Computability -1 (2018) 1–28
DOI 10.3233/COM-180090
IOS Press

Expressiveness, meanings and machines

Joe Davidson *
GoodAI, Prague, Czech Republic
joseph.davidson@goodai.com

Greg Michaelson
Heriot-Watt University, MACS, Edinburgh, Scotland
G.Michaelson@hw.ac.uk
http://www.macs.hw.ac.uk/~greg

Abstract. There is considerable interest in constructing succinct programs and much debate over which languages are most expressive. However, such debates tend to revolve around syntactic notions of program size and language constructs, without considering the complex interrelations of syntax, semantics, and implementation.

We hypothesise that languages with larger semantics have more succinct programs. To explore this, we have conducted an empirical study of the expressive properties of Turing and Random Access Stored Program machines, taking into account their Structural Operational Semantics, universal machines, implementations in each other, and physical realisations as Field Programmable Gate Arrays (FPGAs). We conclude that our intuitions about the relationship between the expressiveness of a language and the typical succinctness of programs are largely correct. We also conclude that the information content of abstract models of computations is a good indicator of the component count for FPGA realisations.

Keywords: Elegance, computability, expressiveness, semantics, FPGA, primitive recursion, turing machine, RASP machine universality

1. Introduction

There is extensive folklore surrounding and connecting efficiency, succinctness and expressiveness. In particular, it is widely held that low level programs are more efficient and less succinct than high level programs because: all programs run ultimately as machine code on hardware platforms; higher level languages are more expressive than lower level languages; machine code is maximally inexpressive; and automatically compiling from a more expressive higher level to a less expressive lower level introduces extraneous artefacts ("code bloat") which can reduce the succinctness compared with hand crafted lower level code.

This paper empirically investigates the apparent link between the expressiveness of a computational model and the succinctness of a program written in the language of that model. This link is articulated by Felleisen as the "Conciseness Conjecture" [8]. We pose six hypotheses based on this conjecture (Section 1.3), define the semantics of four models of computation (Section 2), and determine the program sizes of six arithmetical, five list processing, and two universal functions (Section 3.3). We then resolve the hypotheses by comparing the sizes of the programs relative to the sizes of the semantics.

The models that we make use of are the Turing machine (TM, Section 2.1) and the Random Access Stored Program machine (RASP, Section 2.2). We define metrics based on Kolmogorov/Chaitin information theory [3,14], and use these to attempt to quantify the expressiveness of a computational model relative to the information in its semantics.

We formalise the Turing and RASP machines in Structural Operational Semantics (SOS). These semantics include rules to parse and execute textual representations of the TM and RASP. We also investigate how a small semantic change to a model affects the size of programs written for that model. We make further minor semantic changes to the RASP model to produce two new models, RASP2 and RASP3, which are thought to be more expressive than the RASP.

* Joe Davidson performed the work while attending Heriot-Watt university.

To make an equitable comparison between models we have to choose some common meta-language with which to characterise languages. However, as Stoy notes [21] (p. 296), a problem in language definition based on operational semantics is that of the defined language inheriting characteristics of the meta-language.

Consider a defined language with nested expressions. Then the definition of expression evaluation must account for the evaluation of sub-expressions. Typically this is defined compositionally by a meta-linguistic rule where the evaluation of distinct sub-expression is identified as separate rule invocations. These invocations must be performed in some order, so the defined language evaluation order may be established implicitly by the meta-language evaluation order. For example, if the meta-language is normal order (leftmost outermost) rather than applicative order (leftmost innermost) it may be easier to establish termination but harder to prove sequential properties of programs in the defined language.

To confront how our choice of meta-language influences the constructs of our models, we ground the models in physical realisations of universal machines as Field Programmable Gate Arrays (FPGAs, Section 4). The size measurements of these circuits offer another perspective on the Conciseness Conjecture, independent of our SOS implementations.

We investigate a small space within these possibilities, to try to find empirical measures of how choices of language and meta-language affects the conciseness of programs, as follows. First, we define the semantics of Turing machines and RASP machines using Structural Operational Semantics, a variant of ZF set theory. Next, we write meta-interpreters for TMs as a TM, and for RASP programs as a RASP program. We then write interpreters for TMs as a RASP program, and for RASP programs as a TM. Finally, we write interpreters for TMs and RASP machines in VHDL, for directly configuring Field Programmable Gate Arrays.

As we discuss in the following sections, exploring the normalised sizes of these different approaches gives interesting insights into both conciseness and expressive power. However, we do not formally prove the equivalences of all of these different ways of realising TMs and RASP machines.

Our preliminary results in comparing computational models via our information metrics are as follows. First of all, we have found that the Turing Machine requires less information, on average, than the RASP machines for the functions from the arithmetic hierarchy. However, the RASP machines require less information than the TM for the list processing functions. Furthermore, these comparisons imply that there is a link between the relative average sizes of programs written for each model, and the size of the semantics for those models. This trend follows such that larger semantics beget smaller programs, which is as predicted by the Conciseness Conjecture. Finally, for our test set, there is a correlation between the sizes of a model's semantics and programs, and the number of components required to implement that model on the FPGA.

1.1. Expressiveness, elegance and succinctness

Intuitively, expressiveness is to do with how easy it is to denote some computation in a language, with the sense that there should be some way to quantify or measure this from program text or programming effort. Naively, if language X is more expressive than language Y then one might expect that a program in X is smaller, less complex, or takes less time to construct than the equivalent program in Y.

1.1.1. Empirical approaches

There have been many attempts to capture this notion of expressiveness. For example, Berkholz [2] tries to compare how often and how much programs in different languages are changed. The assumption is that the number of bugs in a program is related to the number of source lines, so programs in more expressive languages will be relatively smaller and so will have fewer bugs. Berkholz analyses 500,000 open source projects written over 20 years in around 100 languages [1], counting, for each program, how many lines have been changed each time a new version of program is committed.

We think that this approach is highly unsatisfactory. Lines of code is a very poor measure of program succinctness, being strongly dependent on layout and lexical choices. Furthermore, code is considered en masse, with no control for what problems are tackled or programmer skill.

Berkholz cites the Halstead's Complexity Measure [9], which is based on proportions of the numbers of distinct operators and operands, and the total numbers of operators and operands. In turn, these are used to derive measures of the length and volume of programs, and of the difficulty and effort in constructing them. Halstead's measure feels language independent and seems to capture the notion that more expressive languages will have richer, more abstract operators, leading to smaller programs for the same computation. It also avoids layout and lexical discrepancies.

However, as with all measures that focus on program text alone, no account is taken of language semantics or of alternative ways of expressing the same computation using different operator and operand combinations. Finally, there is no correspondence with computational complexity. For example, an $O(N)$ linear search may be less complex on this measure than an $O(\log N)$ binary search.

As we shall see in Section 1.1.3, Chaitin also uses a textual measure of program size, but in an information theoretic manner which considers only the most succinct program in any language for any particular function.

1.1.2. Formalisations of expressiveness

Within formal language theory, the Chomsky hierarchy distinguishes formal grammars, and hence languages, of types 3, 2, 1 and 0 with corresponding automata: finite state machines, push down automata, linear bounded automata and Turing machines, respectively [11]. Here, a grammar class of type N includes all the grammars of type $N + 1$. This is demonstrated by constructing a grammar of type N for which there is no type $N+1$ equivalent. For example, it is possible to construct a type 2 grammar, but not a type 3, that accepts only matched pairs of symbols. Equivalently, the corresponding machine for Type N can perform at least all the computations realisable on a machine for Type $N + 1$.

Similarly, recursive function theory [16] distinguishes general recursive functions, which can describe unbounded recursion, from primitive recursive functions, which describe bounded recursion. As Ackerman demonstrated, there are recursive functions which are not primitive recursive.

Thus, for a notion of expressiveness concerned with what can or cannot be expressed, rather than how large the expression is, Type N languages are more expressive than type $N + 1$ languages, and general recursion is more expressive than primitive recursion. Alternatively, we might view a Type N machine as having greater computational power than a Type $N + 1$ machine.

Felleisen [8] provides a comprehensive account of the application of this notion of expressiveness to programming languages, building on work by Kleene [13] and Troelstra [22]. Suppose two languages have a common core but the first contains additional constructs. Given a program in the first language, if the additional constructs can be eliminated by the substitution of constructs from the second language without affecting operational behaviour, then the second language can express the same meanings as the first. The first language is termed a conservative extension of the second and is more expressive in the sense of enabling smaller expressions of the same meanings.

Note that Felleisen's formalisation of expressiveness is orthogonal to notions of computational power. While an extension can increase the computational power of a language, an extension which makes a language more expressive need not increase its computational power, as for example when a Turing complete language is extended.

Felleisen goes on to explore the case where the first language can express more meanings than the second, whose extension may disrupt core semantic properties. He then deploys what is termed a common language universe to explore the expressiveness of different conservative extensions of a core language.

Felleisen's work captures a strong notion of expressiveness but does not consider program succinctness. One approach might be to count elimination steps in transforming a program between languages across a common language universe. However, this seems beset by substantial challenges of soundness and termination, and will not be considered further here.

