**Hybrid Denotational/Operational Semantics for Physically Implemented Computations**

by PolR

The purpose of this text is to introduce a mathematical notation for defining the abstract computations carried out by computers and networks. This notation may be used in two manners at least: (1) to write the specification of the computation or (2) to formalize the semantics of existing devices. Specification of computations comes before implementation while formalization of existing devices is post facto.

In both scenarios the notation defines an abstract mathematical computation in terms that are independent from the laws and phenomenons of physics. A major difference between the scenarios is the reference defining correctness. When a machine doesn't conform a specification we consider the machine is wrong while when a formalization of an existing machine doesn't match the machine we say the formalization is wrong.

Often, the definition of the computation carried out by a single computer will correspond in a large part to its instruction set architecture. In mathematical terms this is a random access stored program (RASP).[1] It is best to extend the RASP with a mathematical definition of the architecture of the I/O hardware interfaces to cover the functionality of all the computer hardware available.

It is also possible that the computation is carried out by multiple cooperating computers. Then the mathematical definition will be an abstract network.

The notation may be viewed as a denotational semantics because the formulas denote mathematical entities which are the semantics of the hardware and its activities. However this denotational semantics has a strong operational feel because the definitions correlate tightly to the hardware components and theirs operations.

See [Stoy 1981]. This is a classic textbook on the denotational semantics of programming languages. On page 12 Stoy describes an operational semantics as follows:

> We define an “abstract machine”, which has a state, possibly with several components, and some set of primitive instructions. We define the machine by specifying how the components of the state are changed by each of the instructions. Then we define the semantics of our particular programming language in terms of that.

This paper explains how to do this, with the difference that the semantics is not assigned to a programming language. It is assigned to the arrangement of hardware in a computing machine or a network of computing machines.

Then Stoy describes a denotational semantics as follows: (emphasis in the original)

> We give “semantic valuation functions”, which map syntactic constructs in the program to the abstract values (numbers, truth values, functions etc.) which they denote. These valuation functions are usually recursively defined: the value denoted by a construct is specified in terms of the values denoted by its syntactic subcomponents, and it is this emphasis on the values *denoted* by all these constructs that gives the approach its name.

This paper also explains how to do this with the difference, once again, that the semantics is not assigned to a programming language. It is assigned to the arrangement of hardware in a computing machine or a network of computing machines. Instead of syntactic constructs of the programming language we have hardware components from a computer or network. Each component denotes something in a mathematical world of meanings. The semantics of the whole computer or network is defined in terms of the semantics of its components and how they relate to each other. But because we work with a machine (or network), the denotational semantics ends up describing an abstract machine (or network). In this sense the denotational semantics is also an operational semantics.

However there is a difference between the operational and denotational approaches. In a true operational semantics the exact steps and their order of execution count. Operational semantics is often preferred for programming language when the language designer wants the exact execution steps and the exact execution order to be part of the specification. In a denotational semantics it is the mathematical function which is denoted. Any execution order or any computation that is mathematically equivalent, which means gives the same answers for the same inputs, is accepted.

The method proposed here is primarily denotational. I make no effort to capture the exact execution steps and the exact execution order. Any implementation that is mathematically equivalent will carry the correct semantics. If you prefer an operational approach lambda-calculus is not the best language. I would suggest to use tiered definitions where some primitives are defined in a denotational manner with lambda-calculus and then the abstract machine is defined in an operational manner in terms of these primitives. This allows to anchor the definition in the mathematical roots of lambda-calculus without giving up the ability of specifying execution steps and execution order. This suggestion is not put in practice in this paper.

This semantics is not a description of the hardware. It doesn't contain the information necessary to describe the physical aspects of the hardware. Several very different implementations of the same semantics are usually possible including 'virtual' implementations entirely in software. The denotational semantics is a statement of the meanings the bits and the computations must have in the universe of mathematics. The only property of the hardware that is described is that it has to convey the specified semantics.

I call this phenomenon “Minsky's ruthless abstraction” because Marvin Minsky has said it best:[2]

> [I]t is important to understand from the start that our concern is with questions about the ultimate theoretical capacities and limitations of machines rather than with the practical engineering analysis of existing mechanical devices.
>
> To make such a theoretical study, it is necessary to abstract away many realistic details and features of mechanical systems. For the most part, our abstraction is so ruthless that it leaves only a skeleton representation of the structure of sequences of events inside a machine – a sort of “symbolic” or “informational” structure. We ignore, in our abstraction, the geometric or physical composition of mechanical parts. We ignore questions about energy. We even shred time into a sequence of separate disconnected moments, and we totally ignore space itself! Can such a theory be a theory be a theory of any “thing” at all? Incredibly, it can indeed. By abstracting out only what amounts to questions about the logical consequences of certain cause-effect relations, we can concentrate our attention sharply and clearly on a few really fundamental matters. Once we have grasped these, we can bring back to the practical world the understanding, which we could never obtain while immersed in inessential detail and distraction.

The notation is designed to be flexible. It has the semantical capability to define a mathematical denotation for the computational effects of a diversity of physical phenomenons including the operations implemented in digital circuits, but also other physically occurring events such as noise on communication lines, powering off devices, disconnecting cables and random events such as the arbitration of which CPU will access main memory next in a symmetric multiprocessing (SMP) system.

**Introducing the Mathematical Notation**

Let's start with some conventions on mathematical notation that will be used in the rest of this document. The notation is a syntactic sugared version of lambda-calculus. This means the universe of mathematical denotations is a mathematical universe described by lambda-calculus. I assume the reader is already familiar with lambda-calculus. If this is not the case, [Hankin 2004] is a good introductory textbook.

It will help if the reader has some knowledge of domain theory as it is used in denotational semantics.[3] In this notation lambda-calculus variables and expressions will sometimes be given a type. Such types are called *domains*. The notation does not explicitly provide a formal type system but, informally, we ensure each expression is associated with a domain. Then the syntax of lambda-calculus expressions correspond to denotations in domains. In this sense, the expressions have types. Stating the domain of an expression also helps document the intended meaning of the calculations.

Lambda-calculus has been proven to be equivalent to general recursive functions, and by consequence to Turing machines in terms of which mathematical function all these models are able to compute. (For instance see [Hankin 2004] pp. 98-101) Therefore all definitions written in this notation are Turing-computable.

*Atomic Domains*

The domain ***None*** (in boldface) which contains the single value *None* which is used to represent the absence of data.

*Bool* is the *Boolean domain*. It contains the two Boolean values *True* and *False*, sometimes written 1 and 0 respectively. Members of domain *Bool* are called bits. The usual boolean operations *Not*, *And*, *Or* and *Xor* are supported. The conditional *if a then b else c* is the expression that has value *b* when *a* is *True* and has value *c* when *a* is *False*.

The domain ***N*** contains the *natural numbers* 0, 1, 2 . . . This domain supports the ordinary arithmetic operations.

If *n* is a natural number then ***n*** (in boldface) is the domain containing the natural numbers smaller than *n*. For instance **8** is the domain containing the numbers 0 to 7 inclusively.

*Function Domains and Some Related Syntactic Sugar*

If *C* and *D* are domains then $C \to D$ is the *function domain* of functions from *C* to *D*.

If *f* is a function in domain $C \to D$ and *x* is in domain C then the *application* of *f* to *x* is written by juxtaposition with a separating white space as in *f x*. It denotes the value of *f* in *D* when the function is applied to the argument *x*.

Lambda-calculus supports *high order* functions, that is functions whose values are themselves functions. For example if *C*, *D* and *E* are domains then $C \to D \to E$ is a domain. The → operator associates to the right, that is $C \to D \to E$ is equivalent to $C \to (D \to E)$. If f is in domain $C \to D \to E$, *x* is in domain *C* and *y* is in domain *D* then *f x y* denote the application of function *f x* to *y*. That is application associates to the left. Parentheses can be used to specify a different order in the usual manner.

While lambda-calculus uses single letter variable names this notation uses multiple alphanumeric characters names to make them descriptive, like is customary in programming languages. This is why when application doesn't use explicit parenthesis whites space must be used to separate variable in a juxtaposition denoting application, e.g. *f*(*x*) or *f x* must be used and not *fx* to distinguish this application from the long variable name *fx*.

Let *M* be a lambda-expression which may or may not involve the variable *x*. Then $\lambda x.M$ is the *abstraction* of *x* from *M*. It denotes the function that takes a value as a parameter and return the value *M* evaluates to when *x* assumes this parameter value. In the case *M* does not involve the variable *x* the function will be a *constant function* which always evaluates to the same value.

The semicolon is the *composition* operator, e.g. *f*;*g* means $\lambda x.g(f(x))$. Please note the order. The first function listed is the first one applied. Composition is associative, that is *f*;*g*;*h* = (*f*;*g*);*h* = *f*;(*g*;*h*). Also application has a higher priority than composition, that is *f*;*g x* = *f*;(*g x*) and *f x*;*g* = (*f x*); *g*.

The *let* construct assigns a temporary name to a value for use in a lambda-calculus expression *M*. The syntax is *let* (*x*=*v*)*M* and it is synonymous with $(\lambda x.M)(v)$.

Definitions are allowed to list their parameters on the left-hand side. For example, assuming *M* is some lambda-calculus expression one may write *f*(*x*) = *M* instead of *f* = $\lambda x.M$.

*Cartesian Products and Currying*

Let *C* and *D* be domains, then the Cartesian product of *C* and *D* is a domain written $C \times D$.

The members of $C \times D$ are ordered pairs $\langle a, b \rangle$ where *a* is in domain *C* and *b* is in domain *D*.

Cartesian products may be iterated *n* times for any finite $n \geq 1$. For example $C \times D \times E$ is a Cartesian product of three domains *C*, *D* and *E*.

Members of Cartesian products are tuples, denoted by comma separated lists within parenthesis. For example $(a, b, c)$ represents a triple whose values are respectively $a$, $b$ and $c$. If this triple is in the Cartesian product $A \times B \times C$ then $a$ is in $A$, $b$ is in $B$ and $c$ is in $C$.

The projection of the $i^{th}$ element of a tuple of size $n+1$, $0 \leq i \leq n$ is written as $\pi_{i,n}$. For instance assuming $t=\langle a, b, c\rangle$ then $\pi_{0,2}(t)=a$, $\pi_{1,2}(t)=b$ and $\pi_{2,2}(t)=c$.

The substitution of a value for the $i^{th}$ element of a tuple of size $n+1$, $0 \leq i \leq n$ is written as $Upd_{i,n}$. This is also known as an "update" operation, hence the notation. For instance, assuming $t=\langle a, b, c\rangle$ then $Upd_{0,2}(t, z) = (z, b, c)$, $Upd_{1,2}(t, z) = (a, z, c)$ and $Upd_{2,2}(t, z) = (a, b, z)$.

In ordinary lambda-calculus functions depending on multiple parameters are *curried*, which means each parameter is a separate abstraction that must be applied in succession. This is the ordinary addition $x + y$ of two natural numbers would be written $Add\ x\ y$ where $Add$ is a curried version of '+', i.e. $Add = \lambda x \lambda y.x + y$.

As a matter of syntactic sugar, this notation allows the use of functions in uncurried form. Then the expectation is that the function expect a single parameter which belong to a Cartesian product domain. Definitions with parameters identified on the left-hand side are allowed in both curried and uncurried form. For example $f(x)(y)= M$ means $f = \lambda x \lambda y.M$ and $f(x, y) = M$ means $f = \lambda z.(\lambda x \lambda y.M)(\pi_{0,1}(z))(\pi_{1,1}(z))$.

When the same domain is involved in a Cartesian product with itself and exponent notation is used to describe the product. For instance $B \times B$ may be written $B^2$ and similarly $B \times \ldots B$ with n occurrences of $B$ may be written $B^n$.

The domain $\boldsymbol{B}$ denotes the Cartesian product of eight bits $Bool^8$, or the domain of bytes,

*Disjoint Unions*

Let $C$ and $D$ be domains, then the disjoint union of $C$ and $D$ is a domain written $C + D$.

The members of $C + D$ are either members $a$ of domain $C$ or members $b$ in domain $D$. The union is called disjoint because when a domain is united with itself, as in $C + C$, we distinguish between the members $a$ of the left-hand side domain $C$ from members $b$ in the right-hand side domain $C$.

Disjoint unions may be iterated $n$ times for any finite $n \geq 1$. For example $C + D + E$ is a disjoint union of three domains $C$, $D$ and $E$.

Members of disjoint unions are members from the individual domains labeled with which of the domain they come from. If $U$ is a disjoint union of $n$ domains $D_0 + \ldots D_n$ then

- $which(x)$ is a function in domain $U \rightarrow \boldsymbol{n}$ identifying for every $x$ in $U$ the index number of the individual domain in the union the value of $x$ comes from. Please note that the indexes start counting at 0.
- $inj_i(x)$ is a function $D_i \rightarrow U$ which return the value in $U$ corresponding to $x$ in $D_i$. This function is called the *injection* function from $D_i$ into $U$.
- $sel_i(x)$ is a function $U \rightarrow D_i$ which return the value in $D_i$ that was originally injected in $U$. If $x$ wasn't originally injected from $D_i$ that is it came from $D_j$ with $j \neq i$, then $sel_i(x)$ is $\perp_{Di}$ where $\perp_{Di}$ denotes the bottom element of the domain $D_i$ according to domain theory. This function is called the selection from $U$ into $D_i$.

Selection and injection are (almost) inverse operations. The equation $x = sel_i(inj_i(x))$ holds and the other equation $x = inj_i(sel_i(x))$ holds whenever $which(x) = i$.

The result of a selection operation cannot be tested for $\perp_{Di}$ because such a test function cannot be defined in lambda-calculus.[4] It is best to test for the appropriate domain with *which* before doing a selection with $sel_i$.

As a matter of syntactic sugar, the injection and selection operators are omitted from the notation when the context unambiguously identify which $inj_i$ or $sel_i$ operator should be used.

As a matter of syntactic sugar whenever the domain $D$ occurs only once in a disjoint union, the function $isD$ in domain $U \rightarrow Bool$ stands for a test of the which operator corresponding to $D$, that is $isD(x) = (which(x)=i)$

where $i$ is the index of $D$ in the disjoint union. For example if $U = \boldsymbol{N} + \boldsymbol{None}$ then $isNone(x)$ is *True* when $which(x)$ is 1 and *False* otherwise.

*Recursive Domains and Recursion*

Domains may be defined by means of recursive equations using previously defined domain and the operators $\rightarrow$, $\times$ and $+$. See [Stoy 1981] pp. 138-141. For example $L = (\boldsymbol{N} \times L) + \boldsymbol{None}$ defines a domain containing the lists of natural numbers. The value *None* represents the empty list and a pair $\langle n, r \rangle$ represents a list where $n$ is the first number in the list and $r$ is the rest of the list.

The **Y** combinator is the lambda-calculus expression satisfying the fixed-point equation $f = \mathbf{Y}\ f$. This expression is written $\lambda f.(\lambda x.f(x\ x))(\lambda x.f(x\ x))$. Generally speaking if $f$ belong to the domain $D \rightarrow D$ then $\mathbf{Y}\ f$ belongs to the domain $D$.

