Computing with Pattern Substitution

Leon Rische

[2018-11-24 Sat 11:59]

Contents

In "The Art of Computer Programming", Volume 1, Section 1.1, Knuth introduces formalism a for describing algorithms based on pattern-matching and string substitutions.

There each algorithm is made up of $N \in \mathbb{N}$ rules $R_j = (\text{pattern}_j, \text{substitution}_j, \text{else}_j, \text{then}_j), 0 \leq jN$ and works on a state of the form (s, j).[1]

$\text{pattern}_j, \text{substitution}_j$ and s are strings over an alphabet A (elements of $A^\star$),
$\text{then}_j, \text{else}_j$ and j are numbers in $[0, N]$.

At each step of the computation, we check if s contains the pattern pattern_j. If so, its first occurrence is replaced with substitution_j and j of the state is set to then_j. Otherwise s remains unchanged and j is set to else_j.

When $j = N$ the computation terminates, s is the result.

1 First Example

Input: a^n (n times the character a)
Task: Determine if n is even.
Output: *even* if n is even, *odd* otherwise.

[1] In the book the parts are named differently

This can be accomplished using the following three rules with $N = 3$ (ε is the empty string):

- $R_0 = (aa, \varepsilon, 1, 0)$
- $R_1 = (a, odd, 2, 3)$
- $R_2 = (\varepsilon, even, 3, 3)$

Start by removing two as at a time until it is not possible anymore. If a single a remains output *odd*, otherwise output *even*.

1. $(aaaa, 0) \to (aa, 0) \to (\varepsilon, 0) \to (\varepsilon, 1) \to (\varepsilon, 2) \to (even, 3)$
2. $(aaaaa, 0) \to (aaa, 0) \to (a, 0) \to (a, 1) \to (odd, 3)$

2 Interpreter

Writing algorithms in this style is tedious because of the need to keep track of the rule indices, using labels instead would make it much easier to write and refactor programs.

I've come up with a simple format

```
label1
  pattern1 substitution1 label_else1 label_then1
label2
  pattern2 substitution2 label_else2 label_then2
...
```

`pattern` and `substitution` can be strings or `_` (the empty string ε) and `label_else/then` can be one of the other labels or `end`, a placeholder for N.

The rules from before could be rewritten as:

```
remove_aa
  aa _ check_remaining remove_aa
check_remaining
  a odd output_even end
output_even
  _ even end end
```

Below is a simple parser that converts this format to rules with indices as else_j and then_j (plus some bonus features like comments).[2]

language=Ruby,label= ,caption= ,captionpos=b,numbers=none Rule = Struct.new(:label, :pattern, :substitution, :$j_e lse, : j_t hen$)

def parse(program) Remove comments and empty lines pairs = program.lines .reject |l| l.match($/^*. * /).reject|l|l.match(/^*/)$ .map(:rstrip) .each$_s lice(2)$

Create a mapping from label to rule index labels = Hash.new $|_h ash, key|raise"Unknownlabelkey"la$

$-1 pairs.each_w ith_i ndex|(label,_b ody), i|labels[label] = i$

Convert the (label, body) pairs to rules pairs.map do |label, body| pattern, substitution, $l_e lse, l_t hen = body.lstrip.split('')substitution ='' if substitution ==''_{pattern=''ifpattern}$

Rule.new(label, pattern, substitution, labels[$l_e lse], labels[l_t hen])endend$

The code below is a direct translation of the formal description from earlier with optional logging of each step of the computation.

language=Ruby,label= ,caption= ,captionpos=b,numbers=none def run(state, rules, logging = false) j, steps = 0, 0

Get the length of the longest rule for nicer formatting $\max_l ength = rules.map|r|r.label.length.maxuntilj == -1dorule = rules[j]$

puts "rule.label.ljust($\max_l ength,' ')|state"iflogging$

Check if ‘state‘ contains the ‘pattern‘ if (index = state.index(rule.pattern)) state.sub!(rule.pattern, rule.substitution) j = rule.$j_t henelsej = rule.j_e lseendsteps+ = 1end$

puts "’end’.ljust($\max_l ength,' ')|state"iflogging puts"Steps : steps"iflogging state end$

Let’s make sure it works:

language=Ruby,label= ,caption= ,captionpos=b,numbers=none def encode$_i nput(n)'a' * nend$

run(encode$_i nput(5), parse(program), true)$

```
remove_aa       | aaaaa
remove_aa       | aaa
remove_aa       | a
check_remaining | a
end             | odd
Steps: 4
```

[2] -1 is used instead of N to signify the end of the computation, this way we don’t need to know how many rules there are

3 Greatest Common Divisor

One of the exercises in the book is to think of a set of rules that calculates $a^{gcd(m,n)}$ given the input $a^m b^n$ using a modified version of **Euclid's algorithm**.

