UNIVERSITY AT ALBANY
State University of New York

College of Engineering and Applied Sciences
Computer Science Department

Recursive Path Ordering on Strings: Two Implementations

(Master's Project)

By

Gayathri Matha Mandadi

(gmandadi@albany.edu)

Under the Guidance of
Professor

Dr. Paliath Narendran

(pnarendran@albany.edu)

Abstract

We have implemented an algorithm to determine whether a given pair of strings can be made RPO-comparable by selecting a partial order on their function symbols. The algorithm is implemented in `Python3`.

Acknowledgements

I would like to express my deepest gratitude to my advisor and mentor Prof. Paliath Narendran for helping me by giving constant support and allowing me to take up this project under his guidance. I would also like to thank Josue Ruiz whose research work helped me immensely in understanding all the concepts. I would to express my sincere gratitude for Kavya Kakkera who helped me in understanding her past work which is useful in my project. Last, but not the least, I would like to thank my family for their deep consideration for the finances and undying support by providing words of encouragement throughout the semester.

1 Introduction and Basic Theory

Termination orderings play a very important role in automated reasoning, in showing that a sequence of rewrite steps will eventually terminate. One of the most influential is the *recursive path ordering* introduced by Nachum Dershowitz. The key idea is to extend an ordering $\succ$ on the symbols to terms. Restricted to strings, the ordering is defined as follows:

Let Σ be an alphabet and $\succ$ be an ordering on Σ. Then $x >_{rpo} y$ if and only if one of the following conditions hold:

(1) $y = \varepsilon$ and $|x| > 0$.

(2) $x = au$, $y = av$, and $u >_{rpo} v$.

(3) $x = au$, $y = bv$, and either

(3a) $u \geq_{rpo} y$, or

(3b) $a \succ b$ and $x >_{rpo} v$.

One well-known property of the *rpo* is

Lemma 1. $\forall x \forall y \forall z : (xz >_{rpo} yz)$ *if and only if* $(x >_{rpo} y)$.

Thus before comparing any two strings their common suffixes can be removed.

If the symbol ordering $\succ$ happens to be total (i.e., the ordering is linear), an alternative characterization was introduced in [3]. Let $max(w, \Sigma)$ stand for the *maximal* (highest) symbol of Σ that occurs in w and let $mul(w, \Sigma)$ be the number of times *this* symbol occurs in w, i.e., $\#_{max(w,\Sigma)}(w)$. Now $w >_{rpo} w'$ iff one of the following holds:

1. $max(w, \Sigma) \succ max(w', \Sigma)$
2. $max(w, \Sigma) = max(w', \Sigma)$ and $mul(w, \Sigma) > mul(w', \Sigma)$
3. $a = max(w, \Sigma) = max(w', \Sigma)$, $mul(w, \Sigma) = mul(w', \Sigma)$,

$$w = w_0 w_1 w_2 \ldots w_k$$

$$w' = u_0 u_1 u_2 \ldots u_k$$

and there exists $0 \leq i \leq k$ such that $w_i >_{rpo} u_i$ and for all $j > i$ we have $w_j = u_j$.

If the strings do not have any common suffixes, then the last condition can be changed to

3'. $a = max(w, \Sigma) = max(w', \Sigma)$, $mul(w, \Sigma) = mul(w', \Sigma)$,

$$w = w_0 w_1 w_2 \ldots w_k$$

$$w' = u_0 u_1 u_2 \ldots u_k$$

and $w_k >_{rpo} u_k$.

2 Implementation

The program for Basic Theory is implemented as follows:

The program takes the input as the pair of strings and symbol ordering.

1. The program calls `rpo()` function with the input of string1, string2 and precedence ordering where it checks if pair of two strings are equal or not. If pair of two strings are equal then its return false else it calls `rpoeq()` function recursively.
2. `rpoeq()` is function which is having the input of string1, string2 and precedence ordering and is checks for the below conditions:
   a. Firstly, It removes the common suffix by using function `remove_common_suffix()`.
   b. Secondly, checks whether the pair two strings are equal, then its return True.
   c. If Not, it will checks for the `is_subseq()` function of two strings.
   if string1 is sub-sequence of string 2 then return False
   if string2 is sub-sequence of string 1 then return True
   d. Then If the first symbol of the pair strings are equal then, then It compares the remaining part of the strings by calling the `rpoeq()` function
   else `higher_prec()` function compare the first symbols based on precedence ordering.
   e. `higher_prec()` function returns True then it will call the `rpoeq()` function with the input of string1, remaining part of the string2 and precedence ordering.
   `higher_prec()` function returns False then It will call the `rpoeq()` function with the input of remaining part of the string1, string2 and precedence ordering.