The **Y** combinator is used to define recursion. For example the *Double* function which doubles every member of a list of natural numbers may be written:

$Double = \mathbf{Y}\ \lambda f.\lambda x.$(if $isNone(x)$ then *None* else $(2*\pi_{0,2}(x),\ f(\pi_{0,2}(x)))$ )

Generally speaking when $f$ belongs to the domain $(C \rightarrow D) \rightarrow (C \rightarrow D)$ then $\mathbf{Y}\ f$ is a recursively defined function in the domain $C \rightarrow D$. As a matter of syntactic sugar, recursive definitions may be written as recursive calls to the defined function. For example let's suppose that $M[f, x]$ is an expression involving $f$ and $x$ such that $\lambda x.M[f, x]$ belongs to domain $C \rightarrow D$, then $f(x) = M[f, x]$ means $f = \mathbf{Y}\ \lambda g.\lambda x.M[g, x]$ . Using this convention the definition of *Double* may be written as:

$Double(x)$ = if $isNone(x)$ then *None* else $(2*\pi_{0,2}(x),\ Double(\pi_{0,2}(x)))$ )

**The Steps to Define the Semantics**

Let's refresh our memory of the task at hand. One possible way to define an abstract machine is done by starting with a complete description of a computer and removing all references to physical elements until we are left only with a description of the information in the abstract. For example a flag is not a flag, it is a bit of boolean data. A register is not a register, it is a sequence of boolean data. An instruction is not affecting registers and flags. It is a mathematical operation on boolean data. When we are done with this removal process no reference to anything physical is left. We have a formal mathematically defined abstract machine that operates on mathematical information. To carry out this formalization with accuracy would require a complete documentation of the compute hardware including the CPU and the hardware interfaces to the peripherals.

It is equally possible to specify the abstract machine beforehand and then build a computer to match this specification.

For the sake of readability I use an unusual convention. Language referring to physical components as if they were mathematical entities should be understood as referring to a corresponding abstract mathematical entity. For example language similar to “define (some particular) computer hardware mathematically” actually mean “define the abstract mathematical device that corresponds to (some particular) computer hardware”. Without this convention the repeated use of phrases like “abstract mathematical device that corresponds to (some particular) computer hardware” will grow pedantic and tiresome. Besides this kind of language conveniently define the exact semantical correspondence between hardware and abstract device. This is in accordance with the goal of this paper which is to show how to define an operational/denotational semantics for the hardware.

You actually define the abstract machine by carrying out the following steps.

1. Make an inventory of all streams of input and output and define the corresponding mathematical domains.
2. Make an inventory of all the places where information is stored and define the corresponding mathematical domains. Then define the domain for the machine states.
3. Make an inventory of all operations changing the state of the machine of and define the

corresponding mathematical transitions.
4. Identify the loops that execute the machine transitions and define the corresponding mathematical formulas for the body of these loops.
5. Define what is a run of a program on the abstract machine.
6. (Abstract networks only) Connect the inputs and outputs of abstract machines to define the corresponding abstract network.

Step no 6 is carried out only when we need a mathematical definition of a abstract network corresponding to a network of computers.

This is a definition of the computation that doesn't consider some details of the hardware which are transparent to the software. For example it is assumed the CPUs access memory directly. If there is a cache or even multiple level of caches in the CPU the mathematical definition of the abstract machine shows no sign of it. CPU features such as pipelining that are transparent to the definition of the instructions are not considered. The specifics of memory bus arbitration are absent beyond the basic assumption that CPUs and DMA peripherals access the memory one CPU at the time. Such hardware features are transparent to software in the sense that they don't affect the computation resulting from the execution of the machine instructions. Their only effect is speedier execution of the same instructions resulting into a faster but otherwise identical computation. Therefore they don't need to be included in a mathematical definition of the computation.

Let's review the five (or six) steps in details.

**Step 1**: *Make an inventory of all streams of input and output and define the corresponding mathematical domains.*

This step is about defining where are the boundaries of the abstract machine. Input is information that comes from the outside into the abstract machine. Output is information that goes from the abstract machine to the outside. The boundaries are the places where the abstract machine connects to the outside. There are two kinds of boundaries to consider. The first kind corresponds to signals that are either received (input) or emitted (output) over physical connections when the machine is actually implemented in the real world. The second kind is nondeterministic events that may affect the computation. In the notation proposed in this paper nondeterminism is treated as a source of input to an otherwise deterministic computation.

*Signals*

Let's consider the signals first. They typically correspond to often some kind of digital or analog interface that defines each hardware boundary. The task is to identify all such interfaces and draw mathematical boundaries for the abstract machine that are meaningful in terms of the hardware machine transitions that manipulate the information. If we draw the boundaries at places that are not meaningful to the machine transitions they will be awkward to define mathematically when we get to step 3.

For example is the keyboard part of a generic computer? A typical CPU knows no instruction for directly handling keyboards. It may have access to some memory mapped input port and I/O register to interact with the keyboard. It uses some I/O specific instructions to manipulate the ports and registers. You can use these instructions to write drivers for the keyboard. Drivers are software. It is the individual instructions that will be included in the abstract machine transitions. Therefore the boundaries of the computing device should be the input port and register for the keyboard interface. The keyboard itself is an independent device. As a rule of thumb *it is usually best to put the boundaries at the I/O ports and registers and leave the peripheral devices outside the abstract device*.

It is possible to define an abstract machine that performs the computational task of a peripheral. A disk controller manages the storage on disk and has a corresponding abstract machine that manages the information. A keyboard has a corresponding abstract machine that sends signals over a wire in response to key presses. The peripherals connected to the central computing unit may be viewed as a network. Then the complete computer, peripherals included, can be translated into a abstract network using the techniques that will be discussed when we get to step 6. This implies that buses like USB ans SCSI are treated mathematically like networks, but this is also true of point to point connections like serial lines. But this analysis is not what is being done in this step 1. I mention it only to clarify that putting the boundary of the computation device at the I/O ports and registers is not overly limiting. It is possible to give a mathematical semantics to a computer complete with its peripherals.

*Nondeterministic Events*

The other kind of boundaries to the abstract machine is the nondeterministic operations which influence the computation. They are treated as inputs to the formulas defining the computation. We could define a probabilistic abstract machine which from time to time makes a probabilistic choice in the course of the computation. This would qualify as a mathematical algorithm of the probabilistic kind. But probabilistic choice is not part of lambda-calculus[5] and we can't use this language to define this sort of algorithms. The solution used in this paper is to treat any source of nondeterministic events as something external to the algorithm which produces data to be supplied as input to the computation. Then the algorithm is defined in a deterministic manner *relative* to these external sources.

Assuming we extent lambda-calculus with the capability of making probabilistic choice, whenever the notation reads a tape used as a source of nondeterministic events we may replace it with a probabilistic choice of some untold distribution. Both formalisms are mathematically equivalent.

Example of nondeterministic events is the choice of which CPU access main memory next in a symmetric multiprocessing system (SMP) and, the moment when power is turned off.

*A Random Number Generator as an Example of an Input Tape*

Let's take a look at how to define mathematically input. For the sake of making a simple example, consider a hardware random number generator made of an analog to digital converter that measures the thermal noise on an otherwise unused circuit. Analog to digital conversion is input. The input is always provided on request from the computer. The random number generator never takes the initiative to supply any data. The data is unpredictable and introduces an element of nondeterminism in the computation. This is mathematically represented by an abstract “tape.”

A random number generator will produce a sequence of number $n_0$, $n_1$, $n_2$, etc. I use numeric indices to reflect the fact that one random number comes first, then another, and another etc because the random number generator is called at multiple occurrences in time. This sequence is potentially infinite if we consider that the computer may run into an infinite loop. Mathematically speaking, this sequence is the tape. When the computer executes the instruction that reads a random number from the input port, it corresponds to a transition in the abstract machine where the tape is read to find the number $n_i$ for the next index in the sequence. We provide unpredictability and nondeterminism by requiring the abstract machine to obey two rules when using the tape.

1. The tape as a whole is an input parameter to the formulas defining the abstract machine. Every new run of the machine uses a different sequence of numbers because it is assumed a new tape parameter is being supplied.
2. Also the abstract machine never asks the tape for $n_i$ twice for the same $i$. The abstract machine must request the values in order starting with $n_0$ then $n_1$ and so on respecting the sequence. Then every tape read request is guaranteed to return an independent and usually different value.

This technique replicates the behavior of a random number generator without making an explicit use of a probabilistic choice. Unpredictability and nondeterminism is hidden by the fact the tape is not defined from within the abstract machine. It is provided separately as a parameter.

The tape technique will work regardless whether the tape follows a perfectly random statistical distribution, a distribution that has a bias or the distribution for a completely broken generator that always picks the same number. In all scenarios we may assume the tape content will reflect the distribution. The abstract machine does not contain any reference to probability. Such reference is implicit because the abstract machine is defined *relative* to the tape and this definition works the same for all tapes.

The tape technique also works for any external source of sequential information regardless whether it is deterministic or not. All external sources generate a similar sequence of values $n_0$, $n_1$, $n_2$, etc that is passed as an input parameter to the formulas defining the abstract machine.

*Outputs and Networking*

Tapes may be used in three flavors: input, output and networked. We have just seen how an input tape works. The source of the input is assumed to be outside of the computation and is not defined by the formulas for the abstract machine. The tape is assumed to contain a record of all the information that will be produced by the external source and the abstract machine reads the tape when it gets to it.

Output is treated symmetrically. A sequence of values $n_0, n_1, n_2,$ etc is generated and emitted by the abstract machine. The same tape abstraction is being used except that for outputs the tape is written by the abstract machine instead of being read. The recipient of the information is outside of the computation and is not defined by the formulas for the abstract machine. At the end of the computation the output tape contains a record of all the information generated by the abstract machine. If the computation runs into an infinite loop an infinite quantity of information will be written on the tape.

Bidirectional communication is a pair of tapes going in opposite direction. One input tape is read for incoming data while the other tape is an output tape for the outgoing data.

Networked tape is when both the reading and writing ends of the tape are part of the computation. There may be one abstract machine writing the tape and another reading from the tape in the same abstract network. The writing machine generates a sequence of values $n_0, n_1, n_2,$ etc which is read in the same order by the reading machine. At the end of the computation the tape contains a record of all the information that has been written. This may be an infinite quantity of information in the case of an infinite loop.

*Storage*

There are physical devices that can store information and let the computer retrieve it at a later time. For example a hard disk connected to a SCSI bus. This information is not included in the definition of the abstract machine because they are stored in peripherals. The proposed abstraction in such situation is an abstract network. Both the computer and hard disk are abstract machines that communicate over an abstract communication link. The abstract computer writes I/O requests on a network “tape”. The abstract disk reads this tape as input and writes answers on another network tape which will be read by the computer. In other words there are two tapes. One that moves data from the abstract computer to the abstract hard disk and one in the other direction. How to model this mathematically will be discussed in step 6.

*The Domains for Tapes*

A tape is mathematically a sequence $n_0, n_1, n_2,$ etc of elements from a domain $D$. Associated with the sequence there will be two counters that keep track of where we are in the reading and writing sequences to ensure the abstract machine respect the constraints of the tape metaphor. When the abstract machine reads from an input tape, it asks for the element $n_c$ in the sequence that matches the current value of the read counter $c$. Then the counter must be incremented so the next time the machine reads the next value is read. Similarly for an output tape a write counter keeps track of where in the sequence the next value will be written. Again this counter must be incremented after each write. A network tape will use both a read and a write counters with the additional constraint that the read counter is never allowed to get past the already written portion of the tape.

The mathematical domain for the counter is **N**. This is the natural numbers 0, 1 ,2 etc to infinity. The mathematical domain for the tape is $\mathbf{N} \rightarrow D + \textbf{\textit{None}}$ where $D$ stands for whatever domain is applicable to an individual element in the sequence. The value *None* is a fictitious value used to represent the lack of information. Reading a blank tape or attempts to read a yet unwritten portion of the tape will return *None*.

The value *None* is used, for instance, to help define the computation that occurs when a device driver attempts to read data that hasn't been yet supplied by a external source. For example consider a keyboard. Data can be read from a keyboard only when someone presses a key. What happens if a buggy keyboard driver tries to read a key stroke from the input port of the keyboard when the user hasn't pressed any key? It depends on what the port circuitry is designed to do in such circumstances. It may hang until the user presses a key and then proceed with the input. It may trigger a hardware exception. It may return some unpredictable data. The mathematical semantics of this circuitry has to provide for this behavior[6]. In this notation this is done by including in the formula for the semantics of I/O instructions a test of whether the value being returned is *None* and then make the calculation proceed accordingly.

Tapes are tuples defined in the following domains:

***ReadCounter*** = **N**
***WriteCounter*** = **N**

***InputTapeD*** = (**N** → (*D* + ***None***)) x ***ReadCounter***
***OutputTapeD*** = (**N** → (*D* + ***None***)) x ***WriteCounter***
***NetworkTapeD*** = (**N** → (*D* + ***None***)) x ***ReadCounter*** x ***WriteCounter***

In these definitions the first element is the sequence and the other elements are the counters. The domain *D* is whatever domain correspond to the individual values being read and/or written. A similar set of domains must be defined for each different domain *D* of values recorded on tapes.

A generic domain for all three types of tapes is:

***TapesD*** = ***InputTapeD*** + ***OutputTapeD*** + ***NetworkTapeD***

One may test which kind of tape with the functions *IsInputTapeD*(*t*), *IsOutputTapeD*(*t*), *IsNetworkTape*(D*t*) which are defined according to the previously mentioned syntactic sugar convention regarding the use of the *which* function.

A blank network tape is the triple (λ*x*.*None*, 0, 0). A blank input or output tape is the pair (λ*x*.*None*, 0).

The exact formulas for reading and writing on tapes will be given later.

*Examples of What Needs to be Inventoried*

The inventory of inputs and outputs must be completely exhaustive. It must include interrupt signals. The CPU constantly monitors the interrupt lines. When the interrupt signal arrives it temporarily suspend the execution of whatever routine is currently executing and transfers control to an interrupt handling routine. The abstract machine must reproduce this behavior. This requires a (**N** → *Bool* + ***None***) for each interrupt signal.

Elements of the computer architecture which are undefined should be treated as nondeterminstic events. Therefore a tape input must be provided for each of them. For example on some CPU some instructions leave some flags in undefined states. This means the vendor makes no commitment as to what values the flags will hold and warns the programmers not to rely on these flags after such instructions are executed. How do we define mathematically “undefined”? This is treated like a random number generator. If the abstract machine reads the flag values from an input tape it will make them the mathematical equivalent of “undefined”.

Another example of this kind of tapes occurs when multiple piece of circuitry have access to shared memory. For example in SMP systems where there are several CPUs running concurrently. Or when the CPU shares memory access with DMA peripherals. In these circumstances there is a need to determine which of the components has the next access to memory. This decision depends on the timing of the activity of the components and, in the case of conflicting timing, on a possibly nondeterminsitic decision of some arbitration circuit. In this notation we eschew the difficulty by using an input tape to indicate which component will access memory next.

**Step 2**: *Make an inventory of all the places where information is stored and define the corresponding mathematical domains. Then define the domain for the machine states.*

Mathematical domains are abstractions[7] for the information held in storage. A computer stores bits in many of its components. Each CPU flag contains one bit. A register will contain 16, 32, 64 or whatever number of bits that makes the size of the register. Memory will contain a number of bytes. There are bits in I/O registers and graphics memory. The mathematical semantics of the computer must include a mathematical representation of every bit stored in the computer. This implies to make an inventory of every places where the computer stores bits and find the corresponding mathematical domain.

