2013-10-29: Addition on Turing Machines

by Jay McCarthy

> The source for this post is online at 2013-10-29-tmadd.scrbl.

Categories: Languages Algorithms Theory Computer Science

Ever since my time as an undergraduate in computer science, I've been fascinated by automata and Turing machines in particular. However, something I always found disappointing against Turing machines as a model of computation, in contrast to the lambda calculus, was that I was never presented with useful Turing machines that did work anyone could care about. In this blog post, I try to rectify this by presenting a variety of machines that do addition.

-

1 Turing Machines

Formally, a Turing machine is a seven tuple: the finite set of states (Q), the finite set of symbols (Gamma), the blank symbol, the finite set of input symbols (that cannot contain the blank), the starting state (which must be in Q), the final states (a subset of Q), and the transition function. The transition function consumes a Q and a Gamma and returns a Q, Gamma, and the symbol `L` or `R`. The machine is interpreted relative to an infinite tape that contains all blank symbols, except just after the head, which contains a string of the input symbols. As the transition function is called, it overwrites symbols on the tape and moves the head.

A Turing machine gets its computational power from the infiniteness of the tape and the ability to move around on that infinite tape. It stays practical because the set of states and symbols are all finite, so you can't play too many games representing complex

situations. (For instance, you can easily have a Turing machine that "remembers" that it read a particular number, by having a state for "read the number 7". However, you cannot have a Turing machine with the state "read the number `n`" for every natural number `n`, because there are an infinite number of them.)

If the Turing machine is still too abstract for you, I hope these example machines will clarify it.

In order to help me in preparing these examples machines, I wrote a little simulation environment for Turing machines in Racket. It allows you to specify the seven tuple directly, but I found it a bit tedious to do that. I wrote a helper that allows you to just specify the transition function and it derives all the sets from the things that occur in the transition function. The only cases were this doesn't really work are for the blank symbol (which I assume will be `_`), the initial state (which I assume is the first one listed), and the final states (which I assume is named `HALT`).

2 Unary Addition

Let's look at the first example: unary addition.

A number represented in unary is essentially a string that is a long as the number. For instance, `**` would represent the number `2`.

The unary addition machine takes strings of the form `* ... + * ...` and reduces them to a single unary number which represents the sum of the two numbers. For instance, `* * + * * *` is rewritten to `* * * * *`. For convenience, all the machines I present will relocate the head to the start of the number before they halt.

The idea of the machine is reframe the problem as purely textual. It needs to replace the `+` with a `*` and then change the last `*` to `_`.

```
(define implicit-unary-addition
  (implicit-tm
   [consume-first-number
    [* (consume-first-number * R)]
    [+ (consume-second-number * R)]]
```

```
    [consume-second-number
     [* (consume-second-number * R)]
     [_ (override-last-* _ L)]]
    [override-last-*
     [* (seek-beginning _ L)]]
    [seek-beginning
     [* (seek-beginning * L)]
     [_ (HALT _ R)]]))
```

This process just has four states. In the first one, we skip over the first number (because the Turing machine has to write every transition, it just writes what it read). We've reached the end of the first number when we see `+`, so we write `*` and then go to the next state. In the second state, we skip over the second number. We know we've reached the end of when we see `_`, but the number is one `*` too long, so, we go back one `*` and replace it with `_`. Then, we go back to the beginning, skipping over the `*`s.

This process takes 15 steps to perform the addition `2 + 3`. The trace looks like this:

```
consume-first-number :  [*]*****  { 0}
consume-first-number :  *[*]****  { 1}
consume-first-number :  **[+]***  { 2}
consume-second-number:  ***[*]**  { 3}
consume-second-number:  ****[*]*  { 4}
consume-second-number:  *****[*]  { 5}
consume-second-number:  ******[_] { 6}
override-last-*      :  *****[*]_ { 7}
seek-beginning       :  ****[*]__ { 8}
seek-beginning       :  ***[*]*__ { 9}
seek-beginning       :  **[*]**__ {10}
seek-beginning       :  *[*]***__ {11}
seek-beginning       :  [*]****__ {12}
seek-beginning       : [_]*****__ {13}
HALT                 : _[*]****__ {14}
```

We don't really need to go back to the beginning, but it seems polite to and it makes the programs a little harder to write.

Something that is very important to understand about this machine is that the symbol `+` has no meaning. The Turing machine is *not* interpreting one arithmetic operation out of many possible operations it can perform. The `+` is just an arbitrary delimiter between the two numbers.

3 Binary Increment

Unary numbers are very unexciting, particularly because the problem of addition is just a textual operation. Binary numbers seem less like this, so let's switch to adding numbers in binary. However, it's going to be pretty complicated, so we'll start small: just incrementing a binary number by 1.

