of the program p (of size 2^n) and the instruction set mapping i.

Unlike the BB problem for TMs, the RASP version is computable because the halting problem for finite RASPs is computable.

Theorem 3 (Halting problem decidability)**.** *The Halting problem for the finite RASP is decidable.*

Proof. Consider a finite n-bit RASP machine M. We define the state of M to be the entire memory at a particular time, and each fetch-decode-execute cycle as a transition from one state to another. Since there is only a finite range of values for a finite number of memory locations, we can calculate the maximum number of possible states for any given machine $numStates(n) = n^n$.

Because each fetch-decode-execute cycle performs a transition between states $S \to S'$ we can run the machine for at most $numStates(n)$ cycles, storing each visited state as it is encountered and checking the store for the new state after every state transition. If we encounter the same state twice, a loop has occurred and can conclude that for some state X which is entered during execution of the machine, there exists a transitive closure over a relation R such that XR^+X. From which we can conclude that M will never halt. □

A.3 Finding the Champions

Assuming the RASP has eight instructions, the number of unique instruction set mapping for an n bit machine is:

$$PI(n) = \prod_{n-8<i\leq n} i$$

Each potential program is a sequence of 2^n natural numbers. Of these, the PC, IR and ACC are initialised at $\{3\ 0\ 0\}$. Each program is a base n number of length $2^n - 3$ so that that the formula to calculate the number of possible initial RASP machines is $PR(n) = (2^n)^{2^n-3}$.

A.3.1 Brute Force Methods

For 3 bit RASP machines, $PR(3) \times PI(3) = 1,321,205,760$. This is a feasible number to search through in a parallel brute force manner.

The parallel architecture was designed as a pseudo-task farm. Each node has an unique identifying integer (id) and knows how many nodes are working on the problem. The node with an id of zero was designated the master node.

Upon initialisation of the search, the nodes use their ids to work out which block of instruction set mappings they should explore. They proceed to run each of their assigned mappings against every n-bit RASP machine, recording the highest shifter and highest 'OUT' executor. Once a node has searched though all of the mappings and has its champion machines, it returns them to the master node which finds the overall champions and outputs them. Non-halting behaviour is detected by storing each state in a binary tree. If a state is already in the tree when visited, the machine is forcibly halted and discarded.

This entire procedure takes around 6 minutes on 32 cores of a 256 core Beowulf cluster consisting of 8 core Intel Xeon CPUs clocked at 2.13GHz. Figures A.1a and A.1b show the top scoring machines for $\Sigma(3) = 47$ and $S(3) = 112$ respectively.

Instr	I Label
3	:PC
0	:IR
0	:ACC
INC	:start
INC	
OUT	
OUT	
INC	

(a) The best 3-bit OUT machine

Instr	I Label
3	:PC
0	:IR
0	:ACC
INC	:start
OUT	
DEC	
INC	
JGZ	

(b) The best 3-bit steps machine

The instruction sets for these machines are:

$A.1a$ $\{0 \mapsto OUT, 1 \mapsto LOAD, 2 \mapsto DEC, 3 \mapsto INC, 4 \mapsto CPY, 5 \mapsto STO, 6 \mapsto HALT, 7 \mapsto JGZ\}$

$A.1b$ $\{0 \mapsto DEC, 1 \mapsto LOAD, 2 \mapsto STO, 3 \mapsto JGZ, 4 \mapsto OUT, 5 \mapsto HALT, 6 \mapsto INC, 7 \mapsto CPY\}$

The machine space $PR(4) = 4,503,599,627,370,496$ is an infeasible number of machines to search through in any reasonable time. So more advanced methods must be employed.

A.3.2 Genetic Algorithms

A genetic algorithm is a problem solving strategy which models natural selection [30]. It begins with an initial *pool* of (often randomly generated) solutions to some problem. Each potential solution is evaluated for *fitness* to determine how effective they are at solving the problem.

A subset of solutions are selected and bred together by means of *crossover* and *mutation*. Those not selected for reproduction are killed off and breeding refills the pool of candidates. The fitness of a solution improves the chance of it being selected for reproduction, but doesn't guarantee it.

A.3.2.1 Selection and Breeding

A solution for the RASP busy beaver is a pair of the program and the instruction set mapping. These are represented in memory as two arrays of length $2^n - 3$ and 8 respectively. We refer to these two arrays as *chromosomes* and the individual elements of the arrays as *genes*.

The fitness scores of a candidate is calculated as the number of steps/number of 'OUT's (dependent on whether our search is for $S(n)$ or $\Sigma(n)$) if the machine halts, otherwise it is 1.

Selection is handled through roulette wheel selection [30]. Imagine a roulette wheel sized such that it accommodates all candidates and each candidate has a 'slice' of the wheel proportional to its fitness (Figure A.2).

When selecting a candidate, we conceptually bounce a ball over the surface of the wheel. The distance that the ball can bounce is calculated as a random proportion of sums of all the fitnesses. As it moves round the wheel and passes

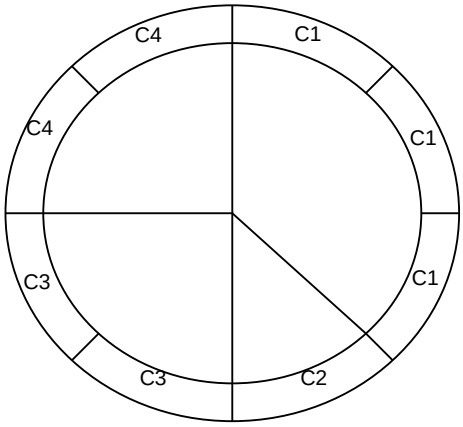


Figure A.2: A depiction of the roulette wheel we use to select candidates. C1 is the candidate with the highest fitness, so it gets the highest proportion of the wheel. C2 has the lowest fitness. C3 and C4 are equal in fitness.

Bits	Results	Comments
3	$S(3) = 112, \Sigma(3) = 47$	Exact values found through brute force searching.
4	$S(4) \geq 3413, \Sigma(4) \geq 1483$	Genetic, Pool: 100000, Generations: 1000, Islands: 32

Table A.2: Current records for numbers of shifts and outputs.

over candidates, it uses up its allowable distance. Once all of the distance has been used, it stops. The candidate that it stops on is then removed from the wheel, the wheel is resized, and the process starts again until the breeding population target has been met.

Crossing chromosomes involves picking two of the solutions and choosing a random point on one of them. The new chromosome is created by taking the genes of the first parent up to the random point, then taking the genes of the second parent past that point. Mutation of the program picks a random gene in a chromosome and changes it to some other gene. Mutation of the instruction set swaps two genes to maintain an injective mapping.

Repopulating the pool picks two parents at random and selects a parent to be 'dominant'. There is a 1/3 chance that the programs get crossed, a 1/3 chance that the instruction sets get crossed (while still adhering to the injective rules for the instruction sets) and a 1/3 chance that both get crossed. If a chromosome isn't to be crossed, the chromosome from the dominant parent is copied. There is a small (5%) chance that the program or instruction set will be mutated.

A.3.2.2 Current Results

Table A.2 shows the current results of the investigation while Table A.3 demonstrates the record holding instruction sets and programs.

The optimal strategy to evolve good machines seems to stem from repeatedly seeding the current champion machine into the algorithm. What this does is seed the initial pools with the current champion machine in the hope that it will be

Record Held	Instruction Set	Program
$S(3) = 112$	{5,6,0,1,2,4,3,7}	{6,4,0,6,3}
$\Sigma(3) = 47$	{6,3,2,1,5,0,7,4}	{3,3,0,0,3}
$S(4) \geq 3413$	{14,3,13,0,6,9,4,15}	{9,3,6,3,4,12,9,13,6,9,3,4,7}
$\Sigma(4) \geq 1483$	{2,6,7,5,1,3,4,0}	{3,3,6,3,3,4,4,5,1,7,1,11,4}

Table A.3: Instruction sets and programs of record holding machines. Instruction mapping is {HALT,INC,DEC,LOAD,STO,OUT,JGZ,CPY}.

improved upon. This is a manual version of the migration strategy laid out above and the author has seen success with hand constructing a seed and letting the algorithm evolve it into a better version.

A.4 Reflection

The investigation outlined was not as enlightening as one would hope. This section reflects on how we structured our algorithm and hardware and what we should do differently for a fresh investigation.

A.4.1 Landscape and Fitness

As with all informed search methods, there is the danger of local maxima. Randomly generating and evolving solutions can achieve good results, but with a search space as large as $n > 3$ we cannot hope to obtain a statistically beneficial initial 'spread' of candidates across the solution landscape. Furthermore, the landscape itself is *exceptionally* jagged. The fitness function is not nearly sophisticated enough to effectively navigate the space. For example, changing any one of the record machines instructions to a HALT (say $\{6, 4, 0, 6, 3\} \leftrightarrow \{6, 4, 0, 5, 3\}$ where $5 \mapsto HALT$) will ruin the fitness score of the machine.

We could apply filters to our machine generator so that it accepts a HALT or unmapped natural number in the body of the machine only if it comes immediately after a LOAD, STO, JGZ, or CPY. This way, we would produce machines that don't instantly halt and that would need to compute, or specifically jump to some halting numeral before it will stop.

Another approach we could try comes from the field of computer security. Self modification is a typical obfuscation technique to disguise malicious code and attempts to combat it had resulted in the development of semantic models which decompose a self modifying binary into phases. These phases are statically analysed for malicious behaviour as normal [22].

We could possibly adopt this approach for larger spaces ($n > 6$). However we would have to experiment to ensure that this decomposition and analyses is faster than, or provides considerably more information than, just running the machine. Otherwise we will incur a greater time overhead per machine in a space where speed of execution is arguably more important.

Advanced static analyses as described above coupled with (non)halting detection could direct a genetic algorithm to target a specific neighbourhood of a candidate. If an n-bit candidate doesn't quite halt, but is otherwise a champion

machine, the problem could perhaps be narrowed down to k registers which need modification. A narrow number of registers can conceivably be brute forced for larger n's than what we've investigated so far. This very specific modification method strays into the remit of Genetic Programming [48].

A.4.2 Architecture and Seeding

The genetic algorithm was parallelised as the brute force algorithm. Each process contains its own pool, the best solutions are evolved from the pool. Once the process has evolved a solution for n generations, it is sent back to the master process which judges the best overall solution.

This 'isolated island' approach tends to exhibit speciation (local maxima) across processes. A better approach may be to migrate the top solutions from the pools every few generations [8]. This re-seeds the pool with the current best solution to the problem, increasing the chances of evolving the current solution into an even better one.

Appendix B

Full Programs

This appendix presents the measured programs for each of the models investigated in this thesis. The programs here are what is measured to obtain the character counts exemplified in Table 4.2 et al. and are analysed in Chapter 6.

B.1 RASP

The RASP programs are presented in two ways: the "programming language form" as seen all throughout this thesis, and the "array form" which is what is actually measured.

B.1.1 Addition

Instr	Data	I Label	D Label
LOAD	3	:addStart	;x
JGZ	'adding		
HALT			
DEC		:adding	
STO	'x		
LOAD	4		;y
INC			
STO	'y		
LOAD	1		
JGZ	'addStart		

```
3,5,6,8,0,2,4,4,3,8,1,4,12,3,1,6,3,0,0,0,0,0,0,0,0,0,0,0
```

B.1.2 Subtraction

Instr	Data	I Label	D Label
LOAD	4	:sub_start	;sub_2
JGZ	'subbing		
HALT			
DEC		:subbing	
STO	'sub_2		
LOAD	7		;sub_1
JGZ	'subbing2		
HALT			
DEC		:subbing2	
STO	'sub_1		
JGZ	'sub_start		

```
3,4,6,8,0,2,4,4,3,7,6,16,0,2,4,12,3,1,6,3,0,0,0,0,0,0,0,0,0
```

B.1.3 Equality

Instr	Data	I Label	D Label
LOAD	6	:dec1	;cmp1
DEC			
STO	'cmp1		
LOAD	5		;cmp2
DEC			
STO	'cmp2		
JGZ	'dec1		
CPY	'cmp1		
JGZ	0		
LOAD	1		
HALT			

```
3,6,2,4,4,3,5,2,4,9,6,3,7,4,6,0,3,1,0,0,0,0,0,0,0,0,0,0,0
```

B.1.4 Multiplication

Instr	Data	I Label	D Label
CPY	'multiplier		
JGZ	'return		
HALT			
LOAD	5	:return	;multiplicand
JGZ	'mul_start		
HALT			
DEC		:mul_start	
STO	'multiplicand		
LOAD	5		;multiplier
STO	'tmp		
LOAD	0	:loop	;tmp
JGZ	'add		
LOAD	1		
JGZ	'return		
DEC		:add	
STO	'tmp		
LOAD	0		;runningTotal
INC			
STO	'runningTotal		
LOAD	1		
JGZ	'loop		

```
7,17,6,8,0,3,5,6,13,0,2,4,9,3,5,4,21,3,0,6,28,3,1,6,8,2,4,21,
3,0,1,4,32,3,1,6,20,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0
```

B.1.5 Division

Instr	Data	I Label	D Label
LOAD	3	:start	;divisor
JGZ	'div_start		
HALT			
STO	'tmp	:div_start	
LOAD	7		;num
STO	'remainder		
LOAD	0	:loop	;tmp
JGZ	'sub		
LOAD	1		
JGZ	'return		
DEC		:sub	
STO	'tmp		
CPY	'num		
JGZ	'nl		
HALT			
DEC		:nl	
STO	'num		
LOAD	1		
JGZ	'loop		
LOAD	0	:return	;quotient
INC			
STO	'quotient		
JGZ	'start		
0		:remainder	

```
3,3,6,8,0,4,15,3,7,4,44,3,0,6,22,3,1,6,37,2,4,15,7,11,6,30,0,
2,4,11,3,1,6,14,3,0,1,4,38,6,3,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0
```

B.1.6 Exponentiation

Instr	Data	I Label	D Label
LOAD	1	:start	;power
JGZ	'continue		
HALT			
DEC		:continue	
STO	'power		
LOAD	1		;runningTotal
STO	'multiplicand		
LOAD	0		
STO	'runningTotal		
LOAD	0	:return	;multiplicand
JGZ	'mulStart		
LOAD	1		
JGZ	'start		
DEC		:mulStart	
STO	'multiplicand		
LOAD	1		;multiplier
STO	'tmp		
LOAD	0	:loop	;tmp
JGZ	'add		
LOAD	1		
JGZ	'return		
DEC		:add	
STO	'tmp		
CPY	'runningTotal		
INC			
STO	'runningTotal		
LOAD	1		
JGZ	'loop		

```
3,1,6,8,0,2,4,4,3,1,4,20,3,0,4,12,3,0,6,27,3,1,6,3,2,4,20,3,
1,4,35,3,0,6,42,3,1,6,19,2,4,35,7,12,1,4,12,3,1,6,34,0,0,0,
0,0,0,0,0,0,0
```

B.1.7 List Membership

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer	:cmp_pointer_target	
STO	'indir_pointer		
CPY	0		;indir_pointer
STO	'cmp_1		
LOAD	0		;target
STO	'cmp_2		
LOAD	'end_test		
STO	'cmp_return_1		
LOAD	'equal		
STO	'cmp_return_2		
LOAD	0	:cmp_start	;cmp_1
DEC			
STO	'cmp_1		
LOAD	0		;cmp_2
DEC			
STO	'cmp_2		
JGZ	'cmp_start		
CPY	'cmp_1		
JGZ	0		;cmp_return_1
LOAD	1		
JGZ	0		;cmp_return_2
LOAD	0	:end_test	;pointer
STO	'cmp_1		
LOAD	'listend		
STO	'cmp_2		
LOAD	'inc_pointer		
STO	'cmp_return_1		
LOAD	'list_ended		
STO	'cmp_return_2		
JGZ	'cmp_start		
LOAD	1	:equal	
HALT			
CPY	'pointer	:inc_pointer	
INC			
JGZ	'cmp_pointer_target		
LOAD	0	:list_ended	
HALT			
			:listStart
			:listend

```
3,73,4,46,4,10,7,0,4,26,3,0,4,31,3,45,4,40,3,63,4,44,3,0,2,4,
26,3,0,2,4,31,6,25,7,26,6,0,3,1,6,0,3,0,4,26,3,72,4,31,3,66,4,
40,3,71,4,44,6,25,3,1,0,7,46,1,6,5,3,0,0,0,0,0,0,0,0,0,0,0,0,
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.1.8 Linear Search

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer	:cmp_pointer_target	
STO	'indir_pointer		
CPY	0		;indir_pointer
STO	'cmp_1		
LOAD	0		;target
STO	'cmp_2		
LOAD	'end_test		
STO	'cmp_return_1		
LOAD	'equal		
STO	'cmp_return_2		
LOAD	0	:cmp_start	;cmp_1
DEC			
STO	'cmp_1		
LOAD	0		;cmp_2
DEC			
STO	'cmp_2		
JGZ	'cmp_start		
CPY	'cmp_1		
JGZ	0		;cmp_return_1
LOAD	1		
JGZ	0		;cmp_return_2
LOAD	0	:end_test	;pointer
STO	'cmp_1		
LOAD	'listend		
STO	'cmp_2		
LOAD	'inc_pointer		
STO	'cmp_return_1		
LOAD	'list_ended		
STO	'cmp_return_2		
JGZ	'cmp_start		
CPY	'pointer	:equal	
STO	'cmp_1		
LOAD	'listStart		
STO	'cmp_2		
LOAD	'finish		
STO	'cmp_return_1		
STO	'cmp_return_2		
JGZ	'cmp_start		
CPY	'cmp_1	:finish	
HALT			
CPY	'pointer	:inc_pointer	

Instr	Data	I Label	D Label
INC			
JGZ	'cmp_pointer_target		
LOAD	'listend	:list_ended	
HALT			
		:listStart	
		:listend	

```
3,89,4,46,4,10,7,0,4,26,3,0,4,31,3,45,4,40,3,63,4,44,3,0,2,4,
26,3,0,2,4,31,6,25,7,26,6,0,3,1,6,0,3,0,4,26,3,88,4,31,3,82,
4,40,3,87,4,44,6,25,7,46,4,26,3,89,4,31,3,79,4,40,4,44,6,25,
7,26,0,7,46,1,6,5,3,88,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.1.9 List Reversal

Instr	Data	I Label	D Label
LOAD	'listEnd		
STO	'cpyPointer		
INC			
INC			
STO	'writePointer		
LOAD	0	:main	;writePointer
STO	'writeSTO		
LOAD	0		;cpyPointer
STO	'cpyLOC		
CPY	0		;cpyLOC
STO	0		;writeSTO
CPY	'writePointer		
INC			
STO	'writePointer		
CPY	'cpyPointer		
STO	'tmp1		
LOAD	'listStart		
STO	'tmp2		
LOAD	0	:loop	;tmp1
DEC			
STO	'tmp1		
LOAD	0		;tmp2
DEC			
STO	'tmp2		
JGZ	'loop		
CPY	'tmp1		
JGZ	'decWritePointer		
HALT			
CPY	'cpyPointer	:decWritePointer	
DEC			
STO	'cpyPointer		

Instr	Data	I Label	D Label
JGZ	'main		
		:listStart	
		:listEnd	

```
3,59,4,16,1,1,4,12,3,0,4,22,3,0,4,20,7,0,4,0,7,12,1,4,
12,7,16,4,37,3,58,4,42,3,0,2,4,37,3,0,2,4,42,6,36,7,37
,6,53,0,7,16,2,4,16,6,11,0,0,0,0
```

B.1.10 Stateful List Reversal

Instr	Data	I Label	D Label
LOAD	'listStart		
STO	'pointer1		
LOAD	'listEnd		
STO	'pointer2		
LOAD	0	:main	;pointer1
STO	'cmp1		
LOAD	0		;pointer2
STO	'cmp2		
LOAD	0	:loop	;cmp1
DEC			
STO	'cmp1		
JGZ	'compare2		
LOAD	0		;cmp2
DEC			
JGZ	'swap		
HALT			
CPY	'cmp2	:compare2	
DEC			
STO	'cmp2		
JGZ	'loop		
HALT			
CPY	'pointer1	:swap	
STO	'swpref1		
STO	'writeref1		
CPY	0		;swpref1
STO	'swp		
CPY	'pointer2		
STO	'swpref2		
STO	'writeref2		
CPY	0		;swpref2
STO	0		;writeref1
LOAD	0		;swp
STO	0		;writeref2
CPY	'pointer1		
INC			
STO	'pointer1		

Instr	Data	I Label	D Label
CPY	'pointer2		
DEC			
STO	'pointer2		
JGZ	'main		
		:listStart	
		:listEnd	

```
3,74,4,12,3,75,4,16,3,0,4,20,3,0,4,27,3,0,2,4,20,6,32,3,
0,2,6,40,0,7,27,2,4,27,6,19,0,7,12,4,47,4,59,7,0,4,61,7,
16,4,57,4,63,7,0,4,0,3,0,4,0,7,12,1,4,12,7,16,2,4,16,6,
11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.1.11 Bubble Sort

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer1		
INC			
STO	'pointer2		
LOAD	0		
STO	'flag		
LOAD	0	:cmp_pointers	;pointer1
STO	'p1ref		
CPY	0		;p1ref
STO	'cmp1		
LOAD	0		;pointer2
STO	'p2ref		
CPY	0		;p2ref
STO	'cmp2		
LOAD	'inc_pointers		
STO	'cmpOther		
STO	'equal1		
LOAD	'swap		
STO	'cmp1Greater		
LOAD	0	:cmp_start	;cmp2
DEC			
STO	'cmp2		
JGZ	'cmp1dec		
CPY	'cmp1		
DEC			
JGZ	0		;cmp1Greater
LOAD	1		
JGZ	0		;equal1
LOAD	0	:cmp1dec	;cmp1
DEC			
STO	'cmp1		

Instr	Data	I Label	D Label
JGZ	'cmp_start		
LOAD	1		
JGZ	0		;cmpOther
CPY	'pointer1	:inc_pointers	
INC			
STO	'pointer1		
CPY	'pointer2		
STO	'cmp2		
LOAD	'listend		
STO	'cmp1		
LOAD	'return_to_inc		
STO	'cmp1Greater		
LOAD	'foundEnd		
STO	'equal1		
STO	'cmpOther		
JGZ	'cmp_start		
CPY	'pointer2	:return_to_inc	
INC			
STO	'pointer2		
JGZ	'cmp_pointers		
LOAD	0	:foundEnd	;flag
JGZ	'start		
HALT			
CPY	'pointer2	:swap	
STO	'p2SwpRef		
STO	'p2WriteRef		
CPY	0		;p2SwpRef
STO	'swp		
CPY	'pointer1		
STO	'p1SwpRef		
STO	'p1WriteRef		
CPY	0		;p1SwpRef
STO	0		;p2WriteRef
LOAD	0		;swp
STO	0		;p1WriteRef
LOAD	1		
STO	'flag		
JGZ	'inc_pointers		
:listStart			
:listend			