The domain that corresponds to a single bit, say a CPU flag, is *Bool.*

The domain that corresponds to a byte is $Bool^{8}$, ie a Cartesian product of eight bits. this domain is also denoted by $\boldsymbol{B}$. Similarly $Bool^{16}$, $Bool^{32}$, $Bool^{64}$, corresponds to words of 16, 32 and 64 bits respectively.

The domain that corresponds to 1 kilobyte of data is $\boldsymbol{B}^{1024}$.

Once we have inventoried all the locations where data is stored it is possible to define a mathematical domain for the possible machine states. This is defined as the sum (actually a Cartesian product) of all information stored in the computer as well as the current states of the tapes. A abstract machine operates by performing mathematically defined transitions on this state. These transitions are the mathematical semantics of various actions taken by the computer on the information.

The definition of the mathematical domain for states is a prerequisite to step 3 where will make an inventory of all these actions and define the corresponding transitions.

We have an inventory of all the places where information is stored in the computer. Let's say there are $i$ such places. This gives us $i$ mathematical domains called $M_0$ to $M_{i-1}$. We also have an inventory of all tapes. Let's say there are $j$ of them. This gives us $j$ mathematical domains called $Tape_0$ to $Tape_{j-1}$. These domains are aggregated in a Cartesian product:

$M_0 \times ... M_{i-1} \times Tape_0 \times ... \ Tape_{j-1}$

For reasons that will be apparent during step 5, we add to these mathematical domain a domain called ***Status*** which holds three possible values: *Running* and *Stopped*. This status domain will be used to define what is a run of a program on the abstract machine and also will be helpful in defining a computation run on an abstract network of abstract machines.

Adding the ***Status*** domain to the Cartesian product gives us the domain **S** of abstract machine states:

$\mathbf{S} = \boldsymbol{Status} \times M_0 \times ... M_{i-1} \times Tape_0 \times ... \ Tape_{j-1}$

**Step 3**: *Make an inventory of all operations changing the state of the machine of and define the corresponding mathematical transitions.*

The task in step 3 is to make an inventory of all possible operations a computer may perform on its state and define their semantics in terms of mathematical transition.

The operations are actions taken by the hardware that either change the information in memory, reads input or write output. An obvious example of such operations are the instructions recognized by the CPU. But the list of CPU instructions is by no means the complete list of computer operations. Some of the operations are not instructions. For example interrupts will cause the CPU to jump to the interrupt handling routine but this action is not an instruction stored in memory. Some operations are not performed by the CPU at all. They may be the instructions for a co-processor, a GPU or the activity of an I/O device with DMA capability. Any component capable of independent activity will have corresponding operations that must be inventoried in this step 3.

The operations must be atomic in the sense that when started they are always performed through completion without being interrupted for the execution of intervening work. Some operations could be large. They may be the equivalent of a subroutine but hard coded in hardware. Then the question is could they be interrupted mid execution? Non atomic operations must be broken down into atomic components. The reason is the abstract machine needs to reflect the actual computation done by the computer. For example it has to ask tapes for interrupts in order to reflect that aspect of the computation. In step 4 we will discuss how this is done. It happens that the mathematical treatment of the transitions assumes they are atomic and non-interruptible. Therefore the transitions must corresponds to atomic and non-interruptible computer operations.

A similar question will pertain to DMA transfers. Do they seize exclusive access to memory for the duration of the transfer? Or are other components allowed to interact with memory while the transfer occurs? In the latter case the DMA transfer is not atomic and should be broken down into smaller atomic transfers.

*Giving the Notation a Programmatic Feel*

The notation allows to define the transitions in a manner that is similar to what one would write in a programming language. This is by design. This feature makes it easy to keep track of the correspondence between the physical machine and its mathematical semantics.

We give each component of this Cartesian product a mnemonic name. For example the instruction pointer may be called *IP* and the stack pointer *SP*. These mnemonics are given a numeric code, as in:

*IP* = 0
*SP* = 1
etc ...

This enables us to define a *ValueOf1*(*S*) in the domain **S** → **N** → ($M_0$ + ... $M_{i-1}$ + $Tape_0$ + ... $Tape_{j-1}$) as follow:

*ValueOf1*(*S*)(*Element*) = If *Element* = 0 then $\pi_{0,i+j-1}(S)$
else if *Element* = 1 then $\pi_{1,i+j-1}(S)$
. . .
else $\pi_{i+j-1,i+j-1}(S)$

The function *ValueOf1*(*S*)(*Element*) returns the value of the element in the state *S*.

Another useful function is *Upd1* which allows to change a state into another state on an element-wise manner. Assuming that n is the mnemonic

*Upd1*(*Element*, *newValue*)(*S*) = If *Element* = 0 then $Upd_{0,i+j-1}$(*S, newValue*)
else if *Element* = 1 then $Upd_{1,i+-1j}$(*S, newValue*)
. . .
else $Upd_{i+j-1,i+j-1}$(*S, newValue*)

Notice that the state parameter *S* is curried in this definition of *Upd1*. The enable to use the composition operator, the semicolon, to aggregate change of states on individual elements into a larger composite transition. For example and operation of swapping the values of two registers *r1* and *r2* may be written as:

*Swap*(*r1*, *r2*)(*S*) = *Upd1*(*r1*, *S*(*r2*)); *Upd1*(*r2*, *S*(*r1*))

or equivalently:

*Swap*(*r1*, *r2*)(*S*) = (*Upd1*(*r1*, *S*(*r2*));
*Upd1*(*r2*, *S*(*r1*))
)

The intent is to treat the *Upd1* function as a form of assignment statement. *Upd1*(*r1*, *value*) means the element *r1* is assigned the value in the state. This is much like the C language statement *r1* = *value*. But there is a difference with C in that the state being updated is given an explicit name which is being referenced in the definition of the *Swap* transition. The function call *Upd1*(*r2*, *S*(*r1*)) assigns to *r2* the value *r1* has in the original state *S* because the name *S* is referenced. This is different from the pair of C assignments *r1* = *r2*; *r2* = r1 which would result into *r2* retaining it original value. So while the notation has a programmatic feel the fact that the state is explicit means this definition is a mathematical expression and not a programming language instruction.

Assuming that *F* and *G* are expressions denoting functions in domain **S** → **S** and test is an expression evaluating to a value in domain *Bool*, and that *test*, *F* and *G* are allowed to depend on a variable *S* ranging over states in **S**; then these two idioms may be used in a composition.

- (If *test* then *F*)(*S*) = if test(*S*) then *F*(*S*) else *S*

  This idiom can be used in a formula like (If *S*(*Zflag*) then *Upd1*(*x*, v)) which means update only when

the flag Z is set otherwise do nothing. It is assumed that *Zflag* is the mnemonic name that locates the flag Z in the state.

- (If *test* then *F* else *G*)(*S*) = if *test*(*S*) then *F*(*S*) else G(*S*)

  This idiom gives a choice of which function *F* or *G* is included in the composition depending on the condition.

This if then else conditional composition can be used, for example, to swap the content of three registers in such a way that the content of *r1* is the smallest number, then *r2* holds the middle number and *r3* the largest number.

```
Order3(r1, r2, r3)(S) = (If (S(r1) < S(r2))
                            then (if (S(r1) < S(r3))
                                     then ( if (S(r3) < S(r2))
                                               then Upd1(r2, S(r3));
                                                    Upd1(r3, S(r2))
                                            )
                                     else ( Upd1(r1, S(r3));
                                            Upd1(r2, S(r1));
                                            Upd1(r3, S(r2))
                                          )
                                  )
                             else (if (S(r2) < S(r3))
                                     then (if (S(r3) < S(r1))
                                              then (Upd1(r1, S(r2));
                                                    Upd1(r2, S(r3));
                                                    Upd1(r3, S(r1))
                                                   )
                                              else ( Upd1(r1, S(r2));
                                                     Upd1(r2, S(r1))
                                                   )
                                  )
                         )(S)
```

*Order3* is just an example of what we can do with this notation[8].

With large strings of compositions like this we need to spread the formula on multiple lines and indent for readability. A noted before this style of writing mathematical formulas looks very much like code in an imperative programming language. The *Upd1* function is similar to an assignment and the semicolon looks like a statement separator. The “If then else” construct looks like conditional statements. Composable functions like *Swap* work like procedure calls.

This is not accidental. I have borrowed this technique from denotational semantics which is a method for defining mathematically the semantics of source code.[9] The trick is that carefully crafted high order functions when composed behave in a similar manner as the statements of programming languages. If you define the mathematical domain correctly then specially crafted high order functions will have the same semantics as programming language statements except that they define transitions on mathematically defined states and not operations on a physical machine.

We bring Turing-completeness to the notation by introduction the while loop construct:

*While*(*test*, *body*)(S) = if *test*(S) then (*body*; *While*(*test*, *body*))(S) else S

Here *test* must be an expression which evaluates to a value in domain *Bool* and body be a function in $D \rightarrow D$ for some domain *D*. This expression keeps iterating *body* until *test*(S) is *False*. Then whatever value S turned out to become after the repeated of *body* is the returned value.

*Memory Access and Updates*

There is an isomorphism between $D^n$. and $\boldsymbol{n} \to D$. There is a one-to-one correspondence between the tuples $t$ in $D^n$ and the functions $f$ in $\boldsymbol{n} \to D$ such that $f(i)=\pi_{i,n}(t)$ for each pair of matching $t$ and $f$. It is advantageous to use this isomorphism to represent in a function domain instead of a Cartesian product information that has a number of more granular elements. Main memory is a good candidate. This simplifies the notation for memory updates. The reason is the projection functions $\pi_{i,n}$ are a discrete arrays of functions in the dialect of lambda-calculus this notation is using. In order to define a uniform function that takes integers $i$ as parameters and produce a corresponding $\pi_{i,n}$ in $D^n \to D$ one would need a huge cascade of "if" expressions with one if for each value of $i$. A similar observations hold for the $Upd_{i,n}$. While such definitions are possible they are overly bulky and inconvenient to handle.

This problem does not occur when one uses the domain $\boldsymbol{n} \to D$. For example if we use the domain $\boldsymbol{n} \to \boldsymbol{B}$ for main memory, then $Memory(S)(i)$ is a uniform function that takes an integer i, $0 \le i \le n$, as a parameter and produces the same value as the discrete array of functions $\pi_{i,n}(Memory(S))$ would have produced had the domain $\boldsymbol{B}^n$ been used.

Also assuming $M$ is in $\boldsymbol{n} \to \boldsymbol{B}$ then $Upd1(i, x)(M) = \lambda y.$if $y=i$ then $x$ else $M(j)$. This is the same as $Upd_{i,n}(M, x)$ had $M$ be in $D^n$ except that in this case $Upd1$ is a function accepting $i$ as a parameter in $\boldsymbol{n}$ instead of being an array of discrete functions $Upd_{i,n}$.

Please note that $Upd1$ is overloaded, meaning that the name is reused for different definitions depending on the domain of its last parameter. When S is in $D^n$ then $Upd1(Element, x)(S)$ is in domain $(\boldsymbol{n} \times D) \to D^n \to D^n$ while when $M$ is in $(\boldsymbol{n} \to D)$ then $Upd1(i, x)(M)$ is in domain $(\boldsymbol{n} \times D) \to (\boldsymbol{n} \to D) \to (\boldsymbol{n} \to D)$.

The function $Upd2$ assigns a value to an element within an element. For example $Upd2(Memory, x, v)(S)$ means to assign value $v$ to the memory cell $x$ within the $Memory$ element of state $S$. Assuming the element is in domain $\boldsymbol{n} \to D$ the domain of $Upd2$ is $((\boldsymbol{n} \times \boldsymbol{n} \times D) \to \mathbf{S} \to \mathbf{S}$ and its definition is:

$$Upd2(Element, item, value)(S) = \text{Let } (x = Upd1(item, value)(\text{S(Element)}) \text{ )}\\ Upd1(Element, x)(\text{S})$$

Finally we combine the functions $Upd1$ and $Upd2$ into a single function $Upd$. The first argument will be either an element mnemonic for element that are modified as a whole or a pair of a mnemonic and and index within the element to be modified. For example $Upd(SP, x)(S)$ means update the stack point to value $x$ while $Upd(\ (Memory, i), x)(S)$ means update location $i$ within main memory to value $x$. The domain for $Upd$ is $(\ (\mathbf{N} + (\mathbf{N} \times \mathbf{N})) \times D) \to \mathbf{S} \to \mathbf{S}$ and its definition is:

$$Upd(address, value)(S) = \text{if } which(address) = 0 \text{ then } Upd1(address, value)(S)\\ \text{else } Upd2(\ \pi_{0,1}(address), \pi_{1,1}(address), value)(S)$$

The following function will sometimes be useful. It builds a pair suitable for use with $Upd$ out of a numeric address $x$ in memory.

$$address(x) = (Memory, x)$$

We also define a function $ValueOf(S)(address)$ which takes a state and returns the value corresponding to an address within this state. This function is in domain $\mathbf{S} \to (\mathbf{N} + (\mathbf{N} \times \mathbf{N})) \to D$ where $D$ is the domain of the element stored at this address.

$$ValueOf(S)(address) = \text{if } which(address) = 0 \text{ then } ValueOf1(S)(address)\\ \text{else } ValueOf1(S)(\pi_{0,1}(address))(\pi_{1,1}(address))$$

As a matter of syntactic sugar we write $S(address)$ for $ValueOf(S)(address)$. This convention is justifiable because the domain of states is clearly not a function domain. The application of a state as if it were a function will always unambiguously denote an implicit use of $ValueOf$.

Two more operations on Cartesian products are useful: slice and concatenation. Slice is the operation of extracting a section of the elements from the tuple. Slice is written with square brackets enclosing the range. For example ⟨a, g, h, i, b, c⟩[1:3] = ⟨g, h, i⟩. Concatenation is gluing two tuples together to make a bigger tuple. Concatenation is written with a + sign. For example ⟨a, b⟩ + ⟨c, d⟩ = ⟨a, b, c, d⟩.

Slices and concatenation are used to express mathematical operations on word size larger then a byte and make them fit with byte size memory. They enable to define mathematical operations like:

*Read08*(S, *x*) = S(*address*(*x*))
*Read16*(S, *x*) = S(*address*(*x*)) + S(*address*(x+1))
*Read32*(S, *x*) = S(*address*(*x*)) + S(*address*(x+1)) + S(*address*(x+2)) + S(*address*(x+3))

*Write08*(*x*, *v*)(S) = *Upd*(*address*(*x*), *v*)(S)
*Write16*(*x*, *v*)(S) = ( *Upd*(*address*(*x*), *v*[0:7]) ; *Upd*(*address*(*x*+1), *v*[8:15]) )(S)
*Write32*(*x*, *v*)(S) = ( *Upd*(*address*(*x*), *v*[0:7]) ; *Upd*(*address*(*x*+1), *v*[8:15]) ;
*Upd*(*address*(*x*+2), *v*[16:23]) ; *Upd*(*address*(*x*+3), *v*[24:31])
)(S)

Assuming Memory is the mnemonic for main memory *Readxx* reads the corresponding number of bits from memory and return these bits as the value of the function. *Writexx* writes the corresponding number of bits in memory. These definitions are valid for little endian architectures. For big endian systems we need to define like this instead:

*Read08*(S, *x*) = S(*address*(*x*))
*Read16*(S, *x*) = S(*address*(x+1)) + S(*address*(x))
*Read32*(S, *x*) = S(*address*(x+3)) + S(*address*(x+2)) + S(*address*(x+1)) + S(*address*(x))

*Write08*(*x*, *v*)(S) = *Upd*(*address*(*x*), *v*)(S)
*Write16*(*x*, *v*)(S) = ( *Upd*(*address*(*x*+1), *v*[8:15]) ; *Upd2Memory, x*, *v*[0:7]) )(S)
*Write32*(*x*, *v*)(S) = ( *Upd*(*address*(*x*+3), *v*[24:31]) ; *Upd*(*address*(*x*+2), *v*[16:23]) ;
*Upd*(*address*(*x*+1), *v*[8:15]) ; *Upd*(*address*(*x*), *v*[0:7])
)(S)

Note that the *WriteXX* functions have been curried. They can be composed with *Upd* using the semicolon operators.

