A Universal Machine

Timothy Murphy
<tim@maths.tcd.ie>
School of Mathematics, Trinity College Dublin

April 21, 2003

Contents

CERTAIN TURING MACHINES have the remarkable property that they can emulate, or imitate, every other Turing machine. These *universal* machines play a central rôle in our theory. From our previous work, we can construct such a machine in any of four ways: as a Turing machine, as a program in the language $\mathcal{T}$, as a stack machine, or as a program in the language $\mathcal{S}$. We opt for the last, writing a program for a 4-stack machine U with the required universal property.

1 Pre-requisites

We have previously introduced the notion of a *Turing machine* T (following Chaitin's model), and the *Turing function*

$$T : \mathbb{S}^* \to \mathbb{S}^*$$

which it implements. Here $\mathbb{S}$ denotes the set of all finite strings of 0's and 1's, and $\mathbb{S}^* = \mathbb{S} \cup \perp$, while $T(s) = \perp$ means that $T(s)$ is undefined — either T enters an infinite loop when s is input, or T tries to read past the end of s, or T halts before reading in the whole of s.

We have also considered 3 other ways of implementing Turing functions:

- by a program in the language $\mathcal{T}$ (which is just another way of defining a Turing machine);
- by a stack machine S (using a number of stacks rather than a tape);
- by a program in the language $\mathcal{S}$ (which is just another way of defining a stack machine).

We shall define our machine by a program in $\mathcal{S}$. Recall that such a program consists of a number of statements, each of which takes one of the forms

$$\texttt{put0}, \texttt{put1}, \texttt{read}, \texttt{write}, \texttt{push}i, \texttt{pop}i, \texttt{jump} \pm j$$

followed by ';'. Here i is the number of a stack (in our case 0–3), while $\texttt{jump} \pm j$ invokes a jump from statement number n to statement number $n \pm j$.

We also allow labelled statements, eg

$$\textit{loop} : \texttt{read};$$

with a corresponding `jumpto`, eg

$$\texttt{jumpto}\ \textit{loop};$$

However, we do not regard this as part of the language $\mathcal{S}$ (or $\mathcal{T}$). It is understood that a 'pre-processor' will change the `jumpto` to an appropriate `jump`.

We also assume it known that

1. Turing machines (ala Chaitin) and 2-stack machines are equivalent in the sense that to each Turing machine T there corresponds a 2-stack machine S such that

$$S(s) = T(s) \quad \forall s \in \mathbb{S}^*;$$

   while conversely to each to each 2-stack machine S there corresponds a Turing machine T such that

$$T(s) = S(s) \quad \forall s \in \mathbb{S}^*.$$

2. For each $n \geq 2$, n-stack machines and 2-stack machines are equivalent, ie to each to each n-stack machine S there corresponds a 2-stack machine S' such that

$$S'(s) = S(s) \quad \forall s \in \mathbb{S}^*.$$

2 Universal Turing machines

Definition 1. *The Turing machine U is said to* emulate *the Turing machine T if there exists a string t such that*

$$U(ts) = f(s)$$

for every string $s \in \mathbb{S}$.

The Turing machine U is said to be universal *if it implements every Turing machine T.*

3 Existence

Theorem 1. *There exists a universal Turing machine.*

Proof. We have seen that each Turing function $f(s)$ can be defined by a 2-stack machine S:

$$f(s) = S(s)$$

for all $s \in \mathbb{S}$.

We are going to construct a 4-stack machine U such that

$$U(\langle S \rangle s) = S(s)$$

for every 2-stack machine S and every $s \in \mathbb{S}$, where $\langle S \rangle$ is the code for S described below.

This will establish the result, since we know that every stack machine implements a Turing function.

We shall define the stack machine U through a program in our language $\mathcal{S}$.

4 A code for 2-stack machines

Recall that a stack machine S is defined by a map

$$S : Q \times \mathbb{B} \to A \times Q,$$

where Q is the set of states $\{q_0, \dots, q_n\}$, $\mathbb{B} = \{0, 1\}$ are the bits that can appear in the accumulator, and A is the set of possible actions. Thus $S(q_i, b) = (a, q_j)$ means: if the machine is in state q_i and bit b is in the accumulator then action a is taken and the machine moves into state q_j.