1. Set $r \leftarrow abs(m-n), n \leftarrow min(m,n)$[3]
2. If $r = 0$, n is the result
3. Set $m \leftarrow n, n \leftarrow r$ and go to **step 1**

After a few hours I've come up with the following solution:

Duplicate the input

```
copy_as
  a ce copy_bs copy_as
copy_bs
  b df move_cs_left copy_bs
move_cs_left
  ec ce move_ds_left move_cs_left
move_ds_left
  fd df move_ds_left2 move_ds_left
move_ds_left2
  ed de calc_abs_diff move_ds_left2
```

This converts the state to $(ce)^m(df)^n$ and then moves the cs and ds around to yield $c^m d^n e^m f^n$.

Calculate $abs(m-n)$

```
calc_abs_diff
  cd _ test_r_zero_c calc_abs_diff
test_r_zero_c
  c c test_r_zero_d calc_min
test_r_zero_d
  d d return_r_remove_es calc_min
return_r_remove_es
  e _ return_r_convert_fa return_r_remove_es
return_r_convert_fa
  f a end return_r_convert_fa
```

[3] $abs(n)$ is the absolute value, e.g. $abs(-2) = 2$ and $abs(3) = 3$

Delete cds until either c^r or d^r with $r = abs(m - n)$ remains.

If both `test_r_zero_c` and `test_r_zero_d` fail, $r = abs(m - n)$ is zero and we need to return f^n as the result by removing all es and changing all fs to as.

After this step, the state is $c^r e^m f^n$ or $d^r e^m f^n$.

Calculate $min(m, n)$

If the result of the previous calculation has the form d^r n must be greater than m
(there were more ds than cs), remove the fs and convert the es to as. Otherwise, remove the es and convert the fs to as.

```
calc_min
  d d remove_es remove_fs
remove_es
  e _ convert_fa remove_es
convert_fa
  f a convert_cb convert_fa
remove_fs
  f _ convert_ea remove_fs
convert_ea
  e a convert_cb convert_ea
```

After this step, the state is $c^r a^{min(m,n)}$ or $d^r a^{min(m,n)}$.

Next iteration

First change the c^r or d^r to b^r, then move the as to the front.

```
convert_cb
  c b convert_db convert_cb
convert_db
  d b move_as_left convert_db
move_as_left
  ba ab copy_as move_as_left
```

After this step, the state is $a^{min(m,n)} b^r$ ($a^m b^n$ with $m \leftarrow n (= min(m, n))$ and $n \leftarrow r$) and we can jump back to the start (`copy_as`).

3.1 Output

A log of the 75 steps needed to compute $gcd(2, 4)$:

```
copy_as               | aabbbb
copy_as               | ceabbbb
copy_as               | cecebbbb
copy_bs               | cecebbbb
copy_bs               | cecedfdfdfdf
move_cs_left          | cecedfdfdfdf
move_cs_left          | cceedfdfdfdf
move_ds_left          | cceedfdfdfdf
move_ds_left          | cceeddffdfdf
move_ds_left          | cceeddfdffdf
move_ds_left          | cceedddfffdf
move_ds_left          | cceedddffdff
move_ds_left          | cceedddfdfff
move_ds_left          | cceeddddffff
move_ds_left2         | cceeddddffff
move_ds_left2         | ccededddffff
move_ds_left2         | ccdeedddffff
move_ds_left2         | ccdededdffff
move_ds_left2         | ccddeeddffff
move_ds_left2         | ccdddededffff
move_ds_left2         | ccdddeedffff
move_ds_left2         | ccdddedeffff
move_ds_left2         | ccddddeeffff
calc_abs_diff         | ccddddeeffff
calc_abs_diff         | cdddeeffff
calc_abs_diff         | ddeeffff
test_r_zero_c         | ddeeffff
test_r_zero_d         | ddeeffff
calc_min              | ddeeffff
remove_fs             | ddeeffff
remove_fs             | ddeefff
remove_fs             | ddeeff
remove_fs             | ddeef
remove_fs             | ddee
convert_ea            | ddee
convert_ea            | ddae
convert_ea            | ddaa
convert_cb            | ddaa
convert_db            | ddaa
convert_db            | bdaa
```

```
convert_db          | bbaa
move_as_left        | bbaa
move_as_left        | baba
move_as_left        | abba
move_as_left        | abab
move_as_left        | aabb
copy_as             | aabb
copy_as             | ceabb
copy_as             | cecebb
copy_bs             | cecebb
copy_bs             | cecedfb
copy_bs             | cecedfdf
move_cs_left        | cecedfdf
move_cs_left        | cceedfdf
move_ds_left        | cceedfdf
move_ds_left        | cceeddff
move_ds_left2       | cceeddff
move_ds_left2       | ccededff
move_ds_left2       | ccdeedff
move_ds_left2       | ccdedeff
move_ds_left2       | ccddeeff
calc_abs_diff       | ccddeeff
calc_abs_diff       | cdeeff
calc_abs_diff       | eeff
test_r_zero_c       | eeff
test_r_zero_d       | eeff
return_r_remove_es  | eeff
return_r_remove_es  | eff
return_r_remove_es  | ff
return_r_convert_fa | ff
return_r_convert_fa | af
return_r_convert_fa | aa
end                 | aa
```