```
3,130,4,15,1,4,23,3,0,4,100,3,0,4,19,7,0,4,57,3,0,4,27,7,0,
4,41,3,67,4,66,4,55,3,104,4,51,3,0,2,4,41,6,56,7,57,2,6,0,
3,1,6,0,3,0,2,4,57,6,40,3,1,6,0,7,15,1,4,15,7,23,4,41,3,132,
4,57,3,92,4,51,3,99,4,55,4,66,6,40,7,23,1,4,23,6,14,3,0,6,3,
0,7,23,4,111,4,123,7,0,4,125,7,15,4,121,4,127,7,0,4,0,3,0,4,
0,3,1,4,100,6,67,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
```

```
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0
```

B.1.12 Universal TM

Instr	Data	I Label	D Label
CPY	'CURR_ST	:PStart	
STO	'SE_ST		
CPY	'CHP		
STO	'SYMBOL_READ		
CPY	5		;SYMBOL_READ
STO	'SE_SY		
LOAD	'MSearchRET		
STO	'SeRetLoc		
JGZ	'SEStart		
LOAD	0	:MSearchRET	;SeResLoc
JGZ	'Vsearch		
HALT			
INC		:Vsearch	
INC			
STO	'NST_Read		
INC			
STO	'NSY_Read		
INC			
STO	'NDIR_READ		
CPY	0		;NST_Read
STO	'CURR_ST		
CPY	'CHP		
STO	'HP		
CPY	0		;NSY_Read
STO	0		;HP
CPY	0		;NDIR_READ
DEC			
JGZ	'DIR_RIGHT		
CPY	'CHP		
DEC			
STO	'CHP		
JGZ	'CONTINUE		
CPY	'CHP	:DIR_RIGHT	
INC			
STO	'CHP		
CPY	'CURR_ST	:CONTINUE	
JGZ	'PStart		
HALT			
LOAD	'SYT_START	:SEStart	
STO	'currentLoc		

Instr	Data	I Label	D Label
LOAD	0	:search_loop	;currentLoc
STO	'lc		
CPY	3		;lc
JGZ	'Valid_Tuple		
LOAD	1		
JGZ	'Not_Found		
LOAD	0	:Valid_Tuple	;SE_ST
STO	'CMP1		
CPY	'currentLoc		
STO	'tabcomp1		
CPY	5		;tabcomp1
STO	'CMP2		
LOAD	'cmp1_return		
STO	'CMP_RET_LOC		
JGZ	'CMP_START		
CPY	'CMP_RET	:cmp1_return	
JGZ	'nTupleSt		
CPY	'currentLoc		
INC			
STO	'currentLoc		
STO	'tabcomp2		
CPY	5		;tabcomp2
STO	'CMP1		
LOAD	0		;SE_SY
STO	'CMP2		
LOAD	'cmp2_return		
STO	'CMP_RET_LOC		
JGZ	'CMP_START		
CPY	'CMP_RET	:cmp2_return	
JGZ	'nTupleSy		
CPY	'currentLoc		
DEC			
STO	'SeResLoc		
LOAD	1		
JGZ	'searchExit		
CPY	'currentLoc	:nTupleSt	
INC			
STO	'currentLoc		
CPY	'currentLoc	:nTupleSy	
INC			
INC			
INC			
INC			
STO	'currentLoc		
JGZ	'search_loop		
LOAD	0	:Not_Found	
STO	'SeResLoc		

Instr	Data	I Label	D Label
LOAD	1	:searchExit	
JGZ	5		;SeRetLoc
LOAD	0	:CMP_START	;CMP1
DEC			
STO	'CMP1		
LOAD	0		;CMP2
DEC			
STO	'CMP2		
JGZ	'CMP_START		
CPY	'CMP1		
JGZ	'NotEqual		
STO	'CMP_RET	:Equal	
LOAD	1		
JGZ	'CMP_EXIT		
LOAD	1	:NotEqual	
STO	'CMP_RET		
JGZ	0	:CMP_EXIT	;CMP_RET_LOC
0	:CMP_RET		
'TAPE_START		:CHP	
1		:CURR_ST	
		:SYT_START	
		:TAPE_START	

```
7,202,4,87,7,201,4,12,7,5,4,120,3,21,4,171,6,70,3,0,6,26,0,1,
1,4,37,1,4,45,1,4,49,7,0,4,202,7,201,4,47,7,0,4,0,7,0,2,6,60,
7,201,2,4,201,6,65,7,201,1,4,201,7,202,6,3,0,3,194,4,75,3,0,4,
79,7,3,6,86,3,1,6,164,3,0,4,173,7,75,4,95,7,5,4,178,3,104,4,
199,6,172,7,200,6,142,7,75,1,4,75,4,116,7,5,4,173,3,0,4,178,3,
129,4,199,6,172,7,200,6,142,7,75,2,4,22,3,1,6,168,7,75,1,4,75,
4,150,7,2,6,142,7,75,1,4,75,4,161,7,0,6,74,3,0,4,22,3,1,6,5,3,
0,2,4,173,3,0,2,4,178,6,172,7,173,6,194,4,200,3,1,6,198,3,1,4,
200,6,0,0,196,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.1.13 Universal RASP

Instr	Data	I Label	D Label
LOAD	'PC_P		
INC			
INC			
INC			
STO	'OFF_PC		
CPY	'OFF_PC	:SIM_START	
STO	'INSLOC		
CPY	4		;INSLOC
STO	'IR_P		
STO	'Decoder_Ins		

Instr	Data	I Label	D Label
JGZ	'dec1		
HALT		:none	
DEC		:dec1	
JGZ	'dec2		
CPY	'ACC_P		
INC			
STO	'x		
STO	'ACC_P		
CPY	'MAX_INT		
STO	'y		
LOAD	'ACC_P		
STO	'sto_location		
LOAD	'done		
STO	'return_location		
JGZ	'TEST_LOOP		
DEC		:dec2	
JGZ	'dec3		
CPY	'ACC_P		
JGZ	'dc		
CPY	'MAX_INT		
STO	'ACC_P		
JGZ	'decST		
DEC		:dc	
STO	'ACC_P	:decST	
LOAD	1		
JGZ	'done		
DEC		:dec3	
JGZ	'dec4		
LOAD	'LOAD_RETURN		
STO	'FETCH_RETURN		
JGZ	'FETCH		
CPY	'IR_P	:LOAD_RETURN	
STO	'ACC_P		
LOAD	1		
JGZ	'done		
DEC		:dec4	
JGZ	'dec5		
LOAD	'STO_RETURN		
STO	'FETCH_RETURN		
JGZ	'FETCH		
CPY	'IR_P	:STO_RETURN	
STO	'OINT		
LOAD	'STO_O_RETURN		
STO	'OFFSET_RETURN		
JGZ	'OFFSET		
CPY	'OINT	:STO_O_RETURN	
STO	'sloc		

Instr	Data	I Label	D Label
CPY	'ACC_P		
STO	0		;sloc
LOAD	1		
JGZ	'done		
DEC		:dec5	
JGZ	'dec6		
OUT			
JGZ	'done		
DEC		:dec6	
JGZ	'dec7		
LOAD	'JGZ_RETURN		
STO	'FETCH_RETURN		
JGZ	'FETCH		
CPY	'ACC_P	:JGZ_RETURN	
JGZ	'JGZ_JUMP		
LOAD	1		
JGZ	'done		
CPY	'IR_P	:JGZ_JUMP	
STO	'PC_P		
STO	'OINT		
LOAD	'JGZ_O_RETURN		
STO	'OFFSET_RETURN		
JGZ	'OFFSET		
CPY	'OINT	:JGZ_O_RETURN	
STO	'OFF_PC		
JGZ	'SIM_START		
DEC		:dec7	
JGZ	'none		
LOAD	'CPY_RET		
STO	'FETCH_RETURN		
JGZ	'FETCH		
CPY	'IR_P	:CPY_RET	
STO	'OINT		
LOAD	'CPY_O_RET		
STO	'OFFSET_RETURN		
JGZ	'OFFSET		
CPY	'OINT	:CPY_O_RET	
STO	'cpyloc		
CPY	0		;cpyloc
STO	'ACC_P		
LOAD	1		
JGZ	'done		
LOAD	'SIM_START	:done	
STO	'INC_FR		
CPY	'PC_P	:INC_PC	
INC			
STO	'PC_P		

Instr	Data	I Label	D Label
STO	'x		
CPY	'MAX_INT		
STO	'y		
LOAD	'PC_P		
STO	'sto_location		
LOAD	'T_INC_RET		
STO	'return_location		
LOAD	1		
JGZ	'TEST_LOOP		
CPY	'PC_P	:T_INC_RET	
JGZ	'INC_OFFSET		
LOAD	'PC_P		
STO	'OFF_PC		
LOAD	1		
JGZ	'INC_EXIT		
CPY	'OFF_PC	:INC_OFFSET	
INC			
STO	'OFF_PC		
LOAD	1	:INC_EXIT	
JGZ	0		;INC_FR
LOAD	'PC_P	:OFFSET	
STO	'f		
LOAD	0	:OFFSET_LOOP	;OINT
INC			
STO	'OINT		
LOAD	0		;f
DEC			
STO	'f		
JGZ	'OFFSET_LOOP		
LOAD	1		
JGZ	0		;OFFSET_RETURN
LOAD	'fetch_r	:FETCH	
STO	'INC_FR		
JGZ	'INC_PC		
CPY	'OFF_PC	:fetch_r	
STO	'FETCH_VAR		
CPY	0		;FETCH_VAR
STO	'IR_P		
LOAD	1		
JGZ	0		;FETCH_RETURN
LOAD	0	;x	:TEST_LOOP
DEC			
STO	'x		
LOAD	0		;y
DEC			
STO	'y		
JGZ	'xtest		

Instr	Data	I Label	D Label
LOAD	1		
JGZ	'xtest2		
CPY	'x	:xtest	
JGZ	'TEST_LOOP		
LOAD	1		
JGZ	'RETURN		
CPY	'x	:xtest2	
JGZ	'INVALID		
LOAD	1		
JGZ	'RETURN		
LOAD	0	:INVALID	
STO	5		;sto_location
LOAD	1	:RETURN	
JGZ	0		;return_location
0	:Decoder_Ins		
0	:OFF_PC		
4	:MAX_INT		
		:PC_P	
		:IR_P	
		:ACC_P	

```
3,306,1,1,1,4,314,7,314,4,15,7,4,4,308,4,313,6,23,0,2,6,47,7,
310,1,4,274,4,310,7,315,4,279,3,310,4,308,3,187,4,312,6,273,2,
6,67,7,310,6,60,7,315,4,310,6,61,2,4,310,3,1,6,187,2,6,84,3,76,
4,272,6,255,7,308,4,310,3,1,6,187,2,6,115,3,93,4,272,6,255,7,
308,4,240,3,103,4,254,6,235,7,240,4,110,7,310,4,0,3,1,6,187,2,
6,121,5,6,187,2,6,156,3,130,4,272,6,255,7,310,6,138,3,1,6,187,
7,308,4,306,4,240,3,150,4,254,6,235,7,240,4,314,6,10,2,6,22,3,
165,4,272,6,255,7,308,4,240,3,175,4,254,6,235,7,240,4,180,7,0,
4,310,3,1,6,187,3,10,4,234,7,306,1,4,306,4,274,7,315,4,279,3,
306,4,308,3,214,4,312,3,1,6,273,7,306,6,226,3,306,4,314,3,1,6,
231,7,314,1,4,314,3,1,6,0,3,306,4,245,3,0,1,4,240,3,0,2,4,245,
6,239,3,1,6,0,3,261,4,234,6,191,7,314,4,266,7,0,4,308,3,1,6,0,
3,0,2,4,274,3,0,2,4,279,6,289,3,1,6,297,7,274,6,273,3,1,6,309,
7,274,6,305,3,1,6,309,3,0,4,5,3,1,6,0,0,0,4,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.2 RASP2

The RASP2 programs are presented in two ways: the "programming language form" as seen all throughout this thesis, and the "array form" which is what is actually measured.

B.2.1 Addition

Instr	Data
LOAD	x
ADD	y

```
3,5,1,8,0
```

B.2.2 Subtraction

Instr	Data	I Label	D Label
LOAD	y	:subStart	;y
JGZ	'subbing		
HALT			
SUB	1	:subbing	
STO	'y		
LOAD	x		;x
JGZ	'subbing2		
HALT			
SUB	1	:subbing2	
STO	'x		
LOAD	1		
JGZ	'subStart		

```
3,4,6,8,0,2,1,4,4,3,7,6,17,0,2,1,4,13,3,1,6,3,0,0,0,0,0,0,0
```

B.2.3 Equality

Instr	Data	I Label	D Label
LOAD	6		;num1
SUB	6		;num2
JGZ	'out		
HALT			
LOAD	1	:out	

```
3,6,2,6,6,10,0,3,1,0,0,0,0
```

B.2.4 Multiplication

Instr	Data	I Label	D Label
LOAD	5		;multiplier
JGZ	'return		
HALT			
LOAD	5	:return	;multiplicand
JGZ	'start		
HALT			
SUB	1	:start	
STO	'multiplicand		
CPY	'multiplier		
ADD	0		;runningTotal
STO	'runningTotal		
LOAD	1		
JGZ	'return		

```
3,5,6,8,0,3,5,6,13,0,2,1,4,9,7,4,1,0,4,20,3,1,6,8,0,0,0,0,0
```

B.2.5 Division

Instr	Data	I Label	D Label
LOAD	y	:start	;y
JGZ	'divStart		
HALT			
STO	'tmp	:divStart	
LOAD	x		;x
STO	'remainder		
LOAD	0	:loop	;tmp
JGZ	'sub		
LOAD	1		
JGZ	'return		
DEC		:sub	
STO	'tmp		
CPY	'x		
JGZ	'nl		
HALT			
DEC		:nl	
STO	'x		
LOAD	1		
JGZ	'loop		
LOAD	0	:return	;quotient
INC			
STO	'quotient		
JGZ	'start		
0		:remainder	

```
3,0,6,8,0,4,15,3,7,4,47,3,0,6,22,3,1,6,39,2,1,4,15,7,11,6,
31,0,2,1,4,11,3,1,6,14,3,0,1,1,4,40,6,3,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0
```

B.2.6 Exponentiation

Instr	Data	I Label	D Label
LOAD	1	:start	;power
JGZ	'continue		
HALT			
SUB	1	:continue	
STO	'power		
LOAD	1		;runningTotal
STO	'multiplicand		
LOAD	0		
STO	'runningTotal		
LOAD	0	:return	;multiplicand
JGZ	'mulStart		
LOAD	1		
JGZ	'start		
SUB	1	:mulStart	
STO	'multiplicand		
LOAD	1		;multiplier
STO	'addition		
CPY	'runningTotal		
ADD	0		;addition
STO	'runningTotal		
LOAD	1		
JGZ	'return		

```
3,1,6,8,0,2,1,4,4,3,1,4,21,3,0,4,13,3,0,6,28,3,1,6,3,2,1,
4,21,3,1,4,39,7,13,1,0,4,13,3,1,6,20,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0
```

B.2.7 List Membership

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer	:cmp_pointer_target	
STO	'indir_pointer		
CPY	0		;indir_pointer
STO	'cmp_1		
LOAD	4		;target
SUB	0		;cmp_1
JGZ	'end_test		
LOAD	1		
HALT			
LOAD	0	:end_test	;pointer
SUB	'listend		
JGZ	'inc_pointer		
LOAD	0		
HALT			
CPY	'pointer	:inc_pointer	
ADD	1		
JGZ	'cmp_pointer_target		
		:listStart	
		:listend	

```
3,35,4,23,4,10,7,0,4,16,3,4,2,0,6,22,3,1,0,3,0,2,36,6,31,
3,0,0,7,23,1,1,6,5,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0
```

B.2.8 Linear Search

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer	:cmp_pointer_target	
STO	'indir_pointer		
CPY	0		;indir_pointer
STO	'cmp_1		
LOAD	4		;target
SUB	0		;cmp_1
JGZ	'end_test		
CPY	'pointer		
SUB	'listStart		
HALT			
LOAD	0	:end_test	;pointer
SUB	'listend		
JGZ	'inc_pointer		
LOAD	'listend		
HALT			
CPY	'pointer	:inc_pointer	
ADD	1		
JGZ	'cmp_pointer_target		
		:listStart	
		:listend	

```
3,37,4,25,4,10,7,0,4,16,3,4,2,0,6,24,7,25,2,37,0,3,0,2,
38,6,33,3,38,0,7,25,1,1,6,5,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0
```

B.2.9 List Reversal

Instr	Data	I Label	D Label
LOAD	'listEnd		
STO	'cpyPointer		
ADD	2		
STO	'writePointer		
LOAD	0	:main	;writePointer
STO	'writeSTO		
LOAD	0		;cpyPointer
STO	'cpyLOC		
CPY	0		;cpyLOC
STO	0		;writeSTO
CPY	'writePointer		
ADD	1		
STO	'writePointer		
LOAD	'listStart		
STO	'lsSub		
CPY	'cpyPointer		
SUB	0		;lsSub
JGZ	'decWritePointer		
HALT			
CPY	'cpyPointer	:decWritePointer	
SUB	1		
STO	'cpyPointer		
JGZ	'main		
		:listStart	
		:listEnd	

```
3,47,4,16,1,2,4,12,3,0,4,22,3,0,4,20,7,0,4,0,7,12,1,1,4,
12,3,46,4,36,7,16,2,0,6,40,0,7,16,2,1,4,16,6,11,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0
```

B.2.10 Stateful List Reversal

Instr	Data	I Label	D Label
LOAD	'listStart		
STO	'pointer1		
LOAD	'listEnd		
STO	'pointer2		
LOAD	0	:main	;pointer1
STO	'cmp1		
LOAD	0		;pointer2
STO	'cmp2		
LOAD	0	:loop	;cmp1
SUB	1		
STO	'cmp1		

Instr	Data	I Label	D Label
JGZ	'compare2		
LOAD	0		;cmp2
SUB	1		
JGZ	'swap		
HALT			
CPY	'cmp2	:compare2	
SUB	1		
STO	'cmp2		
JGZ	'loop		
HALT			
CPY	'pointer1	:swap	
STO	'swpref1		
STO	'writeref1		
CPY	0		;swpref1
STO	'swp		
CPY	'pointer2		
STO	'swpref2		
STO	'writeref2		
CPY	0		;swpref2
STO	0		;writeref1
LOAD	0		;swp
STO	0		;writeref2
CPY	'pointer1		
ADD	1		
STO	'pointer1		
CPY	'pointer2		
SUB	1		
STO	'pointer2		
JGZ	'main		
		:listStart	
		:listEnd	

```
3,79,4,12,3,80,4,16,3,0,4,20,3,0,4,28,3,0,2,1,4,20,6,34,3,
0,2,1,6,43,0,7,28,2,1,4,28,6,19,0,7,12,4,50,4,62,7,0,4,64,
7,16,4,60,4,66,7,0,4,0,3,0,4,0,7,12,1,1,4,12,7,16,2,1,4,16,
6,11,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.2.11 Bubble Sort

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer1		
ADD 1			
STO	'pointer2		
LOAD	0		
STO	'flag		

Instr	**Data**	**I Label**	**D Label**
LOAD	0	:cmpPointers	;pointer1
STO	'p1ref		
CPY	0		;p1ref
STO	'cmp1		
LOAD	0		;pointer2
STO	'p2ref		
CPY	0		;p2ref
STO	'cmp2		
LOAD	'incPointers		
STO	'cmpOther		
STO	'equal1		
LOAD	'swap		
STO	'cmp1Greater		
LOAD	0	:cmpStart	;cmp2
SUB	1		
STO	'cmp2		
JGZ	'cmp1dec		
CPY	'cmp1		
SUB	1		
JGZ	0		;cmp1Greater
LOAD	1		
JGZ	0		;equal1
LOAD	0	:cmp1dec	;cmp1
SUB	1		
STO	'cmp1		
JGZ	'cmpStart		
LOAD	1		
JGZ	0		;cmpOther
CPY	'pointer1	:incPointers	
ADD	1		
STO	'pointer1		
CPY	'pointer2		
STO	'p2sub		
LOAD	'listend		
SUB	0		;p2sub
JGZ	'returnToInc		
LOAD	0		;flag
JGZ	'start		
HALT			
CPY	'pointer2	:returnToInc	
ADD	1		
STO	'pointer2		
JGZ	'cmpPointers		
CPY	'pointer2	:swap	
STO	'p2SwpRef		
STO	'p2WriteRef		
CPY	0		;p2SwpRef

Instr	Data	I Label	D Label
STO	'swp		
CPY	'pointer1		
STO	'p1SwpRef		
STO	'p1WriteRef		
CPY	0		;p1SwpRef
STO	0		;p2WriteRef
LOAD	0		;swp
STO	0		;p1WriteRef
LOAD	1		
STO	'flag		
JGZ	'incPointers		
		:listStart	
		:listend	

```
3,128,4,16,1,1,4,24,3,0,4,88,3,0,4,20,7,0,4,60,3,0,4,28,
7,0,4,42,3,71,4,70,4,58,3,100,4,54,3,0,2,1,4,42,6,59,7,
60,2,1,6,0,3,1,6,0,3,0,2,1,4,60,6,41,3,1,6,0,7,16,1,1,4,
16,7,24,4,84,3,129,2,0,6,92,3,0,6,3,0,7,24,1,1,4,24,6,15,
7,24,4,107,4,119,7,0,4,121,7,16,4,117,4,123,7,0,4,0,3,0,
4,0,3,1,4,88,6,71,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.2.12 Universal TM

Instr	Data	I Label	D Label
CPY	'C_STATE	:PStart	
STO	'SE_ST		
CPY	'CHP		
STO	'SY_R		
CPY	5		;SY_R
STO	'SE_SY		
LOAD	'M_SE_RET		
STO	'SE_R_LOC		
JGZ	'SE_ST		
LOAD	0	:M_SE_RET	;SRL
JGZ	'V_SE		
HALT			
ADD	2	:V_SE	
STO	'N_STR		
CPY	4		;N_STR
STO	'C_STATE		
CPY	'SRL		
ADD	3		
STO	'N_SYR		

Instr	Data	I Label	D Label
CPY	'CHP		
STO	'HP		
CPY	5		;N_SYR
STO	4		;HP
CPY	'SRL		
ADD	4		
STO	'N_DIRR		
CPY	1	;N_DIRR	
SUB	1		
JGZ	'DIR_RIGHT		
CPY	'CHP		
SUB	1		
STO	'CHP		
JGZ	'CONTINUE		
CPY	'CHP	:DIR_RIGHT	
ADD	1		
STO	'CHP		
CPY	'C_STATE	:CONTINUE	
JGZ	'PStart		
HALT			
LOAD	'SY_TABLE	:SE_ST	
STO	'currentLoc		
LOAD	0	:search_loop	;SE_ST
STO	'CMPState		
LOAD	0		;currentLoc
STO	'tabcomp1		
CPY	5		;tabcomp1
SUB	0		;CMPState
JGZ	'nTupState		
CPY	'currentLoc		
ADD	1		
STO	'currentLoc		
STO	'tabcomp2		
CPY	5		;tabcomp2
STO	'CMPSymbol		
LOAD	0		;SE_SY
SUB	0		;CMPSymbol
JGZ	'nTupSym		
LOAD	1		
JGZ	'found		
CPY	'currentLoc	:nTupState	
ADD	1		
STO	'currentLoc		
CPY	'currentLoc	:nTupSym	
ADD	4		
JGZ	'nextTuple		
CPY	'currentLoc	:found	

Instr	Data	I Label	D Label
SUB	1		
STO	'SRL		
JGZ	'searchExit		
STO	'currentLoc	:nextTuple	
JGZ	'search_loop		
STO	'SRL	:Not_Found	
LOAD	1	:searchExit	
JGZ	0		;SE_R_LOC
'TAPE_START		:CHP	
1		:C_STATE	
		:SY_TABLE	
		:TAPE_START	