It also useful to define a few stack related operations, *SP* being the stack pointer. These operations have the same endianess as the *Readxx* and *Writexx* they use.

*Push08*(*v*)(S) = ( *Upd*(*SP*, S(*SP*)-1) ; *Write08*(S(*SP*), v*)* )(S)
*Push16*(*v*)(S) = ( *Upd*(*SP*, S(*SP*)-2) ; *Write16*(S(*SP*), v*)* )(S)
*Push32*(*v*)(S) = ( *Upd*(*SP*, S(*SP*)-4) ; *Write32*(S(*SP*), v*)* )(S)

*Pop08*(*dest*)(S) = ( *Upd*(*dest*, *Read08*(S, *SP*); *Upd*(*SP*, S(*SP*)+1) )(S)
*Pop16*(*dest*)(S) = ( *Upd*(*dest*, *Read16*(S, *SP*); *Upd*(*SP*, S(*SP*)+2) )(S)
*Pop32*(*dest*)(S) = ( *Upd*(*dest*, *Read32*(S, *SP*); *Upd*(*SP*, S(*SP*)+4) )(S)

Note that this stack grows downward. For a computer where the stack grows upward use this instead:

*Push08*(*v*)(S) = ( *Upd*(*SP*, S(*SP*)+1) ; *Write08*(S(*SP*), v*)* )(S)
*Push16*(*v*)(S) = ( *Upd*(*SP*, S(*SP*)+2) ; *Write16*(S(*SP*), v*)* )(S)
*Push32*(*v*)(S) = ( *Upd*(*SP*, S(*SP*)+4) ; *Write32*(S(*SP*), v*)* )(S)

*Pop08*(*dest*)(S) = ( *Upd*(*dest*, *Read08*(S, *SP*); *Upd*(*SP*, S(*SP*)-1) )(S)
*Pop16*(*dest*)(S) = ( *Upd*(*dest*, *Read16*(S, *SP*); *Upd*(*SP*, S(*SP*)-2) )(S)
*Pop32*(*dest*)(S) = ( *Upd*(*dest*, *Read32*(S, *SP*); *Upd*(*SP*, S(*SP*)-4) )(S)

*Reading and Writing Tapes*

For convenience let's define a few primitives that clarifies the tape semantics. The primitives would be:

- *Sequence* which takes as parameters a state and the mnemonic for the tape and returns the sequence component of the tape. In symbol form this is the domain (**S × N**) → (**N** → (*D* + ***None***)) where *D* is the domain of the information items written on the tape.
- *readCounter* which takes as parameters a state and the mnemonic for a tape and returns the current value of the read counter if there is one, or 0 if there is none. This is the domain (**S × N**) → **N**.
- *writeCounter* wich returns the write counter for the tape but is otherwise identical to *readCounter*.

Only input and network tapes have read counters and only output and network tapes have write counters. Giving that tapes are in the domain ***TapesD*** = ***InputTapeD*** + ***OutputTapeD*** + ***NetworkTapeD*** then the *which* function identifies which kind of tape this is.

The three tape primitives are defined as follow:

*Sequence*(S, *tapeMnemonic*) = $\pi_{0,1}$(S( *tapeMnemonic* ))

*readCounter*(S, *tapeMnemonic*) = let (*t*=S( *tapeMnemonic* ))
( if *which*(*t*) = 0 then $\pi_{1,1}$(*t*)
else if *which*(*t*) = 2 then $\pi_{1,2}$(*t*)
else 0
)

*writeCounter*(S, *tapeMnemonic*) = let (*t*=S( *tapeMnemonic* ))
( if *which*(*t*) = 1 then $\pi_{1,1}$(*t*)
else if *which*(*t*) = 2 then $\pi_{2,2}$(*t*)
else 0
)

We use these primitives to define more explicit tape operations.

- *WriteTape* writes some information on an output or network tape identified by its mnemonic. It does nothing on input tapes. This is a function in domain (**N** × *D*) → **S** → **S** where *D* is the base domain for the items written on the tape.
- *ReadTape* reads the current content of an input or networked tape identified by its mnenomic. It returns *None* for an output tape. It also checks whether we get past the last written information on a network tape and returns *None* if this is the case. This is a function in domain (**S × N**) → *D* where *D* is the base domain for the items written on the tape.
- *MoveTape* increments the read counter of an input or networked tape identified by its mnenomic. This is equivalent to moving the reading head forward one position. It does nothing to an output tape. This is a function in domain **N** → **S** → **S**.

As a matter of convention we require that in machine states the read counter always point to the next element in the sequence to be read and likewise the write counter always point to the next element to be written. A write operation always increment the write counter to point to the next location. However the read operation is a function that returns the value at the current tape location. It doesn't affect the machine state. We require that every machine transition invoking *ReadTape* must immediately invoke *MoveTape* to comply with this convention.

The definition of the tape operations are:

*ReadTape*(S, *tapeMnemonic*) =
let (*t*=S(*tapeMnemonic*) )
( if *which*(*t*)=0 then $\pi_{0,1}$( *readCounter*(S, *tapeMnemonic*) )
else if *which*(*t*)=1 then *None*
else if *readCounter*(S, *tapeMnemonic*) >= *writeCounter*(S, *tapeMnemonic*) then *None*
else *readCounter*(S, *tapeMnemonic*)
)

The function call *which*(*t*) tells us whether the tape is input, output or network.

For input tapes this returns the value of the sequence at the current read counter location.
For output tapes this returns *None* because reading one is invalid and shouldn't happen.
For network tapes we first check whether the read counter is greater than equal to the write counter. If so then we are reading past the written part of the tape and *None* is returned. Otherwise the value of the sequence at the current read counter location is returned.

*MoveTape*(*tapeMnemonic*)(S) =
let (*t*=S(*tapeMnemonic*) )
( if *which*(*t*)=0 or ( *which*(*t*)=2 and
*readCounter*(S, *tapeMnemonic*) < *writeCounter*(S, *tapeMnemonic*) )
then *Upd*( ( *tapeMnemonic*, 1), *readCounter*(S, *tapeMnemonic*)+1)
)

This applies only to input or network tapes. It advances the position of the reading head within the machine state by incrementing the counter. This is done unconditionally for input tapes. For network tapes we must first verify the reading head will not get past the writing read by ensuring the read counter is smaller than the write counter. Nothing is done when the reading head is at the end of the written part of the tape.

*WriteTape*(*tapeMnemonic, x*)(S) =
let (*t*=S(*tapeMnemonic*) )
( if *which*(*t*)=1 or *which*(*t*)=2 then
let (*s*=*Upd*( *writeCounter*(S, *tapeMnemonic*), *x*)(*Sequence*(*S, tapeMnemonic*)) )
(*Upd*( ( *tapeMnemonic*, 0), *s*)
*Upd*( ( *tapeMnemonic*, 2), *writeCounter*(S, *tapeMnemonic*)+1)
)
)

This applies only to output and network tapes. The value *x* is first written on the current writing location on the tape and then the writing head is unconditionally advanced by incrementing the write counter.

*MoveTape* and *WriteTape* are operations that modify the machine state, effecting the the named operation on the specified tape element within the machine. *ReadTape* just obtain the value from the tape without any effect on the state.

*Examples of Machine Transitions*

This step 3 is where we go through the machine functionality and identify all functions that modify the machine state. For each function we define mathematically the corresponding transition using the notation we have just defined. Let's go through a suitably representative selection of machine functions to show the power of the notation. This is not an encyclopedia of all possible computer functions for every possible computer imaginable. This is a selection of examples of how this notation when used by a competent person will achieve the desired result.

The notation works because lambda-calculus is a Turing-complete model of computation. The notation is designed in such manner that it is visible by mere inspection of the formulas that they match the expected behavior of the machine. This is why the formulas capture the semantics of an actual computer.

It is possible that the definition of some transition does not match what you would expect based on some system you are familiar with. Please keep in mind that these examples are for illustrative purpose only. The goal is to show the power of the notation. If you don't like the definition provided here, feel free to write your own. The method is flexible and has the ability to define any particular transition that you may require.

*JUMP Instructions*

A JUMP instruction moves the destination of the jump into the instruction pointer (IP). This cause the

execution of the program to transfer to the target of the jump.

$$Jump(target)(S) = Upd(IP, target)(S)$$

A conditional jump performs the jump only when a condition is met. Otherwise the instruction is just skipped over.

$$JumpZ(target)(S) = (Upd(IP, S(IP)+size);\ \text{if } S(Zflag) \text{ then } Upd(IP, target))(S)$$

In this example the jump is performed is the zero flag is set, otherwise we skip to the next instruction by incrementing the instruction pointer by the size of the instruction in byte. In a real life computer, the incrementation of *IP* is done as we fetch the instruction from memory and before it is decoded and executed. The mathematical formula reflects this computation order. We may as well write the equivalent formula:

$$JumpZ(target)(S) = \text{if } S(Zflag)\ \text{then } Upd(IP, target)\ )(S)\ \text{else } Upd(IP, S(IP)+size)(S)$$

It would be mathematically equivalent but the computation order is not the same. In future examples I will not care about the computation order when it adds unnecessary complexity in the formula without changing its meaning. Rewriting the formula to reflect the real life order is not a problem. It is only uselessly tedious.

*Interrupts*

Not all transitions are triggered by instructions. Transitions may also be triggered by incoming signals from the outside world. An example would be interrupts. Here is how you may possibly define the transition from an interrupt, assuming the hardware has a priority interrupt system. Other types of interrupt systems will require to adapt the formula.

$$Interrupt(number)(S) = (Push32(S(IP));\ Push08(S(Flags));\ Push32(S(imask));\ Upd(imask, S(imask)\ \text{AND}\ Read32(baseVector+(number*8)));\ Upd(IP, Read32(baseVector+(number*8)+4)))(S)$$

Upon receiving an interrupt the CPU usually saves on the stack the current instruction pointer, the values of the flags and the interrupt mask. With each interrupt is associated a number. This number is used to calculate the location of an interrupt vector. The first 32 bits are the new interrupt mask and the next 32 bits is the address of the interrupt handling routine. However we must make sure the interrupts that are currently off in the interrupt mask remain off after loading the new mask. This is why we AND the current mask with the new mask. The order of the pushes and the size of the data will vary according to the processor.

When an interrupt is done servicing the REI (Return from Exception or Interrupt) instruction brings the computer back to where it was before the interrupt.

$$Rei(S) = (Pop32(imask);\ Pop08(Flags);\ Pop32(IP))(S)$$

*MOVE Instructions*

This is a MOVE instruction that moves 32 bits of data from memory to a 32 bits register.

$$Move32M2R(register, location)(S) = (\ Upd(IP, S(IP)+size)\ ;Upd(register, Read32(S, location))\ )(S)$$

The opposite movement from a 32 bit register to memory would be:

*Move32R2M*(*location, register*)(S) = ( *Upd*(*IP*, S(*IP*)+*size*); *Write32*(*location*, S(*register*)) )(S)

Both instructions start by moving the instruction pointer to the next instruction. This should be done systematically on any instruction that is not a jump unless there is a good reason to do otherwise.

If you wanted to move 16 bits data instead of 32 bits you would use *Read16* and *Write16* respectively but the mathematical definition would otherwise be the same.

Variants of MOVE may do addressing modes. For example here is how you would rewrite the above definitions if the memory address is using indexed addressing, that is it adds an offset to the content of a register to obtain the address.

*MoveIndex32M2R*(*destination, reg*, *offset*)(S) = ( *Upd*(*IP*, S(*IP*)+*size*) ;
*Upd*(*destination*, *Read32*(S, S(*reg*)+*offset*))
)(S)

*MoveIndexx32R2M*(*reg*, *offset , source*)(S) = ( *Upd*(*IP*, S(*IP*)+*size*);
*Write32*(S(*reg*)+*offset*, S(*source*))
)(S)

*Virtual Memory*

We can also do virtual memory. This requires to rewrite *Read32* and *Write32* to use translation pages. We also need a page fault generation primitive. Let's first work out the preliminaries. The function *paged* calculates the effective address of *x* based on the content of the page table *PT*. The page index is extracted from the bits *startpagebit*: *maxbit* of the address. This function assumes that the page has a valid page table entry. It is required that every instruction validates the page table before making use of the effective address and generate a page fault if no valid page table entry is found.

*paged*(*x*, S) = *address*(*Read32*(S, *PT*+*x*[*startpagebit*: *maxbit*]) + *x*[0:*startpagebit*-1])

We need a way to test whether a page table entry points to a valid page. Let's assume we have some hardware where the page zero is never valid. By convention the page table contains 0 in entries that don't point to a valid page. Then we have a test:

*invalid*(S, *x*) = (*paged*(S, *x*)=0)

In this last definition the left hand side “=” is referring to the mathematical definition when the left-hand side “=” in (paged(S, x)=0) is a test for equality. the mathematical notation is a little confusing here. A C programmer would have used “==”.

If the hardware uses a different convention to identify invalid entries then you will have to adapt the definition accordingly.

Here are paged versions of *Read32* and *Write32* (little endian).

*pRead32*(S, *x*) = S(*paged*(S, *x*+3)) +S(*paged*(S, *x+2)*) +S(*paged*(S, *x*+1)) + S(S, *paged*(*x*))
*pWrite32*(*x*, *v*)(S) = ( *Upd*(*paged*(S, *x*), *v*[0:7]) ;
*Upd*(*paged*(S, *x*+1), *v*[8:15]) ;
*Upd*(*paged*(S, *x*+2), *v*[16:23]) ;
*Upd*(*paged*(S, *x*+3), *v*[24:31])
)(S)

Here is a page fault function. It is an interrupt generated by the CPU, otherwise called an exception. It uses the previously defined *Interrupt* transition.

*Pagefault*(S) = *Interrupt*(*pagefaultno*)(S)

With these definitions in place lets make a paged version of the MOVE instruction of 32 bits of data from memory to register.