1.1.3. Kolmogorov–Chaitin complexity and elegance

Kolmogorov–Chaitin complexity is a measure of the regularity or randomness of a string. For a string s of length $|s|$, and some language L, the minimal description function $d_L(s)$ returns the shortest possible program in L to generate s. The complexity function $K_L(s) = |d_L(s)|$ returns the size in characters of the shortest possible program which outputs the string s [14].

The function K allows us to more concretely define what randomness is. Given a string s, if $K_L(s) \geqslant |s|$ then s is termed *incompressible* (i.e. cannot be generated by a shorter program) or *algorithmically random* if the string s is infinite [20].

Chaitin has extensively explored information theoretic measures of program size or 'elegance' [3]:

Definition 1 (Elegance)**.** Given a language L and a function F, an *elegant program* P, for F from L, is one which computes F and there is no shorter program which computes F in L^*, the power set of language L. Here, the Kleene star $*$ [11] constructs the set of all programs that can belong to L.

That is, P is elegant because it is the shortest program which can be written in L to solve F. We would like to establish P's elegance from some meta-linguistic function d such that, for all y in the domain of F, d returns the shortest program which computes the corresponding x: $P = d(x|y)$.

Chaitin proved that elegance is undecidable, by giving an upper bound on the size of programs which can be determined to be elegant [3]. That is, the function d above cannot be expressed in a Turing complete meta-language.

Elegance may also be shown to be undecidable from the general undecidability of program equivalence [19]: establishing that one program is more elegant than another involves establishing that they both compute the same characterising function. Finally, the undecidability of elegance may be derived from Blum's Axioms [5].

This result markedly constrains what we can reasonably ask about programs. However, Chaitin's elegance is a property of programs in some specific language, so elegance cannot be directly compared across languages. Moreover, elegance does not take into account expressiveness i.e. how the language semantics affects how easily arbitrary algorithms can be realised.

1.1.4. Succinctness

Rather than seeking elegance, we base our study on programs we consider to be *succinct*, a less stringent property. We think that succinctness is to do with the balance between program and semantics. For a given semantics, we aim to explore small instances of programs but we wish to avoid obfuscation. That is, our programs capturing common algorithms should correspond to the standard or natural way of expressing them, rather than depending on cunning encodings or obscure features of the semantics.

1.2. The Conciseness Conjecture, size and information

The intuitive notion that more abstract languages tend to have shorter programs than those languages which are more concrete, but have larger semantics, is captured by Felleisen's *Conciseness Conjecture* [8]:

Definition 2 (The Conciseness Conjecture)**.** Programs in more expressive programming languages that use the additional facilities in a sensible manner contain fewer programming patterns than equivalent programs in less expressive languages.

Here a programming pattern is some algorithmic skeleton which is necessarily and repeatedly employed. An example from later in this paper is addition using increment, decrement, and jump (Section 3.2). Adding any two numbers in a model with these instructions requires a loop over both variables, incrementing one and decrementing the other until the second is zero.

This pattern could possibly occur many times in a program, as many times as there are additions, so to make the model more expressive we may define an add function. Changing the program to make use of the additional facilities provided by add will eliminate the occurrences of the now redundant pattern.

To gain more precision in the notion of "facilities", we use the notion of a *model* of computability, with *symbols*, *abstract syntax*, and *semantics*. This is akin to a a formal language with a vocabulary, rules of formation and rules of interpretation.[1] The symbols and abstract syntax enable the construction of *programs* and the semantics provide rules for *executing* programs.

[1] Note that a model **of** computability is not the same as a model **for** a formal language, which is a domain of interpretation.

Thus, modifying the facilities (or instructions) available in a model requires some change to the model's semantics, so adding or removing instructions would thereby increase or decrease the size of the semantics.

We now need some standard way to measure the size of programs and semantics. In information theory, the size of a program is measured by the number of characters required to write the program out [3,14]. For semantics, if the model is characterised in some reference language, then we could measure the size in terms of reference language instructions. However if we are to compare two models with very little overlap in their instruction sets and methods of operation, this approach would be challenging. Alternatively, we could view the model as a program in the reference language. Thus we take the information theoretic approach of counting characters for both programs and semantics. We discuss this in more detail below.

We define the related notions of *Semantic Information*, *Program Information*, and *Total Information* as the number of characters required to represent the semantics of a model, a program, and the sum of the semantic information and program information respectively:

Definition 3 (Semantic Information)**.** The *Semantic Information* (SI) for a model of computation is the size of the semantics of that model in characters.

Definition 4 (Program Information)**.** The *Program Information* (PI) is the size of a program in characters.

Definition 5 (Total Information)**.** The *Total Information* (TI) is SI + PI.

If P is some program written in the language defined by model M, the PI of P is the amount of information that is needed to represent P's combination of M's instructions. The SI of M is the information needed to explain how the instructions of M are executed. So assuming knowledge of the semantic formalism representing the semantics of M, given M, P, and some input to P, P can be run on the input using the semantics of M to achieve the desired result.

Given the definitions above, we can rank various models and programs according to their sizes. If a model A has less PI/SI/TI than model B, then A requires fewer characters than B to represent its programs or semantics, and vice versa. We also say that some program written in some model *requires* a certain amount of information, as there is some necessary number of characters which represent both the program and the semantics of the model.

Of course, when counting characters, one can trivially inflate sizes by adding white space, and redundant operators like brackets, to programs and semantics, If we are to make comparisons with respect to size, we would wish to compare the smallest possible lexically distinguishable representations.

1.3. Hypotheses

Given the Conciseness Conjecture above, we would like to test if larger semantics indeed beget smaller programs. To do so, we venture three hypotheses for Turing Complete models, and their equivalents for Field Programmable Gate Arrays.

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

Here, the term "more semantic information" refers to the number of characters in the semantics. In the above case, model A will require more characters to describe its semantics than model B. The SI Hypothesis is a more formal wording of the Conciseness Conjecture making use of the definition for semantic information.

We wish to compare the average size of programs, ideally covering the whole set of computable functions, but because we cannot perform an empirical study over this set, we have chosen a number of functions such that any "real" program should incorporate at least one of them.

This set consists of six arithmetical functions: Addition, Subtraction, Equality, Multiplication, Division, and Exponentiation, and five list processing functions: Membership, Linear Search, Reversal via creating a new list,

Reversal through in-place substitution, and Bubble Sort. We also measure the sizes of the universal Turing Machine, and the Universal RASP machine and include them in the comparisons.

We informally state that the arithmetic functions are 'simpler' than the list or universal functions. We think this distinction is justified because list operations require significantly more complex memory management operations than the nested incrementation and decrementation in the arithmetic functions.

This brings us to the Total Information Hypotheses:

Hypothesis 2 (Total Information; Arithmetic)**.** For two Turing Complete models; if model A has more semantic information than model B, then when computing the arithmetic functions, model B will require *less* total information than model A.

Hypothesis 3 (Total Information; List)**.** For two Turing Complete models; if model A has more semantic information than model B, then when computing the list functions, model B will require *more* total information than model A.

These hypotheses are concerned with how much of the semantic information is useful in expressing arithmetic and list functions. In the case of the models used here, we expect to find that, while the TM has smaller semantics than the RASP, it needs less total information on average to represent its semantics and the arithmetic functions. In contrast, the RASP is expected to not need all of its richer facilities in calculating the arithmetic functions, so the extra semantic information is surplus to requirements.

However, the RASP's richer semantic information should be useful for the list functions. Here we expect to see that the TI for the RASP is lower than the TI for the TM, as the RASP semantics (supporting random memory access) better addresses the complexity of memory management inherent in list processing. In contrast, the semantics of the TM do not consider data structures other than the sequential tape, so memory manipulation must be crafted explicitly for list processing.

We also define hypotheses for the FPGA implementations in a similar manner with similar justifications for the hypothesised results:

Hypothesis 4 (Semantic Circuit Size)**.** For two Turing Complete models; if model A requires a larger circuit to express its semantics than model B, then model A will need on average smaller circuits to implement functions.

Hypothesis 5 (Total Circuit Size; Arithmetic)**.** For two Turing Complete models; if model A requires a larger circuit to express its semantics than model B, then when computing the arithmetic functions, model B will require a *smaller* circuit than model A to represent both the semantics and the programs.

Hypothesis 6 (Total Circuit Size; List)**.** For two Turing Complete models; if model A requires a larger circuit to express its semantics than model B, then when computing the list functions, model B will require a *larger* circuit than model A to represent both the semantics and the programs.

We formulate the FPGA hypotheses like this in order to link the notions of circuit size via number of components with our metrics of SI/PI/TI via number of characters. As for sizes of circuits, we define a metric analogous to the character counting above where we count electronic components instead. Because of the overlapping nature of these metrics, if we obtain similar results from these two separate approaches, we will have some assurance that our methodology is correct.

1.4. Method

Given the hypotheses above, we present the method. We define the semantics of the models using structured operational semantics, and measure their sizes in characters. We have what we consider to be a representative sample of functions drawn from arithmetic and list processing functions. These are implemented and also measured in characters. For the FPGA hypotheses, we implement each model in VHDL and compile them to FPGA circuitry, taking note of the number of components required to realise the circuit.