```
7,150,4,84,7,149,4,12,7,5,4,110,3,21,4,148,6,79,3,0,6,
26,0,1,2,4,31,7,4,4,150,7,22,1,3,4,45,7,149,4,47,7,5,4,
4,7,22,1,4,4,55,7,1,2,1,6,68,7,149,2,1,4,149,6,74,7,149,
1,1,4,149,7,150,6,3,0,3,147,4,88,3,0,4,94,3,0,4,92,7,5,
2,0,6,119,7,88,1,1,4,88,4,106,7,5,4,112,3,0,2,0,6,125,3
,1,6,131,7,88,1,1,4,88,7,88,1,4,6,139,7,88,2,1,4,22,6,
145,4,88,6,83,4,22,3,1,6,5,146,1,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0
```

B.2.13 Universal RASP

Instr	Data	I Label	D Label
LOAD	'PC_P		
ADD	3		
STO	'OFF_PC		
CPY	'OFF_PC	:SIM_ST	
STO	'INSLOC		
CPY	4		;INSLOC
STO	'IR_P		
STO	'Decoder_Ins		
JGZ	'dec1		
HALT		:none	
SUB	1	:dec1	
JGZ	'dec2		
CPY	'ACC_P		
ADD	1		
STO	'x		
STO	'ACC_P		
CPY	'MAX_INT		
STO	'y		
LOAD	'ACC_P		

Instr	Data	I Label	D Label
STO	'sto_location		
LOAD	'done		
STO	'return_location		
JGZ	'TEST_LOOP		
SUB	1	:dec2	
JGZ	'dec3		
CPY	'ACC_P		
JGZ	'dc		
CPY	'MAX_INT		
STO	'ACC_P		
JGZ	'decST		
SUB	1	:dc	
STO	'ACC_P	:decST	
LOAD	1		
JGZ	'done		
SUB	1	:dec3	
JGZ	'dec4		
LOAD	'L_RET		
STO	'FE_RET		
JGZ	'FETCH		
CPY	'IR_P	:L_RET	
STO	'ACC_P		
LOAD	1		
JGZ	'done		
SUB	1	:dec4	
JGZ	'dec5		
LOAD	'S_RET		
STO	'FE_RET		
JGZ	'FETCH		
CPY	'IR_P	:S_RET	
STO	'stoadd		
LOAD	'PC_P		
ADD	0		;stoadd
STO	'sloc		
CPY	'ACC_P		
STO	0		;sloc
LOAD	1		
JGZ	'done		
SUB	1	:dec5	
JGZ	'dec6		
OUT			
JGZ	'done		
SUB	1	:dec6	
JGZ	'dec7		
LOAD	'J_RET		
STO	'FE_RET		
JGZ	'FETCH		

Instr	Data	I Label	D Label
CPY	'ACC_P	:J_RET	
JGZ	'JGZ_JUMP		
LOAD	1		
JGZ	'done		
CPY	'IR_P	:JGZ_JUMP	
STO	'PC_P		
STO	'jgzadd		
LOAD	'PC_P		
ADD	0		;jgzadd
STO	'OFF_PC		
JGZ	'SIM_ST		
SUB	1	:dec7	
JGZ	'none		
LOAD	'C_RET		
STO	'FE_RET		
JGZ	'FETCH		
CPY	'IR_P	:C_RET	
STO	'cpyadd		
LOAD	'PC_P		
ADD	0		;cpyadd
STO	'cpyloc		
CPY	0		;cpyloc
STO	'ACC_P		
LOAD	1		
JGZ	'done		
LOAD	'SIM_ST	:done	
STO	'I_FRET		
CPY	'PC_P	:INCREMENT_PC	
ADD	1		
STO	'PC_P		
STO	'x		
CPY	'MAX_INT		
STO	'y		
LOAD	'PC_P		
STO	'sto_location		
LOAD	'TI_RET		
STO	'return_location		
LOAD	1		
JGZ	'TEST_LOOP		
CPY	'PC_P	:TI_RET	
JGZ	'I_OFF		
LOAD	'PC_P		
STO	'OFF_PC		
LOAD	1		
JGZ	'INC_EXIT		
CPY	'OFF_PC	:I_OFF	
ADD	1		

Instr	Data	I Label	D Label
STO	'OFF_PC		
LOAD	1	:INC_EXIT	
JGZ	0		;I_FRET
LOAD	'fetch_r	:FETCH	
STO	'I_FRET		
JGZ	'INCREMENT_PC		
CPY	'OFF_PC	:fetch_r	
STO	'FETCH_VAR		
CPY	0		;FETCH_VAR
STO	'IR_P		
LOAD	1		
JGZ	0		;FE_RET
LOAD	0	:TEST_LOOP	;x
SUB	1		
STO	'x		
LOAD	0		;y
SUB	1		
STO	'y		
JGZ	'xtest		
LOAD	1		
JGZ	'xtest2		
CPY	'x	:xtest	
JGZ	'TEST_LOOP		
LOAD	1		
JGZ	'RETURN		
CPY	'x	:xtest2	
JGZ	'INVALID		
LOAD	1		
JGZ	'RETURN		
LOAD	0	:INVALID	
STO	5		;sto_location
LOAD	1	:RETURN	
JGZ	0		;return_location
0		:Decoder_Ins	
0		:OFF_PC	
15		:MAX_INT	
		:PC_P	
		:IR_P	
		:ACC_P	

```
3,286,1,3,4,293,7,293,4,14,7,4,4,286,4,292,6,22,0,2,1,6,48,7,
286,1,1,4,251,4,286,7,294,4,257,3,286,4,287,3,183,4,291,6,250,
2,1,6,70,7,286,6,62,7,294,4,286,6,64,2,1,4,286,3,1,6,183,2,1,
6,88,3,80,4,249,6,232,7,286,4,286,3,1,6,183,2,1,6,116,3,98,4,
249,6,232,7,286,4,105,3,286,1,0,4,111,7,286,4,0,3,1,6,183,2,1,
6,123,5,6,183,2,1,6,155,3,133,4,249,6,232,7,286,6,141,3,1,6,
183,7,286,4,286,4,150,3,286,1,0,4,293,6,9,2,1,6,21,3,165,4,249,
```

```
6,232,7,286,4,172,3,286,1,0,4,176,7,0,4,286,3,1,6,183,3,9,4,231,
7,286,1,1,4,286,4,251,7,294,4,257,3,286,4,287,3,211,4,291,3,1,6,
250,7,286,6,223,3,286,4,293,3,1,6,228,7,293,1,4,293,3,1,6,0,3,238,
4,231,6,187,7,293,4,243,7,0,4,286,3,1,6,0,3,0,2,1,4,251,3,0,2,1,4,
257,6,268,3,1,6,276,7,251,6,250,3,1,6,288,7,251,6,284,3,1,6,288,3,
0,4,5,3,1,6,0,0,0,15,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.3 RASP3

The RASP3 programs are presented in two ways: the "programming language form" as seen all throughout this thesis, and the "array form" which is what is actually measured.

B.3.1 Addition

Instr	Data	I Label
LOAD	x	
ADD	'label	
y		:label

```
3,5,1,8,0,8,0,0,0,0,0,0,0,0
```

B.3.2 Subtraction

Instr	Data	I Label	D Label
LOAD	4	:sub_start	;sub_2
JGZ	'subbing		
HALT			
SUB	'subTarget	:subbing	
STO	'sub_2		
LOAD	7		;sub_1
JGZ	'subbing2		
HALT			
SUB	'subTarget	:subbing2	
STO	'sub_1		
LOAD	1		;subTarget
JGZ	'sub_start		

```
3,4,6,8,0,2,22,4,4,3,7,6,17,0,2,22,4,13,3,1,6,3,0,0,0,0,0,0,0
```

B.3.3 Equality

Instr	Data	I Label	D Label
LOAD	6		;num1
SUB	'num2		
JGZ	'out		
HALT			
LOAD	1	:out	
HALT			
5	:num2		

```
3,6,2,13,6,10,0,3,1,0,6,0,0
```

B.3.4 Multiplication

Instr	Data	I Label	D Label
LOAD	5		;multiplier
JGZ	'return		
HALT			
LOAD	5	:return	;multiplicand
JGZ	'start		
HALT			
SUB	'one	:start	
STO	'multiplicand		
LOAD	0		;runningTotal
ADD	'multiplier		
STO	'runningTotal		
LOAD	1		;one
JGZ	'return		

```
3,5,6,8,0,3,5,6,13,0,2,24,4,9,3,0,1,4,4,18,3,1,6,8,0,0,0,0,0
```

B.3.5 Division

Instr	Data	I Label	D Label
LOAD	0	:start	;divisor
JGZ	'div_start		
HALT			
STO	'tmp	:div_start	
LOAD	7		;num
STO	'remainder		
LOAD	0	:loop	;tmp
JGZ	'sub		
LOAD	1		
JGZ	'return		
SUB	'one	:sub	
STO	'tmp		
CPY	'num		
JGZ	'nl		
HALT			
SUB	'one	:nl	
STO	'num		
LOAD	1		;one
JGZ	'loop		
LOAD	0	:return	;quotient
ADD	'one		
STO	'quotient		
JGZ	'start		
0		:remainder	

```
7,9,6,8,0,3,3,4,17,3,7,4,49,3,0,6,24,3,1,6,41,2,38,4,17,7,13,
6,33,0,2,38,4,13,3,1,6,16,3,0,1,38,4,42,6,8,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0
```

B.3.6 Exponentiation

Instr	Data	I Label	D Label
LOAD	1	:start	;power
JGZ	'continue		
HALT			
SUB	'one	:continue	
STO	'power		
LOAD	1		;runningTotal
STO	'multiplicand		
LOAD	0		
STO	'runningTotal		
LOAD	0	:return	;multiplicand
JGZ	'mulStart		
LOAD	1		
JGZ	'start		
SUB	'one	:mulStart	
STO	'multiplicand		
CPY	'runningTotal		
ADD	'multiplier		
STO	'runningTotal		
LOAD	1		;one
JGZ	'return		
1	:multiplier		

```
3,1,6,8,0,2,39,4,4,3,1,4,21,3,0,4,13,3,0,6,28,3,1,6,3,2,39,4
,21,7,13,1,42,4,13,3,1,6,20,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0
```

B.3.7 List Membership

Instr	Data	I Label	D Label
LOAD	'ls	:start	
STO	'pointer	:cmp_pointer_target	
STO	'cmp_1		
LOAD	2		;target
SUB	'pointer		;cmp_1
JGZ	'end_test		
LOAD	1		;one
HALT			
LOAD	0	:end_test	;pointer
SUB	'listend		;inc_sub
JGZ	'inc_pointer		
LOAD	0		
HALT			
CPY	'pointer	:inc_pointer	
ADD	'one		
JGZ	'cmp_pointer_target		
'le		:listend	
		:ls	
		:le	

```
3,29,4,19,4,12,3,2,2,19,6,18,3,1,0,3,0,2,33,6,27,3,0,0,7,19,
1,16,6,5,31,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0
```

B.3.8 Linear Search

Instr	Data	I Label	D Label
CPY	'listStart	:start	
STO	'pointer	:cmp_pointer_target	
STO	'cmp_1		
LOAD	2		;target
SUB	'pointer		;cmp_1
JGZ	'end_test		
CPY	'pointer		
SUB	'listStart		
HALT			
LOAD	0	:end_test	;pointer
SUB	'listend		;inc_sub
JGZ	'inc_pointer		
LOAD	'listend		
HALT			
CPY	'pointer	:inc_pointer	
ADD	'one		
JGZ	'cmp_pointer_target		
1		:one	
'ls		:listStart	
'le		:listend	
		:ls	
		:le	

```
7,36,4,21,4,12,3,2,2,21,6,20,7,21,2,36,0,3,0,2,37,6,29,3,37,0,
7,21,1,35,6,5,1,33,34,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0
```

B.3.9 List Reversal

Instr	Data	I Label	D Label
LOAD	'listEnd		
STO	'cpyPointer		
ADD	'two	:one	
STO	'writePointer		
LOAD	0	:main	;writePointer
STO	'writeSTO		
LOAD	0		;cpyPointer
STO	'cpyLOC		
CPY	0		;cpyLOC
STO	0		;writeSTO
CPY	'writePointer		
ADD	'one		
STO	'writePointer		
LOAD	'listStart		
SUB	'cpyPointer	:two	
JGZ	'decWritePointer		
HALT			
CPY	'cpyPointer	:decWritePointer	
SUB	'one		
STO	'cpyPointer		
JGZ	'main	:listStart	;listEnd

```
3,43,4,16,1,45,4,12,3,0,4,22,3,0,4,20,7,0,4,0,7,12,1,44,4,12,3,
42,2,16,6,36,0,7,16,2,44,4,16,6,11,1,2,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0
```

B.3.10 Stateful List Reversal

Instr	Data	I Label	D Label
LOAD	'listStart		
STO	'pointer1		
LOAD	'listEnd		
STO	'pointer2		
LOAD	'listEnd		
SUB	'ls		
ADD	'one		
STO	'listsize		
LOAD	0	:main	;pointer1
SUB	'pointer2		
JGZ	'continue		
HALT			
LOAD	0	:continue	;listsize
SUB	'swaps		
JGZ	'swap		

Instr	Data	I Label	D Label
HALT			
CPY	'pointer1	:swap	
STO	'swpref1		
STO	'writeref1		
CPY	0		;swpref1
STO	'swp		
LOAD	0		;pointer2
STO	'swpref2		
STO	'writeref2		
CPY	0		;swpref2
STO	0		;writeref1
LOAD	0		;swp
STO	0		;writeref2
CPY	'pointer1		
ADD	'one	:one	
STO	'pointer1		
CPY	'pointer2		
SUB	'one	:two	
STO	'pointer2		
LOAD	0		;swaps
ADD	'two		
STO	'swaps		
JGZ	'main		
'listStart		:ls :listStart :listEnd	

```
3,75,4,20,3,76,4,44,3,76,2,79,1,77,4,27,3,0,2,44,6,26,0,3,0,2,70,
6,33,0,7,20,4,40,4,52,7,0,4,54,3,0,4,50,4,56,7,0,4,0,3,0,4,0,7,20,
1,77,4,20,7,44,2,77,4,44,3,0,1,78,4,70,6,19,1,2,75,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0
```

B.3.11 Bubble Sort

Instr	Data	I Label	D Label
LOAD	'listStart	:start	
STO	'pointer1		
ADD	'one		
STO	'pointer2		
LOAD	0		
STO	'flag		
LOAD	0	:cmp_pointers	;pointer1
STO	'p1ref		
CPY	0		;p1ref
STO	'cmp1		
LOAD	0		;pointer2

Instr	Data	I Label	D Label
STO	'p2ref		
CPY	0		;p2ref
STO	'cmp2		
LOAD	'inc_pointers		
STO	'cmpOther		
STO	'equal1		
LOAD	'swap		
STO	'cmp1Greater		
LOAD	0	:cmp_start	;cmp2
SUB	'one		
STO	'cmp2		
JGZ	'cmp1dec		
CPY	'cmp1		
SUB	'one		
JGZ	0		;cmp1Greater
LOAD	1		;one
JGZ	0		;equal1
LOAD	0	:cmp1dec	;cmp1
SUB	'one		
STO	'cmp1		
JGZ	'cmp_start		
LOAD	1		
JGZ	0		;cmpOther
CPY	'pointer1	:inc_pointers	
ADD	'one		
STO	'pointer1		
LOAD	'listend		
SUB	'pointer2		
JGZ	'return_to_inc		
LOAD	0		;flag
JGZ	'start		
HALT			
CPY	'pointer2	:return_to_inc	
ADD	'one		
STO	'pointer2		
JGZ	'cmp_pointers		
CPY	'pointer2	:swap	
STO	'p2SwpRef		
STO	'p2WriteRef		
CPY	0		;p2SwpRef
STO	'swp		
CPY	'pointer1		
STO	'p1SwpRef		
STO	'p1WriteRef		
CPY	0		;p1SwpRef
STO	0		;p2WriteRef
LOAD	0		;swp

Instr	Data	I Label	D Label
STO	0		;p1WriteRef
LOAD	1		
STO	'flag		
JGZ	'inc_pointers		
		:listStart	
		:listend	

```
3,124,4,16,1,56,4,24,3,0,4,84,3,0,4,20,7,0,4,60,3,0,4,28,7,0,4,
42,3,71,4,70,4,58,3,96,4,54,3,0,2,56,4,42,6,59,7,60,2,56,6,0,3,
1,6,0,3,0,2,56,4,60,6,41,3,1,6,0,7,16,1,56,4,16,3,125,2,24,6,88,
3,0,6,3,0,7,24,1,56,4,24,6,15,7,24,4,103,4,115,7,0,4,117,7,16,4,
113,4,119,7,0,4,0,3,0,4,0,3,1,4,84,6,71,0,0
```

B.3.12 Universal TM

Instr	Data	I Label	D Label
CPY	'CS	:P_START	
STO	'SS	:four	
CPY	'CHP		
STO	'SY_READ		
CPY	5		;SY_READ
STO	'S_SY		
LOAD	'MSR	:three	
STO	'STL		
JGZ	'SE_ST	:six	
LOAD	0	:MSR	;SRL
JGZ	'VS		
HALT			
ADD	'two	:VS	
STO	'N_ST_R		
CPY	4		;N_ST_R
STO	'CS		
CPY	'SRL		
ADD	'three	:one	
STO	'NEW_SY_READ		
CPY	'CHP		
STO	'HP		
CPY	5		;NEW_SY_READ
STO	4		;HP
CPY	'SRL		
ADD	'four		
STO	'N_D_R		
CPY	1		;N_D_R
SUB	'one	:two	
JGZ	'DIR_RIGHT		
CPY	'CHP		
SUB	'one		

Instr	Data	I Label	D Label
STO	'CHP		
JGZ	'CONTINUE		
CPY	'CHP	:DIR_RIGHT	
ADD	'one		
STO	'CHP		
CPY	'CS	:CONTINUE	
JGZ	'P_START		
HALT			
LOAD	'SYTABST	:SE_ST	
STO	'currentLoc		
LOAD	0	:search_loop	;SS
SUB	0		;currentLoc
JGZ	'nTupSt		
CPY	'currentLoc		
ADD	'one		
STO	'currentLoc		
STO	'CMPSymbol		
LOAD	0		;S_SY
SUB	0		;CMPSymbol
JGZ	'nTupSy		
LOAD	1		
JGZ	'found		
CPY	'currentLoc	:nTupSt	
ADD	'one		
STO	'currentLoc		
CPY	'currentLoc	:nTupSy	
ADD	'four		
JGZ	'nextTuple		
CPY	'currentLoc	:found	
SUB	'one		
STO	'SRL		
JGZ	'searchExit		
STO	'currentLoc	:nextTuple	
JGZ	'search_loop		
STO	'SRL	:Not_Found	
LOAD	1	:searchExit	
JGZ	5		;STL
'T_ST		:CHP	
1		:CS	
		:SYTABST	
		:T_ST	

```
7,138,4,84,7,137,4,12,7,5,4,98,3,21,4,136,6,79,3,0,6,26,0,1,56,
4,31,7,4,4,138,7,22,1,15,4,45,7,137,4,47,7,5,4,4,7,22,1,5,4,55,
7,1,2,26,6,68,7,137,2,26,4,137,6,74,7,137,1,26,4,137,7,138,6,3,
0,3,138,4,86,3,0,2,0,6,107,7,86,1,26,4,86,4,100,3,0,2,0,6,113,3,
1,6,119,7,86,1,26,4,86,7,86,1,5,6,127,7,86,2,26,4,22,6,133,4,86,
```

```
6,83,4,22,3,1,6,5,138,1,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.3.13 Universal RASP

Instr	Data	I Label	D Label
LOAD	'PC_P	:three	
ADD	'three		
STO	'OFF_PC		
CPY	'OFF_PC	:SIM_ST	
STO	'INSLOC		
CPY	4		;INSLOC
STO	'IR_P		
STO	'Decoder_Ins		
JGZ	'dec1		
HALT		:none	
SUB	'one	:dec1	
JGZ	'dec2		
CPY	'ACC_P		
ADD	'one		
STO	'x		
STO	'ACC_P		
CPY	'MAX_INT		
STO	'y		
LOAD	'ACC_P		
STO	'sto_location		
LOAD	'done		
STO	'return_location		
JGZ	'TEST_LOOP		
SUB	'one	:dec2	
JGZ	'dec3		
CPY	'ACC_P		
JGZ	'dc		
CPY	'MAX_INT		
STO	'ACC_P		
JGZ	'decST		
SUB	'one	:dc	
STO	'ACC_P	:decST	
LOAD	1		
JGZ	'done		
SUB	'one	:dec3	
JGZ	'dec4		
LOAD	'L_RET		
STO	'F_RET		
JGZ	'FETCH		
CPY	'IR_P	:L_RET	

Instr	**Data**	**I Label**	**D Label**
STO	'ACC_P		
LOAD	1		
JGZ	'done		
SUB	'one	:dec4	
JGZ	'dec5		
LOAD	'S_RET		
STO	'F_RET		
JGZ	'FETCH		
LOAD	'PC_P	:S_RET	
ADD	'IR_P		
STO	'sloc		
CPY	'ACC_P		
STO	0		;sloc
LOAD	1		
JGZ	'done		
SUB	'one	:dec5	
JGZ	'dec6		
OUT			
JGZ	'done		
SUB	'one	:dec6	
JGZ	'dec7		
LOAD	'J_RET		
STO	'F_RET		
JGZ	'FETCH		
CPY	'ACC_P	:J_RET	
JGZ	'JGZ_JUMP		
LOAD	1		
JGZ	'done		
CPY	'IR_P	:JGZ_JUMP	
STO	'PC_P		
LOAD	'PC_P		
ADD	'IR_P		
STO	'OFF_PC		
JGZ	'SIM_ST		
SUB	'one	:dec7	
JGZ	'none		
LOAD	'C_RET		
STO	'F_RET		
JGZ	'FETCH		
LOAD	'PC_P	:C_RET	
CPY	'IR_P		
STO	'cpyloc		
CPY	0		;cpyloc
STO	'ACC_P		
LOAD	1		
JGZ	'done		
LOAD	'SIM_ST	:done	

Instr	Data	I Label	D Label
STO	'I_FRET		
CPY	'PC_P	:INCREMENT_PC	
ADD	'one		
STO	'PC_P		
STO	'x		
CPY	'MAX_INT		
STO	'y		
LOAD	'PC_P		
STO	'sto_location		
LOAD	'TI_RET		
STO	'return_location		
LOAD	1		
JGZ	'TEST_LOOP		
CPY	'PC_P	:TI_RET	
JGZ	'INC_OFFSET		
LOAD	'PC_P		
STO	'OFF_PC		
LOAD	1		
JGZ	'INC_EXIT		
CPY	'OFF_PC	:INC_OFFSET	
ADD	'one		
STO	'OFF_PC		
LOAD	1	:INC_EXIT	
JGZ	0		;I_FRET
LOAD	'fetch_r	:FETCH	
STO	'I_FRET		
JGZ	'INCREMENT_PC		
CPY	'OFF_PC	:fetch_r	
STO	'FETCH_VAR		
CPY	0		;FETCH_VAR
STO	'IR_P		
LOAD	1		
JGZ	0		;F_RET
LOAD	0	:TEST_LOOP	;x
SUB	'one		
STO	'x		
LOAD	0		;y
SUB	'one		
STO	'y		
JGZ	'xtest		
LOAD	1		
JGZ	'xtest2		
CPY	'x	:xtest	
JGZ	'TEST_LOOP		
LOAD	1		
JGZ	'RETURN		
CPY	'x	:xtest2	

Instr	Data	I Label	D Label
JGZ	'INVALID		
LOAD	1		
JGZ	'RETURN		
LOAD	0	:INVALID	
STO	5		;sto_location
LOAD	1	:RETURN	;one
JGZ	0		;return_location
0		:Decoder_Ins	
0		:OFF_PC	
4		:MAX_INT	
		:PC_P	
		:IR_P	
		:ACC_P	