For example, if you have `0 0 1 0`, then it increments to `0 0 1 1`, which itself increments to `0 1 0 0`. If you study examples like this, you should see that when you increment, you just need to turn all the `1`s on the right into `0`s and turn the first `0` into a `1`. This is trivial to implement in a Turing machine:

```
(define implicit-binary-add1
  (implicit-tm
   [find-end
    [0 (find-end 0 R)]
    [1 (find-end 1 R)]
    [_ (zero-until-0 _ L)]]
   [zero-until-0
    [1 (zero-until-0 0 L)]
    [0 (HALT 1 L)]]))
```

It takes seven steps to increment `2` to `3`. And we don't even move the head back to the start!

```
find-end    : [0]010  {0}
find-end    : 0[0]10  {1}
find-end    : 00[1]0  {2}
find-end    : 001[0]  {3}
find-end    : 0010[_] {4}
zero-until-0: 001[0]_ {5}
HALT        : 00[1]1_ {6}
```

I hope that you're starting to see that it is useful to ignore the *meaning* of your Turing machine problem and focus on what the problem is textually.

4 Binary Decrement

Next, let's go the other direction and decrement binary numbers. For example, if you have `0 1 0 0`, then it decrements to `0 0 1 1`,

which itself decrements to `0 0 1 0`.

The key here is to switch the number into `1`s-complement (swapping all the bits), then increment that number, then redo the `1`s-complement. For convenience and to conserve steps, we can combine a few of the steps together. For instance, in the process of doing the `1`s-complement, we will find the end, so we can immediately to the increment starting from the `zero-until-0` state. However, we'll have to go back to the right from wherever we end up in the number to do the complement again in the other direction, which will conveniently place the head back at the left.

This machine is considerably longer than the previous ones, but I hope it is still comprehensible:

```
(define implicit-binary-sub1
  (implicit-tm
   [ones-complement
    [0 (ones-complement 1 R)]
    [1 (ones-complement 0 R)]
    [_ (add1:zero-until-0 _ L)]]
   [add1:zero-until-0
    [1 (add1:zero-until-0 0 L)]
    [0 (add1:find-end 1 R)]]
   [add1:find-end
    [0 (add1:find-end 0 R)]
    [1 (add1:find-end 1 R)]
    [_ (ones-complementR _ L)]]
   [ones-complementR
    [0 (ones-complementR 1 L)]
    [1 (ones-complementR 0 L)]
    [_ (HALT _ R)]]))
```

The trace of decrementing `3` to `2` takes 13 steps:

```
ones-complement  :  [0]011  { 0}
ones-complement  :  1[0]11  { 1}
ones-complement  :  11[1]1  { 2}
ones-complement  :  110[1]  { 3}
ones-complement  :  1100[_] { 4}
add1:zero-until-0:  110[0]_ { 5}
add1:find-end    :  1101[_] { 6}
ones-complementR :  110[1]_ { 7}
ones-complementR :  11[0]0_ { 8}
ones-complementR :  1[1]10_ { 9}
ones-complementR :  [1]010_ {10}
```

```
ones-complementR : [_]0010_ {11}
HALT             : _[0]010_ {12}
```

The thing I find most interesting about this machine is the way we logically "call another machine" to do the increment. Unfortunately, Turing machines lack any concept of a "call" or even something like a stack to record what to do after the increment "routine" is done. You could imagine simulating the stack with some additional information on the tape, but it's definitely not trivial. Instead we have to directly embed the increment code, including hard-wiring what it "returns to" when it is done, namely `ones-complementR`.

---

5 Binary Addition

At this point, we're ready to do real binary addition. For example, the input `0 0 1 0 + 0 0 1 1` (`2 + 3`) will halt at the tape `0 1 0 1` (`5`).

Here's the algorithm the Turing machine will implement, as Racket code:

```
(define (add x y)
  (if (zero? x)
    y
    (add (sub1 x) (add1 y))))
```

This trivial Racket program is pretty excruciating to write down as a Turing machine. First, we can't really call other functions, so we need to inline the code and hard-wire the "returns". (The same inlining idea applies to the `if`.) Second, there aren't really "arguments" in a Turing machine, there is just "the tape", so we have discover when one number ends and the other number starts by looking for the delimiter (or the blank symbol). Finally, we can't really just return one of the numbers, we have to erase the other number.