As expected the result is 2 (or rather a^2).

4 A Better Solution

To my surprise, Knuth's solution to this problem has only five rules. What is he doing differently?

One key observation is that the step `calc_abs_diff` to calculate $abs(m-n)$ is executed exactly $min(m,n)$ times, with some careful coding both values can be calculated simultaneously.

Furthermore, $r = abs(m-n) = 0 \implies min(m,n) = m = n$, so there is no need to keep a copy of the original values around.

Implementation

Start by replacing abs with c and moving the c to the left.

```
start
  ab c convert_ab move_c_left
move_c_left
  ac ca start move_c_left
```

After this, the state is either $c^{min(m,n)}a^{abs(m-n)}$ or $c^{min(m,n)}b^{abs(m-n)}$.

```
convert_ab
  a b convert_ca convert_ab
convert_ca
  c a test_r_zero convert_ca
test_r_zero
  b b end start
```

Then rename the as to bs and the cs to as.[4]

Now the state is $a^{min(m,n)}b^{abs(m-n)}$.

If there are no bs in the state, r must be zero, and we can jump to `end` because $a^{min(m,n)}$ is the correct result n.

In this improved version $gcd(2,4)$ only needs 22 steps:

```
start       | aabbbb
move_c_left | acbbb
move_c_left | cabbb
start       | cabbb
move_c_left | ccbb
start       | ccbb
convert_ab  | ccbb
convert_ca  | ccbb
convert_ca  | acbb
convert_ca  | aabb
test_r_zero | aabb
```

[4]`convert_ab` is only needed in the case $m > n$

```
start       | aabb
move_c_left | acb
move_c_left | cab
start       | cab
move_c_left | cc
start       | cc
convert_ab  | cc
convert_ca  | cc
convert_ca  | ac
convert_ca  | aa
test_r_zero | aa
end         | aa
```

5 One Step Further: Primality Testing

Let's try to solve one more problem: given an input a^n, determine whether n is a prime number or not.

There are many (and easier) ways of doing this, but to show how algorithms in this formalism can be combined to solve more complicated problems, I'll use the *gcd* procedure from the previous section.

A number n is prime if each number m from 2 to $n-1$ is coprime to it ($gcd(m, n) = 1$). In our formalism the check $n > m$ is hard to do, instead we can count down from $m - 1$.

1. Count down a value m from $n - 1$ to 1
2. If $m = 1$ return *prime*
3. At each step, if $gcd(m, n) \neq 1$ return *notprime*, otherwise continue with **step 1**

Implementation

First, we need to handle the special case $n = 1$ and copy the as to create the counter m.

```
check1
  aa aa np_clear_c copy_input
copy_input
  a cd move_d_right copy_input
move_d_right
  dc cd subtract1 move_d_right
```

Then we decrement m (it starts at $n-1$) and output *prime* if the result is one.

```
subtract1
  dd d check_done check_done
check_done
  dd dd p_clear_c copy_c
```

`p_clear_c` and its counterpart `np_clear_c` clear the state and output either *prime* or *notprime*.

```
p_clear_c
  c _ p_clear_d p_clear_c
p_clear_d
  d _ p_clear_a p_clear_d
p_clear_a
  a _ p_output p_clear_a
p_output
  _ prime end end

np_clear_c
  c _ np_clear_d np_clear_c
np_clear_d
  d _ np_clear_a np_clear_d
np_clear_a
  a _ np_output np_clear_a
np_output
  _ notprime end end
```

Because the *gcd* procedure destroys the state, we need to make a copy of m and n.