In the case of a 2-stack machine there are 8 possible actions, which we code by 3 bits according to the following table:

`put0`	000
`put1`	001
`read`	010
`write`	011
`push0`	100
`push1`	101
`pop0`	110
`pop1`	111

Equivalently, if the machine has n states then it can be defined by $2n$ 'rules'

$$R[0,0], R[0,1], R[1,0], R[1,1], \dots, R[n-1,0], R[n-1,1],$$

where the rule $R[i,b]$ specifies the action $a = A(i,b)$ to be taken if the machine is in state i with b in the accumulator, as well as the new state $j = Q(i,b)$ it moves into.

We code this rule by

$$\langle R \rangle = \langle a \rangle [[]j] = \langle a \rangle 1^j 0,$$

where $\langle a \rangle$ is the code for the action defined above; and we code the 2-stack n-state machine S

$$\langle S \rangle = \langle R[0,0] \rangle 1 \langle R[0,1] \rangle 1 \langle R[1,0] \rangle 1 \cdots \langle R[n-1,1] \rangle 0.$$

Note that we mark the end of a rule with a 1, allowing us to signal the end of the rules with a 0.

Our universal machine U will have the property that

$$U(\langle S \rangle s) = S(s)$$

for every 2-stack machine S and every string $s \in \mathbb{S}$. Note that this implies in particular that $U(\langle S \rangle s)$ is defined if and only if $S(s)$ is defined.

5 The stack convention

We follow the (usual) convention of denoting the contents of a stack by a string s, with *the top of the stack corresponding to the end of the string.* Thus if

$$\texttt{stack}0 = 011,$$

then bit 1 is on the top of the stack; so `pop`0 would pop the 1 from the top of the stack and place it in the accumulator, eg

$$\texttt{pop}0 : (\textit{accumulator}, \texttt{stack}0) = (0, 011) \mapsto (1, 01),$$

while

$$\texttt{push}0 : (0, 011) \mapsto (0, 0110).$$

6 Winding and rewinding

When we read a string into a stack — or transfer a string from one stack to another — the order of its bits is reversed:

$$s \mapsto \overline{s}.$$

Thus, if we want the string in its original order, we must transfer it twice.

We shall use the term *winding* to describe the transfer of a rule (or other string) from reverse order on one stack (normally stack 2) to correct order on another stack (normally stack 3); and we shall use the term *rewinding* for the converse operation — transfer of a string in correct order on stack 3 to reverse order on stack 2.

When a rule appears in reverse order, we need to modify its code slightly, to

$$\overline{\langle R\rangle} = [[]j]\overline{\langle a\rangle} = 1^j 0\overline{\langle a\rangle};$$

and we modify the code for the machine similarly to

$$\overline{\langle S\rangle} = 1\overline{\langle R[n-1,1]\rangle}0\cdots0\overline{\langle R[0,0]\rangle}.$$

7 The 4 stacks and their functions

Each of the 4 stacks in U is assigned a specific function, as follows:

Stack 0 will hold the contents of S's stack 0.

Stack 1 will hold the contents of S's stack 1.

Stack 2 (the *rule stack*) will hold the — or part of the code — $\overline{[S]}$ for the stack machine S begin emulated, in reverse order (so that rule $R[0,0]$ is at the top of the stack).

Stack 3 (the *auxiliary stack*) will store part of $\langle S\rangle$ while we examine the appropriate rule on stack 2.

Stacks have the nice property that material can be stored temporarily on the top of the stack, provided we ensure that the material is cleared when we want to access the 'real' data at the bottom of the stack.

We use the tops of stacks 0 and 1 in this way. For example, we store the current S-state i temporarily as a string 01^i on top of stack 0; and we store the current S-bit on top of stack 1. Of course we must remove these temporary strings before emulating the action of S, which might involve pushing onto or popping from stacks 0 or 1.

8 Literate programming

While developping TeX, Donald Knuth introduced what he called "literate programming", a system in which a single file (called a Web file) contains both documentation and program code. The Web file can be processed in two ways; by the program *weave* to produce the documentation (as for example this document); or by the program *tangle* to produce the program code.

The Web file consists of a number of *chunks*, each of which has a TeX part and a program part (either of which may be empty). One program chunk can contain references to other program chunks, with the understanding that the latter are to be incorporated into the former when writing the program code.

The present document illustrates this system in practice. The program code produced by *tangle* is listed at the end of the document.