```
3,281,1,3,4,284,7,284,4,14,7,4,4,281,4,283,6,22,0,2,280,6,48,
7,281,1,280,4,242,4,281,7,285,4,248,3,281,4,278,3,173,4,282,
6,241,2,280,6,70,7,281,6,62,7,285,4,281,6,64,2,280,4,281,3,1,
6,173,2,280,6,88,3,80,4,240,6,223,7,281,4,281,3,1,6,173,2,280,
6,112,3,98,4,240,6,223,3,281,1,281,4,107,7,281,4,0,3,1,6,173,2,
280,6,119,5,6,173,2,280,6,149,3,129,4,240,6,223,7,281,6,137,3,
1,6,173,7,281,4,281,3,281,1,281,4,284,6,9,2,280,6,21,3,159,4,
240,6,223,3,281,7,281,4,166,7,0,4,281,3,1,6,173,3,9,4,222,7,281,
1,280,4,281,4,242,7,285,4,248,3,281,4,278,3,201,4,282,3,1,6,241,
7,281,6,213,3,281,4,284,3,1,6,219,7,284,1,280,4,284,3,1,6,0,3,
229,4,222,6,177,7,284,4,234,7,0,4,281,3,1,6,0,3,0,2,280,4,242,
3,0,2,280,4,248,6,259,3,1,6,267,7,242,6,241,3,1,6,279,7,242,6,
275,3,1,6,279,3,0,4,5,3,1,6,0,0,0,4,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,
0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0
```

B.4 TM

B.4.1 Addition/Subtraction/Equality

```
1,1,2,0,R
2,1,2,1,R
2,0,0,1,L
```

(a) Addition

```
1,1,1,1,R
1,0,2,0,R
2,1,2,1,R
2,0,3,0,L
3,1,4,0,L
3,0,0,0,R
4,1,4,1,L
4,0,5,0,L
5,1,5,1,L
5,0,6,0,R
6,1,1,0,R
6,0,7,0,R
7,0,8,0,R
7,1,7,0,R
8,0,0,0,R
```

(b) Subtraction

```
1,0,7,0,R
2,1,2,1,R
2,0,3,0,R
3,1,3,1,R
3,0,4,0,L
4,1,5,0,L
4,0,9,0,L
5,1,5,1,L
5,0,6,0,L
6,1,6,1,L
6,0,1,0,R
7,0,0,0,R
7,1,8,0,R
8,1,8,0,R
8,0,0,1,R
9,0,0,0,L
9,1,10,0,L
10,1,10,0,L
10,0,0,1,L
```

(c) Equality

B.4.2 Multiplication/Division

```
1,0,10,0,R
1,1,2,0,R
2,1,2,1,R
2,0,3,0,R
3,#,3,#,R
3,1,4,#,R
3,0,8,0,L
4,1,4,1,R
4,0,5,0,R
5,1,5,1,R
5,0,6,1,L
6,1,6,1,L
6,0,7,0,L
7,#,7,#,L
7,1,7,1,L
7,0,3,0,R
8,#,8,1,L
8,0,9,0,L
9,1,9,1,L
9,0,1,0,R
10,1,10,0,R
10,0,0,0,R
```

(a) Multiplication

```
1,1,2,#,R
1,#,1,#,R
1,0,6,0,L
2,1,2,1,R
2,0,3,0,R
3,#,3,#,R
3,1,4,#,L
3,0,9,0,L
4,#,4,#,L
4,0,5,0,L
5,#,5,#,L
5,1,5,1,L
5,0,1,0,R
6,#,6,1,L
6,0,7,0,L
7,1,7,1,L
7,0,8,1,R
8,1,8,1,R
8,0,1,0,R
9,#,9,0,L
9,0,10,0,L
10,#,12,0,L
10,1,11,0,L
11,1,11,0,L
11,#,12,0,L
12,#,12,1,L
12,0,0,0,L
```

(b) Division

B.4.3 Exponentiation

```
1,0,0,0,R
1,1,2,0,R
2,1,2,1,R
2,0,3,0,R
3,#,3,#,R
3,0,14,0,R
3,1,4,#,R
4,1,4,1,R
4,0,5,0,R
5,0,5,0,R
5,1,6,#,R
6,1,6,1,R
6,0,7,0,R
7,1,7,1,R
7,0,8,1,L
8,1,8,1,L
8,0,9,0,L
9,#,9,#,L
9,1,9,1,L
9,0,10,0,R
10,1,6,#,R
10,#,10,#,R
10,0,11,0,L
11,#,11,1,L
11,0,12,0,L
12,0,12,0,L
12,#,13,#,L
12,1,13,1,L
13,1,13,1,L
13,#,13,#,L
13,0,3,0,R
14,0,14,0,R
14,1,15,0,R
15,1,15,0,R
15,0,16,0,L
16,0,16,0,L
16,#,17,1,L
17,#,17,1,L
17,0,18,0,L
18,1,18,1,L
18,0,1,0,R
```

B.4.4 List Membership

```
1,1,1,B,R
1,0,1,A,R
1,*,2,*,L
2,1,2,B,L
2,0,2,A,L
2,*,2,*,L
2,A,2,A,L
2,B,2,B,L
2,T,3,T,R
3,0,3,0,R
3,1,3,1,R
3,B,4,1,R
3,A,7,0,R
3,*,8,*,L
4,A,4,A,R
4,B,4,B,R
4,*,4,*,R
4,1,5,B,L
4,0,6,A,R
5,A,5,A,L
5,B,5,B,L
5,1,5,1,L
5,0,5,0,L
5,*,5,*,L
5,T,3,T,R
6,0,6,0,R
6,1,6,1,R
6,*,2,*,L
6,E,0,E,R
7,A,7,A,R
7,B,7,B,R
7,*,7,*,R
7,0,5,A,L
7,1,6,B,R
8,0,8,0,L
8,1,8,1,L
8,T,9,T,L
9,0,0,1,L
```

B.4.5 Linear Search

```
1,1,1,B,R
1,0,1,A,R
1,#,2,#,L
2,A,2,A,L
2,B,2,B,L
2,0,2,A,L
2,1,2,B,L
2,#,2,#,L
2,*,2,*,L
2,T,3,T,R
3,1,3,1,R
3,0,3,0,R
3,B,4,1,R
3,A,9,0,R
3,#,10,#,L
4,A,4,A,R
4,B,4,B,R
4,#,4,#,R
4,*,4,*,R
4,1,5,B,L
4,0,6,A,R
5,A,5,A,L
5,B,5,B,L
5,1,5,1,L
5,0,5,0,L
5,#,5,#,L
5,*,5,*,L
5,T,3,T,R
6,0,6,0,R
6,1,6,1,R
6,*,6,*,R
6,#,2,#,L
6,E,7,E,L
7,0,7,0,L
7,1,7,1,L
7,A,7,A,L
7,B,7,B,L
7,*,7,*,L
7,#,7,#,L
7,T,8,T,L
8,0,8,*,L
8,E,0,E,R
9,A,9,A,R
9,B,9,B,R
9,#,9,#,R
9,*,9,*,R
9,1,6,B,R
9,0,5,A,L
10,0,10,A,L
10,1,10,B,L
10,T,10,T,L
10,E,11,E,R
11,A,12,0,R
11,T,0,T,R
12,A,12,A,R
12,B,12,B,R
12,*,12,*,R
12,#,12,#,R
12,T,12,T,R
12,0,13,A,L
12,1,14,B,L
13,A,13,A,L
13,B,13,B,L
13,*,13,*,L
13,#,13,#,L
13,T,13,T,L
13,0,11,0,R
14,A,14,A,L
14,B,14,B,L
14,*,14,*,L
14,#,14,#,L
14,T,14,T,L
14,0,11,1,R
```

B.4.6 List Reversal

1,#,1,#,L
1,A,1,A,L
1,B,1,B,L
1,$,1,$,L
1,0,9,0,L
1,1,9,1,L
1,*,2,*,R
1,E,0,0,R
2,A,2,A,R
2,B,2,B,R
2,1,7,B,R
2,0,5,A,R
2,#,3,#,R
2,$,3,$,R
3,A,3,A,R
3,B,3,B,R
3,#,3,#,R
3,$,3,$,R
3,0,4,$,L
4,A,4,A,L
4,B,4,B,L
4,#,4,#,L
4,$,4,$,L
4,*,1,$,L
5,A,5,A,R
5,B,5,B,R
5,0,5,0,R
5,1,5,1,R
5,$,5,$,R
5,#,6,#,R
6,A,6,A,R
6,B,6,B,R
6,$,6,$,R
6,0,1,A,L
7,A,7,A,R
7,B,7,B,R
7,0,7,0,R
7,1,7,1,R
7,$,7,$,R
7,#,8,#,R
8,A,8,A,R
8,B,8,B,R
8,$,8,$,R
8,0,1,B,L
9,1,9,1,L
9,0,9,0,L
9,A,9,A,L
9,B,9,B,L
9,*,2,*,R
9,E,2,E,R

B.4.7 Stateful List Reversal

```
1,0,2,Z,R
1,1,5,Z,R
1,*,7,*,R
2,0,2,0,R
2,1,2,1,R
2,A,2,A,R
2,B,2,B,R
2,*,2,*,R
2,E,3,E,R
3,A,3,A,R
3,B,3,B,R
3,0,4,A,L
4,A,4,A,L
4,B,4,B,L
4,0,4,0,L
4,1,4,1,L
4,*,4,*,L
4,E,4,E,L
4,Z,1,Z,R
5,0,5,0,R
5,1,5,1,R
5,A,5,A,R
5,B,5,B,R
5,*,5,*,R
5,E,6,E,R
6,A,6,A,R
6,B,6,B,R
6,0,4,B,L
7,A,7,A,R
7,B,7,B,R
7,0,7,0,R
7,1,7,1,R
7,*,7,*,R
7,E,8,E,L
8,A,8,A,L
8,B,8,B,L
8,*,8,*,L
8,1,9,1,R
8,0,9,0,R
8,Z,14,Z,R
9,*,10,*,L
9,E,10,E,L
10,0,11,Z,L
10,1,13,Z,L
10,*,14,*,R
11,A,11,A,L
11,B,11,B,L
11,0,11,0,L
11,1,11,1,L
11,*,11,*,L
11,Z,12,A,R
12,A,12,A,R
12,B,12,B,R
12,0,12,0,R
12,1,12,1,R
12,*,12,*,R
12,Z,10,Z,L
13,A,13,A,L
13,B,13,B,L
13,0,13,0,L
13,1,13,1,L
13,*,13,*,L
13,Z,12,B,R
14,Z,14,Z,R
14,A,14,A,R
14,B,14,B,R
14,*,14,*,R
14,E,14,E,R
14,0,15,0,L
15,A,16,0,L
15,B,17,0,L
15,E,18,E,L
16,E,16,E,L
16,A,16,A,L
16,B,16,B,L
16,*,16,*,L
16,Z,14,A,R
17,E,17,E,L
17,A,17,A,L
17,B,17,B,L
17,*,17,*,L
17,Z,14,B,R
18,A,18,A,L
18,B,18,B,L
18,*,18,*,L
18,0,19,0,L
18,1,19,1,L
18,E,0,E,R
19,0,19,0,L
19,1,19,1,L
19,*,19,*,L
19,A,20,A,R
19,B,20,B,R
20,*,1,*,R
```

B.4.8 Bubble Sort

1,*,2,$,R
1,1,1,1,R
1,0,1,0,R
2,0,2,0,R
2,1,2,1,R
2,*,3,$,L
2,#,13,#,L
3,0,3,0,L
3,1,3,1,L
3,A,3,A,L
3,B,3,B,L
3,$,4,$,R
4,1,5,B,R
4,0,8,A,R
4,A,4,A,R
4,B,4,B,R
4,$,12,$,R
5,0,5,0,R
5,1,5,1,R
5,$,6,$,R
6,1,7,B,L
6,0,15,0,L
6,A,6,A,R
6,B,6,B,R
7,A,7,A,L
7,B,7,B,L
7,$,3,$,L
8,0,8,0,R
8,1,8,1,R
8,$,9,$,R
9,1,10,1,L
9,0,7,A,L
9,A,9,A,R
9,B,9,B,R
10,A,10,0,L
10,B,10,1,L
10,$,11,*,L
11,A,11,0,L
11,0,11,0,L
11,B,11,1,L
11,1,11,1,L
11,$,1,*,R
12,A,12,A,R
12,B,12,B,R
12,*,10,*,L
13,0,13,0,L
13,1,13,1,L
13,*,13,*,L
13,$,13,*,L
13,#,14,#,L
14,0,0,0,R
14,1,14,0,R
14,#,1,#,R
15,A,15,0,L
15,B,15,1,L
15,$,16,$,R
16,Z,16,Z,R
16,0,17,Z,R
16,1,20,Z,R
16,*,22,*,L
16,#,22,#,L
17,0,17,0,R
17,1,17,1,R
17,*,17,*,R
17,#,18,#,R
18,A,18,A,R
18,B,18,B,R
18,0,19,A,L
19,Z,19,Z,L
19,0,19,0,L
19,1,19,1,L
19,A,19,A,L
19,B,19,B,L
19,*,19,*,L
19,#,19,#,L
19,$,16,$,R
20,0,20,0,R
20,1,20,1,R
20,*,20,*,R
20,#,21,#,R
21,A,21,A,R
21,B,21,B,R
21,0,19,B,L
22,Z,22,Z,L
22,$,23,*,L
23,A,23,0,L
23,B,23,1,L
23,0,23,0,L
23,1,23,1,L
23,*,23,*,L
23,Z,23,Z,L
23,$,24,$,R
24,Z,24,Z,R
24,0,25,Z,R
24,1,26,Z,R
24,*,27,*,R
25,0,25,0,R
25,1,25,1,R
25,*,25,*,R
25,Z,23,0,L
26,0,26,0,R
26,1,26,1,R
26,*,26,*,R
26,Z,23,1,L
27,0,27,0,R
27,1,27,1,R
27,*,27,*,R
27,#,28,#,R
28,A,28,A,R
28,B,28,B,R
28,0,29,0,L
29,A,30,0,L
29,B,31,0,L
29,#,32,#,L
30,A,30,A,L
30,B,30,B,L
30,0,30,0,L
30,1,30,1,L
30,*,30,*,L
30,#,30,#,L
30,Z,27,0,R
31,A,31,A,L
31,B,31,B,L
31,0,31,0,L
31,1,31,1,L
31,*,31,*,L
31,#,31,#,L
31,Z,27,1,R
32,0,32,0,L
32,1,32,1,L
32,*,32,*,L
32,$,32,$,L
32,#,33,#,L
33,0,34,1,R
33,1,34,1,R
34,#,34,#,R
34,0,34,0,R
34,1,34,1,R
34,*,34,*,R
34,$,1,*,R

B.4.9 Universal TM

```
1,0,1,A,L
1,1,1,B,L
1,Y,5,Y,R
1,X,1,X,L
1,B,1,B,L
1,A,1,A,L
2,Y,0,Y,R
2,X,1,X,L
2,1,2,1,R
2,0,2,0,R
3,1,2,B,R
3,0,4,A,L
3,B,3,B,R
3,A,3,A,R
3,X,3,X,R
4,Y,5,Y,R
4,X,4,X,L
4,A,4,A,L
4,B,4,B,L
4,1,4,1,L
4,0,4,0,L
5,A,3,0,R
5,B,6,1,R
5,X,23,X,R
5,0,5,0,R
5,1,5,1,R
6,1,4,B,L
6,0,2,A,R
6,A,6,A,R
6,B,6,B,R
6,X,6,X,R
7,S,1,A,L
7,0,7,0,R
7,1,7,1,R
7,Y,7,Y,R
8,S,1,B,L
8,0,8,0,R
8,1,8,1,R
8,Y,8,Y,R
9,0,7,M,R
9,1,8,M,R
10,0,7,M,R
10,1,8,M,R
11,B,9,0,R
11,A,10,0,L
11,0,11,0,L
11,1,11,1,L
11,Y,11,Y,L
12,B,9,1,R
12,A,10,1,L
12,0,12,0,L
12,1,12,1,L
12,Y,12,Y,L
13,0,11,S,L
13,1,12,S,L
13,B,13,1,L
13,A,13,0,L
13,X,13,X,L
13,Y,13,Y,L
14,0,13,0,L
14,1,13,1,L
14,A,14,A,R
14,B,14,B,R
14,X,14,X,R
14,Y,14,Y,R
15,B,15,1,R
15,A,15,0,R
15,X,14,X,R
15,0,15,0,R
15,1,15,1,R
15,Y,15,Y,R
16,M,15,A,R
16,A,16,A,L
16,B,16,B,L
16,Y,16,Y,L
16,0,16,0,L
16,1,16,1,L
17,M,15,B,R
17,A,17,A,L
17,B,17,B,L
17,Y,17,Y,L
17,0,17,0,L
17,1,17,1,L
18,X,17,X,L
18,0,19,B,R
18,1,19,B,R
18,A,18,A,R
18,B,18,B,R
19,X,23,X,R
19,1,19,1,R
19,0,19,0,R
20,0,19,A,R
20,1,19,A,R
20,X,16,X,L
20,A,20,A,R
20,B,20,B,R
21,Y,20,Y,R
21,B,21,B,L
21,A,21,A,L
21,X,21,X,L
21,1,21,1,L
21,0,21,0,L
22,Y,18,Y,R
22,A,22,A,L
22,B,22,B,L
22,X,22,X,L
22,1,22,1,L
22,0,22,0,L
23,1,22,B,L
23,0,21,A,L
23,A,23,A,R
23,B,23,B,R
23,X,23,X,R
```

B.4.10 Universal RASP

1,0,1,A,R
1,1,1,B,R
1,#,53,#,L
1,S,1,S,R
53,A,53,A,L
53,B,53,B,L
53,1,53,1,L
53,0.253,0,L
53,S,53,S,L
53,#,53,#,L
53,P,2,P,R
2,1,2,1,R
2,0,2,0,R
2,#,3,#,L
3,0,3,A,L
3,1,3,B,L
3,A,3,A,L
3,B,3,B,L
3,P,4,P,R
3,*,3,*,L
3,#,3,#,L
3,S,3,S,L
3,I,3,I,L
3,X,3,X,L
4,0,4,0,R
4,1,4,1,R
4,A,5,0,R
4,B,12,1,R
4,#,15,#,R
5,A,5,A,R
5,B,5,B,R
5,#,6,#,R
6,S,6,S,R
6,A,6,A,R
6,B,6,B,R
6,1,6,1,R
6,0,6,0,R
6,#,7,#,R
7,I,7,I,R
7,#,7,#,R
7,*,7,*,R
7,A,7,A,R
7,B,7,B,R
7,X,7,X,R
7,0,9,A,L
7,1,8,B,R
8,I,8,I,R
8,X,8,X,R
8,*,8,*,R
8,1,8,1,R
8,0,8,0,R
8,E,13,E,L
8,#,3,#,L
9,S,9,S,L
9,I,9,I,L
9,X,9,X,L
9,*,9,*,L
9,#,9,#,L
9,1,9,1,L
9,0,9,0,L
9,A,9,A,L
9,B,9,B,L
9,P,4,P,R
10,I,10,I,R
10,#,10,#,R
10,*,10,*,R
10,A,10,A,R
10,B,10,B,R
10,X,10,X,R
10,1,9,B,L
10,0,8,A,R
11,S,11,S,R
11,A,11,A,R
11,B,11,B,R
11,1,11,1,R
11,0,11,0,R
11,#,10,#,R
12,A,12,A,R
12,B,12,B,R
12,#,11,#,R
13,1,13,1,L
13,0,13,0,L
13,A,13,A,L
13,B,13,B,L
13,S,13,S,L
13,I,13,I,L
13,X,13,X,L
13,I,13,I,L
13,#,13,#,L
13,*,13,*,L
13,P,14,P,L
14,#,14,#,L
14,0,23,P,L
15,S,15,S,R
15,B,16,1,R
15,A,16,0,R
15,#,21,#,R
16,#,16,#,R
16,*,16,*,R
16,A,16,A,R
16,B,16,B,R
16,I,16,I,R
16,X,16,X,R
16,0,17,A,L
16,1,19,B,L
17,#,17,#,L
17,*,17,*,L
17,A,17,A,L
17,B,17,B,L
17,X,17,X,L
17,I,17,I,L
17,0,17,0,L
17,1,17,1,L
17,S,18,S,R
18,A,18,A,R
18,B,18,B,R
18,#,21,#,R
18,1,15,A,R
18,0,15,A,R
19,#,19,#,L
19,*,19,*,L
19,A,19,A,L
19,B,19,B,L
19,X,19,X,L
19,I,19,I,L
19,0,19,0,L
19,1,19,1,L
19,S,20,S,R
20,A,20,A,R
20,B,20,B,R
20,#,21,#,R
20,0,15,B,R
20,1,15,B,R
21,0,21,0,L
21,1,21,1,L
21,A,21,A,L

21,B,21,B,L
21,*,21,*,L
21,#,21,#,L
21,X,21,X,L
21,I,21,I,L
21,S,21,S,L
21,P,22,P,L
22,#,22,#,L
22,0,23,B,L
23,E,24,E,R
24,P,24,P,R
24,#,24,#,R
24,*,24,*,R
24,0,24,0,R
24,1,24,1,R
24,A,24,A,R
24,B,24,B,R
24,S,24,S,R
24,I,24,I,R
24,X,24,X,R
24,E,25,E,L
25,P,26,P,L
25,#,25,#,L
25,*,25,*,L
25,0,25,0,L
25,1,25,1,L
25,A,25,0,L
25,B,25,1,L
25,S,25,S,L
25,I,25,I,L
25,X,25,X,L
26,P,26,P,L
26,#,26,#,L
26,*,26,*,L
26,0,26,0,L
26,1,26,1,L
26,A,26,A,L
26,B,26,B,L
26,S,26,S,L
26,I,26,I,L
26,X,26,X,L
26,E,27,E,R
27,1,54,0,R
27,A,46,0,R
27,B,28,0,R
27,P,38,0,R
27,S,86,0,R
27,#,208,0,R
28,P,28,P,R
28,0,28,0,R
28,1,28,1,R
28,#,28,#,R
28,*,28,*,R
28,A,28,A,R
28,B,28,B,R
28,S,29,S,R
29,A,29,A,R
29,B,29,B,R
29,0,30,A,R
29,1,33,B,R
29,#,36,#,R
30,0,30,0,R
30,1,30,1,R
30,A,30,A,R
30,B,30,B,R
30,*,30,*,R
30,#,30,#,R
30,I,31,I,R
31,0,32,A,L
31,1,32,A,L
31,A,31,A,R
31,B,31,B,R
32,0,32,0,L
32,1,32,1,L
32,A,32,A,L
32,B,32,B,L
32,*,32,*,L
32,#,32,#,L
32,I,32,I,L
32,S,29,S,R
33,0,33,0,R
33,1,33,1,R
33,A,33,A,R
33,B,33,B,R
33,*,33,*,R
33,#,33,#,R
33,I,34,I,R
34,0,35,B,L
34,1,35,B,L
34,A,34,A,R
34,B,34,B,R
35,0,35,0,L
35,1,35,1,L
35,A,35,A,L
35,B,35,B,L
35,*,35,*,L
35,#,35,#,L
35,I,35,I,L
35,S,29,S,R
36,0,36,0,L
36,1,36,1,L
36,A,36,A,L
36,B,36,B,L
36,#,36,#,L
36,S,36,S,L
36,P,37,P,L
37,#,37,#,L
37,0,23,A,L
38,P,39,P,R
38,#,38,#,R
39,#,45,#,L
39,A,39,A,R
39,B,39,B,R
39,0,40,A,R
39,1,43,B,R
40,1,40,1,R
40,0,40,0,R
40,#,40,#,R
40,S,41,S,R
41,A,41,A,R
41,B,41,B,R
41,0,42,A,L
41,1,42,A,L
42,0,42,0,L
42,1,42,1,L
42,B,42,B,L
42,A,42,A,L
42,S,42,S,L
42,#,42,#,L
42,P,39,P,R
43,1,43,1,R
43,0,43,0,R
43,#,43,#,R
43,S,44,S,R
44,A,44,A,R
44,B,44,B,R