Lemma1 Implementation:

1. The program calls `rpo()` function with the input of string1, string2 and precedence ordering.

2. `rpo()` checks for below conditions:

    a. Condition1: If pair two strings are equal, then its return false.

    b. Condition2: If string1 is empty and string2 has one or more symbols, then it will return the false. If it is vice versa, then returns true.

    c. Condition3: If the pair of strings are not equal then

        i. Firstly, It removes the common suffix by using function `remove_common_suffix()`.

        ii. Secondly, If the pair of strings have different max symbols, then max symbol of string1 is higher than max symbol of string2.

        iii. Next, If the pair of strings are having same max symbols, then `counter_function()` calculates the length of each maximal symbol for both the inputs and next, It will compare the both strings based on the precedence ordering.

        iv. After comparing both the strings, if strings is having the same number of max symbols, then call split () function at maximal symbols and compare the strings again.
        After calling the split () function, Compare last piece the string that are present in list by calling recursively.

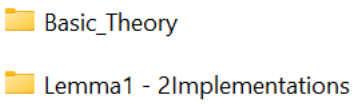
3 Requirements and Usage

1. Python 3.x.: You have to install Python 3 and related packages from https://www.python.org/downloads/.

2. standard build environment (make, gcc, etc.). To test the API, in command prompt use $make test

3. This algorithm is implemented in Python 3. Check for the python versions in your system using commands “python –version” or “python3 –version”

Folder structure:



Run the code:

At the command prompt, use command

python3 Lemma1.py
python3 Basic_Theory.py

4 Conclusion

Implemented the two algorithms to determine whether a given pair of strings can be made RPO-comparable by selecting a partial order on their function symbols and compared running time. The Running time of both algorithms is O(n)

References

[1] F, Baader and T. Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[2] N. Dershowitz. Orderings for term-rewriting systems. *Theoretical Computer Science* 17(3): 279–301. 1982.

[3] P. Narendran and M. Rusinowitch. The theory of total unary RPO is decidable. *Computational Logic – CL 2000, First International Conference, London, UK, 24-28 July, 2000, Proceedings*, volume 1861 of *Lecture Notes in Computer Science*, pages 660–672. Springer, 2000.