With the data, we compare the models to each other in pairs discussing each hypothesis in turn, highlighting particular datapoints of interest and gathering evidence which can either confirm or refute the hypotheses. We also calculate the correlation between the character and component metrics to see how well they relate to one another.

2. Models and semantics

2.1. Turing machines

The Turing machine [23] is a seminal model of computation. All TMs have the same core properties, but individual models may be different according to the needs of the user The TM used here consists of a single unbounded tape which the machine works on, a single read/write head, and a symbol table which is described as a set of 5-tuples.

The Turing machine 5-tuples are of the form $\langle st, sy, st', sy', dir\rangle$ which inform the state machine in state st, when reading a symbol sy, about which symbol sy' to write, which state st' to transist to, and in which direction dir to move on the tape.

The initial and halting conventions for the machines in this paper are that the machines all start in state 1, and a transition to state 0 signifies a halt. The machine will also halt if there are no valid transitions for a particular state and symbol pair.

2.2. RASP machines

The Random Access Stored Program (RASP) machine, originally by Elgot and Robinson [7], is a computational model similar to that of a Random Access Memory machine [10] with the distinction that there is no separation between the instructions of the program and the data which the program is acting on. This lack of separation means that there is a single block of memory available to the machine and that the program can potentially overwrite itself while executing.

Though the concept was introduced by Elgot and Robinson, the RASP machine and instructions that we use are closer to Cook and Reckhow's [4], but the one presented here is a finite rather than an infinite machine. A finite RASP machine is expressed in n-bits and is a sequence of 2^n registers, each addressed by a natural number in the interval $[0, 2^n - 1]$. The registers themselves contain a single natural number in the same interval.

Registers, 0, 1 and 2 have specific functions which are required by the internal state machine. Register 0 is the Program Counter (PC) which contains the address of the current instruction or piece of data that is under consideration. 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) upon which all of the arithmetic instructions operate. There are 8 instructions in the RASP machine with each instruction matched to a natural number. Table 1 shows the effects of each instruction on a RASP machine M, where $M[y]$ is the value stored in address y of the machine.

Note that some of the instructions require a parameter x which is assumed to be held in the next register in sequence. 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, so 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 interval [0, 7], the machine will halt.

RASP machines operate via a fetch-execute cycle. As an example, if a machine were to execute the LOAD instruction it would first copy the LOAD instruction from the memory address pointed to by the PC into the IR.

Table 1
Effects of each instruction on a RASP machine M

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

Decoding the LOAD would prompt an increment of the PC and a further fetch of the parameter into the IR. Once this had been done, the LOAD command would be fully executed by setting the ACC to the value which was currently held by the IR. The machine would then increment the PC and continue on to the next instruction.

The RASP is notable for its ability to modify a currently executing program. While the program for a Turing machine is fixed throughout the lifetime of its execution, the program of a RASP can change through modifying a register and then executing its contents. Combining this with the inherent functions defined in the semantics, it is suspected that the RASP requires more succinct programs than an extensionally equivalent TM. Section 3 gives examples of machines for computing the arithmetic functions.

2.2.1. RASP2 and RASP3

While the RASP is Turing complete, recursive functions like addition and subtraction are laborious to define at each point where they occur. One way of handling this in a large program is to encapsulate the desired behaviour in a function, calling it when required 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. Another way is to extend the semantics of the RASP to create a new model.

We can iterate on the RASP, by replacing INC and DEC with ADD x and SUB x, in two different ways, which we term RASP2 and RASP3. Table 2 states the effects of the new instructions. RASP2 will use ADD x and SUB x where x is a value, such that ADD x will add the constant value x to the accumulator and SUB x will subtract it. RASP3 will also use ADD x and SUB x, but the x is a memory address where the required value is held. Thus, the RASP3 ADD is to CPY, as the RASP2 ADD is to LOAD.

2.3. Semantics

2.3.1. Introduction

We implement the semantics of our models using Structured Operational Semantics [17]. We imagine an idealised "SOS machine" which can interpret and run any well-formed program. This machine acts as our baseline from where we take measurements of the semantic information, and then the program information.

Typically for a computational model, the semantics only describes the execution of the commands or operations. Here, we must also consider how the external representations of the programs and data (which we measure as 'program information') map to the internal representation which is evaluated by the semantics.

Thus, the semantics cover two aspects of the model; parsing of programs and data, and execution of programs on data. Because the programs are presented to the model in some external representation, there must be some semantic mechanism which defines the grammar to which valid programs must adhere. If this is not done, one could create arbitrarily complex grammars to compress program meaning without having the complexity accounted for.

To facilitate this, the semantics have been separated into what we term model and language semantics. The model semantics describe both the parser and the basic mechanical functions of the model, such as the RASP fetch-execute cycle which stays constant irrespective of the actual instruction which is executed.

The language semantics are concerned with the functionality of specific instructions which are not covered by the model semantics. The RASP machines have rules in their language semantics for each of their instructions. The TM requires no language semantics, but if one were to reserve some functionality for a symbol (say a wildcard), the language semantics would define this.

Table 2
Effects of the new ADD and SUB instructions on a RASP2/3 machine M

Integer	Command	Effect RASP2	Effect RASP3
1	ADD x	$M[2] \leftarrow M[2] + x$	$M[2] \leftarrow M[2] + M[x]$
2	SUB x	$M[2] \leftarrow M[2] - x$	$M[2] \leftarrow M[2] - M[x]$

The semantic rules for the models are represented as:

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

where the conclusions are satisfied if and only if the premises are. We combine the preconditions and postconditions of each semantic rule in the premises and show the state transitions as the conclusions. This makes it more natural to say: "If the pre- and postconditions hold, then this state transition must have happened".

For the comparisons in this paper, the semantics for the TM, the RASP, the RASP2, and the RASP3 are written in SOS, with extraneous symbols removed, and the number of character are counted. These are the counts which are used in the comparisons in Section 3.3.

2.3.2. Turing machine semantics

The external representation of the TM is a series of tuples followed by a tape as in Fig. 1. The head of the machine either starts at the very left hand side of the input tape, or to the right of the initial head marker (^). These TMs are defined as $\langle Q, \delta, \Gamma, \gamma \rangle$ where Q is the set of states, δ is the transition function, Γ is the tape alphabet, and $\gamma \in \Gamma$ is the blank symbol.

The internal representation of the machine has a tape T, a symbol table δ, a current state st and a head position h. The tape 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 the symbol under the initial head position. The initial tape 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 Fig. 2.

Before we execute the TM, we first have to populate δ and T_0. The external TM is an expression $e \in (\Gamma \cup Q \cup d \cup \{,\})^*$. Here, we do not consider expressions which are invalid. The symbol table parsing rules supplied by the function P_δ are shown in Fig. 3.

Similarly the external tape is an expression $f \in \Gamma^* \cup \{\text{^}\}$. The function P_T parses f into the initial tape T_0 and is shown in Fig. 4. The functions δ and T are constructed recursively by the union of each mapping of input to output. The initial state of a parsed TM is therefore:

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

```
1,1,2,0,R
2,1,2,1,R
2,0,0,1,R
^101
```

Figure 1. External representation of a Turing machine for addition with an input of $1 + 1$.

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

Figure 2. Type definitions for the variables and functions of the TM.

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

(a) Parsing a rule into δ (b) Default rule

Figure 3. Rules for parsing an external TM symbol table e into the internal representation δ.

$$\frac{\begin{array}{c} f \implies f_1\text{\^{}}gf_2 \\ g \in \Gamma \end{array}}{P_T(f,0) = P_{NT}(f_1,-1) \cup \{0 \mapsto g\} \cup P_T(f_2,1)}$$

(a) Finding ^, if it exists

$$\frac{\begin{array}{c} f \implies gf_1 \\ g \in \Gamma \end{array}}{P_T(f,n) = \{n \mapsto g\} \cup P_T(f_1, n+1)} \qquad \frac{}{P_T(f,n) = \{\}}$$

(b) Parsing symbols after the ^ (c) No symbol to parse after

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

(d) Parsing symbols before the ^ (e) No symbol to parse before

Figure 4. Rules for parsing a raw tape into the internal representation T.

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:

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

The Turing machine consists of three rules: for shifting left, for shifting right, and for no defined state and symbol pair. Figure 5 shows the rules for running a TM. The machine halts when there is no 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.

2.3.3. RASP semantics

The RASP machines also require parsing. The external RASP machine is a comma delimited string of 2^{n-3} natural numbers. These numbers are drawn from the set $G : \{0 \ldots 2^n - 1\}$, and the expression e to be parsed is $e \in (G \cup \{,\})^*$.

The internal structure of the machine is a pair, of a function $S : \mathbb{N} \mapsto \mathbb{N}$ which represents the memory of the machine, and an output set $X : \mathbb{N}$ which is modified whenever the machine executes the OUT instruction. The set $I \subseteq G$ is of non-halting instructions.

The type definitions for the RASP are shown in Fig. 6. 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.

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 is combined with the external representation e parsed by P (Fig. 7) which is then evaluated by E:

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

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

(a) 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) A 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) The halting rule

Figure 5. Rules for the TM; left shift, right shift, and halt.

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

Figure 6. 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 single natural out of the expression e

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

(b) Default rule

Figure 7. Semantics for parsing the external representation e.