*pMove32M2R*(*destination*, *source*)(S) =
( If *invalid*( S, *IP* ) then *Pagefault*
else if *invalid*( S, *IP*+1 ) then *Pagefault*
else if *invalid*( S, *IP*+2 ) then *Pagefault*
else if *invalid*( S, *IP*+3 ) then *Pagefault*
else if *invalid*( S, *IP*+4 ) then *Pagefault*
else if *invalid*( S, *source* ) then *Pagefault*
else if *invalid*( S, *source*+1 ) then *Pagefault*
else if *invalid*( S, *source*+2 ) then *Pagefault*
else if *invalid*( S, *source*+3 ) then *Pagefault*
else *Upd*( *IP*, S(*IP*)+*size*) ; *Upd*( *destination*, p*Read32*(*S*, *paged*(S(*source*)) ) )
)(S)

Each address that is needed and could cause a page fault must be checked individually. The *IP* must be checked for page faults five times: once for the opcode of the instruction and four times for the four bytes of the source address for a total size of five instruction bytes. Then the source require four checks, one for each byte. If a single test gives an invalid result then a page fault occurs and the instruction doesn't proceed any further[10]. The *IP* is not increased until all addresses are validated for page faults. This means when the page fault handling routine returns with an REI instruction the MOVE instruction will be retried and hopefully the page table will be valid and a page fault won't occur.

All instructions involving memory references can be similarly adapted to handle paging. I don't do it systematically in this paper because it is tedious and the paging details will obscure the points of the examples.

*Arithmetic Instructions*

I use ADD as an example of how to do arithmetic. Other arithmetic instructions will follow a similar pattern.

An ADD instruction adds to the target the value of the operand in two's complement arithmetic. The CPU flags are set to reflect the result of boolean tests applicable to the result. The instruction must also respect the bit size of the location where the result is stored. This example illustrates how to define arithmetic operators when the limited storage size makes the operation subject to possible overflow conditions. It also illustrates how to set CPU flags according to the result of an arithmetic operation.

This formula works when both the target and operand are registers. With in memory operand or target we need to use the *ReadXX* and/or *WriteXX* functions where appropriate.

*Add*(*target, operand*)(S) =
Let (*r* = *add2complements*( S(*target*), S(*operand*) ) )
Let (*v* = r[0:*maxbit*] )
Let (*t* = *add2complements*( S(*target*)[0: *maxbit-1*], S(*operand*)[0: *maxbit-1*] )(*maxbit*) )
(*Upd*(*IP*, S(*IP*)+*size* );
*Upd*(*target*, *v* );
*Upd*(*Zflag*, *v*=*allzeros* );
*Upd*(*Nflag*, $\pi_{maxbit\ ,maxbit}(v)$ );
*Upd(Cflag,* $\pi_{maxbit+1\ ,maxbit+1}(r)$ )
*Upd(Vflag,* *t* XOR $\pi_{maxbit+1\ ,maxbit+1}(r)$ );
)(S)

This definition assumes the help of a few predefined functions and constants. The function *add2complements* performs two-complement addition on the bit sequences. The result is one bit longer than its arguments. Therefore the result must be trimmed to fit the bit size of the target and the extra bit becomes the carry flag. The constant *maxbit* is the index of the high order bit for the bit size. We use projections to select specific bits in values *v* and *r*. Remember that domains of bit sequences are Cartesian products of bits. The constant *allzeros* is a bit sequence of the right bit size where all bits are zeros. Boolean operators NOT, AND and XOR are applied on bits according to boolean algebra.

If the calculation of the overflow flag seems a bit mysterious, there is a little trick that is useful to know. The V flag is the XOR of the carry which goes into the sign bit with the carry that goes out of the sign bit. The temporary name *t* refers to the formula which computes the carry to goes into the sign bit and *r*(*maxbit*+1) is the carry that goes out of the sign bit. In a real-life computer this apparently complicated formula may be implemented quite simply by capturing the two carry bits on the fly from within the adder circuit and feeding them to an XOR gate.

*Input and Output Instructions*

The input instruction inputs data from an IO port. It assumes two auxiliary functions: *tape*(*port*) that finds the tuple for the tape associated with a port number and, *lastIO*(*port*) that will be explained in a short while. These two functions must be defined when one defines the tapes and the machine state in steps 1 and 2.

*In*(*accumulator*, *port*)(S) = Let (*i* = *ReadTape*(S, *tape*(*port*)))
(*MoveTape*(*tape*(*port*));
*Upd*(*IP*, *size*);
If *i* = *None*
then *Upd*(*accumulator*, S(*lastIO*(*port*)) )
else ( *Upd*(*accumulator*, i); *Upd*(*lastIO*(*port*), *i*)) )
)(S)

The formula reads the tape and checks if the IO device has information ready (the If *i* = *None* clause) and actually stores the input value in the accumulator only if the information is there. If the information is not ready it assumes the machine will present whatever information is leftover from the last successful input on the device. This information must be stored in the abstract machine state (it is held by the hardware) and the auxiliary function *lastIO*(*port*) finds out the mnemonic number. If *i* is not *None* then the information is ready for input. The information is stored in the accumulator and remembered in case it is needed.

Of course this notion of machine remembering the last input is only an example of what actual hardware may do when the input is not ready. The mathematical definition has to be adjusted to match what the hardware actually does.

The output instruction writes data to the IO port. It assumes theauxiliary functions *tape*(*port*) that finds the tuple for the tape associated with a port number.

*Out*(*port*, *accumulator*)(S) = (*Upd*(*IP*, *size*);
*WriteTape*(*tape*(*port*), S(*accumulator*) );
)(S)

*The CMPXCHG, NOOP and Halt Instructions*

The CMPXCHG instruction works like this:

*Cmpxchg*(*mem*, *reg*)(S) =
If S(*mem*) = S(*accumulator*)
then ( *Upd*(*IP*, *size*); *Upd*(*Zflag*(S), 1); *Write32*(*mem*, S(*reg*)) )(S)
else ( *Upd*(*IP*, *size*); *Upd*(*Zflag*(S), 0); *Upd*(*accumulator*, *Read32*(S, *mem*)) )(S)

CMPXCHG is an x86 instruction that is used to implement atomic operations such as semaphores that are necessary to handle concurrent processes. The *accumulator* refers to the x86 accumulator. I include this instruction to show concurrent processes are mathematically being taken care of by the abstract machine.

The NOOP instruction is a do nothing instruction, updating the instruction pointer to point to the next instruction. It works like this:

*Noop*(S) = *Upd*(*IP*, *size*)(S)

The halt instruction (HLT in 80x86) stops the CPU processing until an interrupt is received. The transition is mathematically described as:

*Halt*(S) = S

In other words *Halt* is a transition that does absolutely nothing in a loop. It doesn't even increase the instruction pointer. The loop follows from the difference between NOOP and HLT. NOOP let the computer proceed to the next instruction but HLT doesn't. A CPU whose instruction pointer points to HLT will loop idle with the instruction pointer stuck on HLT until an interrupt is received or power is turned off.[11] I have included this instruction to show it can be handled mathematically by abstract machines. Older computers implemented HLT like this. On computers of more recent vintage HLT generates an interrupt.

*Bulk Data Transfers and DMA*

The next transition is for a byte transfer loop body. Sometimes the hardware will move bytes around in large quantity. For example the CPU may execute a MOVS (move string) instruction or a peripheral is doing a DMA (Direct Memory Access) transfer. But such hardware will not seize control of the memory bus for the entire duration of the transfer. Interrupts are allowed to proceed and other hardware are allowed to continue their operations in parallel. This is implemented by a transition that does one iteration of the transfer loop.

```
Movs(reg1, reg2, count)(S) = (if (S(count) = 0)
                                then Upd(IP, size)
                                else  (Upd(S(reg1), S(S(reg2)));
                                       Upd(reg1, reg1+1);
                                       Upd(reg2, reg2+1);
                                       Upd(count, count-1)
                                      )
                             )(S)
```

This moves a string of bytes in memory pointed to by register *reg2* to the location pointed to by register *reg1*. There is a double indirection because S(*reg2*) gets the content of the register which is an address and S(S(*reg2*)) get the value stored at this address. The number of bytes to be transferred is identified by the register counter. If counter is zero then the instruction is complete and the instruction pointer is updated to point to the next instruction. Otherwise you move the byte, increment the registers in preparation of the next byte transfer and decrement the counter. But you don't increment the instruction pointer. When the CPU comes to execute the “next” instruction it will do the next iteration of the transfer loop. This makes the MOVS instruction interruptible between two byte transfers. This also allows other CPUs or DMA devices to perform work between two byte transfers.

DMA devices may have their own similar looping constructs based on registers. A possible design might be to require the device driver (executed by the CPU) to load the registers on the DMA device and then send a command to start the DMA transfer. Then the DMA circuitry would just loop until the transfer is done. At this point the DMA device may send an interrupt to the CPU to notify the device driver that the transfer is complete. When there is no DMA transfer or other IO in progress the DMA device is in a halt state waiting. There is no concept of instructions for the DMA device in this design. It only reacts to commands from the CPU or to incoming IO signals from the outside peripheral device.

Here is how a DMA input transition might look like:

```
DMAinput(reg, count, port)(S) = (if (S(count) <> 0)
                                   then Let (i = Readtape(S, tape(port) )
                                             MoveTape(tape(port));
                                             (If i <> None
                                                 then
                                                      (Upd(S(reg), i );
                                                       Upd(reg, reg+1);
                                                       Upd(count, count-1)
                                             )
                                 )(S)
```

This is assumed to be invoked in response to the detection by the hardware interface of a just received input

byte. If the counter is zero this is a spurious input[12] and nothing is done. If it is not zero then we check if the input tape is *None.* If so this is a spurious signal[13] and nothing has really be received. The input is discarded. If the signal is genuine then you read the data from the tape and store it in the correct memory location. Then you update the register and counter ready for the next byte.

*Power Transitions*

Another type of useful transitions are those associated with power on and power off. Some computations are designed to continue over several cycles of powering on and powering off the device. In some cases data is stored on persistent storage like Flash ROM and the software is designed to resume the computation based on this data at power on. In other cases we want to define the mathematics of an abstract network and individual devices are allowed to power on and off while the network is running. Either way we need transitions to define the effect of power on and power off on the abstract machine state.

The effect of a power off is to erase all locations in the computer where data is stored in a non-persistent manner. The effect of a power on is to initialize these locations with new values. Everything else stays unchanged. Everything else means the tapes stand still, not being read, not being written and not being moved. It also means the data in persistent storage stay as it is.

Mathematically there is a sequence $L_0$ ... $L_{n-1}$ of $n$ locations referring to the elements of the machine state that are affected. For each of these locations there are two transitions *TurnOff*(i) and *TurnOn*(i) that erases and initializes location $L_i$. This gives rises to the following mathematical considerations:

There is a function called *volatile* in domain $\boldsymbol{n} \rightarrow \mathbf{N}$ such that *volatile*($i$)= $L_i$. In other words *volatile* enumerates the mnemonic for the volatile elements in the states. This function is definable by a cascade of if's.

*volatile*($i$) = if $i$ = 0 then $L_0$
else if $i$=1 then $L_1$
. . .
else $L_{n-1}$

The domains for the volatile elements in the machine state must allow the *None* value to represent their powered off state, ie a powered of machine holds no information in these elements. Therefore if $D_0$ ... $D_{n-1}$ are the domains for the base elements in the volatile component then the domains actually used in a machine state definition must be the disjoint unions ( $D_0$ + ***None*** ), ... ( $D_{n-1}$ + ***None*** ).

There is a function called *InitialValue* in domain $\boldsymbol{n} \rightarrow D_0 + ... D_{n-1}$ which maps each mnemonic for a volatile element to the corresponding initial value after power on.

As an alternative to a definite initial value the machine may set the state of some element to an unpredictable value that change from one power on cycle to another. We have an input tape called *Unpredictable* where suitable unpredictable values in $D_0 + ... D_{n-1}$ may be read.

The definitions of the transitions *TurnOff*(i) and *TurnOn*(i) are:

*TurnOff*(i)(S) = *Upd*(S, *volatile*($i$), *None*)(S)

*TurnOn*(i)(S) = *Upd*(S, *volatile*($i$), $InitialValue_i$)(S) whenever the power on value of $L_i$ is defined by the hardware to be $InitialValue_i$.

*TurnOn*(i)(S) = Let (v=*ReadTape*(*Unpredictable*) )
(*MoveTape*(*Unpredictable*);
*Upd*(S, *volatile*($i$), v)
)(S)

whenever the power on value of $L_i$ is undefined by the hardware and is unpredictable. An tape called *Unpredictable* is used to generate unpredicted values in such circumstances.

The transitions to power off and power on the computer are:

*PowerOff*(S) = (*TurnOffList*(n); *Upd*(*status*, *Stopped*) )(S)

*TurnOffList*(n)(S) = if (n = 0) then *TurnOff*(0)(S) else ( *TurnOffList*(n-1); *TurnOff*(n) )(S)

*PowerOn*(S) = ( *Upd*(*status*, *Running*); *TurnOnList*(n) )(S)
*TurnOnList*(n)(S) = if (n = 0) then *TurnOn*(0)(S) else ( *TurnOnList*(n-1); *TurnOn*(n) )(S)

These definitions are example of how you can use recursion to define loops in this mathematical notation. You power on the abstract machine by initializing all locations in the sequence $L_0$ ... $L_n$ that is affected by power changes. Similarly you power off the abstract machine by erasing the sequence of location. The status is changed from *Stopped* to *Running* or conversely as appropriate.

*Wrapping Up Step 3*

These examples sufficiently illustrate the power of the notation. It should be clear by now that we have to power to describe all computer operations with mathematically defined transitions. The goal of this paper is not to describe a specific computer. It is to convince that we have a method powerful enough to describe any computer if we spend the time and energy and have access to the required technical documentation.

Some people focus on defining mathematical formulas for computer instructions on a per register/memory location basis. Then they notice that many instructions cannot be defined mathematically in this manner by ordinary algebraic means. They erroneously conclude that these instructions are not mathematical. The error is to try doing it on a per register/memory location basis while restricting themselves to ordinary algebra. A similar error occurs when someone tries to define mathematically the individual statements of a programming languages like C by ordinary algebraic means. Then this person notices that the variables in a program do not work like mathematical variables because they can be overwritten. The erroneous conclusion is that programming statements are not mathematical operations. The error is to try to define the formula by means of ordinary algebra. The correct way is to define higher order functions that affect the machine state as a whole using a language with this kind of expressiveness. Once this technique is understood there is no complexity in the hardware that can't be defined mathematically.

Another important technique is to represent as an input tape anything relevant that is not under control of the hardware in a deterministic and predictable manner. Input tapes are parameters to the computation that may be physically provided by uncomputable real-world events. Such events may influence the computation but they are not the computation. This is why they are parameters.

**Step 4**: *Identify the loops that execute the machine transitions and define the corresponding mathematical formulas for the body of these loops.*

Physical machines carry out the computations by means of loops hard wired in the circuitry. The best known example is the instruction cycle. This physical loop replicates the operating principle of the abstract RASP. The details of the loop will vary from one make and brand of CPU to another. It typically looks like this.

1. Read the instruction code from the location in memory indicated by the instruction pointer.
2. Increments the instruction pointer to point past the just read instruction.
3. Decode the instruction code.
4. If operands are required, read the operand data from memory at the location pointed to by the instruction pointer.
5. If required increment the instruction pointer to point pas the operands data.
6. Perform whatever task is described by the instruction code and the operands if any.
7. Test if an interrupt must be serviced. If there is one then perform the following steps:
    - Push the current interrupt mask and CPU flags on the stack.
    - Push the current instruction pointer on the stack.
    - Determine the interrupt number of the currently processed interrupt.
    - Calculate the address of the corresponding interrupt vector.
    - Set the interrupt mask according to the interrupt vector.
    - Set the instruction pointer to the address of the interrupt handler located right after the interrupt mask in the interrupt vector.
1. Return to step 1.

