On the Computational Power of a Continuous-Space Optical Model of Computation

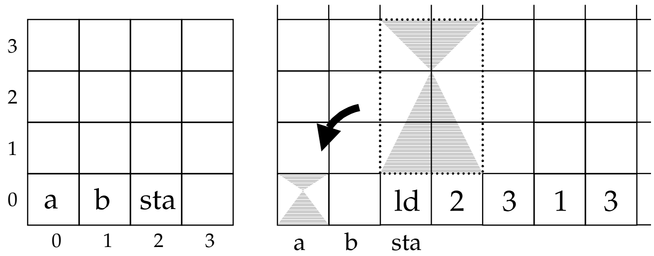
Thomas J. Naughton and Damien Woods

TASS Group, Department of Computer Science, National University of Ireland, Maynooth, Ireland. Corresponding author: dwoods@cs.may.ie

Abstract. We introduce a continuous-space model of computation. This original model is inspired by the theory of Fourier optics. We show a lower bound on the computational power of this model by Type-2 machine simulation. The limit on computational power of our model is nontrivial. We define a problem solvable with our model that is not Type-2 computable. The theory of optics does not preclude a physical implementation of our model.

1 Introduction

In this paper we introduce to the theoretical computer science community an original continuous-space model of computation. The model was developed for the analysis of (analog) Fourier optical computing architectures and algorithms, specifically pattern recognition and matrix algebra processors [3]. The functionality of the model is limited to operations routinely performed by optical scientists thus ensuring it is implementable within this physical theory [10]. The model uses a finite number of two dimensional (2-D) images of finite size and infinite resolution for data storage. It can navigate, copy, and perform other optical operations on its images. A useful analogy would be to describe the model as a random access machine, without conditional branching and with registers that hold images. This model has previously [4, 5] been shown to be at least as computationally powerful as a universal Turing machine (TM). However, its exact computational power has not yet been characterised. To demonstrate a lower bound on computational power we simulate a Type-2 machine. The upper bound is not obvious; the model can decide at least one language that a Type-2 machine can not. This combination of super-Turing power and possible implementation strongly motivates investigation of the model. In Sect. 2, we introduce the optical model of computation. In Sect. 3, we outline some relevant points from Type-2 Theory of Effectivity, and present our working view of Type-2 machines. In Sect. 4, we present our simulation of a Type-2 machine. We finish with a discussion of its super-Turing power and a conclusion (Sects. 5 and 6).



(a) (b)

Fig. 1. Schematics of (a) the grid memory structure of our model of computation, showing example locations for the 'well-known' addresses **a**, **b** and **sta**, and (b) loading (and automatically rescaling) a subset of the grid into grid element **a**. The program [ld | 2 | 3 | 1 | 3 | . . . | *hlt*] instructs the machine to load into default location **a** the portion of the grid addressed by columns 2 through 3 and rows 1 through 3.

2 The Optical Computational Model

Each instance of our machine consists of a memory containing a program (an ordered list of operations) and an input. The memory structure is in the form of a 2-D grid of rectangular elements, as shown in Fig. 1(a). The grid has finite size and a scheme to address each element uniquely. Each grid element holds a 2-D infinite resolution complex-valued image. Three of these images are addressed by the identifiers **a**, **b**, and **sta** (two global storage locations and a program start location, respectively).

The two most basic operations available to the programmer, **ld** and **st** (both parameterised by two column addresses and two row addresses), copy rectangular $m \times n$ ($m, n \in \mathbb{N}$, $m, n \geq 1$) subsets of the grid into and out of image **a**, respectively. Upon such loading and storing the image information is rescaled to the full extent of the target location [as depicted in Fig. 1(b)]. Two additional real-valued parameters z_{lower} and z_{upper}, specifying lower and upper cut-off values, filter the rectangle's contents by amplitude before rescaling,

$$f(i,j) = \begin{cases} z_{\text{lower}} & : \quad |f(i,j)| < z_{\text{lower}} \\ z_{\text{upper}} & : \quad |f(i,j)| > z_{\text{upper}} \\ f(i,j) & : \quad \text{otherwise} \end{cases} .$$

For the purposes of this paper we do not require the use of amplitude filtering and use an all-pass filter represented by the rationals 0/1 and 1/1 (we make use of the symbol '/' for this purpose). The complete set of atomic operations is given in Fig. 2.