The RASP semantics are structured as model and language semantics. The model semantics contain the rules for P and the rules for the fetch-execute cycle (Fig. 8). If a value is a valid instruction, the instruction is applied via the A function to produce a new S and X pair.

The language semantics are ten rules for the seven non halting instructions. For example, Fig. 9 shows the rule for loading a constant into the ACC. It sets the PC of the returned function S' to $PC\_INC(S)$, and the ACC and IR to $S(S(PC))$.

The semantics of the RASP2 and RASP3 use the same model semantics of the RASP. The fetch-execute cycle and parsing semantics do not change, but the language semantics for both models replace the INC and DEC instructions of the RASP with new ADD x and SUB x instructions.

Figure 10 shows the ADD x instruction for both models. Comparing the two, the difference is on the $S'(ACC) = \cdots$ line. RASP2 adds the contents of the IR with "$+S'(IR)$", but the RASP3 adds an indirection with the addition

$$\frac{S(S(PC)) \in I \quad S'(PC) = PC\_INC(S) \quad S'(IR) = S(S(PC)) \quad \langle S'', X' \rangle = A(S', X)}{E(S, X) \implies E(S'', X')}$$

(a) Executing a non-halting instruction

$$\frac{S(S(PC)) \notin I \quad S'(IR) = S(S(PC))}{E(S, X) \implies \langle S', X \rangle}$$

(b) A halting instruction/value

Figure 8. Rules for the F–E cycle of the RASP.

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

Figure 9. LOAD instruction.

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

(a) RASP2 ADD instruction

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

(b) RASP3 ADD instruction

Figure 10. ADD instructions for the RASP2 and RASP3.

of another enclosing S, "$+S(S'(IR))$". These six extra characters, three for each rule, distinguish the models from each other.

2.3.4. Measuring semantics

Let us now consider the status of SOS. We tend to privilege semantic formalisms over programming languages as being more mathematical and more abstract. In particular, semantic formalisms appear to lack notions of time or evaluation order; that is a semantics applied to a language instance is in some sense instantaneously the instance's meaning. While we disagree with this stance, time independence is a desirable property of a formalism for making explicit the time dependent properties of concrete languages.

Given there are vastly many different potential semantic notations, we may again reasonably ask which is the most succinct, that is which offers the most elegant characterisations of defined languages. For example, here we have used SOS, but we could equally well have deployed denotational semantics or an algebraic semantics or an axiomatic semantics. Commonly, number theoretic predicate calculus or set theory are claimed as lingua franca of mathematical logic.

Indeed, the choice of a formalism as a baseline is largely arbitrary and selected due to familiarity. But we could pick any Recursively Enumerable (r.e.) language as a baseline for our measurements and realise that the semantics for a model is just a universal machine for that model written in that semantic scheme. While it is convenient for us to use semantic schemes to describe the models, there is no reason why we cannot equally well use the SOS to implement the programs, and take the RASP and TM models as baselines by expressing the semantics of the SOS as RASP/TM programs.

Insisting that the r.e. languages used to represent the semantics are somehow different from the r.e. languages used to represent the programs, and that their information should be measured in a different way, is to insist that all r.e. languages are sufficiently different from each other and require their own measures. This disallows any kind

of comparison and is, we think, an untenable position, especially in the light of Section 3.3 which exemplifies comparisons between different models of computation.

The use of a single number to represent the total information content of a particular program in some model has been found to be satisfactory for our two models and single semantic scheme. However, as we include more semantic schemes and models, continuing to use this single figure may cause us to lose meta-information on how the content is proportioned.

For instance, if we wrote a larger set of semantics which enabled us to represent the programs more succinctly, perhaps with some special encoding, using a single number n to denote the information does not show us this development.

We would be better served by a pair of figures $\langle n, m \rangle$ which show n characters in the semantics and m characters in the program. This tuple notation is flexible enough to show the relation between the size of semantic expressions and the size of the programs for any number of semantic schemes and functions. This will not be required here since we have only a single semantic scheme.

Thus, semantics are measured in characters. However, the semantic rules presented above are verbose in order to ease understanding. The measured size arises from a logically equivalent realisation of these semantics where the symbols are all reduced to single characters and expressed in Reverse Polish Notation (RPN) [18]. The advantages of this notation are that we remove the need for bracketed expressions, with the overhead of keeping track of value and operator stacks as we parse the expressions.

Table 3 shows the model+language semantics for RASPs and the model semantics for TM. The language semantics for the RASP2 and RASP3 replace the INC and DEC commands of the RASP with new ADD and SUB instructions. These instructions are more complicated than INC and DEC, requiring an extra fetch to obtain the parameter, so require more information to represent the semantic rules. In total, the semantics for RASP2 are 29 characters larger than the those for RASP. The semantics for RASP3 are only 2 characters larger than the semantics for RASP2. Because we equate the size of the semantics with the expressiveness of the model, we expect (by Hypothesis 1) to see that smaller program sizes follow the relation: TM $\geqslant$ RASP $\geqslant$ RASP2 $\geqslant$ RASP3.

A measurement of the model semantics size excluding the parsing rules is also shown.

Note that the FPGA implementations discussed below do not parse their inputs. They start with the machines already in the correct internal representation, so it is more equitable to compare the FPGA data to the semantics which only evaluate the machines.

3. Comparison of program size versus expressiveness

3.1. Exemplars

The arithmetic functions which we measure here are: addition, subtraction, equality, multiplication, division, and exponentiation. These primitive recursive arithmetic functions [6] exemplify the arithmetic hierarchy and one may plausibly assume that at least one is included in any program. This provides some reasonable baseline of tasks for a model of computation to solve.

In addition, we also measure a set of five list functions. These are: list membership, linear search, list reversal via in-place swapping, list reversal by creating a new list, and the bubble sort. These functions build upon the arithmetic functions and introduce non-trivial memory manipulation.

The most complex tasks we measure here are interpreters for both the RASP and Turing machines. This shows both the computational equivalence of the models, and their simulation overhead in terms of program information.

Table 3
SOS sizes of the four models

	RASP	RASP2	RASP3	TM
SOS Size	228 + 328	228 + 357	228 + 359	335
Ex. Parsing	156 + 328	156 + 357	156 + 359	146

3.2. Addition

Assuming that succ(a) and pred(a) are the successor and predecessor functions respectively, addition can be defined as:

$$\text{add}(x, y) = \begin{cases} x & : y = 0 \\ \text{add}(\text{succ}(x), \text{pred}(y)) & : \text{otherwise} \end{cases}$$

Addition is the only program here where the TM has a shorter program than the RASP (Fig. 11). The input is two numbers in unary, separated by a single blank symbol. The TM operates by removing the first symbol from the leftmost number, shifting right until it finds the middle blank symbol, replacing it with a 1 symbol and halting. The TM uses 3 states and 2 symbols: $TM_{\text{add}}(3, 2)$

The RASP program is shown in Table 4. For convenience, those integers representing instructions for execution are written out in full, whereas integers that are not to be executed remain as integers. Similarly, the labelling of instruction and data addresses are purely abstract for readability; labels in the "Data" column are the addresses which the labels reference.

The two integers to be added are referenced by the labels "add2" and "add1" which are 3 and 5 respectively. We first test add2 for zero. If it is we halt, but if not we decrement it, store it back in add2, increment the value in add1, and jump back to addStart. After the machine halts, the final result can be found in add1. The RASP2 and RASP3 natively support addition, and its usage is shown in Table 5.

The rest of the functions and implementations measured in this paper are covered in detail in [5]. While we cannot guarantee the succinctness of any one program, we have used standard representations of the functions and consistent translations to TM and RASP.

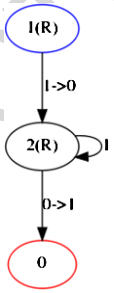

Figure 11. State diagram for the addition TM. State 1 is initial, 0 is halt.

Table 4
The RASP program for addition. Showing the instructions, data, and their respective labels

Instruction	Data	Instruction Label	Data Label
LOAD	3	:addStart	;add2
JGZ	'adding		
HALT			
DEC		:adding	
STO	'add2		
LOAD	5		;add1
INC			
STO	'add1		
LOAD	1		
JGZ	'addStart		

Table 5
Addition in (a) The RASP2 and (b) The RASP3

(a) RASP2 addition

Instruction	Data
LOAD	3
ADD	5

(b) RASP3 addition

Instruction	Data	Instruction Label
LOAD	3	
ADD	'label	
HALT		
5		:label

3.3. Comparisons

This section presents the resolution of Hypotheses 1–3 which are recapped below. We use size measurements of the programs and semantics following the Chaitin–Kolmogorov convention of measuring the program only, and do not consider other metrics such as the growth rate of the input sizes of the programs. An in depth analysis of how these programs influence their input sizes is out of scope for this investigation, but is briefly discussed in Section 6.

The textual forms of the RASP and TM are measured as shown in Section 2.3. The RASP is a comma delimited string of $2^n - 3$ integers, and the TM is a string of tuples of the form st, sy, st', sy', dir delimited by newline characters.

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

Hypothesis 2 (Total Information; Arithmetic)**.** For two Turing Complete models; if model A has more semantic information than model B, then when computing the arithmetic functions, model B will require *less* total information than model A.