Much of this loop is built into the transitions as we have defined them in the example. All the “increment the instruction pointer” stuff is included. The interrupt handling is defined as a transition. What this loop tells us is how the transitions are glued together in an ongoing process.

Mathematically the body of the loop looks like this:

*CPULoopBody*(S) = ( *DoInstruction*; *DoInterrupt* )(S)

Every iteration of the loop first do an instruction and then checks whether an interrupt should be processed. If so then it is processed.

*DoInstruction*(S) = Let ( *code* = *Read08*(S, S(*IP*) )
Let ( *T* = *FindTransition*(*code*) )
Let ( *Op* = *FindOperand*(*code*)(S) )
*T*(*Op*)(S)

Doing an instruction is first find the one byte instruction opcode based on the instruction pointer and the data in memory. Then find the mathematically transition that matches the opcode. This means we must have built in step 3 a table that associates with each instruction opcode the corresponding mathematical transition. Such a table is a mathematical function and it is called *FindTransition* in this example. Then we need to find the arguments to supply to the transitions. They are the operands of the instruction. Each opcode is associated with a method of extracting the operands from the memory. This yield another table called *FindOperand*. Then the transition is applied to the operand and the computer state. Both *FindTransition* and *FindOperand* may be written in lambda-calculus as a series of if statements.

If more than one operated is required then we can treat the whole group as a tuple and define *FindOperand* as an operation that retrieves the tuple and apply *T* to the tuple. This way we can reduce the case of multiple operands to the handling of a single entity which is a tuple.

*DoInterrupt*(S) = Let (*number* = *Interrupted*(S))
( *MoveInterruptTapes*; if (*number* <> *None*) then *Interrupt*(*number*) )(S)

The *Interrupted*(S) function finds the number of the highest priority interrupt that needs servicing. It returns *None* when no interrupt need servicing. If there is an interrupt then we invoke an *Interrupt* transition such as the one that was defined in the examples in step 3. The function *MoveInterruptTapes* moves the tapes that were read in the evaluation of *Interrupted*(S) according to the convention about tape information never been read more than once.

The loop itself is to calculate *CPULoopBody*(S) on the current state S and use the result as the state for the next iteration of the loop. Repeat this forever, or at least until you get bored and power off the computer. The change in the state on each iteration of the loop are the calculations done by the computer.

Let's see another example. Let's consider a simplified DMA device. It has a control register whose content tells the device which DMA process is ongoing if any. If a process is ongoing then other registers are used to control the progress of the DMA process. For an example see the *DMAinput* transition in the preceding section. The DMA loop body may look like:

*DoDMAtransition*(S) = Let ( *code* = *DMAprocess*(S(*ControlReg*)) )
Let ( *T* = *FindDMATransition*(*code*) )
Let ( *Op* = *FindDMAOperand*(*code*)(S) )
*T*(*Op*)(S)

The *DoDMAtransition* looks exactly like the *DoInstruction* function for the CPU. The differences are:

- It doesn't read its code from the same place.
- It has its own transition table called *FindDMATransition*.
- It has its own operand identification table called *FindDMAOperand*.

- The DMA body of the loop only have a transition part. It has no interrupt part like the CPU.

There is a difference also in how the loop is iterated. The CPU has a clock cycle. Each instruction takes a few cycles. The rate at which instructions are executed depends on the clock speed, the number of clock cycles that are required by the instructions and the rate at which interrupts are received.

The DMA device has its own clock cycle that may be and usually is different from the CPU. It may also have wait states that depends on the peripheral readiness for physical input or output. The two loops coexists, they affect the same memory, but how do we determine in which sequence to interleave their operation? The answer depends on the relative speed of the circuitry, external events triggered by the peripherals and also the arbitration circuitry for memory access. All of this is outside the control of the software. Like in any situation where software is affected by factors outside its control we define a tape. This is simpler than trying to define in the minute details the effect of all the laws of electronics that may apply. The name of the tape is *WhoGoesNext*. We define the consolidated loop body like this:

*LoopBody*(S) = Let ( *device* = *ReadTape*(S, *WhoGoesNext*) )
( *MoveTape*(*WhoGoesNext*);
*FindLoopBody*(*device*)
)(S)

When we wrote all the loop bodies we have built at the same time a table that associates the device with the corresponding loop body. This table is the function *FindLoopBody*(*device*). Then when you query the tape for the device that goes next you can find the corresponding loop body and apply it to the state. This loop would work whatever the number of looping processes that runs in the compute.

**Step 5**: *Define what is a run of a program on the abstract machine.*

I begin with a simple scenario where the machine is powered on to start the computation and powered off to stop it. Other scenarios will follow.

*The Base Power On–Power Off Scenario*

A complete run of the computer starts at power on and stops at power off. This requires four things: (1) a definition of what is an acceptable start state for the machine before power on in terms of information located on the tapes, and, (2) a transition that generates initial values for the non-tapes elements of the abstract machines, and, (3) a power input tape.

The first item means we need to define a formula called *isStart*(S) that will test *True* when the state S qualifies as a valid starting state and *False* otherwise. A valid computation is allowed to start at any valid state. There is usually more than one valid start state because the content of several of the tapes are the inputs which may vary from computation to computation. Tapes used for output will be initially blank. Therefore formula *isStart*(S) identifies the valid inputs for the computation.

The second item is an *PowerOn*(S) transition that produces the initial state of a computation from a valid power off start state. It corresponds to the act of powering on the machine. We have seen its definition in step 3.

The third item is a tape called *Power* that reads *True* to represent the fact that there is power and reads *False* otherwise. The computation continues as long as the *Power* tape stays *True* and stops as soon as it is *False*. This is an input tape which must be included among the machine tapes..

Let's start with the *isStart*(S) formula. Recall that **S** is defined as follow:

$\mathbf{S} = \boldsymbol{Status} \times M_0 \times \ldots M_{i-1} \times Tape_0 \times \ldots \ Tape_{j-1}$

Among the elements $M_0 \ldots M_{i-1}$ some have a well defined state at power on, for instance the content of ROM, while others are undefined and will land in some unpredictable state. It is assumed that each domain $M_i$ takes the form $\boldsymbol{D}_i + \boldsymbol{None}$ for some domain $\boldsymbol{D}_i$. Therefore we can define the permitted powered off state of an abstract machine as follow:

- The status component is *Stopped*

- For each element $M_i$ the value is *None*
- For input tapes the counter starts at zero but any sequence of values is permissible. As an option some additional constraints may be imposed on the sequences of values, like requiring the *Power* tape to stay on during the entire computation and eventually turn off. The admissible constraints must be ones which reflect real life constraints applicable to a physical machine and not application constraints that are checked by the program. The point is to define the semantics of the machine and not the semantics of the program.
- For network tapes used for input the input counter starts at zero but any sequence of values and any output counters are permissible. As an option some additional constraints may be imposed on the sequences of values.
- For output tapes the counter starts at zero and the tape must be be all blank, ie. all tape entries are *None*.
- For network tapes used for output the both counters starts at zero and the tape must be be all blank, ie. all tape entries are *None*.

Remember that S(*n*) is the $n^{th}$ component within the state S. The *isStart*(S) formula for valid starting states is the "and" of these clauses:

- S ϵ **S**, obviously
- S(0) = *Stopped*
- S(*n*) = *None* for all *n*, 1 <= *n* <= *i*; for locations corresponding to locations where information is stored.
- *Sequence*(S(*n*)) ϵ $SM_n$ for all *n*, *i*+1 <= n < *j*; When *n* corresponds to input tapes and network tapes used for input; the sets $SM_n$ being chosen to meet constraints one may wish to impose on acceptable input tapes. If no constraints are imposed $SM_n$ will include all elements of the sequence domain.
- *Sequence*(S(*n*))(*m*) = *None*; for all *n*, *i* <= n < *j*; and for all *m*; When *n* corresponds to output tapes and network tapes used for outputs
- *readCounter*(S(*n*)) = 0; for all *n*, i <= *n* < *j*; for all tapes
- *writeCounter*(S(*n*)) = 0; for all *n*, i <= *n* < *j*; When *n* corresponds to output tapes and network tapes used for outputs

A computation is a sequence of states in **N** → **S** such that

- *isStart*($S_0$) is *True* and
- $S_1$ = *PowerOn*($S_0$)
- $S_{i+2}$ = (*MoveTape*(*Power*); *LoopBody*)($S_{i+1}$) provided *ReadTape*($S_{i+1}$, *Power*) is *True*. Otherwise the computation stops at $S_{i+1}$.

According to this definition the computation to be performed by the machine will vary depending on (a) the input as defined by the tapes and (b) the information that happens to be stored in uninitialized parts of the computer at power on. A well programmed computer will not normally rely on the uninitialized parts[14] and will depend only on input tapes but this mathematical semantics of the machine will accept a poorly written program.

The above definition is a mathematical representation of this loop:

1. The initial state is *PowerOn*(S) for any member S of domain **S** such that the formula *isStart*(S) is *True*.
2. If *ReadTape*(S, *Power*) is false then stop the execution otherwise proceed with step 3.
3. A new state S' = (*MoveTape*(*Power*); *LoopBody*)(S) is computed.
4. Go back to step 2 with S' in the role of S.

In other words the computation is the sequence of all states the abstract machine goes through starting from the power on and stopping at power off.

Or in the notation of this article the steps 1–4 are the formula below with the requirement that *isStart*(S) is *True* of the initial value of S.

*MachineRun*(S) = (*PowerOn*;
*While*( λS *ReadTape*(S*, Power*),

(*MoveTape*(*Power*); *LoopBody*)
)
)(S)

Documentation note: the λS in the loop condition is necessary to make the loop condition apply to the current state of the loop iteration.

*The Per Routine or Per Program Scenario*

We may want the definition of a computation from running a specific program or a specific portion of a program like a block of code, procedure, method or function within a program. Whenever the computer runs only one program from power on to power off there is no problem. A run of the computer is the same thing as a run of the program in such scenario. But most real life computers also run an operating system, device drivers and plenty of other programs and utilities. The computation done by a specific program must be isolated from the computations done by other software in the same computer run.

This could be done with filtering. The trick is to view the run of the computer as a sequence of successive states generated one after another. The run of the program or block of code is the succession of these states where this particular program or code executes.

The sequence of states from a run of the computer is defined by the pair of recursive equations:

*SeqRun*(S)(0) = *PowerOn*(S)
*SeqRun*(S)(n+1) = if *not ReadTape*(S*, Power*) then S else (*MoveTape*(*Power*); *LoopBody*)(*SeqRun*(S)(n))

Assuming that *ProgramFilter*(S) is a boolean function which test whether a state belongs to the execution of a program or block of code then a run of this program or block of code is defined with these three functions.

*NextStateAndCount*(S)(*n*) = If *not ReadTape*(S*, Power*)
then *None*
else if *ProgramFilter*(*SeqRun*(S)(n))
then ⟨*SeqRun*(S)(n+1), *n*⟩
else *NextStateAndCount*(S)(*n*+1)

*ProgramRunList*(S)(*n*) = Let (*x* = *NextStateAndCount*(S)(*n*))
if *isNone*(*x*)
then *None*
else ⟨$\pi_{0,1}$(*x*), *ProgramRunList*(S)($\pi_{1,1}$(*x*))⟩ )

*ProgramRun*(S) = *ProgramRunList*(S)(0) when state S shouldn't be in the list or
⟨S, *ProgramRunList*(S)(0)⟩ when state S should be included in the list

The idea is to generate a list of all states that belongs to the program run. Function *NextStateAndCount*(S)(*n*) will find the next such state and index number that comes after state number *n* in a program run starting with initial state S. If no such state is found before power off then the *None* value is returned. There is a quirk in this definition: the test is applied to state number *n* but the state returned is number *n*+1. The reason is that the *ProgramFilter* test is usually based on components such as the content of the program counter (*PC*) which indicate which execution will execute next. Therefore when the test passes its result is applicable to the next state, hence the returned state number is *n*+1. A consequence is that the initial state will never included in the list by this mechanism. This state has to be included explicitly if need be.

Function *ProgramRunList*(S)(*n*) builds a list of all states belonging to the program run that comes after state number *n*. Then function *ProgramRun* builds the complete list of states starting at state number 0 and this is the entire program run.

A possible test for *ProgramFilter*(S) might be to verify that the program counter is in the correct range. For example:

*ProgramFilter*(S) = *lowPCvalue* ≤ S(PC) ≤ *highPCvalue*

If the system uses virtual memory we need to add a clause to ensure the page table register points to the page table that corresponds to the address space where the program is loaded.

*ProgramFilter*(S) = (*lowPCvalue* ≤ S(PC) ≤ *highPCvalue*) and (S(*PTR*) = *programPageTable*)

More filters may be devised to match the many possible program circumstances.

*Computations Across Multiple Cycles of Powering On and Off*

Sometimes we want to define a computation that runs across multiple cycles of power on and power off events, resuming from where it stopped at the next power on. We will use this kind of computations when we define abstract network because individual nodes are allowed to shutdown and restart while the rest of the network keeps running. The computation starts when the computer is first connected to the network and powered on, then it is suspended when the computer is turned off, resumed when it is turned on again and after a number of cycles going off and on again eventually it stops definitively when the computer is permanently removed from the network.

This definition requires on top of *isStart*($S_0$) and the *Power* tape a *inService* tape that is true while the computer is in service even if powered off and becomes false when the computer is permanently put out of service. It also requires to include in the state whether the abstract machine is currently powered or not.

We must adjust *LoopBody*(S) to account for transitions involving turning power on and off. The new loop body is:

```
LoopBodyWithPower(S) = Let (currentStatus = S(Status))
                       Let (stayPowered = Readtape(S, Power))
                           (MoveTape(Power);
                            if currentStatus=Running
                               then if stayPowered
                                       then LoopBody
                                       else PowerOff
                               else if stayPowered
                                       then PowerOn
                           )(S)
```

The rule is quite simple. When the power was on and is still on the abstract machine does the *LoopBody*. When the power was on and becomes off the abstract machine does a power off transition. When the power was off and becomes on the abstract machine does a power on transition. When the power was off and stays off the abstract machine does nothing but read the *Power* tape.

Notice the use of the *Status* information to determine whether power transitions are required. This is where this information is used.

We define the computation as a sequence of states in **N** → **S** such that

- *isStart*($S_0$) is *True* and
- $S_1$ = *PowerOn*($S_0$)
- $S_{i+2}$ = (*MoveTape*(*inService*); *LoopBodyWithPower*)($S_{i+1}$) provided *ReadTape*($S_{i+1}$, *inService*) is *True*. Otherwise the computation stops at $S_{i+1}$.

The above definition mathematically represents a loop like this:

1. The initial state is *PowerOn*(S) for any member S of domain **S** such that the formula *isStart*(S) is *True*.
2. If *ReadTape*(S, *inService*) is *False* then stop the execution otherwise proceed with step 3.

3. A new state S' = (*MoveTape*(*inService*); *LoopBodyWithPower*)(S) is computed.
4. Go back to step 2 with S' in the role of S.

Compare the definitions in the case the run is from power on to power off with the case the run goes across power on-off cycles. The formulas look the same. We have replaced the *Power* tape with the *inService* tape and the *LoopBody* function with *LoopBodyWithPower*. Everything else is exactly the same. This means that in both cases the abstract machine representing the work of the computer shares the same abstract pattern.