```
44,0,42,B,L
44,1,42,B,L
45,#,45,#,L
45,A,45,A,L
45,B,45,B,L
45,P,45,P,L
45,0,23,B,L
46,#,46,#,R
46,P,46,P,R
46,0,46,0,R
46,1,46,1,R
46,P,46,P,R
46,S,47,S,R
47,1,47,1,R
47,0,47,0,R
47,#,48,#,L
48,1,51,1,L
48,0,49,0,L
49,0,49,0,L
49,1,49,1,L
49,S,49,S,L
49,#,49,#,L
49,P,50,P,L
50,#,50,#,L
50,0,23,S,L
51,0.251,0,L
51,1,51,1,L
51,S,51,S,L
51,#,51,#,L
51,P,52,P,L
52,#,52,#,L
52,0,23,1,L
54,#,54,#,R
54,P,55,P,R
55,1,55,1,R
55,0.255,0,R
55,#,56,#,L
56,1,56,1,L
56,P,57,P,R
56,0.257,1,R
57,1,57,0,R
57,#,58,#,R
58,0.258,A,L
58,1,58,B,L
58,#,71,#,L
58,S,59,S,R
59,1,59,1,R
59,0.259,0,R
59,#,60,#,L
60,0,60,A,L
60,1,60,B,L
60,A,60,A,L
60,B,60,B,L
60,P,61,P,R
60,*,60,*,L
60,#,60,#,L
60,S,60,S,L
60,I,60,I,L
60,X,60,X,L
61,#,72,#,R
61,0,61,0,R
61,1,61,1,R
61,A,62,0,R
61,B,69,1,R
62,A,62,A,R
62,B,62,B,R
62,#,63,#,R
63,S,63,S,R
63,A,63,A,R
63,B,63,B,R
63,1,63,1,R
63,0,63,0,R
63,#,64,#,R
64,I,64,I,R
64,#,64,#,R
64,*,64,*,R
64,A,64,A,R
64,B,64,B,R
64,X,64,X,R
64,0,66,A,L
64,1,65,B,R
65,I,65,I,R
65,X,65,X,R
65,*,65,*,R
65,1,65,1,R
65,0,65,0,R
65,E,70,E,L
65,#,60,#,L
66,S,66,S,L
66,I,66,I,L
66,X,66,X,L
66,*,66,*,L
66,#,66,#,L
66,1,66,1,L
66,0,66,0,L
66,A,66,A,L
66,B,66,B,L
66,P,61,P,R
67,I,67,I,R
67,#,67,#,R
67,*,67,*,R
67,A,67,A,R
67,B,67,B,R
67,X,67,X,R
67,1,66,B,L
67,0,65,A,R
68,S,68,S,R
68,A,68,A,R
68,B,68,B,R
68,1,68,1,R
68,0,68,0,R
68,#,67,#,R
69,B,69,B,R
69,#,68,#,R
70,1,70,1,L
70,0,70,0,L
70,A,70,A,L
70,B,70,B,L
70,I,70,I,L
70,X,70,X,L
70,I,70,I,L
70,#,70,#,L
70,*,70,*,L
70,S,82,S,R
71,A,71,A,L
71,B,71,B,L
71,1,71,1,L
71,0,71,0,L
71,S,71,S,L
71,#,71,#,L
71,P,59,P,R
72,S,72,S,R
72,B,72,B,R
72,A,72,A,R
72,#,72,#,R
72,*,72,*,R
```

72,I,73,I,R
73,0,73,0,R
73,1,73,1,R
73,A,73,A,R
73,B,73,B,R
73,#,74,#,R
74,0,80,0,L
74,1,80,1,L
74,A,75,A,L
74,B,75,B,L
75,A,75,A,L
75,B,75,B,L
75,I,76,I,R
75,0,75,0,L
75,1,75,1,L
75,#,75,#,L
76,1,76,1,R
76,0,76,0,R
76,A,77,0,R
76,B,77,0,R
76,#,80,#,L
77,A,77,A,R
77,B,77,B,R
77,X,77,X,R
77,#,77,#,R
77,*,77,*,R
77,0,78,A,L
77,1,79,B,L
78,A,78,A,L
78,B,78,B,L
78,#,78,#,L
78,*,78,*,L
78,X,78,X,L
78,0,75,0,L
78,1,75,0,L
79,A,79,A,L
79,B,79,B,L
79,#,79,#,L
79,*,79,*,L
79,X,79,X,L
79,0,75,1,L
79,1,75,1,L
80,A,80,A,L
80,B,80,B,L
80,1,80,1,L
80,0,80,0,L
80,#,80,#,L
80,*,80,*,L
80,S,80,S,L
80,X,80,X,L
80,I,80,I,L
80,P,81,P,L
81,#,81,#,L
81,0,23,S,L
82,A,82,A,R
82,B,82,B,R
82,#,82,#,R
82,*,82,*,R
82,I,83,I,R
83,A,83,0,R
83,B,83,0,R
83,#,84,#,L
84,A,84,A,L
84,B,84,B,L
84,1,84,1,L
84,0,84,0,L
84,*,84,*,L
84,#,84,#,L
84,S,84,S,L
84,I,84,I,L
84,P,85,P,L
85,#,85,#,L
85,0,23,S,L
86,#,86,#,R
86,P,86,P,R
86,0,86,0,R
86,1,86,1,R
86,S,87,S,R
87,1,87,1,R
87,0,87,0,R
87,#,88,#,L
88,1,89,1,L
88,0,90,0,L
89,1,91,1,L
89,0,94,0,L
90,1,93,1,L
90,0,92,0,L
91,1,96,1,L
91,0,95,0,L
92,1,101,1,L
92,0,102,0,L
93,1,99,1,L
93,0,100,0,L
94,1,97,1,L
94,0,98,0,L
95,0,95,0,L
95,S,174,S,R
95,1,100,1,L
96,0,96,0,L
96,1,100,1,L
96,S,143,S,R
97,0,97,0,L
97,1,100,1,L
97,S,131,S,R
98,0,98,0,L
98,1,100,1,L
98,S,123,S,R
99,0,99,0,L
99,1,100,1,L
99,S,108,S,R
100,0,100,0,L
100,1,100,1,L
100,S,0,S,R
101,0,101,0,L
101,1,100,1,L
101,S,103,S,R
102,S,113,S,R
102,0,102,0,L
102,1,100,1,L
103,1,103,1,R
103,0,103,0,R
103,#,103,#,R
103,*,103,*,R
103,I,103,I,R
103,X,104,X,R
104,0,104,0,R
104,1,104,1,R
104,#,105,#,L
105,1,105,1,L
105,X,106,X,R
105,0,106,1,R
106,1,106,0,R
106,#,107,#,L
107,1,107,1,L
107,0,107,0,L
107,X,107,X,L

107,#,107,#,L
107,*,107,*,L
107,I,107,I,L
107,S,107,S,L
107,P,203,P,R
108,1,108,1,R
108,0,108,0,R
108,#,108,#,R
108,*,108,*,R
108,I,108,I,R
108,X,109,X,R
109,1,109,1,R
109,0,109,0,R
109,#,110,#,L
110,0,110,0,L
110,X,111,X,R
110,1,111,0,R
111,0,111,1,R
111,#,112,#,L
112,1,112,1,L
112,0,112,0,L
112,#,112,#,L
112,*,112,*,L
112,X,112,X,L
112,I,112,I,L
112,S,112,S,L
112,P,203,P,R
113,1,113,1,R
113,0,113,0,R
113,*,113,*,R
113,#,113,#,R
113,I,113,I,R
113,X,114,X,R
114,A,114,A,R
114,B,114,B,R
114,0,115,A,R
114,1,118,B,R
114,#,120,#,R
115,1,115,1,R
115,0,115,0,R
115,#,115,#,R
115,*,115,*,R
115,E,116,E,R
116,A,116,A,R
116,B,116,B,R
116,*,116,*,R
116,0,117,A,L
117,A,117,A,L
117,B,117,B,L
117,0,117,0,L
117,1,117,1,L
117,#,117,#,L
117,*,117,*,L
117,E,117,E,L
117,X,114,X,R
118,1,118,1,R
118,0,118,0,R
118,#,118,#,R
118,*,118,*,R
118,E,119,E,R
119,A,119,A,R
119,B,119,B,R
119,*,119,*,R
119,0,117,B,L
120,1,120,1,R
120,0,120,0,R
120,#,120,#,R
120,*,120,*,R
120,E,121,E,R
121,A,121,A,R
121,B,121,B,R
121,*,121,*,R
121,0,122,*,L
122,A,122,A,L
122,B,122,B,L
122,0,122,0,L
122,1,122,1,L
122,X,122,X,L
122,I,122,I,L
122,S,122,S,L
122,#,122,#,L
122,*,122,*,L
122,E,122,E,L
122,P,203,P,R
123,1,123,1,R
123,0,123,0,R
123,#,123,#,R
123,*,123,*,R
123,I,124,I,R
124,A,124,A,R
124,B,124,B,R
124,1,125,B,R
124,0,129,A,R
124,#,130,#,R
125,1,125,1,R
125,0,125,0,R
125,#,125,#,R
125,*,125,*,R
125,X,126,X,R
126,A,126,A,R
126,B,126,B,R
126,0,127,B,L
126,1,127,B,L
127,0,127,0,L
127,1,127,1,L
127,A,127,A,L
127,B,127,B,L
127,#,127,#,L
127,*,127,*,L
127,X,127,X,L
127,I,124,I,R
128,A,128,A,R
128,B,128,B,R
128,0,127,A,L
128,1,127,A,L
129,0,129,0,R
129,1,129,1,R
129,#,129,#,R
129,*,129,*,R
129,X,128,X,R
130,0,130,0,L
130,1,130,1,L
130,A,130,A,L
130,B,130,B,L
130,#,130,#,L
130,*,130,*,L
130,I,130,I,L
130,S,130,S,L
130,P,203,P,R
131,1,131,1,R
131,0,131,0,R
131,I,131,I,R
131,#,131,#,R
131,*,131,*,R
131,X,132,X,R
132,0,132,0,R

132,1,133,1,L
132,#,142,#,R
133,#,133,#,L
133,*,133,*,L
133,1,133,1,L
133,0,133,0,L
133,X,133,X,L
133,I,134,I,R
134,A,134,A,R
134,B,134,B,R
134,1,136,B,L
134,0,135,A,L
134,#,140,#,L
135,0,135,0,L
135,1,135,1,L
135,I,135,I,L
135,S,135,S,L
135,*,135,*,L
135,#,135,#,L
135,A,135,A,L
135,B,135,B,L
135,P,137,P,R
136,0,136,0,L
136,1,136,1,L
136,I,136,I,L
136,S,136,S,L
136,*,136,*,L
136,#,136,#,L
136,A,136,A,L
136,B,136,B,L
136,P,138,P,R
137,A,137,A,R
137,B,137,B,R
137,0,139,A,R
137,1,139,A,R
138,A,138,A,R
138,B,138,B,R
138,1,139,B,R
138,0,139,B,R
139,0,139,0,R
139,1,139,1,R
139,S,139,S,R
139,#,139,#,R
139,*,139,*,R
139,I,134,I,R
140,A,140,A,L
140,B,140,B,L
140,0,140,0,L
140,1,140,1,L
140,#,140,#,L
140,*,140,*,L
140,S,140,S,L
140,I,140,I,L
140,P,141,P,L
141,#,141,#,L
141,0,23,#,L
142,0,142,0,L
142,1,142,1,L
142,I,142,I,L
142,X,142,X,L
142,*,142,*,L
142,#,142,#,L
142,S,142,#,L
142,P,203,P,R
143,0,143,0,R
143,1,143,1,R
143,*,143,*,R
143,#,143,#,R
143,I,144,I,R
144,0,144,0,R
144,#,158,#,L
144,1,145,1,R
145,1,145,1,R
145,0,145,0,R
145,#,146,#,L
146,0,146,A,L
146,1,146,B,L
146,A,146,A,L
146,B,146,B,L
146,*,146,*,L
146,#,146,#,L
146,X,146,X,L
146,I,147,I,R
147,1,147,1,R
147,0,147,0,R
147,A,148,0,R
147,B,150,1,R
147,#,152,#,R
148,A,148,A,R
148,B,148,B,R
148,#,148,#,R
148,*,148,*,R
148,X,148,X,R
148,0,149,A,L
148,1,151,B,R
149,0,149,0,L
149,1,149,1,L
149,A,149,A,L
149,B,149,B,L
149,#,149,#,L
149,*,149,*,L
149,X,149,X,L
149,I,147,I,R
150,A,150,A,R
150,B,150,B,R
150,#,150,#,R
150,*,150,*,R
150,X,150,X,R
150,1,149,B,L
150,0,151,A,R
151,0,151,0,R
151,1,151,1,R
151,*,151,*,R
151,X,151,X,R
151,E,165,E,L
151,#,146,#,L
152,A,152,A,R
152,B,152,B,R
152,*,152,*,R
152,X,153,X,R
153,A,154,0,R
153,B,154,0,R
153,#,164,#,L
154,A,154,A,R
154,B,154,B,R
154,*,154,*,R
154,#,154,#,R
154,1,155,B,L
154,0,156,A,L
155,A,155,A,L
155,B,155,B,L
155,#,155,#,L
155,*,155,*,L
155,0,153,B,R
156,A,156,A,L
156,B,156,B,L

156,#,156,#,L
156,*,156,*,L
156,0,153,A,R
158,0,158,0,L
158,1,158,1,L
158,A,158,A,L
158,B,158,B,L
158,*,158,*,L
158,#,158,#,L
158,I,158,I,L
158,S,158,S,L
158,X,158,X,L
158,P,159,P,R
159,A,159,A,R
159,B,159,B,R
159,#,164,#,L
159,0,160,A,R
159,1,162,B,R
160,0,160,0,R
160,1,160,1,R
160,S,160,S,R
160,#,160,#,R
160,*,160,*,R
160,I,160,I,R
160,X,161,X,R
161,A,161,A,R
161,B,161,B,R
161,1,158,A,L
161,0,158,A,L
162,0,162,0,R
162,1,162,1,R
162,S,162,S,R
162,#,162,#,R
162,*,162,*,R
162,I,162,I,R
162,X,163,X,R
163,A,163,A,R
163,B,163,B,R
163,0,158,B,L
163,1,158,B,L
164,A,164,0,L
164,B,164,1,L
164,0,164,0,L
164,1,164,1,L
164,X,164,X,L
164,I,164,I,L
164,S,164,S,L
164,*,164,*,L
164,#,164,#,L
164,P,203,P,R
165,1,165,1,L
165,0,165,0,L
165,A,165,A,L
165,B,165,B,L
165,X,165,X,L
165,*,165,*,L
165,#,165,#,L
165,I,166,I,R
166,A,166,0,R
166,B,166,1,R
166,1,166,1,R
166,0,166,0,R
166,#,167,#,L
167,0,167,0,L
167,1,167,1,L
167,I,168,I,R
168,A,168,A,R
168,B,168,B,R
168,#,164,#,L
168,0,169,A,R
168,1,172,B,R
169,0,169,0,R
169,1,169,1,R
169,A,169,A,R
169,B,169,B,R
169,#,169,#,R
169,*,169,*,R
169,X,170,X,R
170,A,171,0,L
170,B,171,0,L
170,0,170,0,R
170,1,170,1,R
171,X,171,X,L
171,A,171,A,L
171,0,171,0,L
171,1,171,1,L
171,B,171,B,L
171,#,171,#,L
171,*,171,*,L
171,I,168,I,R
172,0,172,0,R
172,1,172,1,R
172,A,172,A,R
172,B,172,B,R
172,#,172,#,R
172,*,172,*,R
172,X,173,X,R
173,0,173,0,R
173,1,173,1,R
173,A,171,1,L
173,B,171,1,L
174,1,174,1,R
174,0,174,0,R
174,#,174,#,R
174,*,174,*,R
174,I,175,I,R
175,0,175,0,R
175,#,196,#,R
175,1,176,1,R
176,0,176,0,R
176,1,176,1,R
176,#,177,#,L
177,0,177,A,L
177,1,177,B,L
177,A,177,A,L
177,B,177,B,L
177,#,177,#,L
177,*,177,*,L
177,X,177,X,L
177,I,178,I,R
178,1,178,1,R
178,0,178,0,R
178,A,179,0,R
178,B,182,1,R
178,#,183,#,R
179,A,179,A,R
179,B,179,B,R
179,#,179,#,R
179,*,179,*,R
179,X,179,X,R
179,0,181,A,L
179,1,180,1,R
180,1,180,1,R
180,0,180,0,R
180,*,180,*,R
180,X,180,X,R

180,#,177,#,L
180,E,189,E,L
181,A,181,A,L
181,B,181,B,L
181,X,181,X,L
181,0,181,0,L
181,1,181,1,L
181,#,181,#,L
181,*,181,*,L
181,I,178,I,R
182,A,182,A,R
182,B,182,B,R
182,#,182,#,R
182,*,182,*,R
182,X,182,X,R
182,0,180,A,R
182,1,181,B,L
183,A,183,A,R
183,B,183,B,R
183,*,183,*,R
183,X,184,X,R
184,1,202,1,L
184,0,202,0,L
184,A,185,0,R
184,B,188,1,R
185,A,185,A,R
185,B,185,B,R
185,#,185,#,R
185,*,185,*,R
185,1,186,A,L
185,0,186,A,L
186,A,186,A,L
186,B,186,B,L
186,#,186,#,L
186,*,186,*,L
186,0,186,0,L
186,1,186,1,L
186,X,187,X,R
187,0,187,0,R
187,1,187,1,R
187,A,185,0,R
187,B,188,1,R
187,#,202,#,L
188,A,188,A,R
188,B,188,B,R
188,#,188,#,R
188,*,188,*,R
188,1,186,B,L
188,0,186,B,L
189,1,189,1,L
189,0,189,0,L
189,A,189,0,L
189,B,189,1,L
189,*,189,*,L
189,#,189,#,L
189,X,189,X,L
189,I,190,I,R
190,0,190,0,R
190,1,190,1,R
190,#,190,#,R
190,*,190,*,R
190,X,191,X,R
191,0,192,A,L
191,1,194,B,L
191,A,191,A,R
191,B,191,B,R
191,#,202,#,L
192,1,192,1,L
192,0,192,0,L
192,A,192,A,L
192,B,192,B,L
192,*,192,*,L
192,#,192,#,L
192,X,192,X,L
192,I,193,I,R
193,0,190,A,R
193,1,190,A,R
193,A,193,A,R
193,B,193,B,R
194,1,194,1,L
194,0,194,0,L
194,A,194,A,L
194,B,194,B,L
194,*,194,*,L
194,#,194,#,L
194,X,194,X,L
194,I,195,I,R
195,0,190,B,R
195,1,190,B,R
195,A,195,A,R
195,B,195,B,R
196,0,196,0,R
196,1,196,1,R
196,#,196,#,R
196,*,196,*,R
196,X,197,X,R
196,S,196,S,R
196,I,196,I,R
197,A,197,A,R
197,B,197,B,R
197,0,198,A,L
197,1,200,B,L
197,#,202,#,L
198,1,198,1,L
198,0,198,0,L
198,A,198,A,L
198,B,198,B,L
198,#,198,#,L
198,*,198,*,L
198,I,198,I,L
198,S,198,S,L
198,X,198,X,L
198,P,199,P,R
199,A,199,A,R
199,B,199,B,R
199,0,196,A,R
199,1,196,A,R
200,1,200,1,L
200,0,200,0,L
200,A,200,A,L
200,B,200,B,L
200,#,200,#,L
200,*,200,*,L
200,I,200,I,L
200,S,200,S,L
200,X,200,X,L
200,P,201,P,R
201,A,201,A,R
201,B,201,B,R
201,0,196,B,R
201,1,196,B,R
202,0,202,0,L
202,1,202,1,L
202,A,202,0,L
202,B,202,1,L
202,I,202,I,L

202,S,202,S,L
202,X,202,X,L
202,#,202,#,L
202,*,202,*,L
202,P,203,P,R
203,1,203,1,R
203,0,203,0,R
203,#,204,#,L
204,1,204,1,L
204,P,205,P,R
204,0,205,1,R
205,1,205,0,R
205,#,206,#,L
206,1,206,1,L
206,0,206,0,L
206,P,207,P,L
207,#,207,#,L
207,0,23,#,L
208,#,208,#,R
208,P,209,P,R
209,0,209,0,R
209,1,209,1,R
209,#,1,#,R

B.5 λ-Calculus

B.5.1 Addition

$$\lambda n.\lambda m.\lambda n.(\lambda p.\lambda f.\lambda x.f(p\ f\ x))m$$

B.5.2 Subtraction

$$\lambda m.\lambda n.n(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u))m$$

B.5.3 Equality

$$(\lambda z.(\lambda q.(\lambda a.\lambda m.\lambda n.n\ a\ m(\lambda x.q)z(m\ a\ n(\lambda x.q)z)(n\ a\ m(\lambda x.q)z))$$
$$(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u)))(\lambda x.\lambda y.y))(\lambda x.\lambda y.x)$$

B.5.4 Multiplication

$$\lambda m.\lambda n.\lambda f.m(n\ f)$$

B.5.5 Division

$$(\lambda u.(\lambda z.(\lambda t.(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))(\lambda g.\lambda q.\lambda a.\lambda b.(\lambda n.n(\lambda x.u)z)$$
$$b\ u((\lambda a.\lambda b.\lambda k.\lambda j.a\ t\ b(\lambda x.u)z\ j\ k)a\ b((\lambda x.\lambda y.\lambda f.f\ x\ y)q\ a)$$
$$(g((\lambda n.\lambda f.\lambda x.f(n\ f\ x))q)((\lambda m.\lambda n.n\ t\ m)a\ b)b)))u)$$
$$(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u)))(\lambda x.\lambda y.x))(\lambda x.\lambda y.y)$$

B.5.6 Exponentiation

$$\lambda e.\lambda b.b\ e$$

B.5.7 List Membership

$$(\lambda z.(\lambda t.(\lambda w.((\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.z))b\ z\\ ((\lambda m.\lambda n.n\ w\ m(\lambda x.z)t(m\ w\ n(\lambda x.z)t)(n\ w\ m(\lambda x.z)t))((\lambda p.p\ t)b)\\ c\ t(a((\lambda p.p\ z)b)c)))))(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)\\ (\lambda u.u)))(\lambda x.\lambda y.x))(\lambda x.\lambda y.y)$$

B.5.8 Linear Search

$$(\lambda v.(\lambda z.(\lambda t.(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.z))c(\lambda f.\lambda x.f\ x)\\ ((\lambda m.\lambda n.n\ t\ m(\lambda x.z)v(m\ t\ n(\lambda x.z)v)(n\ t\ m(\lambda x.z)v))((\lambda p.p\ v)c)b)z\\ ((\lambda n.\lambda f.\lambda x.f(n\ f\ x))(a\ b((\lambda p.p\ z)c)))))(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))\\ (\lambda u.x)(\lambda u.u)))(\lambda x.\lambda y.y))(\lambda x.\lambda y.x)$$

B.5.9 List Reversal

$$(\lambda j.(\lambda z.(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))(\lambda g.\lambda a.\lambda l.(\lambda p.p(\lambda x.\lambda y.j))l\\ a(g((\lambda x.\lambda y.\lambda f.f\ x\ y)((\lambda p.p\ z)l)a)((\lambda p.p\ j)l)))(\lambda x.z))(\lambda x.\lambda y.x))(\lambda x.\lambda y.y)$$