Hypothesis 3 (Total Information; List)**.** For two Turing Complete models; if model A has more semantic information than model B, then when computing the list functions, model B will require *more* total information than model A.

Table 6 presents the program information and corresponding total information (semantics size + program size) for each function in this study. Tables 7 and 8 show the arithmetic and geometric means for the categories of functions described. In these tables "AR" denotes the arithmetic functions, while "L" indicates the list functions.

3.3.1. RASP and TM

Hypothesis 1 states that the more semantic information in a model, the smaller the programs are on average. Using the data in Table 6 and comparing the RASP with the TM, we can see that the RASP tends to produces smaller programs than the TM, with the exception of addition.

The arithmetic and geometric means of the program information (Tables 7 and 8) for the RASP show, that for all classes of functions, it requires less information on average to represent these functions than for the TM, supporting Hypothesis 1.

Hypothesis 2 states that it will take less total information to represent the arithmetic functions in TM, compared to the RASP. The means of the total information figures show that the TM requires less total information than the RASP programs, which supports this Hypothesis. Only for the exponentiation function does the TM require more TI than the RASP at 785 characters to 688.

Comparing the list functions of the RASP against those of the TM lends evidence in support of Hypothesis 3. The list functions alone show the arithmetic/geometric average total of the RASPs as 862.6/852.54 opposed to 1209.4/1139.43 of the TM. Comparing the total information average of all of the functions serves to widen this gap.

Table 6
Program and Total information measurements of each function in each model. Measured in characters

Function	RASP		RASP2		RASP3		TM	
	Prog	Total	Prog	Total	Prog	Total	Prog	Total
Addition	58	614	9	594	25	612	29	364
Subtraction	59	615	59	644	61	648	149	484
Equality	57	613	26	611	27	614	179	514
Multiplication	126	682	59	644	60	647	223	558
Division	131	687	131	716	134	721	281	616
Exponentiation	132	688	129	714	131	718	450	785
List Membership	271	827	129	714	131	718	379	714
Linear Search	281	837	132	717	135	722	779	1114
New List Reverse	140	696	135	720	137	724	499	834
In Place Reverse	273	829	273	858	277	864	1049	1384
Bubble Sort	557	1113	549	1134	297	884	1611	1946
Universal TM	613	1169	571	1156	574	1161	1270	1605
Universal RASP	1239	1795	1209	1794	1231	1818	14414	14749
Semantics size	556		585		587		335	

Table 7
Arithmetic means of the data in Table 6

Group	RASP		RASP2		RASP3		TM	
	Prog	Total	Prog	Total	Prog	Total	Prog	Total
Arith	93.83	649.83	68.83	653.83	73	660	218.5	553.5
List	304.4	860.4	243.6	828.6	195.4	782.4	863.4	1198.4
AR+L	189.55	745.55	148.27	733.27	128.64	715.64	511.64	846.64
All	302.85	858.85	262.38	847.38	247.69	834.69	1639.38	1974.38

Table 8
Geometric means of the data in Table 6

Group	RASP		RASP2		RASP3		TM	
	Prog	Total	Prog	Total	Prog	Total	Prog	Total
Arith	86.71	648.84	48.95	652.18	59.27	658.53	167.15	538.68
List	276.67	850.31	202.98	814.57	181.93	778.72	757.23	1123.07
AR+L	146.93	733.70	93.44	721.54	98.68	710.74	332.16	752.26
All	193.22	814.65	130.78	802.48	137.21	793.38	492.16	1002.53

Hypotheses 2 and 3 appear to be confirmed, with the breakpoint where the TI of the RASP becomes lower than that of the TM appearing between the list membership function, and the exponentiation function. We believe that the cause is a combination of both the encodings, and consequently, of the number of loops which have to be defined for the machines. While the RASP can manipulate entire registers to copy, increment, and decrement with one step, the TM is required to copy numbers, or list elements one symbol at a time which necessitates the programmer to define structures which are implicitly defined in the semantics of the RASP.

3.3.2. RASP2/RASP3 and TM

Comparing the RASP2 and RASP3 with the TM reinforces the evidence we have for Hypotheses 1, 2, and 3. All of the functions have smaller RASP2 and RASP3 representations than in the Turing Machine. With larger semantics

Table 9
Number of instructions of programs written in the three versions of RASPs

Function	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 Reverse	57	45	43
In Place Reverse	73	78	77
Bubble Sort	131	127	123
TM Simulator	200	148	137
RASP Simulator	313	292	283
Geometric Mean	57.99	40.79	41.47
Arithmetic Mean	85.46	69.76	67.56

for the RASP2 and RASP3, the total information of the TM for the arithmetic functions is still lower than that of the RASPs again, excepting the exponentiation function.

Similarly, the total information measurements of the RASP2 and RASP3 over the list functions and the set of functions as a whole is lower than the total information of the TM (Tables 7 and 8). Again, this provides evidence in support of Hypothesis 3.

3.3.3. RASPs

It can be observed that the RASP measurements have a tendency to cluster. For instance, the RASP2 functions for linear search, list membership, and new list reverse all have very similar program sizes. These are a result of the RASP having a fixed memory size depending on the number of bits in the machine. Unused instructions at the end of the machine are 'padded' out to size via the HALT instruction.

We might think that RASP 2 and 3 would have significantly smaller programs than the RASP and that Table 6 would show this. But aside from the RASP2/3 figures for addition, multiplication and comparison, the reductions are not as extreme as we might expect.

There are two reasons for this. First of all the gains produced are not large enough to reduce the required number of instructions such that the program can fit on a machine of size 2^{n-1}.

For a RASP machine with a fixed memory (2^n), not every memory location is required for the function to be executed. Any unused memory locations will be padded with the HALT command (0) and any reduction in the number of required instructions that RASP2/3 hold over RASP may be not be obvious. Table 9 presents just the instructions used without padding, which shows that subtraction and division programs require more instructions than the standard RASP.

This brings us to the second reason: there are always incremental and decremental changes to the values in the functions. As an example, take the subtraction function: SUB(x, y). This function decrements both x and y until one of them is 0. Decrementing in the RASPs 2 and 3 takes at least one more instruction than the atomic DEC of the original RASP. Division is a similar case where we need to decrement our dividend until it reaches zero, incrementing the quotient. Because the SUB command in the RASP pays no attention to over and underflow conventions, there is no straightforward method of determining if the SUB command will trigger an underflow. The only alternative is to emulate the DEC instruction using "SUB 1".

Hypothesis 1 predicts that the RASP3 will require smaller programs than the RASP2, which in turn requires smaller programs than the RASP. The geometric and arithmetic means (Tables 8 and 7) of the arithmetic functions show the RASP2 as having the lowest program and total information. The mean measures disagree on the list functions; the arithmetic means show the RASP3 as requiring less information than the RASP2, whereas the geometric means assert the converse. There is a similar case with the program information regarding the AR+L and All categories.

The arithmetic mean is calculated using the absolute difference between elements of the set being averaged. The geometric mean is more concerned with the percentage difference between elements. Here, the difference between the program information measurements of the addition function (9 vs 25) is weighted more significantly than the difference between the measurements of the bubble sort (549 vs 297). With respect to the RASP2/3, the evidence for Hypothesis 1 currently depends on which mean is valued more. More evidence is required before there can be a conclusion either way.

Table 9 offers some insight into what the trend of new functions may be. As with Table 6, the functions are roughly ordered by how complex we perceive them to be and it shows that the list functions for the RASP3 often have fewer instructions than for the RASP2. We believe this trend will continue as the comparison set increases in size.

Hypotheses 2 and 3 predict that the total information of the RASP machines will follow the trend: RASP<RASP2<RASP3 for the arithmetic functions, and RASP3<RASP2<RASP for the list and universal functions. In all cases, the means in Tables 8 and 7 reflect what was hypothesised.

3.4. Results

We have assessed the data displayed in Tables 6, 7, and 8 with respect to the initial hypotheses. The RASP machines are all largely congruent with the hypotheses, but we need more data to resolve Hypothesis 1 with respect to the RASP2 and RASP3.

The strong evidence to support Hypotheses 2 and 3 is very interesting. The TM requiring less total information to represent the arithmetic functions than the RASP reflects the large difference in semantic size between the two, but the RASP models conforming well to the prediction is more of a surprise. There are much narrower gaps (as little as two characters) between the RASP semantics so that we would need much more data before such a clear separation in the total information measurements would appear.

Furthermore, we think that we have identified a trend in the number of instructions required for the RASP2 and RASP3 (Table 9). We speculate that as further complex functions are added to the comparison set, the gap between the number of instructions to implement the functions in the models will continue to grow.

4. Grounding interpreting mechanisms

The discussion above has been of abstract models of computation. However, computations are ultimately intended for realisations on physical computers. Given that such devices are materialisations of symbolic systems for manipulating information, we may expect that Hypotheses 1 to 3 apply to those devices as well.

Thus, we now attend to the construction and measurement of universal machines as physical machines. Here we choose Field Programmable Gate Arrays (FPGAs) as one of the simplest forms of physical device for which we can gain precise measures of information comparable to the abstract information theoretic measures.