9 Program structure

Our program for U starts by reading the rules $[S]$ for the 2-state machine S that we are emulating into the 'rule-stack', stack 2. We want the rules to appear on this stack *in reverse order*, so that rule $R[0,0]$ is on top of the stack. To achieve this we first read the rules onto the auxiliary stack, stack 3, and then rewind them onto the rule stack, stack 2.

The machine U then enters the main loop, each circuit of which corresponds to emulation of a single step of S.

9 ⟨ Initialize stacks 10 ⟩
⟨ Read the rules of S onto the auxiliary stack 11 ⟩
⟨ Rewind rules onto rule stack 15 ⟩
MainLoop: ⟨ Wind through rule stack to correct rule 20 ⟩
⟨ Take specified action, and save S-bit on stack 1 26 ⟩
⟨ Check for end of program 33 ⟩
⟨ Save output S-state on stack 0 34 ⟩
⟨ Rewind rules onto rule stack 15 ⟩
put1 ; *jumpto MainLoop*;

10 Initialization

As explained above, we make temporary use of stacks 0 and 1 — between emulating steps of S — to store the current S-state and S-bit.

Of course these 'appendices' must be removed before we emulate each S-action.

Initially S is in state q_0, with 0 in its accumulator. To this end we place 0 initially at the bottom of stacks 0 and 1. We also place 0 at the bottom of stacks 2 and 3, to mark the end of the rules. (This is not actually necessary in the case of stack 2, the rule stack, since we assume that the rules never specify an output state outside the range for which rules are given.)

10 ⟨ Initialize stacks 10 ⟩ ≡
put0 ; *push0* ; *push1* ; *push2* ; *push3* ;
This code is used in chunk 9.

11 Reading in the rules

Recall that we start by reading the rules into the auxiliary stack (stack 3), before rewinding them onto the rule stack (stack 2).

After we have read into stack 3, stacks 0, 1 and 2 contain a single 0, while stack 3 holds

$$0\langle R[0,0]\rangle'1\langle R[0,1]\rangle'1\langle R[1,0]\rangle'1\cdots 1\langle R[n-1,1]\rangle'1,$$

where

$$\langle R[i,b]\rangle' = \langle A[i,b]\rangle 01^{Q[i,b]},$$

with the 0 on the other side of the 1^q since now we shall be reading from the top of the stack, ie from the end of the string.

11 ⟨ Read the rules of S onto the auxiliary stack 11 ⟩ ≡
⟨ Read action 12 ⟩
⟨ Read *outState* 13 ⟩
⟨ Read bit; if 1 read in next rule 14 ⟩

This code is used in chunk 9.

11.1 Reading the action

Recall that we code the action in 3 bits.

12 ⟨ Read action 12 ⟩ ≡
read; *push3*;
read; *push3*;
read; *push3*;

This code is used in chunk 11.

11.2 Reading the out-state

We read the out-state $[j] = 1^j 0$, and push it (onto stack 3) as 01^j. To simplify the code, we push the final 0 onto the stack, and then pop it.

13 ⟨ Read *outState* 13 ⟩ ≡
put0; *push3*;
read; *push3*; *jump* -2;
pop3;

This code is used in chunk 11.

11.3 Checking for the last rule

Recall that the last rule is signalled by a 0, earlier rules being succeeded by a 1. On the stack we replace the final 0 by a 1.

14 ⟨ Read bit; if 1 read in next rule 14 ⟩ ≡
read; *push3*; *jump* -14;
pop3;

This code is used in chunk 11.

12 Rewinding the rules

It is convenient to express this in a slightly more general form for later use; we assume that a certain number of rules $R[0,0], R[0,1], \ldots, R[i,b]$ are on stack 3 (with $R[0,0]$ on the bottom), while the remaining rules are already on stack 2.

At the end of this phase, all the rules have been transferred to stack 2; stacks 0, 1 and 3 contain a single 0, while stack 2 holds

$$0\langle R[n-1,1]\rangle^* 1 \langle R[n-1,0]\rangle^* 1 \langle R[n-2,1]\rangle^* 1 \cdots 1 \langle R[0,0]\rangle^*,$$

where

$$\langle R[i,b]\rangle^* = 01^{Q]i,b]}\overline{\langle A[i,b]\rangle},$$

$\overline{s}$ denoting the reverse of string s.