The formula for this loop is:

*MachineRun*(S) = (*PowerOn*;
*While*( λS *ReadTape*(S*, inService*),
(*MoveTape*(*inService*); *LoopBodyWithPower*)
)
)(S)

Documentation note: the λS in the loop condition is necessary to make the loop condition apply to the current state of the loop iteration.

**Summarizing Interlude, Abstract Computer**

Let's summarize the definition of the abstract computer in mathematical terms. This summary captures the common mathematical pattern of the various flavors of abstract machines.

The abstract computer is a quintuple P = ⟨**S**, *LoopBody*, *isStart*, *PowerOn*, *Stop*⟩ where the components of the tuple are as follow:

**S** corresponds to an inventory of all the places where information is stored in the computer together with an inventory of all the inputs and outputs the computer may use. **S** is defined as follow:

**S** = ***Status*** × $M_0$ × ... $M_{i-1}$ × $Tape_0$ × ... $Tape_{j-1}$

The abstract computer executes a loop whose body is *LoopBody* in **S** → **S**.

The *isStart*(S) formula defines what constitutes a valid starting point for the loop. It is the “and” of these clauses:

- S ∈ **S**, obviously
- S(0) = *Stopped*
- S(*n*) = *None* for all *n*, 1 <= *n* <= *i*; for locations corresponding to locations where information is stored.
- *Sequence*(S(*n*)) ∈ $SM_n$ for all *n*, *i*+1 <= n < *j*; When *n* corresponds to input tapes and network tapes used for input; the sets $SM_n$ being chosen to meet constraints one may wish to impose on acceptable input tapes. If no constraints are imposed $SM_n$ will include all elements of the sequence domain.
- *Sequence*(S(*n*))(*m*) = *None*; for all *n*, *i* <= n < *j*; and for all *m*; When *n* corresponds to output tapes and network tapes used for outputs
- *readCounter*(S(*n*)) = 0; for all *n*, i <= *n* < *j*; for all tapes
- writeCounter(S(n)) = 0; for all n, i <= n < j; When n corresponds to output tapes and network tapes used for outputs

The *PowerOn*(S) transition is performed immediately upon powering the machine. It is a state transition from domain **S** → **S**.

*Stop* is an input tape that returns a value in *Bool* indicating the termination of the computation.

A computation performed by the machine P is a sequence $S_i$ of states in **N** → **S** such that

- *isStart*($S_0$) is *True* and
- $S_1$ = *PowerOn*($S_0$)
- $S_{i+2}$ = (*MoveTape*(*Stop*); *LoopBody*)($S_{i+1}$) provided *ReadTape*($S_{i+1}$, *Stop*) is *True*. Otherwise the computation stops at $S_{i+1}$.

This computation corresponds to this following function **S** → **S**

*MachineRun*(S) = (*PowerOn*;
*While*( λS. *ReadTape*(S, *Stop*),
(*MoveTape*(*Stop*); *LoopBody*)
)
)(S)

Documentation note: the λS in the loop condition is necessary to make the loop condition apply to the current state of the loop iteration.

**Step 6 (abstract networks only)**: *Connect the inputs and outputs of abstract machines to define the corresponding abstract network.*

There are models of computations that will represent abstract networks better than lambda-calculus but this doesn't prevent lambda-calculus from being able to do it. There is a definite advantage in using the same mathematical formalism for both the individual machines and their aggregation in the network. Therefore I expand the notation to include features targeted specifically at networks.

*Expanding the Notation to Lists*

A domain of lists **L** of elements in domain **D** is a recursively defined domain of the form:

**L** = (**D** × **L**) + ***None***

The empty list consists of the single value *None*. A nonempty list is a pair ⟨*head*, *rest*⟩ where *head* is the first value in the list and *rest* is the rest of the list. This permits the functions:

*isEmpty*(*x*) = *isNone*(*x*)
*First*(⟨*head, rest*⟩) = *head*
*Rest*(⟨*head, rest*⟩) = *rest*

The domains are **L** → *Bool* for *isEmpty*, **D** × **L** → **D** for First and **D** × **L** → **L** for *Rest*.

Neither *First* not *Rest* are defined when the list is empty, or more precisely selecting the (**D** × **L**) domain from an empty list will return the bottom element of the (**D** × **L**) domain. Correspondingly *First* and *Rest* will return the bottom element of the **D** and **L** domains respectively. In domain theory this corresponds to absence of data and *First* and *Rest* are useless in such circumstances. The assumption is we have to test for the empty list before calling these functions to ensure meaningful results.

The function *Append*(L, *v*) appends the value *v* at the end of list L. The domain for *Append* is **L** × **D** → **L**.

*Append*(*None*, *v*) = ⟨*v, None*⟩
*Append*(⟨*head, rest*⟩, *v*) = ⟨*head, Append*(*rest, v*)⟩

The function *AtIndex*(L, *n*) returns the item located at index *n* in list L or *None* if no such item exist. Index 0 corresponds to the first element (the head) of the list. Index 1 corresponds to the second element etc. The

domain for *AtIndex* is **L** × **N** → **D**.+ ***None***.

*AtIndex*(*None*, *n*) = *None*
*AtIndex*(⟨*head, rest*⟩, 0) = *head*
*AtIndex*(⟨*head, rest*⟩, *n*+1) = *AtIndex*(*rest*, *n*)

The function *Replace*(L, *n*, *v*) replaces the value located at index *n* in the list L with value *v*. The domain for *Replace* is **L** × **N** × **D** → **L**.

*Replace*(*None*, *n*, *v*) = *None*
*Replace*(⟨*head, rest*⟩, 0, *v*) = ⟨*v, rest*⟩
*Replace*(⟨*head, rest*⟩, *n*+1, *v*) = ⟨*head, Replace*(*rest*, *n*, *v*)⟩

The function *Delete*(L, *n*) deletes the item located at index *n* in the list L. The domain for *Delete* is **L** × **N** → **L**.

*Delete*(*None*, *n*) = *None*
*Delete*(⟨*head, rest*⟩, 0) = *rest*
*Delete*(⟨*head, rest*⟩, *n*+1) = ⟨*head, Delete*(*rest*, *n*)⟩

*Machine Descriptors*

As was mentioned before an abstract network is a set of abstract machines that are connected. Some output tapes of some abstract machines are the input tapes of other (or same) abstract machines. When tapes are used as both input and output to connect two machines they are called network tapes.

A computation performed by the abstract network is a sequence of transitions from one network state to another. Conceptually a network state is the aggregation of all the states of the abstract machines and all connections between machines in the network. The interpretation is that at every moment during the computation if you take a snapshot of the states of all abstract machines and all connections in the abstract network you get the network state.

One of the difficulties is that the abstract machines are heterogeneous. They have a different internal structure with different component and different transitions. But because they all conform to the same abstract mathematical pattern it is possible to handle this diversity with the help of a mathematical device called the machine descriptor.

A machine descriptor for machine *m* is a quintruple ⟨ $S_m$, $LoopBody_m$, $isStart_m$, $PowerOn_m$, $Stop_m$ ⟩ where $S_m$ is the current machine state and $LoopBody_m$, $isStart_m$, $PowerOn_m$ and $Stop_m$ are the basic primitives used to define the abstract machine computation as explained above. Please note that there is a difference between the definition of an abstract machine as a quintuple P = ⟨**S**, *LoopBody*, *isStart*, *PowerOn*, *Stop*⟩ and a machine descriptor. The first element of the quintuple P is the state domain but the first element of the machine descriptor is the state.

The primitives in the descriptors will vary from machine to machine, therefore they are identified with a subscript. This descriptor belongs to domain:

$\mathbf{MD}_m = \mathbf{S}_m \times (\mathbf{S}_m \to \mathbf{S}_m) \times (\mathbf{S}_m \to Bool) \times (\mathbf{S}_m \to \mathbf{S}_m) \times (\mathbf{N} \to Bool)$

Assuming there are in the abstract network *n*+1 possible machine types with one domain $\mathbf{MD_i}$ for each type of machine the domain for all machines descriptors is:

$\mathbf{MD} = \mathbf{MD_0} + \ldots + \mathbf{MD_n} + \mathbf{None}$

This domain allows to talk about machine descriptors in general while the machines primitives themselves belongs to different domains. We allow a descriptor to be *None* to indicate the absence of a machine.

As a matter of convention each of the primitives has a corresponding function identified with a prime '

symbol. The notation *Function*'(*md*) means the corresponding primitive from a machine descriptor in domain **MD**. The primitives are:

S'(*md*) for the machine state, *Loopbody*'(*md*) for the body of the computing loop, and similarly for *isStart*'(*md*), *PowerOn*'(*md*) and Stop'(*md*).

Another notation is *NewState*(*md*)(S) which changes the state component of the machine descriptor. It is used to implement transitions on machine descriptors based on transitions from the underlying machine.

*NewState*(S)(*md*) = Upd(0, S)(*md*)

*The Definition of Abstract Networks and their Domains*

An abstract network is the collection of all its abstract machines and their connections taken together. This is expressed in formal mathematical terms as tuples from Cartesian products as follow.

> An abstract connection is a quadruple $\langle P, input, Q, output\rangle$ where $P$ and $Q$ are abstract machines, *input* is the index to an input network tape in a state for $P$ and *output* is the index an output tape in a state for $Q$. It is said that $P$ and $Q$ are connected by the connection $C$ when there is an *input* and an *output* such that $C = \langle P, input, Q, output\rangle$ and $P(input) = Q(output)$, that is the same network tape is present in both machines.
>
> An abstract network is a pair $\langle AM, AC\rangle$ where $AM$ represents the abstract machines in the network and $AC$ represents the abstract connections in the network. The abstract machines $AM$ is a list $\langle P_0 , ... \langle P_n, None\rangle ...\rangle$ where each of the $P_i$ in the list is an abstract machine which is said to be a machine in the network. The abstract connections $AC$ is a list $\langle C_0 , ... \langle C_m, None\rangle ...\rangle$ where each of the $C_i$ in the tuple is an abstract connection between two machines in the network. For all connections in the network $C_i = \langle P, input, Q, output\rangle$ we have $P(input) = Q(output)$.

The domains for a list of machine descriptors is defined recursively:

**AN** = (**MD** × **AN**) + ***None***

The domain for a list of abstract connections is defined recursively as a list of quadruples.

**AC** = ((**N** × **N** × **N** × **N**) × **AC**) + ***None***

The first and third element of the quadruple are the indices where to find the connected machines in a list of machine descriptors. The second element is the index where to find the input tape in the first machine sate while the fourth index will likewise locate the output tape in the second machine.

We define significant names for the components of an abstract connection.

*inMachine*(C) = $\pi_{0,3}$(C)
*inTape*(C) = $\pi_{1,3}$(C)
*outMachine*(C) = $\pi_{2,3}$(C)
*outTape*(C) = $\pi_{3,3}$(C)

The domain for an abstract network is as this:

**NS** = **AN** × **AC** × **NT**

The domain **NT** is for a tape containing the descriptions of network transitions. These transitions will be defined shortly.

We define significant names for the components of an abstract network.

*Machines*(N) = $\pi_{0,2}$(N)
*Connections*(N) = $\pi_{1,2}$(N)

*Transitions*(N) = $\pi_{2,2}$(N)

We also define *ReadTransitionTape*(N) and *MoveTransitionTape*(N) analogously for machine state tapes:

*ReadTransitionTape*(N) = let ( *sequence*=$\pi_{0,1}$(*Transitions*(N)) )
let ( *readCounter*=$\pi_{1,1}$(*Transitions*(N)) )
*sequence*( *readCounter* )

*MoveTransitionTape*(N) = let ( *sequence*=$\pi_{0,1}$(*Transitions*(N)) )
let ( *readCounter*=$\pi_{1,1}$(*Transitions*(N)) )
⟨ *Machines*(N), *Connections*(N), ⟨ *sequence*. *readCounter*+1 ⟩ ⟩

*Network Transitions*

Now that we know what an abstract network looks like in terms of the constituent machines and connections we proceed to the definition of what is a computation performed by the network. The computation advances one step at a time by the occurrence of each of any of the following events.

- One of the individual machines makes a transition.
- A machine is added to the network
- A machine is initially powered on
- A machine is removed from the network
- A connection between two machines is established
- A connection between two machines is torn down
- The network is done computing and stops

Each of these events corresponds to a domain for a computation event descriptor:

***MachineTransition*** = **N**
***AddMachine*** = **MD**
***PowerOnMachine*** = **N**
***RemoveMachine*** = **N**
***AddConnection*** = **(N × N × N × N)**
***RemoveConnection*** = **N**
***StopNetwork*** = *Bool*

Together these domains for the basis domain for the tape of network transitions **NT** mentioned above. That is this basis is the domain **NE** of network events defined as follow:

**NE** = ***MachineTransition*** + ***AddMachine*** + ***PowerOnMachine*** + ***RemoveMachine*** +
***AddConnection*** + ***RemoveConnection*** + ***StopNetwork***

A ***MachineTransition*** is described by the index (a natural number) in the list of machine descriptors where we find the machine that will perform the next transition.

The transition ***AddMachine*** is described by a machine descriptor for the machine to be added to the network.

The transition ***PowerOnMachine*** is described by the index in the list of machine descriptors where we find the machine to be powered on initially.

The transition ***RemoveMachine*** is described by the index in the list of machine descriptors where we find the machine to be removed from the network.

The transition ***AddConnection*** is described by the four numbers making the connection to be added.

The transition ***RemoveConnection*** is described by the index in the list of connections where the connection to be removed is located.

The transition ***StartMachine*** is described by the index in the list of connections where the machine is located and the initial state the machine is initialized to once started.

The transition ***StopNetwork*** is a boolean that is True when it is time for the network to stop computing.

Each of the transitions except stopping the network is defined by a function.

The transition event ***MachineTransition*** corresponds to the function *MachineTransition*(*m*)(N) from domain ***MachineTransition*** → **NS** → **NS**. This transition executes one single machine transition on machine at index *m* in the *Machines*(N) list of abstract machines in the network. In the event this transition writes something to some output tape this transition propagates the change to the corresponding input tape of the receiving machine.

The transition works by (a) locating in the list of machines the descriptor of the one that must perform the transition, (b) extracting from the machine descriptor the current machine state, (c) execute the transition on the state, (d) update the descriptor with the new state, (e) update the list of machines with the updated descriptor and (f) finally synchronize the output network tapes with the inputs of the machines that will receive the information.

*MachineTransition*(*m*)(*N*) = let (AN=*Machines*(*N*))
let (AC=*Connections*(*N*))
let (*descriptor=AtIndex*(AN, *m*))
let S = S'(*descriptor*))
let (T = if *not ReadTape*(*Stop*'(*descriptor*))(S)
then ( *MoveTape*(*Stop*'(*descriptor*)); *LoopBody*'(*descriptor*) )(S)
)
let *newAN* = *Replace*(AN, *m, NewState*(*descriptor*)(T) )
⟨*Sync*(*newAN*, AC, *m*), AC⟩

The auxiliary function *Sync*(AN, AC, *m*) is used to synchronize the output network tapes of machine at index *m* with the corresponding input network tapes of the other machines receiving this information. It works by traversing the list of connections and, upon finding a connection that applies, replicating the output network tape in the input machine state. No effort is made to go directly to the relevant connection. The function just checks them all and make sure they are all in sync, trusting that spurious synchronization is equivalent to no action being taken. This mathematical function looks complicated but its real life implementation is achieved by establishing the proper physical connections. No actual computation need being done.