First, we introduce FPGAs through implementations of specimen RASPs and TMs. We next restate our hypotheses for FPGAs, and explore FPGA implementations of universal RASPs and TMs, using low level component counts as indices of information to compare their properties. Finally, we examine the relationships between the information of our abstract formulations of computation and their physical realisations as FPGAs.

4.1. FPGAs

An FPGA is essentially a configurable chip. Rather than converting a High Level Description Language (HDL) specification into something resembling assembly code, the specification is “synthesised” into a Register Transfer

Logic (RTL) diagram which expresses the high level logic as electronics using components such as gates, flip-flops, multiplexers and so forth.

An FPGA consists of blocks and slices depending on the terminology of the manufacturer. These blocks/slices each have a small number of gates, multiplexers and memory with the capability to route signals in an arbitrary path around and between blocks. At configuration time, the board maps the RTL components to the components in the slices and activates routes between them so that signals can be transferred.

This results in a chip that physically performs the task specified by the HDL, though it may not necessarily have any resemblance to the RTL schematic, as the components in the FPGA are distributed. For instance RAM may be constructed by many flip-flops across multiple blocks/slices, with those flip-flops being physically close together.

The use of an FPGA to build RASP and Turing machines provides us with an absolute physical grounding which we can use as a baseline for comparison of the sizes of the models using a tangible measure, rather than the number of characters.

4.2. FPGA environment

The RASPs and TM were specified in VHDL (VHSIC Hardware Description Language [12]) and implemented using a the Xilinx Zynq-7000 [25] all programmable System on a Chip. The software used to compile the design was the Xilinx ISE Design Suite 14.7. The specific programmable logic part is the XC7Z020-CLG484 at speed grade -1, and the software was configured to implement the design using as little area as possible. The above part also includes a dual-core ARM Cortex-A9 based application processing unit, but that was not initialised in the design of RASP and TM, and was therefore not used.

The PL fabric consists of a series of Configurable Logic Blocks (CLBs). Each CLB consists of two slices, where each slice contains 4 look up tables (LUTs). These can be chained to form a 6 or 5 input LUT, which can represent 64k and 32k of ROM respectively. The LUTs can also be used to implement arbitrary logic in concert with those in other slices to produce complex logical systems. Additionally, the slices contain flip-flops (FFs) which can be paired with the LUTs to produce RAMs, rather than ROMs. There are also 4 bit cascadable adders which can be chained with other adders, and a number of multiplexers and assorted small registers.

To create a circuit which is as small as possible, we tune the optimiser so that the resulting circuit utilises the fewest LUTs, FFs, and registers. We create a 'semantic circuit' by expressing the models using only the execution semantics with a minimal memory size, and contrast that with the 'full circuits' where we also include the memories for the instructions loaded onto the device.

4.3. TM and RASP in VHDL

Broadly, the TM and RASP are each composed of three components: the controller which is a state machine representing the read/write/shift rules for the TM and the fetch-decode-execute cycle for the RASPs, the memory as a tape or RAM, and an overall "machine" block which routes signals between the two (Fig. 12).

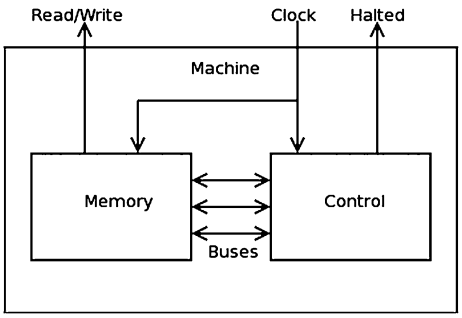

Figure 12. A block level view of the general layout of both the RASP and TM.

All of the components are clocked at 300 MHz, where the controller processes information on the buses every time the clock ticks to 1 (the *rising edge*), and the memory processes the buses whenever the clock tocks to 0 (the *falling edge*).

Inside the control blocks, state machines impose an execution order on the instructions. Each clock cycle prompts the state machine to modify a handful of signals and increment a counter indicating which state the machine is in. The implementation details of the RASP and TM in the FPGA will be briefly covered here, but a more thorough treatment can be found in [5].

The RASP control is a state machine which uses two nested counters to execute the program. The outer counter controls the general fetch-decode-execute cycle which is analogous to the model semantics for the RASP. The inner counter keeps track of the stages of executing an instruction. The memory of the machine is the current mapping of addresses to data, and there are variable signals in the control which specify the memory address to be read or written to.

The TM is controlled by a state machine with a single counter. The tuples of the specific TM are stored at records in ROM, and a look-up function retrieves the correct tuple using the current state and symbol. The memory, despite being random access, is read from and written to as a tape.

4.4. Machine sizes

Synthesis and mapping of the RTL gates to the board produces a utilisation report which explains what has been used in the process. We are not concerned with the sizes of the VHDL code, as we are with those for the programs and semantics in previous sections, but with the number and complexity of the gates that the code maps to. The size of the resultant machines are largely indicated by three different values: the slice registers for storing state information, look up tables (LUTs) for defining control logic and ROM, and flip flops which implement parts of RAM and some of the control logic.

As above, we make comparisons between the TM and all of the RASP machines. The functions for each machine were translated into VHDL and mapped to the FPGA fabric with the optimiser prioritising the minimum use of resources. The optimiser removes extraneous signals and memories from the RTL diagram although, due to its proprietary nature, it is unclear exactly which optimisations are performed. As before, we ignore input sizes but because of the optimiser, we define the tape as two cells for all TMs.

The data in Tables 10 to 12 show that the TM sizes vary for each separate function, while the sizes of the RASPs are constant with respect to their memory size. This is due to the RASP programs being defined in memory which can vary in size while the controller remains constant. Conversely the TM memory here is fixed at a certain number of cells and the controller varies in the size of the symbol table specific to each function.

Table 10
Slice registers required to implement the above programs in the above models on an FPGA

Function	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 New List	32	32	33	23
Reverse In Place	37	37	38	23
Bubble Sort	41	41	38	24
Universal TM	41	41	42	23
Universal RASP	46	45	46	19

Table 11
LUTs required to implement the above programs in the above models on an FPGA

Function	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 New List	74	79	91	32
Reverse In Place	81	86	102	80
Bubble Sort	90	96	102	150
Universal TM	89	96	112	195
Universal RASP	92	108	123	1019

Table 12
FFs required to implement the above programs in the above models on an FPGA

Function	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 New List	32	32	33	23
Reverse In Place	37	37	38	23
Bubble Sort	41	41	38	24
Universal TM	41	41	42	23
Universal RASP	46	45	46	18

4.5. Making comparisons

Hypotheses 4, 5, and 6 make predictions about the number of components required to implement programs on the FPGA. Hypothesis 4 is analogous to Hypothesis 1, in that there is a prediction that models with more components dedicated to their semantics will produce smaller programs. Hypothesis 5 predicts that models with small semantics will produce smaller total circuit sizes for the arithmetic functions than models with larger semantics. Finally Hypothesis 6 predicts that models with larger semantics will produce smaller circuits for the list functions. The hypotheses are recapped below.

Hypothesis 4 (Semantic Circuit Size)**.** For two Turing Complete models; if model A requires a larger circuit to express its semantics than model B, then model A will need on average smaller circuits to implement functions.

Hypothesis 5 (Total Circuit Size; Arithmetic)**.** For two Turing Complete models; if model A requires a larger circuit to express its semantics than model B, then when computing the arithmetic functions, model B will require a *smaller* circuit than model A to represent both the semantics and the programs.

Table 13
Numbers of components required to implement only the semantics of the models

Function	RASP	RASP2	RASP3	TM
Slice Registers	21	21	22	10
LUTs	48	50	63	7
FFs	21	21	22	10

Table 14
Components for programs only on FPGAs

Function	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

Hypothesis 6 (Total Circuit Size; List)**.** For two Turing Complete models; if model A requires a larger circuit to express its semantics than model B, then when computing the list functions, model B will require a *larger* circuit than model A to represent both the semantics and the programs.

The numbers of LUTs and slice registers required to implement the semantics of the models are presented in Table 13. The compiler and optimiser give a breakdown of the components needed to implement the control and memory blocks of the circuit individually. The RASPs were all measured with an empty memory of size 8, and the TM had two tape cells and a single tuple in the symbol table.

Here, the LUTs implement the random logic of the model state machines. Slice registers hold state information and variables. These measurements are ordered in the same sequence as the semantic information measurements earlier, which conforms to our intuition.

To find the number of slice registers and LUTs to implement the program information aspects of the FPGA implementations (Table 14) we subtract the number of semantic components (semantic information, Table 13) from the total components (total information, Tables 10 to 12).

Hypothesis 4 states that models with larger semantic circuits require smaller circuits to implement programs on average. For the arithmetic functions, the TM appears to always require fewer components (Slice registers + LUTs) than the RASPs. For the list and universal functions the RASPs require fewer components.

The RASP data shows that the RASP3 requires more succinct program circuits in general relative to the RASP and RASP2. This supports Hypothesis 4, in contrast to the figures from Table 6 where the RASP2 produced the most succinct programs for the arithmetic functions.