Note that each of the rules on stack 3 is preceded by a 1 (reading from the top of the stack), with the end of the rules signalled by a 0. For simplicity, we start near the end of our cycle, so that we can drop out when we see a 0.

15 ⟨ Rewind rules onto rule stack 15 ⟩ ≡
⟨ Rewind state 16 ⟩
⟨ Rewind action 17 ⟩
⟨ Check for bottom of stack, otherwise read in new rule 18 ⟩

This code is used in chunk 9.

16 ¶ ⟨ Rewind state 16 ⟩ ≡
put0; *push2*;
pop3; *push2*; *jump* -2;
pop2;

This code is used in chunk 15.

17 ¶ ⟨ Rewind action 17 ⟩ ≡
pop3; *push2*;
pop3; *push2*;
pop3; *push2*;

This code is used in chunk 15.

18 ¶ ⟨ Check for bottom of stack, otherwise read in new rule 18 ⟩ ≡
pop3; *push2*; *jump* -14;
pop2; *push3*;

This code is used in chunk 15.

13 The main cycle

At the start of the cycle, if S is in state q_i with b in the accumulator then the top of stack 0 holds 01^{2i+b}.

Our first task is to wind through $2i + b$ rules — the number of 1's on top of stack 0 — to reach the appropriate rule $R[i, b]$.

Note that this process will remove the 01^{2r+b} on top of stack 0.

20 ¶ ⟨Wind through rule stack to correct rule 20⟩ ≡

put1 ; *jumpto Test*;
NextRule: ⟨Wind single rule onto auxiliary stack 21⟩
Test: *pop0* ;
jumpto NextRule;

This code is used in chunk 9.

21 ¶ ⟨Wind single rule onto auxiliary stack 21⟩ ≡

⟨Wind action 22⟩
⟨Wind state 23⟩
⟨Wind marker bit 24⟩

This code is used in chunk 20.

22 ¶ ⟨Wind action 22⟩ ≡

pop2 ; *push3* ;
pop2 ; *push3* ;
pop2 ; *push3* ;

This code is used in chunk 21.

23 ¶ ⟨Wind state 23⟩ ≡

put0 ; *push3* ;
pop2 ; *push3* ; *jump* -2;
pop3 ;

This code is used in chunk 21.

24 ¶ ⟨Wind marker bit 24⟩ ≡

pop2 ; *push3* ;

This code is used in chunk 21.

14 Taking action

The 3-bit code specifying the action is now on top of stack 2. Essentially, we have to implement a simple jump-table — remembering to save the action-bits on stack 3.

26 ¶ $\langle$ Take specified action, and save S-bit on stack 1 26 $\rangle \equiv$

pop2; *push3*; *jumpto PushAndPop*;
$\langle$ Put and IO 27 $\rangle$
PushAndPop: $\langle$ Push and pop 30 $\rangle$
Done:

This code is used in chunk 9.

27 ¶ $\langle$ Put and IO 27 $\rangle \equiv$

pop2; *push3*; *jump*7;
$\langle$ Put 0 or 1 28 $\rangle$
$\langle$ Read or write 29 $\rangle$

This code is used in chunk 26.

28 ¶ $\langle$ Put 0 or 1 28 $\rangle \equiv$

pop1;
pop2; *push3*;
push1;
put1; *jumpto Done*;

This code is used in chunk 27.

29 ¶ $\langle$ Read or write 29 $\rangle \equiv$

pop2; *push3*; *jump*6;
pop1; *read*; *push1*;
put1; *jumpto Done*;
pop1; *write*; *push1*;
put1; *jumpto Done*;

This code is used in chunk 27.

30 ¶ $\langle$ Push and pop 30 $\rangle \equiv$

pop2; *push3*; *jump*14;
$\langle$ Push onto stack 0 or 1 31 $\rangle$
$\langle$ Pop from stack 0 or 1 32 $\rangle$

This code is used in chunk 26.

31 ¶ $\langle$ Push onto stack 0 or 1 31 $\rangle \equiv$

pop2; *push3*; *jump*6;
pop1; *push0*; *push1*;
put1; *jumpto Done*;
pop1; *push1*; *push1*;
put1; *jumpto Done*;

This code is used in chunk 30.

32 ¶ ⟨Pop from stack 0 or 1 32⟩ ≡

pop2; *push3*; *jump*6;
pop1; *pop0*; *push1*;
put1; *jumpto Done*;
pop1; *pop1*; *push1*;
put1; *jumpto Done*;

This code is used in chunk 30.