Since I've already shown how to do increment and decrement, the code should be fairly understandable, but skip ahead for an explanation if it is not clear.

```
(define implicit-binary-add
```

```
(implicit-tm
 [check-if-zero
  [0 (check-if-zero 0 R)]
  [1 (seek-left&sub1 1 L)]
  [+ (seek-left&zero _ L)]]
 [seek-left&sub1
  [0 (seek-left&sub1 0 L)]
  [1 (seek-left&sub1 1 L)]
  [_ (sub1:ones-complement _ R)]]
 [sub1:ones-complement
  [0 (sub1:ones-complement 1 R)]
  [1 (sub1:ones-complement 0 R)]
  [+ (sub1:add1:zero-until-0 + L)]]
 [sub1:add1:zero-until-0
  [1 (sub1:add1:zero-until-0 0 L)]
  [0 (sub1:add1:find-end 1 R)]]
 [sub1:add1:find-end
  [0 (sub1:add1:find-end 0 R)]
  [1 (sub1:add1:find-end 1 R)]
  [+ (sub1:ones-complementR + L)]]
 [sub1:ones-complementR
  [0 (sub1:ones-complementR 1 L)]
  [1 (sub1:ones-complementR 0 L)]
  [_ (seek-right&add1 _ R)]]
 [seek-right&add1
  [0 (seek-right&add1 0 R)]
  [1 (seek-right&add1 1 R)]
  [+ (add1:find-end + R)]]
 [add1:find-end
  [0 (add1:find-end 0 R)]
  [1 (add1:find-end 1 R)]
  [_ (add1:zero-until-0 _ L)]]
 [add1:zero-until-0
  [1 (add1:zero-until-0 0 L)]
  [0 (seek-left&continue 1 L)]]
 [seek-left&continue
  [0 (seek-left&continue 0 L)]
  [1 (seek-left&continue 1 L)]
  [+ (seek-left&continue + L)]
  [_ (check-if-zero _ R)]]
 [seek-left&zero
  [0 (seek-left&zero _ L)]
  [_ (seek-start _ R)]]
 [seek-start
  [_ (seek-start _ R)]
  [0 (move-right-once 0 L)]
```

```
     [1 (move-right-once 1 L)]]
    [move-right-once
     [_ (HALT _ R)]]))
```

The machine starts reading numbers and if it gets to a `1`, then it goes back to the left and then does the decrement operation (where `+` is the end of the number, and not `_`). After the decrement is finished, it seeks to the `+` and then starts the increment operation. Once the increment is over, it seeks the left-most `_`, which puts the head back in front of the equation where it can return to the initial state. If in the initial state, it reaches the `+` without reading a `1`, then the left number must have been `0`, so it goes back to the left and erases the number as it goes, then goes back to the right and puts the head before the right number.

It takes 98 steps for this machine to add `2 + 3`, but it works! It just has one constraint: the numbers must be initialized to the length on the output number. For instance `1 1 + 1 1` will fail, but `0 1 1 + 0 1 1` will succeed.

6 Binary Addition with Three Tapes

There is a variant of a Turing machine called a multi-tape Turing machine where there are many different tapes that can be read and updated simultaneously. If we had such a Turing machine, binary addition could be a lot simpler.

For instance, imagine having three tapes: one for the left operand, one for the right, and one for the answer. Then, store the bits in increasing significance (rather than traditional decreasing significance). Then, a purely right-moving Turing machine could add the numbers in one pass with just two states—one for when there is a carry bit and the other for when there is not.

Every multi-tape Turing machine has a one tape equivalent, according to the theory, but it may increase the number of steps quadratically to simulate the movement of each tape. Fortunately in this case, we can easily write down the one tape equivalent once we realize that the tape only goes in one direction.

We can represent all combinations of the two input tape's values

as four symbols: `(0 0)`, `(0 1)`, `(1 0)`, and `(1 1)`. (It may seem against the rules to do this, but it's not. We can use any finite set as the set Gamma. If you prefer to keep the symbols to a single digit, then you can replace them with `A`, `B`, `C`, and `D`, but it is really unnecessary.)

Since we know the machine doesn't need to read any tape more than once, as we move to the right we can just erase the two tapes with the value on the third tape.

When we put these ideas together, the machine is very simple:

```
(define implicit-binary-add-mt
  (implicit-tm
   [no-carry
    [(0 0) (no-carry 0 R)]
    [(1 0) (no-carry 1 R)]
    [(0 1) (no-carry 1 R)]
    [(1 1) (   carry 0 R)]
    [    _ (    HALT _ L)]]
   [carry
    [(0 0) (no-carry 1 R)]
    [(0 1) (   carry 0 R)]
    [(1 0) (   carry 0 R)]
    [(1 1) (   carry 1 R)]
    [    _ (    HALT 1 R)]]))
```

This machine takes six steps to add `2 + 3`:

```
no-carry: [(0 1)] (1 1) (0 0) (0 0)         {0}
no-carry:    1   [(1 1)](0 0) (0 0)         {1}
carry    :    1      0  [(0 0)](0 0)         {2}
no-carry:    1      0     1  [(0 0)]        {3}
no-carry:    1      0     1     0  [  _  ] {4}
HALT     :    1      0     1  [  0  ]  _    {5}
```

I like the contrast between these two machines a lot. The first binary adder shows the pain of low-level programming with Turing machines: you don't have any ability to have internal memory or function calls, but you can still do something hard. The second shows that if you free yourself from thinking about it as a real machine that is constrained to binary digits, you can think of a clever way to represent the three tapes in one tape and do the work very fast. I think it's a beautiful representation of how Turing machine are simultaneously an abstract mathematical

device *and* a fairly accurate model of real physical automatons.

7 Yo! It's almost time to go!

But first let's remember what we did today!

Turing machines are basic computers implemented in math.

Most problems on Turing machines are easy if you make them about text and not about mathematical values.

Turing machines don't really have internal memory or function calls, which stinks.

Multi-tape Turing machines feel much more powerful that single-tape machines, but they're not.