Each instance of our machine is a quintuple $M = \langle D, L, Z, I, P \rangle$, in which

- $D = \langle x, y \rangle\,,\ \ x, y \in \mathbb{N}$: grid dimensions
- $L = \langle a_{\text{x}}, a_{\text{y}}, b_{\text{x}}, b_{\text{y}}, s_{\text{x}}, s_{\text{y}} \rangle\,,\ \ a, b, s \in \mathbb{N}$: locations of **a**, **b**, and **sta**

$\boxed{\mathrm{ld}}\boxed{c1}\boxed{c2}\boxed{r1}\boxed{r2}\boxed{z_\mathrm{l}}\boxed{z_\mathrm{u}}$	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_\mathrm{l}, z_\mathrm{u} \in \mathbb{Q}$; copy into **a** the rectangle defined by the coordinates ($c1$, $r1$) and ($c2$, $r2$). $(z_\mathrm{l}, z_\mathrm{u})$ is the amplitude filter [we use $(0/0, 1/1)$ everywhere for the purposes of this paper].
$\boxed{\mathrm{st}}\boxed{c1}\boxed{c2}\boxed{r1}\boxed{r2}\boxed{z_\mathrm{l}}\boxed{z_\mathrm{u}}$	: $c1, c2, r1, r2 \in \mathbb{N}$; $z_\mathrm{l}, z_\mathrm{u} \in \mathbb{Q}$; copy the image in **a** into the rectangle defined by the coordinates ($c1$, $r1$) and ($c2$, $r2$).
$\boxed{\mathrm{h}}$	: perform a horizontal 1-D Fourier transform on the 2-D image in **a**. Store result in **a**.
$\boxed{\mathrm{v}}$	: perform a vertical 1-D Fourier transform on the 2-D image in **a**. Store result in **a**.
$\boxed{\cdot}$	: multiply (point by point) the two images in **a** and **b**. Store result in **a**.
$\boxed{+}$	: perform a complex addition of **a** and **b**. Store result in **a**.
$\boxed{*}$	: replace **a** with the complex conjugate of **a**.
$\boxed{\mathrm{br}}\boxed{c1}\boxed{r1}$	: $c1, r1 \in \mathbb{N}$; unconditionally branch to the instruction at the image with coordinates ($c1$, $r1$).
$\boxed{\mathrm{hlt}}$	: halt.

Fig. 2. The set of atomic operations permitted in the model.

- $Z = \langle z_{\mathrm{MIN}}, z_{\mathrm{MAX}}, r \rangle, \;\; z \in \mathbb{C}, \;\; r \in \mathbb{Q}$: global amplitude bounds and amplitude resolution of grid elements
- $I = [(i_{1\mathrm{x}}, i_{1\mathrm{y}}, \psi_1), \ldots, (i_{n\mathrm{x}}, i_{n\mathrm{y}}, \psi_n)], \;\; i \in \mathbb{N}, \;\; \psi \in \mathit{Image}$: the n inputs and their locations, where Image is a complex surface bounded by 0 and 1 in both spatial directions and with values limited by Z
- $P = [(p_{1\mathrm{x}}, p_{1\mathrm{y}}, \pi_1), \ldots, (p_{m\mathrm{x}}, p_{m\mathrm{y}}, \pi_m)], \;\; p \in \mathbb{N}, \;\; \pi \in \{$ld, st, h, v, $*$, $\cdot$, $+$, br, hlt, $/$, $N\} \subset \mathit{Image}$: the m programming symbols, for a given instance of the machine, and each of their locations. $N \in \mathit{Image}$ represents an arbitrary row or column address.

As might be expected for an analog processor, its programming language does not support comparison of arbitrary image values. Fortunately, not having such a comparison operator will not impede us from simulating a branching instruction (see Sect. 4). In addition, address resolution is possible since (i) our set of possible image addresses is finite (each memory grid has a fixed size), and (ii) we anticipate no false positives (we will never seek an address not from this finite set).