15 Checking if the program has ended

Recall that the S-program ends when the machine enters state 0 with 1 in the accumulator. This is signalled by a 0 on the top of stack 2, and a 1 on the top of stack 1.

33 ⟨Check for end of program 33⟩ ≡

pop2; *push2*; *jump*7;
pop1; *push1*; *jump*3;
put1; *jump*2;
halt;

This code is used in chunk 9.

16 Saving the output state

The output state j is now at the top of stack 2, while the accumulator bit b is on top of stack 1. We want to store 01^{2r+b} on top of stack 0, to indicate the number of the rule we need to access during the next cycle.

34 ⟨Save output S-state on stack 0 34⟩ ≡

put0; *push0*; *push3*;
pop2; *push3*; *push0*; *push0*; $jump - 4$;
pop0; *pop0*; *pop3*;
pop1; *push1*; *push0*; *jump*2;
pop0;

This code is used in chunk 9.

17 The program in $\mathcal{S}$

```
/*9:*/
/*10:*/
put0;push0;push1;push2;push3;/*:10*/
/*11:*/
/*12:*/
read;push3;
read;push3;
```

```
read;push3;/*:12*/
/*13:*/
put0;push3;
read;push3;jump-2;
pop3;/*:13*/
/*14:*/
read;push3;jump-14;
pop3;/*:14*//*:11*/
/*15:*/
/*16:*/
put0;push2;
pop3;push2;jump-2;
pop2;/*:16*/
/*17:*/
pop3;push2;
pop3;push2;
pop3;push2;/*:17*/
/*18:*/
pop3;push2;jump-14;
pop2;push3;/*:18*//*:15*/
MainLoop:
/*20:*/
put1;jumpto Test;
NextRule:/*21:*/
/*22:*/
pop2;push3;
pop2;push3;
pop2;push3;/*:22*/
/*23:*/
put0;push3;
pop2;push3;jump-2;
pop3;/*:23*/
/*24:*/
pop2;push3;/*:24*//*:21*/
Test:pop0;jumpto NextRule;/*:20*/
/*26:*/
pop2;push3;jumpto PushAndPop;
/*27:*/
pop2;push3;jump 7;
/*28:*/
pop1;
pop2;push3;
push1;
put1;jumpto Done;/*:28*/
```

```
/*29:*/
pop2;push3;jump 6;
pop1;read;push1;
put1;jumpto Done;
pop1;write;push1;
put1;jumpto Done;/*:29*//*:27*/
PushAndPop:/*30:*/
pop2;push3;jump 14;
/*31:*/
pop2;push3;jump 6;
pop1;push0;push1;
put1;jumpto Done;
pop1;push1;push1;
put1;jumpto Done;/*:31*/
/*32:*/
pop2;push3;jump 6;
pop1;pop0;push1;
put1;jumpto Done;
pop1;pop1;push1;
put1;jumpto Done;/*:32*//*:30*/
Done:/*:26*/
/*33:*/
pop2;push2;jump 7;
pop1;push1;jump 3;
put1;jump 2;
halt;/*:33*/
/*34:*/
put0;push0;push3;
pop2;push3;push0;push0;jump-4;
pop0;pop0;pop3;
pop1;push1;push0;jump 2;
pop0;/*:34*/
/*15:*/
/*16:*/
put0;push2;
pop3;push2;jump-2;
pop2;/*:16*/
/*17:*/
pop3;push2;
pop3;push2;
pop3;push2;/*:17*/
/*18:*/
pop3;push2;jump-14;
pop2;push3;/*:18*//*:15*/
```

```
put1;jumpto MainLoop;/*:9*/
```

18 Example

The following code $\langle S\rangle$ defines a 2-stack machine S which implements the function

$$[m][n] \mapsto [mn],$$

where

$$[n] = \underbrace{1\ldots1}_{n\ 1\text{'s}}0.$$

Now let us append the line

```
1110111110
```

to this file. Then the command

```
java StackProgram UniversalMachine < multiply.stackcode
```

will result in the output

```
1111111111111110
```

19 Summary

We have constructed a universal machine U which performs exactly like the 2-state machine S, provided the program is prefixed by the code $\langle S\rangle$ for the machine:

$$U\left(\langle S\rangle s\right) = S(s).$$

Index

List of Refinements