B.5.10 Stateful List Reversal

$$(\lambda j.(\lambda m.(\lambda k.(\lambda q.(\lambda s.(\lambda v.(\lambda i.(\lambda r.(\lambda z.(\lambda a.(z(\lambda a.\lambda b.\lambda c.\lambda d.(\lambda a.\lambda b.\lambda d.\lambda c.a\ i\ b\\ (\lambda x.j)m\ c\ d)b\ c(a(s\ b)(i\ c)((z(\lambda a.\lambda b.\lambda c.\lambda d.r\ b(v((\lambda a.\lambda b.a\ q\ b\ m)c\ d)\\ ((z(\lambda a.\lambda b.\lambda c.\lambda d.r\ b(v\ c(q\ d))(v(k\ d)(a(i\ b)c(q\ d)))))(i\ c)(k\ d)(q\ d)))\\ (v(k\ d)(a(i\ b)(i\ c)(q\ d)))))b\ c\ d))d))j(i((z(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.j))c\\ b(a(s\ b)(q\ c)))j)a))a))(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x))))(\lambda n.n(\lambda x.j)m))\\ (\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u)))(\lambda x.\lambda y.\lambda f.f\ x\ y))\\ (\lambda n.\lambda f.\lambda x.f(n\ f\ x)))(\lambda p.p\ j))(\lambda p.p\ m))(\lambda x.\lambda y.x))(\lambda x.\lambda y.y)$$

B.5.11 Bubble Sort

$$(\lambda j.(\lambda o.(\lambda u.(\lambda h.(\lambda t.(\lambda i.(\lambda s.(\lambda g.(\lambda f.(\lambda v.(\lambda z.(z(\lambda a.\lambda b.\lambda c.\lambda d.\lambda e.\\ (\lambda m.\lambda n.n\ f\ m(\lambda x.o)j)d(f((z(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.o))c\ b(a(g\ b)(t\ c)))o)e))\\ ((\lambda a.\lambda b.\lambda c.\lambda d.a\ f\ b(\lambda x.o)j\ d\ c)(s\ d\ e)(s\ c\ e)(a\ j(g\ c)(g\ d)\\ ((z(\lambda a.\lambda b.\lambda c.\lambda d.v\ b(i(s\ c\ d)((z(\lambda a.\lambda b.\lambda c.\lambda d.v\ b(i\ c(t\ d))(i(h\ d)(a(f\ b)c(t\ d)))))\\ (f\ c)(h\ d)(t\ d)))(i(h\ d)(a(f\ b)(f\ c)(t\ d)))))c\ d\ e))(a\ b(g\ c)(g\ d)e))\\ (b(a\ o\ o\ u\ e)e)))o\ o\ u)(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x))))(\lambda n.n(\lambda x.o)j))\\ (\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u)))(\lambda n.\lambda f.\lambda x.f(n\ f\ x)))(\lambda a.\lambda b.a\ t\ b\ j))\\ (\lambda x.\lambda y.\lambda f.f\ x\ y))(\lambda p.p\ o))(\lambda p.p\ j))(\lambda f.\lambda x.f\ x))(\lambda x.\lambda y.y))(\lambda x.\lambda y.x)$$

B.5.12 Universal TM

$(\lambda z.(\lambda u.(\lambda l.(\lambda k.(\lambda j.(\lambda i.(\lambda g.(\lambda f.(\lambda e.(\lambda d.(\lambda c.(\lambda b.z(\lambda a.\lambda s.\lambda h.\lambda t.\lambda p.d(c\ s(e\ h\ p)t)$
$p(a(g(c\ s(e\ h\ p)t))(b(g(i(i(c\ s(e\ h\ p)t))))(j\ h)((\lambda n.\lambda f.\lambda x.f(n\ f\ x))h))$
$t((z(\lambda a.\lambda b.\lambda c.\lambda d.(\lambda n.n(\lambda x.l)u)b(k\ c(i\ d))(k(g\ d)(a(j\ b)c(i\ d)))))h$
$(g(i(c\ s(e\ h\ p)t)))p))))(\lambda n.n(\lambda x.l)u))((z(\lambda a.\lambda s.\lambda y.\lambda t.((\lambda p.p\ l\ u)(d\ t))$
$(((\lambda p.\lambda q.p\ q\ p)(f\ s(g(g\ t)))(f\ y(g(i(g\ t)))))(i(i(g\ t)))(a\ s\ y(i\ t)))$
$(k\ l(k\ l(k\ l(\lambda x.u)))))))(\lambda p.p(\lambda x.\lambda y.l)))(\lambda a.\lambda b.a\ i\ b\ u))(\lambda m.\lambda n.n\ j\ m$
$(\lambda x.l)u(m\ j\ n(\lambda x.l)u)(n\ j\ m(\lambda x.l)u)))(\lambda p.p\ u))(\lambda p.p\ l))(\lambda n.\lambda f.\lambda x.n$
$(\lambda g.\lambda h.h(g\ f))(\lambda u.x)(\lambda u.u)))(\lambda x.\lambda y.\lambda f.f\ x\ y))(\lambda x.\lambda y.y))(\lambda x.\lambda y.x))$
$(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))$

B.5.13 Universal RASP

$(\lambda s.(\lambda r.(\lambda q.(\lambda p.(\lambda n.(\lambda l.(\lambda k.(\lambda j.(\lambda i.(\lambda h.(\lambda g.(\lambda f.(\lambda e.(\lambda d.(\lambda c.(\lambda b.s$
$(\lambda a.\lambda m.\lambda o.f(g\ n(d\ m))n(a(c\ r(c\ p(d\ m)))o)(f(g\ n(d\ m))p(a(c\ r(b\ p(d\ m)))o)$
$(f(g\ n(d\ m))(\lambda f.\lambda x.f(f(f\ x)))(a(c\ r((\lambda m.e\ p(g\ n(d(c\ r\ m)))(d(c\ r\ m)))$
$(d\ m)))o)(f(g\ n(d\ m))(\lambda f.\lambda x.f(f(f(f\ x))))(a(c\ r((\lambda m.e(g\ n(d(c\ r\ m)))(g\ p\ m)$
$(d(c\ r\ m)))(d\ m)))o)(f(g\ n(d\ m))(\lambda f.\lambda x.f(f(f(f(f\ x)))))(a(c\ r(d\ m))$
$((\lambda m.\lambda o.(i(g\ p\ m)o))m\ o))(f(g\ n(d\ m))(\lambda f.\lambda x.f(f(f(f(f(f\ x))))))(a(c\ r$
$((\lambda m.(f(g\ p(d(c\ r\ m)))r)(d(c\ r\ m))(b\ r(e\ r(g\ n(d(c\ r\ m)))(d(c\ r\ m)))))$
$(d\ m)))o)(f(g\ n\ (d\ m))(\lambda f.\lambda x.f(f(f(f(f(f\ x)))))))(a(c\ r((\lambda m.e\ p(g(g\ n$
$(d(c\ r\ m)))(d(c\ r\ m)))(d(c\ r\ m)))(d\ m)))o)(i(d\ m)o)))))))))(\lambda d.\lambda m.(f(g\ d\ m)r)$
$(e\ d(k(h\ m))m)(e\ d(k(g\ d\ m))m)))(\lambda d.\lambda m.(f(k(h\ m))(g\ d\ m))(e\ d\ r\ m)$
$(e\ d(l(g\ d\ m))m)))(\lambda m.e\ n(g(g\ r\ m)m)m)))(s(\lambda a.\lambda b.\lambda c.\lambda d.(\lambda n.n(\lambda x.r)q)$
$b(i\ c(j\ d))(i((\lambda p.p\ q)d)(a(k\ b)c(j\ d))))))\ (\lambda m.\lambda n.n\ k\ m(\lambda x.r)q(m\ k\ n$
$(\lambda x.r)q)(n\ k\ m(\lambda x.r)q)))(\lambda a.\lambda b.a\ j\ b\ q))(s(\lambda a.\lambda b.\lambda c.(\lambda p.p(\lambda x.\lambda y.r))c$
$b(a(l\ b)(j\ c)))r))(\lambda x.\lambda y.\lambda f.f\ x\ y))(\lambda p.p\ r))(\lambda n.\lambda f.\lambda x.n(\lambda g.\lambda h.h(g\ f))$
$(\lambda u.x)(\lambda u.u)))(\lambda n.\lambda f.\lambda x.f(n\ f\ x)))\ (\lambda f.\lambda x.f\ x))(\lambda f.\lambda x.f(f\ x)))(\lambda x.\lambda y.x))$
$(\lambda x.\lambda y.y))(\lambda g.(\lambda x.g(x\ x))(\lambda x.g(x\ x)))$

B.6 SKI

B.6.1 Addition

$$SI(K(S(S(KS)K)))$$

B.6.2 Subtraction

$S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))$
$(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))))K$

B.6.3 Equality

$S(S(S(KS)(S(K(S(KS)(S(K(SSK)))))))(S(K(S(K(SS(KK)))K))S))$
$(K(S(K(S(K(S(SI(K(K(KI))))(KK)))))(S(K(S(SI(K(S(K(S(K(S$
$(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S$
$(K(S(K(S(K(SI))K))))(SI)))K))))K))))))K)))$

B.6.4 Multiplication

$$S(KS)KII$$

B.6.5 Division

$$S(K(S(K(S(SSK(S(K(SS(S(SSK))))K))(K(KI))))(S(K(S(K(S(K(S(S(S(SI(K(K(KI))))(KK))(K(KI)))))))))))(S(S(K(S(K(S(K(S(KS)K))S))(S(K(S(KS)(S(KS)))))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(KK)))(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K)))))K))S))(S(KS))))(S(K(S(K(S(SI(K(KI)))(KK)))))))(S(K(S(K(S(K(SS(KK)))K))S))(S(K(S(K(S(SI(K(K(KI))))(KK)))))))(S(K(S(K(S(K(S(K(S(K(S(K(SS(KI)))))))))(S(S(K(S(KS)(S(K(S(K(S(K(S(KS)K))S))K)))))(S(K(SS(K(S(S(KS)K)))))K)))))K))K))(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))))K)))))K)))))(SI(KK))))))K)))))$$

B.6.6 Exponentiation

$$S(K(SI))KII$$

B.6.7 List Membership

$$SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(K(S(S(SI(K(K(K(KI)))))(K(KI)))))))(S(S(KS)(S(K(S(K(SS(K(KK))))(S(S(K(S(S(S(KS)(S(K(S(KS)(S(K(SSK)))))))(S(K(S(K(SS(KK)))K))S))(K(S(K(S(SI(K(K(KI))))(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))))K)))))(SI(KK))))))K)))))(S(K(S(K(SS(K(SI(K(KI))))))K))))$$

B.6.8 Linear Search

$$SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(S(KS)(S(K(S(K(S(K(SS(K(KI)))))(S(S(SI(K(K(K(KI)))))(KI)))))(S(S(K(S(S(S(KS)(S(K(S(KS)(S(K(SSK)))))))(S(K(S(K(SS(KK)))K))S))(K(S(K(S(SI(K(K(KI))))(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))))K)))))(SI(KK))))))K)))))(S(K(S(K(S(S(KS)K)))))))(S(K(S(K(SS(K(SI(K(KI))))))K))))$$

B.6.9 List Reversal

$$SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(S(K(S(KS)(S(SI(K(K(K(KI)))))))K)))(S(K(SS(K(SI(K(KI)))))))))(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))(SI(KK)))))K))S))K))(KK)$$

B.6.10 Stateful List Reversal

$S(S(K(SS(KI)))(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(SS(K(KI))))))$
$(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))$
$(SI)))K))))K)))(SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(S(K(S(KS)$
$(S(SI(K(K(K(KI)))))))K)))(S(K(S(K(SS(K(SI(K(KI))))))K)))))$
$(S(K(SS(K(S(S(KS)K)))))K))(KI))))))K))(S(K(S(K(S(K(S(K(S$
$(SSK(S(K(SS(S(SSK))))K))(K(KI))))(S(K(S(K(S(K(S(K(SS(KI)))))))$
$(S(S(K(S(KS)(S(K(S(KS)K)))))(S(K(S(K(S(K(S(SI(K(KI)))(KK)))))$
$(S(S(K(S(K(S(SI(K(K(KI))))(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS$
$(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K$
$(SI))K))))(SI)))K))))K))))))K)))))K))))))))(S(S(K(S(K(S(KS)$
$(S(K(S(KS)(S(K(S(KS)K)))))))(S(K(S(K(SS(K(S(K(S(K(S(K(SS$
$(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S$
$(K(SI))K))))(SI)))K))))K))))K)))))(S(K(SS(K(S(S(KS)K)))))K)))))K))$
$(S(S(K(S(KS)(S(K(S(KS)(S(KS)))))))(S(K(S(K(S(K(S(K(SS(K(S(K$
$(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))))K))S))K))))(S(K(S(K(S(K$
$(SS(KI)))))))(S(K(S(K(S(K(SS(K(S(KK)(S(K(S(K(SI(KK)))))$
$(SI(K(SI(K(KI)))))))))K))S))(S(K(S(KS)K)))))))))(K(SSK(S$
$(K(SS(S(SSK))))K)(S(K(S(K(S(S(K(S(KS)(S(KS))))(S(K(S(K(S(K(S$
$(K(SS(K(S(K(S(K(SS(K(SI(K(KI))))))K))(S(K(S(K(S(K(S(K(SS$
$(KK)))K))S))(SI)))K)))))K))S))K))(S(SI(K(K(KI))))(KK))))))$
$(S(K(S(K(S(K(S(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))$
$(SI(KK)))))))(S(K(S(K(SS(K(SI(K(KI))))))K))))))))$
$(S(K(SS(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))$
$(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))))K))))$

B.6.11 Bubble Sort

$S(S(K(SS(K(K(SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(K(S(KS)$
$(S(KS))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SS(K(SI(K(KI))))))K))$
$(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K)))))K))S))K))$
$(S(SI(K(K(KI))))(KK))))))(S(K(S(K(S(K(S(S(K(S(K(S(K(S(K$
$(S(K(SS(KK)))K))S))(SI)))K))(SI(KK)))))))(S(K(S(K(SS$
$(K(SI(K(KI))))))K))))))))(S(K(SS(K(S(K(S(K(S(K(SS(K(KI))))))$
$(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))$
$(SI)))K))))K)))))K)))))))(S(K(SS(K(KI))))(S(K(S(K(S(K(S(K(SS$
$(K(K(KI)))))(S(K(S(SSK(S(K(SS(S(SSK))))K))(K(KI)))))))$
$(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(KK)K))))(S(K(SS(KI))))))$
$(S(S(KS)K))))K))(S(S(SI(K(KI)))(K(KI)))(KI))))))(S(K(S(KS)$
$(S(K(S(KS)(S(K(S(K(S(KS)(S(KS))))(S(S(KS)(S(K(S(K(SS(K(S(K$
$(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K$
$(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))(SSK(S(K(SS(S(SSK))))K)$
$(S(K(S(K(S(S(K(S(KS)(S(SI(K(K(K(KI)))))))K)))(S(K(S(K(SS(K$
$(SI(K(KI))))))K)))))(S(K(SS(K(S(S(KS)K)))))K))(KI))))))K))$
$(S(K(S(K(S(SI(K(K(KI))))(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS$
$(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K$
$(SI))K))))(SI)))K))))K))))))K))))))))))))))))))(S(K(S(K(SS$
$(K(S(K(S(K(S(K(S(K(SS(K(S(S(KS)K)))))K)))))))(S(K(S(K(S(KS)K))S))(S(K$
$(S(KS)K)))))K)))))))(S(K(S(K(S(K(S(KS)K))S))(S(K(S(KS)$
$(S(KS))))))))))))(S(S(K(S(K(S(K(S(K(S(K(S(KS)K))S))K))S))$
$(S(K(S(KS)(S(KS)))))))(S(S(K(S(K(S(KS)K))S))(S(K(S(KS)(S(K$
$(S(K(S(K(S(K(S(SI(K(KI)))(KK)))))(S(S(K(S(K(S(SI(K(K(KI))))$
$(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS$
$(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))$
$K))))))K)))))K)))))))(S(KK))))(S(K(S(K(S(K(S(S(K(S(K(S(KS)$
$(S(K(S(KS)(S(K(S(KS)K))))))))(S(K(S(K(SS(K(S(S(KS)K)))))$
$K)))))(S(K(S(K(SS(K(S(S(KS)K)))))K))(SI(KK))))))K))))$
$(S(S(K(S(K(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(KS))))))))))$
$(S(S(KS)(S(K(S(K(S(K(S(KS)K))S))K)))))))K))K)(S(K(S(K(S$
$(K(S(K(S(K(S(K(SS(KI)))))))))(S(S(K(S(K(S(KS)K))S))(S(K$
$(S(KS)K)))))))K))(S(KK))))))))(K(S(K(S(K(SI(KK)))))$
$(SI(K(SI(K(KI)))))))$

B.6.12 Universal TM

$S(K(SSK(S(K(SS(S(SSK))))K)))(S(S(K(S(K(S(K(S(KS)K))S))$
$(S(K(S(K(S(KS)(S(K(S(K(S(KS)(S(KS))))(S(K(S(K(SS(KI)))$
$(S(K(SI(K(K(K(KI))))))))))))))))(S(K(S(K(S(K(SS(KK)))K))S))))))))$
$(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SI(KK)))))))(SI(K(SI(K$
$(KI)))))))))K))S))K))))(S(S(K(S(KS)(S(K(S(KS)(S(K(S(K(S(KS)$
$(S(K(S(KS)(S(KS)))))))(S(K(S(K(SS(KK)))(S(KS)))))))))))))$
$(S(S(K(S(K(S(K(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)$
$(S(KS)))))))))))))(S(S(K(S(K(S(K(S(K(S(K(S(KS)K))S))$
$K))S))K))S))K))))K))(S(K(S(K(S(K(S(K(S(K(SI(KK)))))))))$
$(S(K(S(K(S(K(SS(KK)))K))S)))))))(S(K(S(K(S(K(S(K(SS(K(S(K$
$(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))))K))S))K))))(S(K(S(K(S(KK)$
$(S(K(S(K(SS(K(S(K(S(KK)K))(S(S(KS)K))))))(S(K(S(K(S(K$
$(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K$
$(S(K(SI))K))))(SI)))K))))K))))))(S(K(S(KS)(S(K(S(KS)$
$(S(K(S(K(S(K(S(K(S(SI(K(K(KI))))(KK)))(SI(KK))))(SI(K(KI)))))$
$(SI(K(KI)))))))))))))(S(K(S(K(S(K(SS(KK)))K))S))))))))$
$(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SI(KK)))))(SI(K(SI(K$
$(KI)))))))))K))S))K))))))(S(K(S(KK)(S(K(S(K(S(K(S(K(S(K(S$
$(K(SS(KI)))))))(S(S(K(S(K(S(K(S(KS)K))S))K))(SSK(S(K(SS$
$(S(SSK))))K)(S(K(S(K(S(S(K(S(KS)(S(KS))))(S(K(S(K(S(K(S(K$
$(SS(K(S(K(S(K(SS(K(SI(K(KI))))))K))(S(K(S(K(S(K(S(K(SS$
$(KK)))K))S))(SI)))K)))))K))S))K))(S(SI(K(K(KI))))(KK))))))$
$(S(K(S(K(S(K(S(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))$
$(SI(KK)))))))(S(K(S(K(SS(K(SI(K(KI))))))K))))))))(S(K(SS(K$
$(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K$
$(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))))K)))))))(S(K(S$
$(K(S(K(S(K(SI(KK)))(SI(K(KI))))))))))(S(K(S(K(S(K(SS$
$(KK)))K))S))))))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SI(KK)))))$
$(SI(K(SI(K(KI)))))))))K))S))K))))))(SSK(S(K(SS(S(SSK))))K)$
$(S(K(S(K(S(K(S(K(S(K(SS(K(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))$
$(SI)))K(KI)(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K(KI)(S(K$
$(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K(KI)(KK))))))))(S(S(K(S$
$(SI(K(KI)))(KK)))(SI(K(K(K(KI))))))))))))(S(S(K(S(K(S(KS)$
$(S(KS))))(S(K(SS(K(S(K(S(K(SI(K(KI))))(SI(K(KI)))))$
$(SI(KK)))))))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SS(K(S(K(S$
$(K(SI(KK)))(SI(K(KI)))))(SI(KK))))))K))(S(S(S(KS)(S(K(S(KS)$
$(S(K(SSK)))))))(S(K(S(K(SS(KK)))K))S))(K(S(K(S(K(S(SI(K(K(KI))))$
$(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS$
$(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))$
$(SI)))K))))K))))))K))))))K))S))(S(K(SSK)))))(S(K(S(K(SS(K$