3 Type-2 Theory of Effectivity

Standard computability theory [8] describes a set of functions that map from one countably infinite set of finite symbol sequences to another. In "Type-2 Theory of Effectivity" (TTE) [9], 'computation' refers to processing over infinite sequences of symbols, that is, infinite input sequences are mapped to infinite output sequences. If we use two or more symbols the set of such sequences

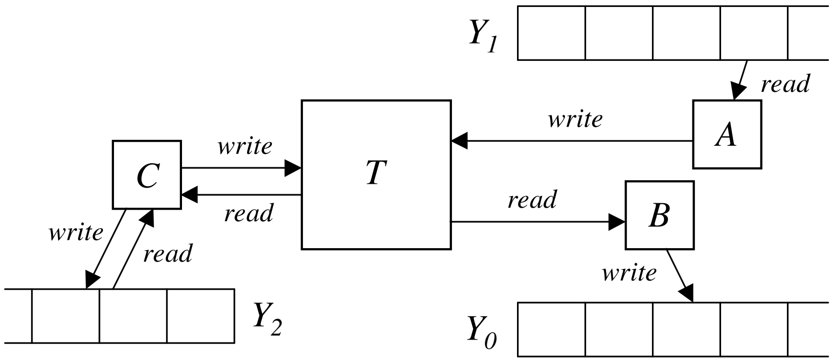


Fig. 3. Our working view of a Type-2 machine: (T) a halting TM; (Y_0) the output tape; (Y_1) the input tape; (Y_2) the nonvolatile 'work tape'. Controls A, B, and C represent the functionality to read from Y_1, write to Y_0, and read/write to Y_2, respectively.

is uncountable; TTE describes computation over uncountable sets and their subsets. The following is a definition of a Type-2 machine as taken from [9].

> **Definition 1.** *A Type-2 machine M is a TM with k input tapes together with a type specification* $(Y_1, \ldots, Y_k, Y_0)$ *with* $Y_i \in \{\Sigma^*, \Sigma^\omega\}$*, giving the type for each input tape and the output tape.*

In this definition, Σ is a finite alphabet of two or more symbols, Σ^* is the set of all finite length sequences over Σ, Σ^ω is the set of all infinite length sequences over Σ. There are two possible input/output tape types, one holds sequences from Σ^* and the other from Σ^ω.

Input tapes are one-way read only and the output tape is one-way write only. If the output tape restriction was not in place any part of an infinite output would not be guaranteed to be fixed as it could possibly be overwritten during a future computation step. Hence, finite outputs from Type-2 computations are useful for approximation or in the simulation of possibly infinite processes. A Type-2 machine either finishes its computation in finite time with a finite number of symbols on its output tape, or computes forever writing an infinite sequence. Machines that compute forever while outputting only a finite number of symbols are undefined in Type-2 theory [9].

3.1 A New View of Type-2 Computations

We maintain that a Type-2 machine can be viewed as a repeatedly instantiated halting TM that has an additional (read-only) input tape Y_1 and an additional (write-only) output tape Y_0. (Without loss of generality, the finite number of input tapes from Def. 1 can be mapped to a single tape.) A nonvolatile 'work tape' Y_2 is used to store symbols between repeated instantiations (runs) of the halting TM. This is illustrated in Fig. 3. T is the halting TM. Control A represents the functionality to read from Y_1. Control B represents the functionality to write to Y_0. Control C represents the functionality to write to and read from Y_2.

A Type-2 computation will proceed as follows. The halting TM T is instantiated at its initial state with blank internal tape(s). It reads a symbol from Y_1. Combining this with symbols (if any) left on Y_2 by the previous instantiations, it can, if required, write symbols on Y_2 and Y_0. T then halts, is instantiated once more with blank internal tape(s), and the iteration continues. The computation will either terminate with a finite sequence of symbols on the output tape or compute forever writing out an infinite sequence. In this light, an infinite Type-2 machine computation corresponds to an infinite sequence of instantiations of a single halting TM (plus extra computation steps for controls A, B, and C).