This is done by converting the state $c^m d^n$ to $(Ca)^m(bD)^n$, moving the as and bs to the center and then renaming C, D back to c, d.[5]

```
copy_c
  c Ca copy_d copy_c
copy_d
  d bD move_a_right copy_d
move_a_right
```

[5]Renaming them in the first place is necessary to avoid an endless loop in the rule

```
  aC Ca move_b_left move_a_right
move_b_left
  Db bD convert_Cc move_b_left
convert_Cc
  C c convert_Dd convert_Cc
convert_Dd
  D d start_gcd convert_Dd
```

Now the state has the form $c^m a^m b^n d^n$ and we can call the *gcd* procedure from the previous section.[6]

```
start_gcd
  ab e convert_ab move_e_left
move_e_left
  ae ea start_gcd move_e_left
convert_ab
  a b convert_ea convert_ab
convert_ea
  e a test_r_zero convert_ea
test_r_zero
  b b test_coprime start_gcd
```

If we can't find the pattern aa afterwards, the numbers are coprime, and we can continue with $n \leftarrow n - 1$, otherwise m is not a prime.

```
test_coprime
  aa aa next np_clear_c
next
  a _ subtract1 subtract1
```

Even for small numbers this needs a lot of steps and I've removed some boring sections from the output:

```
check1       | aaaa
copy_input   | aaaa
copy_input   | cdaaa
copy_input   | cdcdaa
copy_input   | cdcdcda
copy_input   | cdcdcdcd
```

[6]Altered slightly to use e instead of c

```
move_d_right | cdcdcdcd
...
move_d_right | ccccdddd
subtract1    | ccccdddd
check_done   | ccccddd
copy_c       | ccccddd
...
convert_Dd   | ccccaaaabbbddd
start_gcd    | ccccaaaabbbddd
move_e_left  | ccccaaaebbddd
move_e_left  | ccccaaeabbddd
move_e_left  | ccccaeaabbddd
move_e_left  | cccceaaabbddd
start_gcd    | cccceaaabbddd
move_e_left  | cccceaaebddd
move_e_left  | cccceaeabddd
move_e_left  | ccccceeaabddd
start_gcd    | ccccceeaabddd
move_e_left  | cccceeaeddd
move_e_left  | cccceeeaddd
start_gcd    | cccceeeaddd
convert_ab   | cccceeeaddd
convert_ab   | cccceeebddd
convert_ea   | cccceeebddd
convert_ea   | ccccaeebddd
convert_ea   | ccccaaebddd
convert_ea   | ccccaaabddd
test_r_zero  | ccccaaabddd
start_gcd    | ccccaaabddd
move_e_left  | ccccaaeddd
move_e_left  | ccccaeaddd
move_e_left  | cccceaaddd
start_gcd    | cccceaaddd
convert_ab   | cccceaaddd
convert_ab   | ccccebaddd
convert_ab   | ccccebbddd
convert_ea   | ccccebbddd
convert_ea   | ccccabbddd
test_r_zero  | ccccabbddd
start_gcd    | ccccabbddd
```

```
move_e_left  | ccccebddd
start_gcd    | ccccebddd
convert_ab   | ccccebddd
convert_ea   | ccccebddd
convert_ea   | ccccabddd
test_r_zero  | ccccabddd
start_gcd    | ccccabddd
move_e_left  | cccceddd
start_gcd    | cccceddd
convert_ab   | cccceddd
convert_ea   | cccceddd
convert_ea   | ccccaddd
test_r_zero  | ccccaddd
test_coprime | ccccaddd
next         | ccccaddd
subtract1    | ccccddd
check_done   | ccccdd
copy_c       | ccccdd
...
convert_Dd   | ccccaaaabbdd
start_gcd    | ccccaaaabbdd
move_e_left  | ccccaaaebdd
move_e_left  | ccccaaeabdd
move_e_left  | ccccaeaabdd
move_e_left  | cccceaaabdd
start_gcd    | cccceaaabdd
move_e_left  | cccceaaedd
move_e_left  | cccceaeadd
move_e_left  | cccceeaadd
start_gcd    | cccceeaadd
convert_ab   | cccceeaadd
convert_ab   | cccceebadd
convert_ab   | cccceebbdd
convert_ea   | cccceebbdd
convert_ea   | ccccaebbdd
convert_ea   | ccccaabbdd
test_r_zero  | ccccaabbdd
start_gcd    | ccccaabbdd
move_e_left  | ccccaebdd
move_e_left  | cccceabdd
```

```
start_gcd    | cccceabdd
move_e_left  | cccceedd
start_gcd    | cccceedd
convert_ab   | cccceedd
convert_ea   | cccceedd
convert_ea   | ccccaedd
convert_ea   | ccccaadd
test_r_zero  | ccccaadd
test_coprime | ccccaadd
np_clear_c   | ccccaadd
...
np_output    |
end          | notprime
```

For larger numbers and primes the results are correct, too, but I won't include the logs here.