$(S(K(SI(KK)))(SI(KK))))))K))(S(S(S(KS)(S(K(S(KS)(S(K(SSK)))))))$
$(S(K(S(K(SS(KK)))K))S))(K(S(K(S(K(S(SI(K(K(KI))))(KK)))))$
$(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))$
$(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))))K)))))))))$
$(S(K(S(K(S(K(SS(K(SI(K(KI))))))K)))))))$

B.6.13 Universal RASP

$S(S(K(S(K(SS(K(S(KK)(S(K(SS(KI)))(S(K(S(K(SS(K(S(S(K(S$
$(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))(S(K(S(K(SI(KK)))))$
$(SI(K(SI(K(KI)))))(KI)))I))))K))(SI(KI)))))))(S(KS))))$
$(S(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(SSK(S(K(SS$
$(S(SSK))))K))))))))))))(S(S(K(S(K(S(K(S(KS)K))S))(S(K(S(KS)$
$(S(K(S(KS)(S(K(S(K(S(K(S(KS)K))S))(S(KS))))))))))))$
$(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(S(KS)K)))K))))$
$(S(S(K(S(KS)K))(SI(K(KI)))))))(S(S(K(S(KS)K))(SI(K(S(S$
$(KS)K)I)))))))K))))(S(K(S(K(S(K(S(K(S(K(S(KS)K))S))K))S))$
$(S(K(S(KS)K)))))))(S(K(SS(K(KI)))))))(S(K(S(K(S(K(SS$
$(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))K))S))K)))$
$(S(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S$
$(K(S(KS)(S(KS))))))))))))))))))(S(S(K(S(K(S(K(S(K(S$
$(KS)K))S))(S(K(S(K(S(K(S(K(S(K(S(K(S(KS)K))S))K))S))K))S))K))S))$
$(S(K(S(KS)K)))))))(S(K(SS(K(K(S(S(KS)K)I)))))))(S(K(S$
$(K(S(K(SS(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))$
$I))))))K))S))K))(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S$
$(S(K(S(K(S(KS)K))S))K)))K))))(S(S(K(S(K(S(K(S(KS)K))S))K))$
$(SI(K(KI)))))))K))))))(S(K(S(K(S(K(S(K(SS(KK)))K))S))$
$(S(K(S(KS)K)))))))(S(K(SS(K(K(S(S(KS)K)I)))))))(S(K(S$
$(K(SS(K(S(K(S(K(SS(KK)))(S(KS))))(S(K(S(K(SS(K(S(K(S(K$
$(S(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))$
$(S(K(SS(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))))(S$
$(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))))K))))K))S)))))$
$(S(K(S(KS)(S(KS)))))))(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S$
$(K(SS(KK)))(S(KS))))(S(K(S(K(SS(K(S(K(S(KK)(S(K(S(K(S$
$(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S$
$(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))(SSK(S(K(SS(S(SSK))))K)$
$(S(K(S(K(S(S(K(S(KS)(S(SI(K(K(K(KI)))))))))K)))(S(K(S(K$
$(SS(K(SI(K(KI))))))K)))))(S(K(SS(K(S(S(KS)K)))))K))$
$(KI))))))K))S)))))K))S))(S(KS))))(S(K(SS(K(K(KI)))))))$
$(S(K(S(K(S(K(SS(K(S(K(S(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K$
$(KI)))))))K))))K))S))K))))))(S(S(K(S(KS)(S(K(S(KS)$
$(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)$
$(S(KS))))))))))))))))))(S(K(S(K(S(K(S(K(S(K(SS(K(S(K$
$(S(K(S(K(S(K(S(K(S(S(K(S(K(S(KS)K))S))K)))K))))(S(S(K$
$(S(K(S(K(S(KS)K))S))K))(SI(K(KI)))))))))(S(K(S(K(SS$
$(K(S(KK)K))))(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)K)))))))))$
$(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(SSK(S(K(SS(S(SSK))))K)(S(K$
$(S(K(S(S(K(S(KS)(S(KS))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(K$
$(SS(K(SI(K(KI)))))K))(S(K(S(K(S(K(S(K(SS(KK)))K))S))$
$(SI)))K)))))K))S))K))(S(SI(K(K(KI))))(KK))))))(S(K(S(K$
$(S(K(S(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))$
$(SI(KK)))))))(S(K(S(K(SS(K(SI(K(KI))))))K))))))))(S(K$
$(SS(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))$
$(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))))K))I))$

$$(S(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))(S(K(S(K(SI$$
$$(KK)))))(SI(K(SI(K(KI)))))(KI)))I))I))))(SI(K(KI)))))))$$
$$(S(KS)))))))(S(K(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(S$$
$$(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))(S(K(S(K(S$$
$$(SS(K(SI(K(KI))))))K))S))K)))))K))S))K))S))K))(SI(K(S(S$$
$$(KS)K)I))))))))K))S))(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S$$
$$(KS)K))S))K))S))K))S))K))S))(S(K(S(KS)K))))))))(S(K(SS(K(K$$
$$(S(S(KS)K)(S(S(KS)K)I)))))))))(S(K(S(K(S(K(SS(K(S(K(S$$
$$(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))K))S))K)))(S$$
$$(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(K$$
$$(S(KS)(S(KS))))))))))))))))))(S(K(S(K(S(K(S(K(S(K(S(K$$
$$(S(K(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(S(K$$
$$(S(K(S(KS)K))S))K)))K))))(S(S(K(S(K(S(K(S(KS)K))S))K))$$
$$(SI(K(KI)))))))))(S(K(S(K(SS(K(S(KK)K))))(S(K(S(KS)(S(K$$
$$(S(KS)(S(K(S(KS)K)))))))))))(S(K(S(K(SS(K(S(K(S(K(S(K(SS$$
$$(K(SI(K(KI))))))K))S))K))))(S(K(S(K(S(KS)(S(KS))))(S(K$$
$$(SS(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))(S(S$$
$$(KS)K)I)))))))))))(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(S(K$$
$$(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))(S(K(S(K(S(K$$
$$(SS(K(SI(K(KI))))))K))S))K)))))K))S))K))S))K))))))K))S))$$
$$K))S))K))S))K))S))K))S))(S(K(S(KS)K)))))(S(K(SS(K(K(S(K(S$$
$$(S(KS)K)I))(S(S(KS)K)I)))))))(S(K(SS(K(S(K(S(K(S(K(SI(KK)))))$$
$$(SI(K(SI(K(KI)))))I))(S(S(K(SSK(S(K(SS(S(SSK))))K)(S(K(S$$
$$(K(S(S(K(S(KS)(S(KS))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SS(K$$
$$(SI(K(KI))))))K))(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))$$
$$K)))))K))S))K))(S(SI(K(K(KI))))(KK)))))(S(K(S(K(S(K(S$$
$$(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))(SI(KK)))))))$$
$$(S(K(S(K(SS(K(SI(K(KI))))))K)))))))(S(K(SS(K(S(K(S(K(S(K$$
$$(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S$$
$$(K(S(K(SI))K))))(SI)))K))))K)))))K))I))(S(S(K(S(K(S(K(SI(KK)))))$$
$$(SI(K(SI(K(KI)))))))(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))$$
$$(KI)))I))I)))))K))))(S(S(K(S(K(S(K(S(K(S(KS)K))S))(S(K(S(K$$
$$(S(K(S(K(S(K(S(K(S(K(S(KS)K))S))K))S))K))S))(S(K(S(KS)K)))))))$$
$$(S(K(SS(K(K(S(S(KS)K)(S(K(S(S(KS)K)I))(S(S(KS)K)I)))))))))$$
$$(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))$$
$$I))))))K))S))K))(S(K(S(K(S(S(K(S(K(S(KS)(S(K(S(K(S(K(S$$
$$(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(KS)))))))))(S(K(S(K(S(K$$
$$(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))$$
$$(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))(S(S(KS)K)I))))))K))S))$$
$$(S(K(S(KS)K))))))))(S(S(KS)K))))K)))))(S(S(K(S(KS)K))$$
$$(SI(K(KI)))))))K)))))(S(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)$$
$$(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(KS)))))))))))))))))))$$
$$(S(S(K(S(K(S(K(S(K(S(KS)K))S))(S(K(S(K(S(K(S(K(S(K(S(K$$
$$(K(S(KS)K))S))K))S))K))S))(S(K(S(KS)K))))))))(S(K(SS(K(K$$
$$(S(K(S(S(KS)K)(S(S(KS)K)I)))(S(S(KS)K)I)))))))))(S(K(S(K$$
$$(S(K(SS(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))$$

K))S))K))(S(K(S(K(S(K(S(K(S(K(S(K(S(S(K(S(K(S(KS)K))
S))K)))K))))(S(S(K(S(K(S(K(S(KS)K))S))K))(SI(K(KI)))))))))))
(S(K(S(K(SS(K(S(KK)K))))(S(K(S(KS)(S(K(S(KS)(S(K(S
(KS)K))))))))))))))(S(S(K(S(K(S(K(S(KS)K))S))(S(K(S(KS)
(S(KS)))))))(S(K(S(K(SS(K(S(K(S(K(S(K(SS(K(SI(K(KI))))))
K))S))K))))(S(K(S(K(S(KS)(S(KS))))(S(K(SS(K(K(KI))))))))))
(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(SI(KK)))))
(SI(K(SI(K(KI)))))(S(S(KS)K)I)))))))(S(K(S(K(S(K(SS(K(SI
(K(KI))))))K))S))K)))))K))S))K))S))K)))(S(K(S(K(S(K(SS(K
(S(K(S(K(SS(K(S(K(S(K(S(K(SS(K(SI(K(KI))))))K))S))K))))
(S(K(S(KS)(S(KS)))))))(S(K(S(K(S(K(S(K(S(K(S(K(SS(K(S(K
(S(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))
(S(K(S(K(S(K(SS(K(SI(K(KI))))))K))S))K)))))K))S))K))S))K))
(SI(K(KI))))))))(S(K(S(K(S(K(S(K(S(KS)K))S))K))S)))))(S(K
(SS(K(K(KI))))))))(S(K(S(K(SS(K(S(K(S(K(SS(KK)))(S(KS))))
(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S
(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))
(SI)))K))))K)))))(S(S(K(S(K(SI(KK)))))(SI(K(SI(K
(KI)))))))))K))))K))S)))))(S(K(S(KS)(S(KS)))))))(S(K(S(K
(S(K(S(K(S(K(SS(K(S(K(S(K(SS(KK)))(S(KS))))(S(K(S(K(SS(K
(S(K(S(KK)(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))
(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K))))(SSK(S(K
(SS(S(SSK))))K)(S(K(S(K(S(S(K(S(KS)(S(SI(K(K(K(KI))))))))K)))
(S(K(S(K(SS(K(SI(K(KI))))))K)))))(S(K(SS(K(S(S(KS)K)))))K))
(KI))))))K))S)))))K))S))(S(KS))))(S(K(SS(K(K(KI))))))))(S(K
(S(K(S(K(SS(K(S(K(S(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI))))))))
K))))K))S))K))))))))(S(K(S(K(S(K(SS(K(S(K(S(K(S(KK)K))K))
(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K)))))))(S(K(S(KS)
(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)(S(KS))))))))))))))))))(S(K(S
(K(S(K(S(K(S(K(SS(K(S(K(S(K(S(K(S(K(S(S(K(S(K(S(KS)K))
S))K)))K))))(S(S(K(S(K(S(K(S(KS)K))S))K))(SI(K(KI))))))))))
(S(K(S(K(S(K(SS(K(S(KK)K))))(S(K(S(KS)(S(K(S(KS)(S(K(S(KS)
K)))))))))))(S(K(S(K(SS(K(S(K(S(K(S(S(K(SSK(S(K(SS(S
(SSK))))K)(S(K(S(K(S(S(K(S(KS)(S(KS))))(S(K(S(K(S(K(S(K(SS
(K(S(K(S(K(SS(K(SI(K(KI))))))K))(S(K(S(K(S(K(S(K(SS(KK)))
K))S))(SI)))K)))))K))S))K))(S(SI(K(K(KI))))(KK))))))(S(K
(S(K(S(K(S(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))
(SI(KK)))))))(S(K(S(K(SS(K(SI(K(KI))))))K))))))))(S(K(SS
(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K
(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))))K))I))
(S(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))(S(K(S(K(SI
(KK)))))(SI(K(SI(K(KI)))))(KI)))I))I))))(SI(K(KI)))))))(S
(KS)))))))(S(K(S(K(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SS(K(S(K(S(K
(S(S(K(SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(S(K(S(KS)(S(KS))))
(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SS(K(SI(K(KI))))))K))(S(K(S(K

$(S(K(S(K(SS(KK)))K))S))(SI)))K)))))K))S))K))(S(SI(K(K(KI))))$
$(KK))))))(S(K(S(K(S(K(S(S(K(S(K(S(K(S(K(S(K(SS(KK)))K))S))$
$(SI)))K))(SI(KK)))))))(S(K(S(K(SS(K(SI(K(KI))))))K))))))))$
$(S(K(SS(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS(KK)))K)))))$
$(S(K(SS(K(S(K(S(K(S(K(S(K(SI))K))))(SI)))K))))K)))))K))I))$
$(S(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))(S(K(S(K(SI$
$(KK)))))(SI(K(SI(K(KI)))))(KI)))I))I))))(SI(K(KI)))))))$
$(S(K(S(KS)(S(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))))$
$(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))))(S(K(S(K(S$
$(K(SS(K(SI(K(KI))))))K))S))K)))))K))S))K))S))K))(SI(K(S(S$
$(KS)K)I)))))))))K))S))(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S(KS)$
$K))S))K))S))K))S))(S(K(S(KS)K)))))))(S(K(SS(K(K(S(S(KS)K)$
$(S(K(S(S(KS)K)(S(S(KS)K)I)))(S(S(KS)K)I)))))))))(S(K(S(K$
$(S(K(SS(K(S(K(S(K(S(K(SI(KK)))))(SI(K(SI(K(KI)))))I))))))$
$K))S))K))))))))))(S(K(S(K(S(KK)K))))(S(K(S(K(SS(K(S(K(S$
$(K(SS(KI)))))(S(K(SS(K(S(K(S(K(S(S(KS)K)))))(S(K(S(K(SI$
$(KK)))))(SI(K(SI(K(KI)))))))))(S(K(S(KS)K)))))))(S(K(S$
$(KS)(S(KS)))))))(S(K(S(K(S(K(S(K(SS(K(SS(K(K(KI)))))))$
$K))S))(S(KS))))(S(K(S(K(S(K(SS(K(S(K(S(K(SI(KK)))))(SI$
$(K(SI(K(KI)))))))))K))S))(S(K(SS(K(S(K(S(K(S(K(S(K(SS$
$(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S$
$(K(S(K(SI))K))))(SI)))K))))K)))(SSK(S(K(SS(S(SSK))))K)$
$(S(K(S(K(S(S(K(S(KS)(S(SI(K(K(K(KI)))))))K)))(S(K(S(K$
$(SS(K(SI(K(KI))))))K)))))(S(K(SS(K(S(S(KS)K)))))K))$
$(KI))))))K)))))))(K(SSK(S(K(SS(S(SSK))))K)(S(K(S(K(S(S(K$
$(S(KS)(S(KS))))(S(K(S(K(S(K(S(K(SS(K(S(K(S(K(SS(K(SI(K$
$(KI))))))K))(S(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K)))))$
$K))S))K))(S(SI(K(K(KI))))(KK))))))(S(K(S(K(S(K(S(S(K(S$
$(K(S(K(S(K(S(K(SS(KK)))K))S))(SI)))K))(SI(KK)))))))(S(K$
$(S(K(SS(K(SI(K(KI))))))K)))))))(S(K(SS(K(S(K(S(K(S(K(SS$
$(K(KI))))))(S(K(S(K(SS(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S$
$(K(SI))K))))(SI)))K))))K)))))K))))(S(S(S(KS)(S(K(S(KS)(S(K$
$(SSK)))))))(S(K(S(K(SS(KK)))K))S))(K(S(K(S(K(S(SI(K(K(KI))))$
$(KK)))))(S(K(S(SI(K(S(K(S(K(S(K(SS(K(KI))))))(S(K(S(K(SS$
$(KK)))K)))))(S(K(SS(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K(S(K$
$(SI)))K))))K))))))K))))$

Appendix C

VHDL Code

This appendix presents the VHDL which specifies the coordination, control and memory modules for the TM and RASP machines. For the sake of brevity, the general form of the programs for each machine are also presented, rather than every program represented in full VHDL.

C.1 RASP

The RASPs all share the same coordination module. This is the VHDL module which ties the control and memory modules together. It defines and routes the buses and signals between the two modules.

The VHDL code has set of variables which are adjusted for each particular instance of a RASP, these variables are related to the number of bits per register for a machine. For the sake of brevity, the coordination module and memory will be displayed once with these variables uninstantiated.

C.1.1 All RASP Coordination

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity RASPMachine is
    Port ( halted : out STD_LOGIC;
      memrw : out std_logic;
      clk : in std_logic;
      controlOut : out std_logic);
end RASPMachine;
```

```
architecture Behavioral of RASPMachine is

  Component RASPControl is
    Port ( clk : in std_logic;
           halted: out std_logic;
           address : out  STD_LOGIC_vector(x downto 0);
           datain : out std_logic_vector(x downto 0);
           dataout : in std_logic_vector(x downto 0);
           wFlag : out std_logic;
           controlOut : out std_logic );
    end component;



  Component RASPmemory
    Port(address : IN  std_logic_vector(x downto 0);
         datain : IN  std_logic_vector(x downto 0);
         dataout : OUT  std_logic_vector(x downto 0);
         wFlag : IN  std_logic;
         clk : IN  std_logic;
         memrw : out std_logic);
   END Component;


  signal address :  std_logic_vector(x downto 0);
  signal datain :  std_logic_vector(x downto 0);
  signal dataout :  std_logic_vector(x downto 0);
  signal wFlag : std_logic;

begin
  control : RASPControl port map (clk, halted, address, datain,
            dataout, wFlag, controlOut);
  memory : RASPmemory port map (address, datain, dataout,
            wFlag, clk, memrw);
end Behavioral;
```