*Sync*(AN, *None*, *m*) = AN
*Sync*(AN, ⟨*connection, rest*⟩, *m*) = if outMachine(*connection*)=*m*
then *Sync*(*SyncHead*(AN, *connection*, *m*), *rest*, *m*)
else *Sync*(AN, *rest*, *m*)

*SyncHead*(AN, *connection*, m) = let ( *OutputTape*=S'( *AtIndex*(AN, *m*) )(*outTape*(*connection*) ) )
let ( *InputMachineDescriptor=AtIndex*(AN, *InMachine*(*connection*))) )
let ( *InputTapeIndex=inTape*(*connection*) )
let ( S=S'(*InputMachineDescriptor*) )
let ( T=*Upd*( *InputTapeIndex*, *OutputTape*)(S))
let ( *NewInputMachineDescriptor*= *NewState*( *InputMachineDescriptor*)(T)
*Replace*(AN, *InMachine*(*connection*), *NewInputMachineDescriptor*)

The transition event ***AddMachine*** corresponds to the function *AddMachine*(M)(N) from the domain ***AddMachine*** → **NS** → **NS**. This function just take the descriptor of a machine and adds it to the the list of machines in the network.

*AddMachine*(M)(N) = ⟨ *Append*(*Machines*(N), M), *Connections*(N) ⟩

For the sake of integrity it should be assumed that machine are added to the network in a valid powered off

state, that is *isStart*'(M) must be *True*.

The transition event ***PowerOnMachine*** corresponds to the function *PowerOnMachine*(*m*)(N) from domain ***PowerOnMachine*** → **NS** → **NS**. This function locates the machine descriptor and performs a power on transition.

*PowerOnMachine*(*m*)(N) = let ( AN=*Machines*(N) )
let ( *descriptor*=*AtIndex*(AN, *m*) )
let ( T=*PowerOn*(S'(*descriptor*))
⟨ *Replace*(*Machines*(N), *m*, *NewState*(*descriptor*)(T)), *Connections*(N) ⟩

The transition event ***RemoveMachine*** corresponds to the function *RemoveMachine*(m)(N) from domain ***RemoveMachine*** → **NS** → **NS**. This function removes the machine and its associated connections from the abstract network.

*RemoveMachine*(*m*)(N) = Let ( AN=*Machines*(N) )
Let ( AC=*Connections*(N) )
⟨ *Delete*(AN, *m*), *disconnectMachine*(AC, *m*) ⟩

This definition uses the auxiliary function *disconnectMachine*(AC, *m*) removes the connections involving machine *m* from the list of connections AC.

*disconnectMachine*(AC, *m*) = if *isEmpty*(AC)
then *None*
else let (*connection*=*First*(AC))
if *inMachine*(*connection*)=*m or outMachine*(*connection*)=*m*
then *disconnectMachine*(*Rest(*AC), *m*)
else ⟨ *connection*, *disconnectMachine*(*Rest(*AC), *m*) ⟩

The transition ***AddConnection*** corresponds to the function *AddConnection*(C)(N) which adds a connection to the list of established connection in the network.

*AddConnection*(C)(N) = ⟨ *Machines*(N), *Append*(*Connections*(N), C) ⟩

The transition ***RemoveConnection*** corresponds to the function *RemoveConnection*(*n*)(N) which aremoves connection number *n* from the list of established connection in the network.

*RemoveConnection*(*n*)(N) = ⟨ *Machines*(N), *Delete*(*Connections*(N), *n*) ⟩

The transition ***StopNetwork*** doesn't have a corresponding function. It indicates the computation has ended.

*The Computation Carried Out by the Network*

The initial state of the network is the triple ⟨ *None*, *None*, *Events* ⟩ where the lists of machines and connections are empty and *Events* is a tape of network events to occur. This tape is one that corresponds to non deterministic events so it is an ideal candidate for replacement by a probabilistic choice. In fact a probabilistic choice is probably a more intuitive description of the network semantics because events occurs in a network in a very loosely correlated manner. When we use a tape it may be seen as a record of events to occur. Then human intuition may give a pre-determined interpretation to such a record in the sense that the human may understand the events are decided before the computation begins. This interpretation is erroneous because it introduces an operational element (the moment in time when the events are determined) in a denotational definition. This tape is intended to be interpreted as the sequence of nondeterministic choices that will be made in the course of computation. No execution order is implied.

The *Events* tape is subject to a number of conditions:

- *isStart*'(M) must be *True* of each machine M involved in an ***AddMachine*** transition.
- ***MachineTransition***, ***PowerOnMachine*** and ***RemoveMachine*** transitions must always apply to machines that are already in the list of machines.
- ***MachineTransition*** transitions must always apply to machines that are in *Running* status.
- ***PowerOnMachine*** transitions must always apply to machines that are in *Stopped* status.
- ***RemoveConnection*** transitions must always apply to connections that have already been established.
- At most one transition ***StopNetwork*** is present and must always be the boolean value *True*.

Then the computation proceeds by executing the transitions corresponding to the *Events* tape one after another until the network is stopped. This gives us a loop:

1. The initial network state $NS_0$ is $\langle$ *None*, *None*, *Events* $\rangle$ as indicated above.
2. An *event* is read from the tape *Events*.
3. If *event* is in ***StopNetwork*** then the computation stops
4. Else the next network state $NS_{i+1}$ is calculated from the event and network state $NS_i$

This corresponds to the formula

*NetworkRun*(*Events*) = *Execute*( $\langle$ *None*, *None*, *Events* $\rangle$ )

```
Execute(N) = let ( event=readTransitionTape(N) )
                    if isStopNetwork(event)
                       then N
                       else (moveTransitionTape;
                             if isMachineTransition(event) then MachineTransition(event)
                                else if isAddMachine(event) then AddMachine(event)
                                else if isPowerOnMachine(event) then PowerOnMachine(event)
                                else if isRemoveMachine(event) then RemoveMachine(event)
                                else if isAddConnection(event) then AddConnection(event)
                                else if isRemoveConnection(event) then RemoveConnection(event);
                             Execute
                            )(N)
```

The *Execute* function reads a network event descriptor from the tape and then move the tape and execute according to the event descriptor until told to stop the network computation. The *NetworkRun* function starts the *Execute* loop with the initial event list.

This definition allows abstract machines that are turned on and off as the network is running provided they are defined as was discussed previously.

This definition implicitly serialize the event execution due to their sequencing on the tape. The probabilistic choice alternative will have the same effect. This is a limitation of lambda-calculus which is a sequential computation model. This serialization is of no consequence for the parallel execution of independent machine transitions is serializable. But from a philosophical perspective the use of an inherently distributed model of computation should be superior.

*Unreliable Communication Links*

This treatment of network connections assumes that the data transfer between the emission of output and the reception of input is faultless. Real life data exchange is subject to various problems like data loss, data alteration and reception of spurious data. It is also subject to bugs in device drivers that may cause attempts to read the input before the output is ready. If such phenomena need to be factored in the definition. The solution I propose is the use what I call a *mangling receptionist*.

The mangling receptionist is an abstract machine that is inserted between the source of output and the

reception of input to capture the effect on the data of communication problems[15]. The receiving machine doesn't get the data directly from the sender. It gets it from the mangling receptionist who uses a nondeterministic input tape to determine if the data is passed as sent from the output or if some altered version must be used. The abstraction is that an unreliable communication channel has computational properties. It alters the data on a probabilistic basis. The job of the mangling receptionist abstract machine to to do these alterations. If data has been received from the sender the receptionist gives it. If no data has been received and the request for data is premature, the receptionist answers "*None".* If the data is lost, the receptionist doesn't pass it through either and if it has been received altered the receptionist passes the altered version. In case of spurious data injected in the inputs the mangling receptionist reads them from the mangling tape and introduces this information between two pieces of genuine information.

The loop body of the mangling receptionist looks like this:

```
ManglingLoop(S) =
        Let Action = ReadTape(S, ManglingTape)
            (MoveTape(ManglingTape);
             If Action = "Received"
                then (WriteTape(output, ReadTape(S, input) );
                      MoveTape(input);
                      MoveTape(output)
                     )
             If Action = "Dropped"
                then MoveTape(input)
            else if Action = "Altered"
                then (WriteTape(output, ReadTape(S, input) xor ReadTape(S, ManglingTape) );
                      MoveTape(ManglingTape);
                      MoveTape(input);
                      MoveTape(output)
                     )
            else if Action = "Spurious"
                then (WriteTape(output, ReadTape(S, ManglingTape) );
                      MoveTape(ManglingTape);
                      MoveTape(output)
                     )
            else if Action = "NotYetReceived"
                then (WriteTape(output, None );
                      MoveTape(output);
                      )
            ) (S)
```

A possible interpretation of the mangling receptionist is to view the communication channel as an abstract machine with data mangling capabilities. The output tape of the sending machine is connected to the input of the mangling receptionist. The output of the mangling receptionist is connected to the input of the receiving machine. So the mangling receptionist is an eavesdropper on the communication but for the purpose of defining the abstract network with formulas it is treated as a machine in the network as any other machine.

Note that the loop of the mangling receptionist is executing an instruction set except the the instructions are not stored in memory. They are read from the mangling tape.

*Shared Communication Channels*

The definition of abstract network assumes all communication links are unidirectional point to point links. Bidirectional links are represented with a pair of unidirectional links going in opposite directions. Shared channels like wireless, satellite, coaxial ethernet cables and 10Base-T hubs are represented by an abstract machine. Again the abstraction is that the communication channel has some computational property, it replicates the data to many recipients. For purpose of defining the semantics of an abstract network by a mathematical formula the communication channel is treated as a node in the network where the actual machines must connect. The shared channel loop body looks like this:

```
SharedChannelLoop(S) = Let ( InputList = ReadTape(S, WhoIsTalkingNow) )
```

```
                    MoveTape(S, WhoIsTalkingNow);
                    if length(InputList) = 1
                       then Let (Data = ReadTape(S, First(InputList))) )
                                (MoveTape(First(InputList));
                                 WriteAllOutputTapes(Data)
                                )
                       else (MoveAllInputTapes(InputList); WriteAllOutputTapes(Collision))
```

The shared channel is treated like a star shaped point to point network with the hub replicating the data from any single source to all the nodes. The *WhoIsTalkingNow* tape returns a list of the input tapes (called *InputList*) that are sending data at the same time. There must be at most one of them talking, otherwise a collision occurs. If exactly one is sending data then the data it replicated to all output tapes. If more than one is sending data, then the special value *Collision* is sent under the assumption the receiving port at the other end is able to handle this condition. This loop replicates the situation where the collision is seen by all receiving device or none of them which is built into some shared communication channel protocols. As an alternative it is possible to write a loop that will mathematically calculate the propagation delays of signals between nodes and determine whether the collision is seen at one node while the data is successfully received at another node. Another alternative is to let the hub always replicate the data to all nodes and leave it to the mangling receptionists at the receiving ends to determine whether a collision occurred based on their mangling tapes.

This assumes the shared channel has an *OutputList* which includes all output tapes receiving data to the channel. There are three helper functions that handle tape operations on a list of tapes. *WriteAllTapes* replicate the data to all output tapes by calling *WriteAllOutputs* which does the actual job. This is used to implement the hub-like function. *MoveAllTapes* is used to move the input tapes involved in a collision.

```
WriteAllTapes(Data)(S) = WriteAllOutputs(Data, OutputList)(S)

WriteAllOutputs(Data, L)(S) = (If (L=None)
                                  then S
                                  else (WriteTape(First(L), Data);
                                        MoveTape(First(L));
                                        WriteAllOutputs(Data, Rest(L))
                                       )(S)
                               )(S)


MoveAllInputTapes(L)(S) = (If (L=None)
                              then S
                              else (MoveTape(First(L));
                                    MoveAllInputs(Rest(L))
                                          )(S)
                           )(S)
```

This concludes the discussion of abstract networks.

## References

[Aho 1974] Aho, Alfred V., Hopcroft, John E. and Ullman, Jeffrey D. *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Company 1974

[Hankin 2004] Hankin, Chris. *An Introduction to Lambda Calculi for Computer Scientists*, King's College Publications, 2004

[Minsky 1967] Minsky, Marvin L., *Computation, Finite and Infinite Machines*, Prentice-Hall, 1967

[Ramsey 2002] Ramsey, Norman, Pfeiffer, Avi, *Stochastic Lambda Calculus and Monads of Probability Distributions*, Conference Record of the 29th Annual ACM Symposium on Principles of Programming Languages (POPL'02) Available on-line

[Stoy 1981] Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, MIT Press, First Paperback Edition 1981

1 See [Aho 1974] p. 15.
2 See [Minsky 1967] p. 2
3 See [Stoy 1981] chapters 6 and 7 for an explanation of domain theory which is suitable for the purposes of this article. Chapter 8 explains how lambda-calculus is given a semantics in terms of domains.
4 Such a test function is not computable because it would solve the halting problem.
5 There exist a stochastic version of lambda-calculus which seems to provide for probabilistic choice. I don't use it because I am not familiar with it. See [Ramsey 2002]
6 A well written driver normally doesn't do such things. It will read from the keyboard port only when it knows there is valid data to be read, either by testing a status flag in an input register or by responding to an interrupt. But it doesn't matter. The abstract machine still needs to give a mathematical semantics to poorly written drivers. Even buggy software is mathematics.
7 Mathematical domains are independent from the physical means used to store the bits, They are only concerned with the information value of the bits.
8 If you wonder how *Order3* works think of the bubble sort algorithm. It compares every two adjacent pairs of values and if they are in the wrong order it swaps them. Repeat this until every adjacent pairs are in order. If you work out a bubble sort for three values and unroll the loop you get *Order3*. There is no need to exhaustively list all possible permutations of *r1, r2* and *r3* because the ones that are listed suffice to do the work of a bubble sort.
9 [Stoy 1981] is a reference manual for denotational semantics although the techniques used here are somewhat different from Stoy's due to the difference is purposes. He defines a semantics for programming languages while this paper shows how to define a semantics for a physical machine.
10 I am sure hardware engineers will point to better more efficient tests for checking all memory references go through valid page tables than this exhaustive list. The definition should be adapted to whatever is appropriate to match the hardware.
11 An older version of this document has written halt like this: *Halt*(S) = *While*(λS.*True*, λS.S)(S). A reader has commented that this is incorrect because the loop will not transfer control back to the instruction cycle where interrupt processing occurs. Therefore interrupts are not received executed. The corrected version lets the instruction cycle do the looping by not incrementing the instruction pointer. Then execution is stuck on the same HLT instruction until an interrupt cause a change of the value of the instruction pointer.
12 This may correspond to situations where the counter has been initialized wrong. The interface still receive data but it has nowhere to go because according to the counter the transfer should be complete.
13 This situation represents something like noise on the line. The DMA interface detects a signal but when comes the time to read the data nothing shows up. This definition reflects an ability of the hardware to detect such conditions and ignore the spurious signal. If the hardware has no such ability, just omit the test for None and proceed as if the tape will always have data to provide.
14 Some applications like cryptography may properly use the pseudo-randomness inherent in reading non-initialized part of computer memory.
15 There is no obligation to make abstract machines correspond to physical machines. Abstract machines are mathematical devices that may be used to represent all sort of physical phenomena including some that are not machines.