While the data regarding the RASP3 supports Hypothesis 4, the data with respect to the TM is more complex. Unlike the character comparisons above, the TM can realise its programs with fewer components than the RASPs. The list and universal function sizes are more in favour of the RASPs, with only list reversal in the TM still smaller than the RASP implementations

Table 15
Arithmetic means of the models implemented on the FPGA

Group	RASP		RASP2		RASP3		TM	
	Slice R	LUTs	Slice R	LUTs	Slice R	LUTs	Slice R	LUTs
Mean AR	30	70	27.5	68.17	29	79.67	17.17	19
Mean L	36.80	81.40	34.8	83.80	35	95.4	22.80	71
Mean AR+L	33.09	75.18	30.82	75.27	31.73	86.82	19.73	42.64
Mean All	34.69	77.54	32.69	79.38	33.62	91.54	23.55	157.27

Hypothesis 5 predicts that models with small semantic circuits will have smaller overall circuits for the arithmetic functions than models with comparatively larger semantics. This is supported by comparing the TM data from Tables 10 to 12 to the RASPs. The TM has smaller semantics than any of the RASP machines and also uses fewer components to realise the arithmetic functions.

Table 15 shows that the RASP, with smaller semantics than the RASP2 and RASP3, does not require fewer components on average to represent the arithmetic functions relative to the RASP2. Indeed, the RASP2 implementation have fewer components on average than both the RASP and RASP3.

Hypothesis 6 predicts that models with large semantic circuits will produce smaller overall circuits for the list functions than models with comparatively smaller circuits. If the hypothesis were correct, the RASP3 would require fewer components on average to represent the list functions. However, the list function means in Table 15 show that this is not the case. The RASP2 requires the fewest slice registers, and the RASP requires the fewest LUTs.

Unlike Hypotheses 2 and 3, the data for Hypotheses 5 and 6 is not supportive. It is suspected that there are two causes. First, FPGA measurements are too coarse and do not capture the subtle differences between the RASP programs as the character metric does. And second, increasing the sizes of programs incurs an overhead of LUTs and slice registers in the semantics.

The abstract nature of the mathematical representations defines a general and fixed size for the semantics. Once the semantics have been defined in the SOS, they are measured and then assumed to be constant in size because the structures and variables are initialised to a type which encompasses all possible values. In the FPGA realisation, the semantics of the model must be changed to permit larger values to be represented. Larger buses, adders, and state variables all contribute to the overhead of increasing the size of the program. The more complex the semantics, the higher the overhead.

The RASP program measurements on the FPGA are dependent purely on the size of the memory. These 'levels' of resource use do not reflect how many registers have been used past $2^n/2$. In the case of comparing the RASP2 and RASP3, which have equal memory sizes for the majority of their functions, the models with the smaller semantics wins out. The RASP2 may require (say) 30 more instructions than the RASP3, but if the total memory sizes are the same, the measurements are going to favour the RASP2.

5. Reconciling FPGAs with SOS

The implementation and measurement of the models in FPGAs is an effort to corroborate the measurements of the SOS semantics. If the total circuit size of an FPGA implementation is relatable to the total information using SOS, then we can be more confident that the methodology is correct.

The plots in Figs 13–15 show the total information of the programs with semantics in SOS, the number of slice registers required to implement the programs, and the number of LUTs to implement the programs respectively.

For the RASPs, the number of slice registers appears to correlate strongly with the TIs of the programs. Conversely, the slice register count of the TM shows no correlation. The TM TIs correlate strongly with the number of LUTs in the circuit, and the RASP TIs correlates quite well also.

The semantics in the SOS are not entirely representative of the semantics in the FPGA. The FPGA semantics contain no rules for parsing the programs because each program is separately compiled into the circuit so there is no need for parsing to take place. Conveniently, the definitions and rules for parsing can be removed from the SOS semantics for a more accurate comparison.

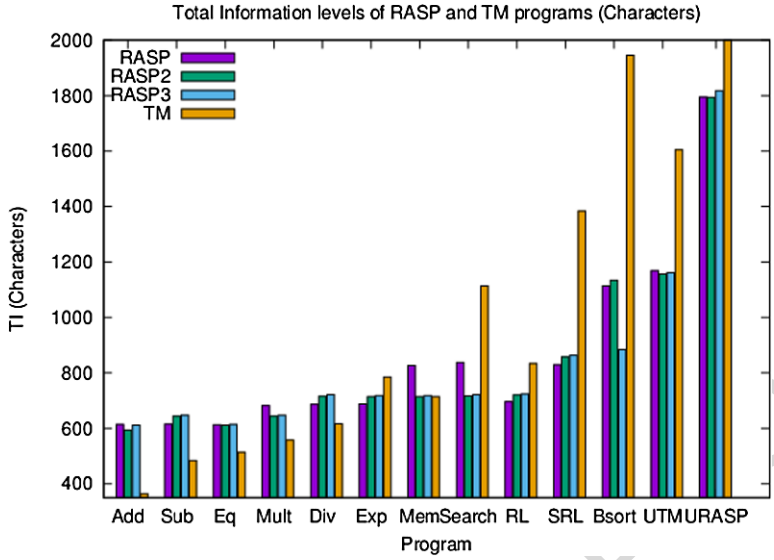

Figure 13. Total Information contents of the programs.

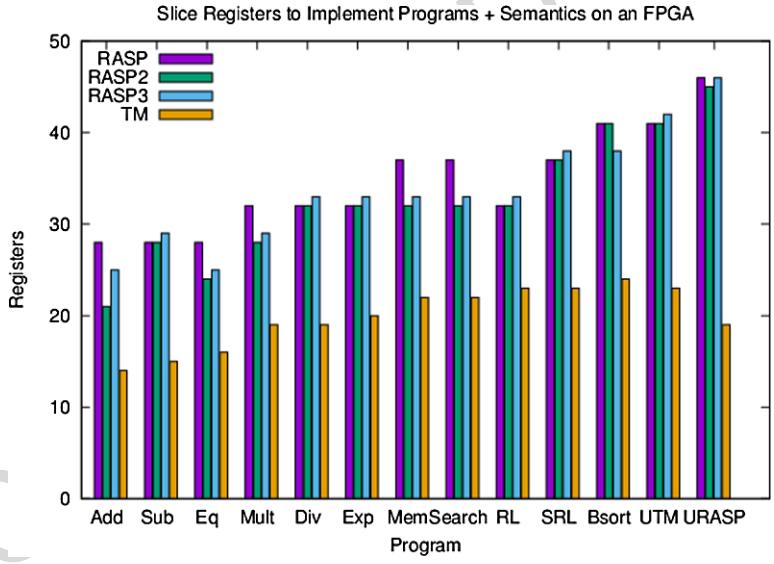

Figure 14. Number of slice registers required to implement the entire circuits.

Table 16 displays the semantic sizes of the models without definitions and rules for parsing. Furthermore, it shows the Pearson correlation coefficient between TI (excluding parsing) figures and the various component counts.

As noted above in the plots, the slice register/flip-flop counts 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 0.98. With a coefficient this high, we can be reasonably sure that there exists a causal link between the TI of a TM and the number of LUTs in the corresponding FPGA implementation.

There is also a high correlation between the TI of the RASP and the number of slice registers. These correlations fluctuate slightly for the RASP2 and RASP3, but still heavily suggest a causal connection.

With the data presented here it can be concluded that the methodology of grounding the computational models

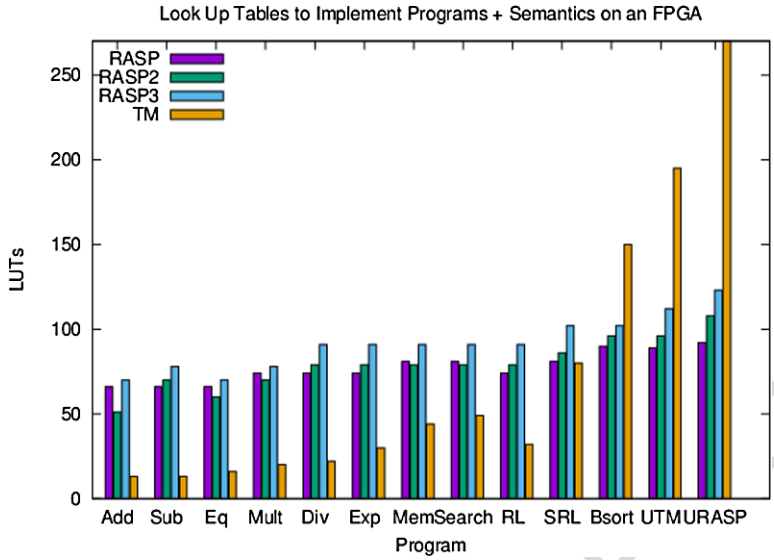

Figure 15. Number of LUTS required to implement the entire circuits.

Table 16
Pearson correlation coefficient of the TI (excluding parsing semantics) vs the components

	RASP	RASP2	RASP3	TM
Semantics (no parsing)	484	513	515	146
Slice Registers	0.79	0.75	0.80	0.0
LUTs	0.70	0.74	0.80	0.98
Flip-Flops	0.77	0.75	0.80	−0.07

in FPGAs is consistent with the methodology of defining the semantics in a mathematical model such as SOS. Measuring the information required to express programs as absolute character counts correlates well with the number of components required to express the programs on the FPGA.