4 Simulation

We use simulation as a technique to measure computational power. If we can show that machine M_0 can simulate every operation that machine M_1 performs, we can say that M_0 is at least as powerful as M_1. Universality for our machine has already been proved [5, 4] following Minsky's arithmetisation of TMs [2] (representing a TM in terms of quadruples of integers). Four images were used to represent Minsky's four registers. In this paper our TM simulation is more efficient, reducing the number of required commands from 52 to 17. We refine this TM simulation into a Type-2 machine simulation. Although straightforward, these simulations are technically nontrivial. We were required to overcome the restriction of no conditional branching in our model and to define an appropriate view of Type-2 computations.

In general, a TM could be simulated by a look-up table and the two stacks **m** and **n**, as shown in Fig. 4. A given TM [such as that in Fig. 4(a)] is written in the imperative form illustrated in Fig. 4(b), where the simulation of state changes and TM tape head movements can be achieved with two stacks and two variables as shown in Fig. 4(c).

In order to simulate a stack we previously effected indirect addressing with a combination of program self-modification and direct addressing. We also simulated conditional branching by combining indirect addressing and unconditional branching [5, 4]. This was based on a technique by Rojas [6] that relied on the fact that our set of symbols is finite. Without loss of generality, in our simulation we will restrict ourselves to three possible symbols, '0', '1' and a blank symbol 'b'. Then, the conditional branching instruction "if (α='1') then jump to address X, else jump to Y" is written as the unconditional branching instruction "jump to address α". We are required only to ensure that the code corresponding to addresses X and Y is always at addresses '1' and '0', respectively (and that we take into account the case where α ='b'). In a 2-D memory (with an extra addressing coordinate) many such branching instructions are possible.

q	s	s'	d	q'
0	0	1	R	1
0	1	0	R	1
0	b	b	R	2
1	b	b	L	3

0: initial state
1: moving left
2: rejecting halt
3: accepting halt

(a)

```
q := initial_state;
halt := false;
while (halt = false) {
  select case (q, s) {
    (0,0): fn(1,R,1);
    (0,1): fn(0,R,1);
    (0,b): fn(b,R,2);
    (1,b): fn(b,L,3);
    else: halt := true;
  }
}
```

(b)

```
void fn(s',d,q') {
  if(d = R) {
    push(m,s');
    s := pop(n);
  } else {
    push(n,s');
    s := pop(m);
  }
  q := q';
}
```

(c)

Fig. 4. Figure showing (a) an example TM table of behaviour. This machine flips the binary value at its tape head and halts in an accepting state. If there is a blank at its tape head it halts in a rejecting state; (b) an illustration of how an arbitrary TM table of behaviour might be simulated with pseudocode; (c) how one might effect a TM computation step with stacks **m** and **n**.

4.1 Shorthand Conventions

To facilitate persons reading and writing programs, a shorthand notation is used (see Fig. 6). In this shorthand, instead of having to specify exact addresses, we give images a temporary name (such as 'x1') and refer to the address of that image with the '&' character. So, when the programmer writes

	0	1	2	3	4
5	st	&y1	br	0	y1

(s)he would intend

	0	1	2	3	4	5	6	7	8	9	10	11	12	13
5	st	13	13	5	5	0	/	1	1	/	1	br	0	

where the blank or undefined image at coordinates $(13, 5)$ will be overwritten with the contents of **a** before the statement to which it belongs is executed. Expansion from this shorthand to the long-form programming language is a mechanical procedure that could be performed as a 'tidying-up' phase by the programmer or by a pre-processor. Unless otherwise stated, we assume that the global bounds on image amplitude values are $z_{\mathrm{MIN}} = 0$ and $z_{\mathrm{MAX}} = 1$. The load and store operations contain 0/1 (= 0) and 1/1 (= 1) for their z_{lower} and z_{upper} parameters, respectively, indicating that the complete image is to be accessed. As a convention we use underlining in program grid elements whose images can be modified by the machine and italics to highlight points of TM termination within the grid.