C.1.2 All RASP Memory

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use std.textio.all;

entity RASPMemory is
    Port ( address : in  STD_LOGIC_VECTOR (x downto 0);
           datain : in  STD_LOGIC_VECTOR (x downto 0);
           dataout : out  STD_LOGIC_VECTOR (x downto 0);
           wFlag : in  STD_LOGIC;
           clk : in  STD_LOGIC;
           memrw : out std_logic);
end RASPMemory;

architecture Behavioral of RASPMemory is

  type mem is array (0 to n) of std_logic_vector (x downto 0);
  signal m : mem := ("...",...);

begin
   process (clk)
     begin
       if falling_edge(clk) then
         if wFlag = '1' then
           m(to_integer(unsigned(address))) <= datain;
           memrw <= '0';
         else
           dataout <= m(to_integer(unsigned(address)));
           memrw <= '1';
         end if;
       end if;
   end process;

end Behavioral;
```

C.1.3 RASP Control

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity RASPControl is
    Port ( clk : in std_logic;
           halted : out std_logic;
           address : out  STD_LOGIC_vector(x downto 0);
           datain : out std_logic_vector(x downto 0);
           dataout : in std_logic_vector(x downto 0);
           wFlag : out std_logic;
           controlOut : out std_logic);
end RASPControl;

architecture Behavioral of RASPControl is
  signal incFlag: std_logic := '1';
  signal currentInstr : std_logic_vector (x downto 0) := "";
  signal temp : std_logic_vector(x downto 0) := "";

 begin
 p: process (clk)
    variable counterOuter : unsigned(2 downto 0) := "000";
    variable counterInner : unsigned (2 downto 0) := "000";
    variable addition: unsigned(x downto 0) :=  "";
    begin
       if rising_edge(clk) then
        controlOut <= '0';
        case counterOuter is
          when "000" =>
            wFlag <= '0';
            incFlag <= '1';
            address <= "";
            counterOuter := counterOuter+1;
          when "001" =>
            address <= dataout;
            counterOuter := counterOuter+1;
```

```
when "010" =>
  address <= "";
  datain <= dataout;
  wFlag <= '1' ;
  currentInstr <= dataout;
  counterOuter := counterOuter+1;
when "011" =>
  wFlag <= '0';
  case currentInstr is
    when "000" =>     -- HALT
      halted <= '1';
      incFlag <= '0';
    when "001" =>              -- INC
      case counterInner is
        when "000" =>
          address <= "010";
          counterInner := counterInner +1;
        when "001" =>
          addition := unsigned(dataout);
          addition := addition + 1;
          datain <=  std_logic_vector(addition);
          wFlag <= '1';
          counterInner := counterInner +1;
        when "010" =>
          wFlag <= '0';
          counterInner := "000";
          counterOuter := counterOuter +1;
          halted <= '0';
        when others => null;
      end case;

    when "010" =>     -- DEC
      case counterInner is
        when "000" =>
          address <= "010";
          counterInner := counterInner +1;
        when "001" =>
          addition := unsigned(dataout);
          addition := addition - 1;
          datain <=  std_logic_vector(addition);
          wFlag <= '1';
          counterInner := counterInner +1;
        when "010" =>
          wFlag <= '0';
          halted <= '0';
          counterInner := "000";
          counterOuter := counterOuter +1;
        when others => null;
      end case;
```

```
when "011" =>     -- LOAD
  case counterInner is
   when "000" =>
     address <= "000";
     counterInner := counterInner +1;
   when "001" =>
     addition := unsigned(dataout);
     addition := addition + 1;
     datain <=  std_logic_vector(addition);
     wFlag <= '1';
     counterInner := counterInner +1;
   when "010" =>
     wFlag <= '0';
     address <= "000";
     counterInner := counterInner +1;
   when "011" =>
     address <= dataout;
     counterInner := counterInner +1;
   when "100" =>
     address <= "001";
     datain <= dataout;
     wFlag <= '1' ;
     counterInner := counterInner +1;
   when "101" =>
     address <= "010";
     wFlag <= '1' ;
     counterInner := counterInner +1;
   when "110" =>
     wFlag <= '0';
     counterInner := "000";
     counterOuter := counterOuter +1;
     halted <= '0';
   when others => null;
  end case;
```

```
when "100" =>     -- STO
  case counterInner is
    when "000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "010" =>
      address <= "000";
      wFlag <= '0';
      counterInner := counterInner +1;
    when "011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "100" =>
      address <= "001";
      datain <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
    when "101" =>
      temp <= dataout;
      wFlag <= '0';
      counterInner := counterInner +1;
    when "110" =>
      address <= "010";
      counterInner := counterInner +1;
    when "111" =>
      datain <= dataout;
      address <= temp;
      wFlag <= '1';
      counterInner := "000";
      counterOuter := counterOuter +1;
      halted <= '0';
    when others => null;
    end case;
```

```
when "101" =>     -- OUT
  halted <= '0';
  controlOut <= '1';
  counterOuter := counterOuter +1;

when "110" =>     -- JGZ
  case counterInner is
    when "000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "010" =>
      address <= "000";
      wFlag <= '0';
      counterInner := counterInner +1;
     when "011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "100" =>
      address <= "001";
      datain <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
    when "101" =>
      temp <= dataout;
      wFlag <= '0';
      counterInner := counterInner +1;
    when "110" =>
      address <= "010";
      counterInner := counterInner +1;
    when "111" =>
      if(dataout = "000") then
        null;
      else
        address <= "000";
        datain <= temp;
        wFlag <= '1';
        incFlag <= '0';
      end if;
      counterInner := "000";
      counterOuter := counterOuter + 1;
      halted <= '0';
    when others => null;
  end case;
```

```
when "111" =>     -- CPY
  case counterInner is
    when "000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "010" =>
      address <= "000";
      wFlag <= '0';
      counterInner := counterInner +1;
    when "011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "100" =>
      address <= "001";
      datain <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
    when "101" =>
      address <= dataout;
      wFlag <= '0';
      counterInner := counterInner +1;
    when "110" =>
      address <= "010";
      datain <= dataout;
      wFlag <= '1';
      counterInner := "000";
      counterOuter := counterOuter + 1;
      halted <= '0';
    when others =>
      halted <= '1';
      incFlag <= '0';
   end case;
when others => null;
end case;
```

```
          when "100" =>
            wFlag <= '0';
            if incFlag = '1' then
              address <= "000";
            end if;
            counterOuter := counterOuter +1;
          when "101" =>
            if incFlag = '1' then
              addition := unsigned(dataout);
              addition := addition + 1;
              datain <=  std_logic_vector(addition);
              wFlag <= '1';
            end if;
            counterOuter := "000";
          when others => null;
          end case;
        end if;
        end process;
end Behavioral;
```

C.1.4 RASP2 Control

The control module for the RASP2 is identical to that for the RASP save that the INC and DEC instructions of the RAPS are replaced by the following ADD and SUB instructions.

```
when "001" =>                    -- ADD
  case counterInner is
    when "000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "010" =>
      address <= "000";
      wFlag <= '0';
      counterInner := counterInner +1;
    when "011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "100" =>
      address <= "001";
      datain <= dataout;
      temp <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
      temp <= dataout;
    when "101" =>
      address <= "010";
      counterInner := counterInner +1;
    when "110" =>
      addition := unsigned(dataout);
      addition := addition + unsigned(temp);
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
   when "111" =>
      wFlag <= '0';
      counterInner := "000";
      counterOuter := counterOuter +1;
      halted <= '0';
   when others => null;
   end case;
```

```
when "010" =>     -- SUB
  case counterInner is
    when "000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "010" =>
      address <= "000";
      wFlag <= '0';
      counterInner := counterInner +1;
    when "011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "100" =>
      address <= "001";
      datain <= dataout;
      temp <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
      temp <= dataout;
    when "101" =>
      address <= "010";
      counterInner := counterInner +1;
    when "110" =>
      addition := unsigned(dataout);
      addition := addition - unsigned(temp);
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "111" =>
      wFlag <= '0';
      counterInner := "000";
      counterOuter := counterOuter +1;
      halted <= '0';
    when others => null;
    end case;
```

C.1.5 RASP3 Control

The control module for the RASP2 is identical to that for the RASP save that the INC and DEC instructions of the RAPS are replaced by the following ADD

and SUB instructions.

```
when "001" =>                -- ADD
  case counterInner is
    when "0000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "0001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "0010" =>
      wFlag <= '0';
      address <= "000";
      counterInner := counterInner +1;
    when "0011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "0100" =>
      address <= "001";
      datain <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
    when "0101" =>
      wFlag <= '0';
      address <= dataout;
      counterInner := counterInner +1;
    when "0110" =>
      temp <= dataout;
      address <= "010";
      counterInner := counterInner +1;
    when "0111" =>
      addition := unsigned(dataout);
      addition := addition + unsigned(temp);
      datain <= std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "1000" =>
      wFlag <= '0';
      counterInner := "0000";
      counterOuter := counterOuter +1;
      halted <= '0';
    when others => null;
  end case;
```

```
when "010" =>     -- SUB
  case counterInner is
    when "0000" =>
      address <= "000";
      counterInner := counterInner +1;
    when "0001" =>
      addition := unsigned(dataout);
      addition := addition + 1;
      datain <=  std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "0010" =>
      wFlag <= '0';
      address <= "000";
      counterInner := counterInner +1;
    when "0011" =>
      address <= dataout;
      counterInner := counterInner +1;
    when "0100" =>
      address <= "001";
      datain <= dataout;
      wFlag <= '1' ;
      counterInner := counterInner +1;
    when "0101" =>
      wFlag <= '0';
      address <= dataout;
      counterInner := counterInner +1;
    when "0110" =>
      temp <= dataout;
      address <= "010";
      counterInner := counterInner +1;
    when "0111" =>
      addition := unsigned(dataout);
      addition := addition - unsigned(temp);
      datain <= std_logic_vector(addition);
      wFlag <= '1';
      counterInner := counterInner +1;
    when "1000" =>
      wFlag <= '0';
      counterInner := "0000";
      counterOuter := counterOuter +1;
      halted <= '0';
    when others => null;
  end case;
```

C.1.6 RASP Programs

The initial state of a program in the RASP is represented in the memory module. The line:

```
signal m : mem := ("...", ...);
```

is filled with the entire contents of the RASP memory, including the initial states of the PC, IR, and ACC. Each numeral is an n-bit binary number, where n is the number of bits in the machine.

The programs are converted from the "array form" in Appendix B into binary and arranged after the register states. As an example, consider the RASP2 addition program from Appendix B.2.1:

```
3,5,1,8,0
```

This program converted to the VHDL form is:

```
signal m : mem := ("011","000","000","011","101","001",
                   "000","000");
```

C.2 TM

As with the RASPs, the TM has the same coordination and memory modules for each TM. The variable (x) in this case refers to the number of symbols which are defined for use of on the tape of the machine.

C.2.1 TM Coordination

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity TuringMachine is
    Port ( clk : in  STD_LOGIC;
           acc : out  STD_LOGIC;
           halted : out std_logic);
end TuringMachine;

architecture Behavioral of TuringMachine is

  component TMControl is
     port(
        clk : in  STD_LOGIC;
        symbolOut : in  integer range 0 to x;
        headPos : out  unsigned (0 to 0);
        wFlag : out  STD_LOGIC;
        symbolIn : out  integer range 0 to x;
        halted : out std_logic);
   end component;

   component TMTape is
        Port (
         headPos : in  unsigned (0 to 0);
         symbolIn : in  integer range 0 to x;
         symbolOut : out  integer range 0 to x;
         wFlag : in std_logic;
         acc    : out std_logic;
         clk : in std_logic);
   end component;

   signal symbolIn : integer range 0 to x := 0;
        signal symbolOut: integer range 0 to x := 0;
        signal wFlag : std_logic := '0';
        signal headPos : unsigned (0 to 0);



begin
   control : TMControl port map (clk, symbolOut, headPos,
              wFlag, symbolIn, halted);
   tape : TMTape port map (headPos, symbolIn, symbolOut,
              wFlag, acc, clk);

end Behavioral;
```

C.2.2 TM Memory

The TM memory is a tape which contains a single symbol. The tape can accept up to x symbols which are represented as integers.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity TMTape is
    Port ( headPos : in  unsigned (0 to 0);
           symbolIn : in  integer range 0 to x;
           symbolOut : out  integer range 0 to x;
           wFlag : in std_logic;
           acc    : out std_logic;
           clk : in std_logic);
end TMTape;

architecture Behavioral of TMTape is
        type mem is array(0 to 1) of integer range 0 to x;
        signal tape : mem := (0,1);

begin
  process (clk)
    begin
           if falling_edge(clk) then
              if wFlag = '1' then
                    tape(to_integer(headPos)) <= symbolIn;
                         acc <= '1';
         else
                   symbolOut <= tape(to_integer(headPos));
                         acc <= '0';
                      end if;
           end if;
  end process;
end Behavioral;
```

C.2.3 TM Control

The TM control houses the semantics of the TM and the symbol table. The variable x is again the number of symbols required for the machine to function, and the new variable n is number of bits required to represent the maximum number of states of the machine. The variable t dictates the number of tuples in the symbol table.

The symbol table *st* is what holds the specific symbol table of each TM. In this example, the symbol table is left unfilled.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;

entity TMControl is
    Port ( clk : in  STD_LOGIC;
      symbolIn : out  integer range 0 to x;
      headPos : out  unsigned (0 to 0);
      wFlag : out  STD_LOGIC;
      symbolOut : in  integer range 0 to x;
      halted :out std_logic);
end TMControl;

architecture Behavioral of TMControl is
    type tuple is record
      stateR : unsigned (n downto 0);
      symbolR : integer range 0 to x;
      stateW : unsigned (n downto 0);
      symbolW : integer range 0 to x;
      dir : std_logic;
    end record;

    signal currentState : unsigned (n downto 0) := "01";
    signal counter : unsigned (2 downto 0) := "000";
    signal hPos : unsigned (0 to 0) := "1";

    type st is array (0 to t) of tuple;
    constant symbolTable : st :=(...,...);
```

```
begin

  process (clk)
    variable found : std_logic := '0';
    variable var : integer range 0 to t := 0;
    begin
      if rising_edge(clk) then
       case counter is
         when "000" =>
           found := '0';
           headPos <= hPos;
           counter <= counter + 1;

         when "001" =>
           if currentState = "00" then
             halted <= '1';
           else
             halted <= '0';
             counter <= counter + 1;
           end if;

          when "010" =>
            for i in symbolTable'RANGE loop
              if symbolTable(i).stateR = currentState
               and symbolTable(i).symbolR = symbolOut then
                found := '1';
                var := i;
                exit;
              end if;
            end loop;
            counter <= counter +1;

           when "011" =>
             if found = '1' then
               headPos <= hPos;
               wFlag <= '1';
               symbolIn <= symbolTable(var).symbolW;
               currentState <= symbolTable(var).stateW;
               counter <= counter +1;
             else
               counter <= "001";
               currentState <= "00";
             end if;
```

```
                when "100" =>
                  wFlag <= '0';
                  if(symbolTable(var).dir = '1') then
                    hPos <= hPos + 1;
                  else
                    hPos <= hPos - 1;
                  end if;
                  counter <= "000";
                 when others =>
                 end case;

            end if;
    end process;
end Behavioral;
```

C.2.4 TM Programs

Programs in the TM are specified in the control module as ROM. The line:

```
constant symbolTable : st :=(...,...);
```

in the control module specifies the symbol table in the "tuple" record which is defined just above this line.

The format of the symbol table itself follows the previous conventions of the thesis laid out in Sections 2.3.1.1 and 3.4.1. There is a straightforward encoding of the TM tuples into the VHDL form. Consider the addition TM in Appendix B.4.1:

1,1,2,0,R
2,1,2,1,R
2,0,0,1,L

This TM is converted to the following VHDL form:

```
constant symbolTable : st := (
("01",1,"10",0,'1'),
("10",0,"00",1,'0'),
("10",1,"10",1,'1'));
```

Each symbol is an integer from 0 to x. The states are converted into binary notation with n bits, where n is the number of bits required to represent the largest state of the TM. Appendix B shows the full tuples for the TM which are converted using the above method and mapped to an FPGA circuit to produce the measurements of this thesis.

Appendix D

Full Semantics

This appendix lists the Reverse Polish Notation [56] expressions of the semantics of the computational models which are featured in this thesis (Section 3.3). These expressions are measured and their sizes shows in Table 4.2 et al. and are analysed in Chapter 6.

The notation used here is obtuse. When reading this Appendix, it is recommended that the reader concurrently follow the appropriate sub-section in Section 3.4. Each RPN rule shown here corresponds to a semantic rule expressed in infix notation in Section 3.4.

D.1 RASPs

D.1.1 RASP Model

S , Y , T , J : ℕℕ ↦
X , F , L : ℕ
G : { 0 . . 2 n ^ 1 − }
IG⊂
: Sℕ ↦
A : SX × SX × ↦
P : G{ , } ∪+ ℕ × S ↦
E : SX × SX × ↦
SZ : 0 S1 + S# %
J{03 ↦, 10 ↦, 20 ↦} =
FØ=
e , k : G{ , } ∪+
⟨S , X⟩ Je2P ∪ FE=

eg , k ⟹
gG∈
enP{ne↦}kn1 + P∪=

enPØ ⟹

0 SSI∈
0YSZ=
1Y0SS=
⟨T , L⟩YXA=
SXETLE ⟹

0 SSI∉
1Y0SS=
SXE⟨Y , X⟩ ⟹

D.1.2 RASP Language

1S1=
2Y2S1+S#%=
SXA⟨Y,X⟩ ⟹

1S2=
Y2S1−S#%=
SXA⟨Y,X⟩ ⟹

1S3=
1Y2Y0SS==
0YSZ=
SXA⟨Y,X⟩ ⟹

1S4=
1Y0SS=
1Y1>
0SSY2S=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S4=
1Y0SS0==
0Y2S=
1T0=
0TYZ=
SXA⟨T,X⟩ ⟹

1S4=
0SS1=
1Y2S=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S5=
1Y=0SS
2S0=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S5=
1Y0Y0SS==
2S0>
SXA⟨Y,X⟩ ⟹

1S6=
FX{2S}∪=
SXA⟨Y,F⟩ ⟹

1S7=
1Y0SS=
2Y1YS=
0YSZ=
SXA⟨Y,X⟩ ⟹

D.1.3 RASP2 Language

1S1=
1Y0SS=
2Y2S1Y+S#%=
0YSP=
SXA⟨Y,X⟩ ⟹

1S2=
1Y0SS=
2Y2S1Y−S#%=
0YSP=
SXA⟨Y,X⟩ ⟹

1S3=
1Y2Y0SS==
0YSZ=
SXA⟨Y,X⟩ ⟹

1S4=
1Y0SS=
1Y1>
0SSY2S=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S4=
1Y0SS0==
0Y2S=
1T0=
0TYZ=
SXA⟨T,X⟩ ⟹

1S4=
0SS1=
1Y2S=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S5=
1Y=0SS
2S0=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S5=
1Y0Y0SS==
2S0>
SXA⟨Y,X⟩ ⟹

1S6=
FX{2S}∪=
SXA⟨Y,F⟩ ⟹

1S7=
1Y0SS=
2Y1YS=
0YSZ=
SXA⟨Y,X⟩ ⟹

D.1.4 RASP3 Language

S1=
1Y0SS=
2Y2S1YS+S#%=
0YSP=
SXA⟨Y,X⟩ ⟹

1S2=
1Y0SS=
2Y2S1YS−S#%=
0YSP=
SXA⟨Y,X⟩ ⟹

1S3=
1Y2Y0SS==
0YSZ=
SXA⟨Y,X⟩ ⟹

1S4=
1Y0SS=
1Y1>
0SSY2S=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S4=
1Y0SS0==
0Y2S=
1T0=
0TYZ=
SXA⟨T,X⟩ ⟹

1S4=
0SS1=
1Y2S=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S5=
1Y=0SS
2S0=
0YSZ=
SXA⟨Y,X⟩ ⟹

1S5=
1Y0Y0SS==
2S0>
SXA⟨Y,X⟩ ⟹

1S6=
FX{2S}∪=
SXA⟨Y,F⟩ ⟹

1S7=
1Y0SS=
2Y1YS=
0YSZ=
SXA⟨Y,X⟩ ⟹

D.2 TM

```
s , r :Q
y : Γ                                          fgm ⟹
h : ℤ                                          gΓ ∈
d : { L , R }                                  fnU { ng↦ } mn1+U∪ ⟹
T , J , X : ℤΓ ↦
δ : QΓ × QΓ × d × ↦                            fnU∅ ⟹
P : eδ ↦
U , I : Zf × T ↦                               fnI∅ ⟹
e , a : ΓQ ∪ d ∪ { , } ∪+
f , k , m : Γ+ { ^ } ∪                         fmg ⟹
E : Q ( ℤΓ ↦ ) × ℤ × ( ℤΓ ↦ ) ↦                 gΓ ∈
δeP=                                           fnI { ng↦ } mn1−U∪=
J1fU0E=

                                               shTδ⟨r , hX , L⟩=
es , y , r , v , d  a ⟹                        sThEfXh1−E ⟹
eP { ⟨s , y \ rangle ⟨r , v , d \ rangle ↦ } aP∪ ⟹
                                               shTδ⟨r , hX , R⟩=
eP∅ ⟹                                          sThErXh1+E ⟹

fk ^ gm ⟹                                      shTδ⟨r , hX , d⟩ ≠
gΓ ∈                                           sThET ⟹
f0Uk−1I { 0g↦ } ∪m1U∪=
```

D.3 λ-Calculus

F , T , J , R , G , L = { z , L , R }
e , f , g ∈ ({ λ , { a . . z }+ , (,) , . , , } V ∪)+
v , m V ∈
V : { a . . z }+
z = B | A | v
P : e T ↦
E : T T ↦
B : T V ↦
Z : T V × V × T ↦
S : T T × v × T ↦
J e P =
F J E =

e ⟹ λ v . f
e P { B , v P , f P } ⟹

e ⟹ f v
e P { A , f P , v P } ⟹

e ⟹ f (g)
e P { A , f P , g P } ⟹

e ⟹ v
e P { v , ∅ , ∅ } ⟹

e ⟹ (f)
e P f P ⟹

T . z A =
T . L . z B =
T . R . z T . L B ∉
T E T . L . R T . R T . L . L . z S ⟹ ; J E

T . z A =
T . L . z B =
H T . L . R B =
T . R . z H ∈
m H ∉
T E T . L . R m T . R . z Z T . R T . L . L . z S ⟹ ; J E

T E { T . z , T . L E , T . R E } ⟹

T ∅ =
T E ∅ ⟹

T G j S { T . z , T . L G j S , T . L G j S } ⟹

T . z j =
T G j S G ⟹

T ∅ =
T G j S ∅ ⟹

T . z B =
T . L . z j =
T G j S T ⟹

T . z B =
T B { T . L . z } T . R B ∪ ⟹

T . z A =
T B ⟹ T . L B T . R B ∪

T B ∅ ⟹

T . z v =
T m k Z { m , ∅ , ∅ } ⟹

T ∅ =
T m k Z ∅ ⟹

D.4 SKI

```
e , f , g : ( Z { ( , ) } ∪ {A} / )+
E : TT↦                          T . zA=
P : eT↦                          TE{A, LE , RE} ⟹
F , T , L , R, J : { z , L , R}
z∈Z                              T . zA=
Z : { S , K, I , A}              T . L . zI=
JeP=                             TE ⟹ T . R ; JE
FJR=
                                 T . zA=
e ⟹ ( f )                        T . L . L . zK=
ePfP ⟹                           TE ⟹ T . L . R ; JE

e ⟹ f ( g )                      T . zA=
eP {A, fP , gP} ⟹                T . L . L . L . zS=
                                 hT . R=
e ⟹ fz                           TE ⟹ {A, { A, T . L . L . R, h} , {A, T . L . R, h} } ; JE
eP {A, fP , zP} ⟹
                                 T∅=
e ⟹ z                            TET ⟹
eP {z , ∅ , ∅} ⟹
```