This data should enable reasonable estimations of the number of LUTs required to represent some TM program in the FPGA using the TI, and vice versa. Similarly one could estimate the number of slice registers required for a RASP or RASP2/3 machine with some accuracy.

6. Conclusions

This paper empirically tests Felleisen's Conciseness Conjecture using two different models of computation, the Turing and RASP machines. We take the information metric of Kolmogorov and Chaitin, the number of characters to represent a program, and use it to measure the information in both the semantics of models, and in programs written for them.

The choice of a semantic system as a baseline from which to compare sizes is problematic, as we could never be sure that the chosen scheme would produce the most concise set of semantics that describe the operation of our machines. We considered implementing the models in different semantic systems, but this would start us on the road to an infinite regress of implementing more and more abstract systems to describe the underlying concrete ones.

Instead we physically grounded our models such that our baseline system is a series of electronic gates using a Field Programmable Gate Array. Measuring the size of the circuits in terms of their utilised components gives us an intuition as to how the space of machine sizes looks with respect to the functions we have selected for comparison.

We stated six hypotheses relating the ratio of information in the semantics to information in the programs. Three of the hypotheses concerned the SOS implementations of the models, while the other three made predictions about the FPGA realisations. We tested the hypotheses by implementing six functions from the arithmetic hierarchy, five list processing functions, and two universal machines.

Hypotheses 1 and 4 predicted that models of computation which had more information/electronic components in their semantics (i.e. more instructions/complexity) produced more succinct programs. Testing Hypothesis 1 showed that the RASP3 (with the largest semantics) eventually produced smaller programs for comparatively large functions (Table 6). In the smaller functions, the average size favoured the RASP2. The much simpler semantics of the TM meant that the program sizes quickly grew with the complexity, eventually creating easily the largest universal programs.

Testing Hypothesis 4 appears to approach the same resolution as Hypothesis 1, although measurements are muddied somewhat by the more stringent coupling of programs and the semantics of the models. There is an overhead involved for the semantic section of the circuit whenever the program grows. This is to support wider data/address busses, structures and so forth. Hence, there is a slower divergence in the number of components between models. Without more data, Hypothesis 4 cannot be resolved either way.

Hypotheses 2 and 5 predict that models with smaller semantics/semantic circuits will have less total information/total circuit size for the arithmetic functions. Testing Hypothesis 2 suggests that the TM has less total information for the arithmetic functions than the RASP machines do. Similarly, the RASP has less total information than the RASP2, which has less total information than the RASP3. These observations fit perfectly with the hypothesis, so, for this dataset, it is confirmed.

Hypothesis 5 is contradicted by the RASP2 requiring less overall components than the RASP, which has smaller semantics than the RASP2, to represent the arithmetic functions (Table 14). The FPGA logic optimiser is an unknown factor here, and may have found a more favourable optimisation strategy for the RASP2 than the RASP. More investigation is required here, so this hypothesis cannot be confirmed.

Hypotheses 3 and 6 predict that for the list functions, models with smaller semantics/semantic circuits will have more total information/total circuit size than models with larger semantics. Like Hypothesis 2, Hypothesis 3 is fully supported by the data. The TM has the smallest semantics and the highest total information of the list functions. In contrast, the RASP3 has the largest semantics and the lowest total information of the list functions.

Hypothesis 6, like Hypothesis 5, is contradicted by the data. The TM has the smallest semantic circuit of all the models, but both the average number of slice registers and LUTs is lower than for any of the RASPs. We believe that the aforementioned semantic overhead of the FPGA may be responsible, but do not yet have enough data to investigate this fully.

6.1. Further work

This paper has explored the first steps of a quantitative method of investigating the information required for computation. We have exposed a number of fertile areas for research and believe that further investigation will deepen our understanding of the relationship between information and computability.

The programs written for this investigation have been controlled by the notion of succinctness, for which we not only expect some reasonable size of the programs, but also some reasonable input. It would be interesting to investigate how removing this restriction would change comparisons. For example, Neary's UTM [15] uses fewer tuples, but has a very high growth factor in the input. Some preliminary work into investigating the relationships in the trifecta of semantics size, program size, and input encoding is present in [5], but a more thorough investigation could shed some light on some underlying no-free-lunch [24] style theorems.

Our work immediately suggests three other areas for further investigation. First of all, we would like to analyse other models of computation, especially from different paradigms, for example λ calculus or Curry combinators. It would also be interesting to explore how the relative total information contents for different models react as an increasing number of programs are compared, in particular, whether or not they converge. Finally, it would be worthwhile to systematically refine the programs/VHDL with non-trivial refactorings and optimisations, to elucidate how different program constructs contribute both to TI based measures and hardware realisations.

Acknowledgements

Joe Davidson's PhD has been supported by Heriot-Watt University, the University of Glasgow, 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'.

We are pleased to thank our colleagues Phil Trinder, Paul Cockshott, Rob Stewart, and Patrick Maier for their constructively critical engagement with this work, and the two anonymous referees whose contribution has improved the quality of the work herein.

References

[1] A. Avram, Study: Clojure, CoffeeScript and Haskell Are the Most Expressive General-purpose Languages, *InfoQ* (2013), http://www.infoq.com/news/2013/03/Language-Expressiveness.

[2] D. Berkholz, Programming languages ranked by expressiveness, Donnie Berkholz's Story of Data, 2012, http://redmonk.com/dberkholz/2013/03/25/programming-languages-ranked-by-expressiveness/.

[3] G.J. Chaitin, *The Unknowable. Discrete Mathematics and Theoretical Computer Science*, Springer, Singapore, 1999.

[4] 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*, ACM, New York, NY, USA, 1972, pp. 73–80. doi:10.1145/800152.804898.

[5] J.R. Davidson, An Information Theoretic Approach to the Expressivness of Programming Languages, PhD thesis, University of Glasgow, 2016.

[6] M. Davis, *Computability & Unsolvability*, Dover Books on Computer Science Series, Dover, 1958.

[7] C.C. Elgot and A. Robinson, Random-access stored-program machines, an approach to programming languages, *J. ACM* **11**(4) (1964), 365–399. doi:10.1145/321239.321240.

[8] M. Felleisen, On the expressive power of programming languages, *Science of Computer Programming* **17**(1–3) (1991), 35–75. doi:10.1016/0167-6423(91)90036-W.

[9] M.H. Halstead, *Elements of Software Science (Operating and Programming Systems Series)*, Elsevier Science Inc., New York, NY, USA, 1977.

[10] J. Hartmanis, Computational complexity of random access stored program machines, *Mathematical Systems Theory* **5**(3) (1971), 232–245. doi:10.1007/BF01694180.

[11] J.E. Hopcroft, R. Motwani and J.D. Ullman, *Introduction to Automata Theory, Languages, and Computation*, 3rd edn, Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[12] IEEE, IEEE Standard VHDL Language Reference Manual, *IEEE Std 1076-2008 (Revision of IEEE Std 1076-2002)* (2009), c1–626.

[13] S.C. Kleene, *Introduction to Metamathematics. Bibl. Matematica*, North-Holland, Amsterdam, 1952.

[14] A.N. Kolmogorov, On tables of random numbers, *Theor. Comput. Sci.* **207**(2) (1998), 387–395. doi:10.1016/S0304-3975(98)00075-9.

[15] T. Neary, Small universal Turing machines, PhD thesis, NUI Maynooth, 2007.

[16] R. Péter and I. Földes, *Recursive Functions, by Dr. Rózsa Péter; Translated by István Földes*, 3rd revised edn, Academic Press, 1967.

[17] G.D. Plotkin, Structural approach to operational semantics, Technical report, Aarhus University, 1981.

[18] H.A. Pogorzelski, J. Lukasiewicz, J. Slupecki and P. Wydawnictwo, Remarks on Nicod's Axiom and on "Generalizing Deduction", *Journal of Symbolic Logic* **30**(3) (1965), 376–377. doi:10.1017/S0022481200128385.

[19] V.J. Rayward-Smith, *A First Course in Computability*, Blackwell, 1986.

[20] C.P. Schnorr, Process complexity and effective random tests, *Journal of Computer and System Sciences* **7**(4) (1973), 376–388. doi:10.1016/S0022-0000(73)80030-3.

[21] J.E. Stoy, Semantic models, in: *Theoretical Foundations of Programming Methodology: Lecture Notes of an International Summer School Directed and Bauer, F. L. and Dijkstra, E. W. and Hoare, C. A. R., Netherlands*, M. Broy and G. Schmidt, eds, Springer, Dordrecht, 1982, pp. 293–325. doi:10.1007/978-94-009-7893-5_10.

[22] A.S. Troelstra, *Metamathematical Investigation of Intuitionistic Arithmetic and Analysis*, Springer, New York, 1973.

[23] A.M. Turing, On computable numbers, with an application to the entscheidungsproblem, *Proc. London Math. Soc.* **2**(42) (1936), 230–265.

[24] D.H. Wolpert and W.G. Macready, No free lunch theorems for optimization, *Trans. Evol. Comp* **1**(1) (1997), 67–82. doi:10.1109/4235.585893.

[25] Xilinx Inc, 7 Series FPGA Configurable Logic Block, Technical report, Xilinx Inc., 2014.