4.2 Push and Pop Routines

Images can be stacked using a stepwise rescaling technique. Take an empty image, representing an empty stack, and an image i to be pushed onto the stack. Place both side-by-side with i to the left and rescale both into a single grid element. This could be regarded as an image stack with one element. Take another image, j, place it to the left of the stack and rescale both into a single grid element once again. This single image is a stack with two elements. A pop operation would involve stretching this stack over two grid elements. The left-hand image will then contain a single image (j in this case) and the right-hand image will contain the remainder of the stack. The stack can be repeatedly rescaled over two images popping a single image each time.

An implementation of such a stack system in our model requires indirectly addressing the stack and employing a third image (in addition to **a** and **b**) named **c**. The low-level details are as follows. In advance of the push operation we ensure that the column number of the stack is stored in **a** and the element to be pushed is stored in **c**. (The column number is sufficient as all stacks will be located on row 99 in Fig. 5.) The push routine begins by copying the contents of the stack into **b** and moving **c** into **a**. Then both **a** and **b** are rescaled into **a** and stored back in the stack's original location. In advance of a pop operation we ensure that the column number of the stack in question will be stored in **a**. The pop routine begins by loading the stack into **a** and rescaling it over both **a** and **b**. Image **a** now contains the top image and **b** contains the remainder of the stack. The top element is temporarily stored in **c**, the contents of the stack stored back in its original location, and the popped element returned to **a** before the routine ends.

We use these routines to simulate the movement of the TM tape head as illustrated in Fig. 4(c). The operational semantics of push and pop (including the use of self-modification to load the contents of a stack into **a**) can be found in rows 7 and 8 of Fig. 5.

4.3 Type-2 Machine Simulation

An arbitrary Type-2 machine is incorporated into our simulation as follows. Firstly, transform the Type-2 machine into a Type-2 machine that operates over our alphabet. Then rewrite the machine to conform to the form shown in Fig. 3. For the purposes of this simulation we represent Y_2 with TM T's internal tape (essentially using the semi-infinite tape to the left of the tape head). When T halts it will either be in an accepting or rejecting state. T's accepting state is equivalent to the simulator's initial state (i.e. T passes control back to the simulator when it halts). At the simulator's initial state it checks if T's tape head was at a non-blank symbol when T halted. If so, it writes that symbol to Y_0. All symbols to the left of the tape head (essentially the contents of Y_2) will be retained for the next instantiation of T. Next, the simulator reads a symbol from Y_1 and writes it on T's tape in the cell being scanned by T's tape head. It then passes control to T, by going into T's initial state. If at any time T halts in a rejecting state we branch to the simulator's halt state. In Fig. 5, we simulate a specific example of a Type-2 machine that flips the bits of its binary input. If the input is an infinite sequence it computes forever, writing out an infinite sequence of flipped bits. If the input is finite it outputs a finite sequence of flipped bits.

	m		s	n		Y_1		Y_0		'0'	'1'	'b'						sta			a	b	c
99	1	$\underline{\emptyset}$	$\underline{\emptyset}$	4	$\underline{\emptyset}$	6	$\underline{\mathbf{?}}$	8	$\underline{\emptyset}$	0	1	2						br	0	2	$\underline{\emptyset}$	$\underline{\emptyset}$	$\underline{\emptyset}$
pop: 8	st	&x1	st	&x2	st	&x3	st	&x4	ld	$\underline{\mathbf{x1}}$	$\underline{\mathbf{x2}}$	st	ab	st	c	ld	b	st	$\underline{\mathbf{x3}}$	$\underline{\mathbf{x4}}$	ld	c	ret
psh: 7	st	&x5	st	&x6	st	&x7	st	&x8	ld	$\underline{\mathbf{x5}}$	$\underline{\mathbf{x6}}$	st	b	ld	c	ld	ab	st	$\underline{\mathbf{x7}}$	$\underline{\mathbf{x8}}$	ret		
mvr: 6	Ph	m	Pp	n	st	s	ret																
mvl: 5	Ph	n	Pp	m	st	s	ret																
acc: 4	br	0	*s																				
rej: 3	hlt																						
2	Pp	Y_1	st	s	br	qS	0																
1	ld	s	Ph	Y_0	br	0	2																
0	ld	s	Ph	Y_0	br	0	2																
	0	1	2	3	4 . . .																		

2					'b'	R	q2		'b'	L	q3		*br*	*rej*		
1					'0'	R	q1								*br*	*acc*
0	br	q0	*s		'1'	R	q1								*br*	*acc*
	qS				q0				q1				q2		q3	

Fig. 5. Simulating Type-2 machines on our model of computation. The machine is in two parts for clarity. The larger is a universal Type-2 machine simulator and the smaller is its halting TM table of behaviour. [The example TM we use here is that in Fig. 4(a).] The simulator is written in a compact shorthand notation. The expansions into sequences of atomic operations are shown in Fig. 6 and the simulation is explained in Sect. 4.4.

(a)

Pp	Y_1	→	ld	Y_1	br	pop		
Pp	m	→	ld	m	br	pop		
Pp	n	→	ld	n	br	pop		
Ph	m	→	st	c	ld	m	br	psh
Ph	n	→	st	c	ld	n	br	psh
Ph	Y_0	→	st	c	ld	Y_0	br	psh

(b)

br	q0	*s	→	ld	s	st	&y1	br	q0	**y1**
br	0	*s	→	ld	s	st	&y2	br	0	<u>**y2**</u>

(c)

‘b’	R	q2	→	ld	‘b’	br	mvr	br	q2	*s
‘0’	R	q1	→	ld	‘0’	br	mvr	br	q1	*s
‘1’	R	q1	→	ld	‘1’	br	mvr	br	q1	*s
‘b’	L	q3	→	ld	‘b’	br	mvl	br	q3	*s

(d)

ld	Y_1	→	ld	5	5	99	99	0	/	1	1	/	1
ld	Y_0	→	ld	7	7	99	99	0	/	1	1	/	1
ld	m	→	ld	0	0	99	99	0	/	1	1	/	1
ld	n	→	ld	3	3	99	99	0	/	1	1	/	1
st	Y_1	→	st	5	5	99	99	0	/	1	1	/	1
st	Y_0	→	st	7	7	99	99	0	/	1	1	/	1
st	m	→	st	0	0	99	99	0	/	1	1	/	1
st	n	→	st	3	3	99	99	0	/	1	1	/	1
ld	s	→	ld	2	2	99	99	0	/	1	1	/	1
ld	b	→	ld	21	21	99	99	0	/	1	1	/	1
ld	c	→	ld	22	22	99	99	0	/	1	1	/	1
st	s	→	st	2	2	99	99	0	/	1	1	/	1
st	b	→	st	21	21	99	99	0	/	1	1	/	1
st	c	→	st	22	22	99	99	0	/	1	1	/	1
ld	ab	→	ld	20	21	99	99	0	/	1	1	/	1
st	ab	→	st	20	21	99	99	0	/	1	1	/	1
ld	‘0’	→	ld	9	9	99	99	0	/	1	1	/	1
ld	‘1’	→	ld	10	10	99	99	0	/	1	1	/	1
ld	‘b’	→	ld	11	11	99	99	0	/	1	1	/	1

(e)

br	pop	→	br	0	8
br	psh	→	br	0	7
br	mvr	→	br	0	6
br	mvl	→	br	0	5
br	*acc*	→	br	0	4
br	*rej*	→	br	0	3

(f)

st	&x1	→	st	9	9	8	8	0	/	1	1	/	1	
ld	<u>**x1**</u>	<u>**x2**</u>	→	ld	<u>**x1**</u>	<u>**x2**</u>	99	99	0	/	1	1	/	1
st	<u>**x3**</u>	<u>**x4**</u>	→	st	<u>**x3**</u>	<u>**x4**</u>	99	99	0	/	1	1	/	1

Fig. 6. Time-saving shorthand conventions used in the simulator in Fig. 5. These are explained in Sect. 4.4.

4.4 Explanation of Figs. 5 and 6

The Type-2 simulation by our model is shown in Fig. 5. It consists of two parts (separated in the diagram for clarity). The larger is the simulator (consisting of a universal TM, functionality A, B, and C from Fig. 3, and stacks Y_1 and Y_0). A TM table of behaviour must be inserted into this simulator [the example TM is from Fig. 4(a)]. It has a straightforward encoding. Notice how for each row of the table of behaviour $\langle q, s, s', d, q' \rangle$ an ordered triple $\langle s', d, q' \rangle$ is placed at the location addressed by coordinates (q, s).

Row 99 of Fig. 5 is the 'well-known' row containing stacks, constants and the locations **a**, **b**, **c**, and **sta**. To permit indirect addressing, locations **m**, **n**, Y_1, and Y_0 store the column numbers of their respective stacks. Y_1 and Y_0 represent the one-way tapes from Fig. 3 (we pop from Y_1 and push to Y_0). The stack **m** encodes all symbols on T's tape to the left of tape head, and the stack **n** encodes all symbols on T's tape to the right of the tape head. Image **s** encodes the symbol currently being scanned by T's tape head. The blank constant 'b' is represented by '2'. Before execution begins, an input will have been encoded in stack Y_1, grid element $(6, 99)$, in place of symbol '**?**'. Control flow begins at the program start location **sta**, grid element $(17, 99)$, and proceeds rightwards until one of **br** or **hlt** is encountered.

In rows 7 and 8 we have the push and pop routines; these are explained in Sect. 4.2. Rows 5 and 6, named **mvl** and **mvr** respectively, contain commands to simulate moving left and right on T's tape. This is effected by a combination of pushes and pops to the stacks **m** and **n** and image **s** that encode T's tape symbols. The 'ret' command returns execution to the point from which a subroutine **mvl**, **mvr**, **psh**, or **pop** was called. Rows 3 and 4 describe what is to happen when the inserted TM halts in a rejecting or accepting state, respectively. In the former case the Type-2 machine simulation halts. In the latter case, the TM is reinstantiated in accordance with the description in Sect. 3.1. If the TM's tape head had written '0' or '1' as it halted, control flow then moves to row 0 or 1, respectively, and the appropriate symbol is written to Y_0 before instantiation. If the tape head writes a '2' (a blank) as it halted, control flow moves to row 2 and the TM is directly reinstantiated.

For our particular example inserted TM [from Fig. 4(a)] if it reads a '0' or '1' it halts in an accepting state and control moves to the beginning of row 4. If the input to our simulation is finite, an instantiation will eventually read a blank symbol, will enter state 'q2', and the Type-2 simulation will halt. Otherwise, it will be repeatedly instantiated, each time flipping one bit.

The simulation is written in a shorthand notation (including shorthand versions of **ld**, **st**, and **br** from Fig. 2) which is expanded using Fig. 6. Figure 6(a) shows the expansion of shorthand notation used in setting up calls to the **psh** and **pop** routines (loading the appropriate stack address into **a** in advance of a pop, and storing into **c** the symbol to be pushed and loading the stack address into **a** in advance of a push). In advance of a **psh** the element to be pushed will be in **a**. After a **pop**, the popped element will be in **a**. Figure 6(b) shows commands for branching to an address where the row is specified by the symbol currently scanned by T's tape head. Figure 6(c) shows routines for simulating the execution of a row of T's table of behaviour. Commands for loading from and storing to locations specified at runtime, and to/from the 'well-known' locations on row 99 are in Fig. 6(d). Figure 6(e) shows the commands for branching to subroutines. Finally, Fig. 6(f) illustrates how all labels are eventually given absolute addresses by a preprocessor. After a first pass of the preprocessor (expanding the shorthand) the modifiable (underlined) references are updated with hardcoded addresses.

5 Super-Turing Capabilities of Our Model

Type-2 machines do not describe all of the computational capabilities of our model. The model's atomic operations operate on a continuum of values in constant time (independent of input size) and would not have obvious TM or Type-2 machine implementations.

Consider the language L defined by its characteristic function $f : \Sigma^\omega \to \{0, 1\}$, where

$$f(p) := \begin{cases} 1 & : \text{ if } p \neq 0^\omega \\ 0 & : \text{ otherwise} \end{cases}$$

and where p is an infinite sequence over alphabet $\{0, 1\}$. This language is acceptable but not decidable by a Type-2 machine [9] (Ex. 2.1.4.6). In our model, we encode a boolean value in an image by letting a δ-function at its origin denote a '1' and an empty image (or an image with low background noise) denote a '0'. An infinite sequence of boolean-valued images could be presented as input concatenated together in one image without loss of information (by definition, images in our machine have infinite spatial resolution). An off-centre peak can be centred for easy detection through Fourier transformation (using the shorthand program $\boxed{\text{ld}}\boxed{\text{Y}_1}\boxed{\text{h}}\boxed{\text{v}}\boxed{\text{st}}\boxed{\text{b}}\boxed{*}\boxed{\cdot}\boxed{\text{h}}\boxed{\text{v}}$). This uses the property that the term at the origin of a Fourier transform of an image, the dc term, has a value proportional to the energy over the entire image. Our model could therefore Fourier transform the continuous input image and then measure the value of the dc term in unit time. A peak would indicate that there is some energy (and therefore at least one '1') somewhere in the image; the corresponding word is in L. An absence of a peak at the origin indicates that there is not a '1' in the image; the corresponding word is not in L.

6 Conclusion

We introduce an original continuous-space model of computation to the computer science community. We show its relationship to existing theory by proving that it is at least as powerful as the Type-2 machine model. We are currently investigating the relationship between our model and piecewise affine maps [1, 7]. As the model remains faithful to the theory of physical optics we propose that it could serve as an implementation of Type-2 machines. Furthermore, the upper bound on the computational power of this model is nontrivial and deserves to be analysed. We present at least one problem, decidable by our model, that is undecidable by a Type-2 machine. The model's computational power, combined with implementation possibilities, strongly motivates continued investigation.

Acknowledgements

We gratefully acknowledge advice and assistance from J. Paul Gibson and the Theoretical Aspects of Software Systems research group, NUI Maynooth. Many thanks also to the reviewers of this paper for their constructive comments.

References

[1] P. Koiran, M. Cosnard, and M. Garzon. Computability with low-dimensional dynamical systems. *Theoretical Computer Science*, 132(1-2):113–128, 1994.

[2] M.L. Minsky. *Computation: Finite and Infinite Machines.* Series in Automatic Computation. Prentice Hall, Englewood Cliffs, New Jersey, 1967.

[3] T. Naughton, Z. Javadpour, J. Keating, M. Klíma, and J. Rott. General-purpose acousto-optic connectionist processor. *Optical Engineering*, 38(7):1170–1177, July 1999.

[4] T.J. Naughton. A model of computation for Fourier optical processors. In *Optics in Computing 2000*, Proc. SPIE vol. 4089, pp. 24–34, Quebec, June 2000.

[5] T.J. Naughton. Continuous-space model of computation is Turing universal. In *Critical Technologies for the Future of Computing*, Proc. SPIE vol. 4109, pp. 121–128, San Diego, California, August 2000.

[6] R. Rojas. Conditional branching is not necessary for universal computation in von Neumann computers. *J. Universal Computer Science*, 2(11):756–768, 1996.

[7] H.T. Siegelmann and E.D. Sontag. On the computational power of neural nets. In *Proc. 5th Annual ACM Workshop on Computational Learning Theory*, pp. 440–449, Pittsburgh, July 1992.

[8] A.M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proceedings of the London Mathematical Society ser. 2*, 42(2):230–265, 1936. Correction in vol. 43, pp. 544–546, 1937.

[9] K. Weihrauch. *Computable Analysis.* Springer, Berlin, 2000.

[10] D. Woods and J.P. Gibson. On the relationship between computational models and scientific theories. Technical Report NUIM-CS-TR-2001-05, National University of Ireland, Maynooth, Ireland, February 2001.