5 Appendix

Python Code

5.1 Basic_Theory-Implementation.py

```
1
2  import time
3
4  #from prettyprint import *
5
6  import re
7
8  start = time.time()
9
10 def find_common_prefix(s1, s2):
11
12     prefix = ''
13     for i in range(min(len(s1), len(s2))):
14         if s1[i] == s2[i]:
15             prefix += s1[i]
16         else:
17             break
18     return prefix
19
20 """
21 >>> find_common_prefix('asd','aqwq')
22 a
23 >>> find_common_prefix('aabaabaa','aabacaa')
24 aaba
25 >>> find_common_prefix('ab','a')
26 a
27 >>> find_common_prefix('abcd','abef')
28 ab
29
30 """
31
32 def find_common_suffix(s1, s2):
33
34     suffix = ''
35     for i in range(1, min(len(s1), len(s2))+1):
36         if s1[-i] == s2[-i]:
37             suffix = s1[-i] + suffix
38         else:
39             break
40     return suffix
41
42 """
43 >>> find_common_suffix('ababab', 'ccabcab')
44 ab
45 >>> find_common_suffix('ababab', 'ccabcb')
46 b
47 >>> find_common_suffix('ababab', 'ab')
48 ab
```

```
"""

def remove_common_prefix(s1, s2):

    prefix = find_common_prefix(s1, s2)
    return (s1[len(prefix):], s2[len(prefix):])

"""
>>> remove_common_prefix('asd','aqwq')
('sd','qwq')
>>> remove_common_prefix('aabaabaa','aabacaa')
('abaa','caa')
>>> remove_common_prefix('ab','a')
('b','')
>>> remove_common_prefix('abcd','abef')
('cd','ef')

"""

def remove_common_suffix(s1, s2):

    suffix = find_common_suffix(s1, s2)
    if (suffix == ''):
        return (s1, s2)
    else:
        return (s1[:-len(suffix)], s2[:-len(suffix)])

"""
>>> remove_common_suffix('ababab', 'ccabcab')
('abab', 'ccabc')
>>> remove_common_suffix('ababab', 'ccabcb')
('ababa', 'ccabc')
>>> remove_common_suffix('ababab', 'ab')
('abab', '')
"""

"""
print("Process executed in {0} s".format(round((end-start),1)))
"""

rcs = remove_common_suffix

end = time.time()

def is_subseq(x, y):  # Stephan Pochmann on the internet
    it = iter(y)
    return all(c in it for c in x)

"""
>>> is_subseq('ac', 'abc')
True
>>> is_subseq('ac', 'abbc')
True
>>> is_subseq('ac', 'bbbbabbc')
True
>>> is_subseq('ac', 'bbbbcab')
False
```

```
>>> is_subseq('', 'ac')
"""

def tail(ls):
    if (len(ls) == 0):
        return None
    else:
        return ls[1:]

"""
>>> tail(['a','d','as'])
['d', 'as']
>>> tail(['aa','ad','aass'])
['ad', 'aass']

"""

def higher_prec(sym1, sym2, prec_list):
    sym = hd(prec_list)
    if (sym1 == sym):
        return True
    elif (sym2 == sym):
        return False
    else:
        return higher_prec(sym1, sym2, tail(prec_list))

"""
>>> higher_prec('b', 'd', pls)
True
>>> higher_prec('b', 'a', pls)
False
>>> higher_prec('d', 'a', pls)
False
"""

def decompose(str, symbol):
    w = re.split(symbol, str)
    return w

"""
>>> decompose('babbaba', 'a')
['b', 'bb', 'b', '']
>>> decompose('babbabccac', 'a')
['b', 'bb', 'bcc', 'c']
>>> decompose('babbaba', 'b')
['', 'a', '', 'a', 'a']
"""
def explode(str):
    return list(str)

'''
>>> explode('ads')
['a', 'd', 's']
>>> explode('qwq')
['q', 'w', 'q']

'''
```

```

def hd(L):
    if type(L) == type([]):
        if len(L) == 0: return None
        else: return L[0]
    else: return None
"""
>>> hd(['ad','a'])
ad
>>> hd(['','a'])

>>> hd(['a','aasa'])
a
"""

def rpoeq(str1, str2, preclist):
    (str1, str2) = remove_common_suffix(str1, str2)
    if (str1 == str2):
        return True
    else:
        if is_subseq(str1, str2):
            return False
        else:
            if is_subseq(str2, str1):
                return True
            else:
                xa = str1[0]
                xb = str2[0]
                if (xa == xb):
                    return rpoeq(str1[1:], str2[1:], preclist)
                else:
                    if higher_prec(xa, xb, preclist):
                        return rpoeq(str1, str2[1:], preclist)
                    else:
                        return rpoeq(str1[1:], str2, preclist)

"""
>>> rpoeq('abc', 'abb', ['a', 'b', 'c'])
False
>>> rpoeq('abc', 'bbbbbac', ['a', 'b', 'c'])
True
>>> rpoeq('abc', 'bbbbbacccc', ['a', 'b', 'c'])
True
"""

def rpo(str1, str2, preclist):
    if (str1 == str2):
        return False
    else:
        return rpoeq(str1, str2, preclist)

"""
>>> rpo('a', 'bbbbb', ['a', 'b', 'c'])
True
>>> rpo('a', 'a', ['a', 'b', 'c'])
False
>>> rpo('ab', 'bbbbbbbac', ['a', 'b', 'c'])
```

```
True
>>> rpo('abba', 'baab', ['a', 'b'])
True
"""
```

5.2 Lemma1_2-Implementations.py

```

import time
# from prettyprint import *
import re
start = time.time()

# Implmentation 1
def pos(x, L):
    if (len(L) == 0):
        return None
    else:
        if (x == L[0]):
            return 0
        else:
            return (1 + (pos(x, L[1:])))

"""
>>> pos('c', ['a','b','c'])
2
>>> pos('b', ['a','b','d'])
1
>>> pos('a', ['a','b'])
0
"""

def rporec(str1, str2, m, n, i, j, flag, symlist):
    if ((i > m) and (j > n)):
        return flag
    else:
        if (j > n):
            return True
        elif (i > m):
            return False
    x = pos(str1[i], symlist)
    y = pos(str2[j], symlist)
    #print(i, " ", j, " ", x, " ", y);
    if (x < y):
        return rporec(str1, str2, m, n, i, j + 1, flag, symlist)
    elif (x == y):
        return rporec(str1, str2, m, n, i + 1, j + 1, flag, symlist
    )
    else:
        if (flag):
            return rporec(str1, str2, m, n, i + 1, j, flag, symlist
    )
        else:
            return rporec(str1, str2, m, n, i + 1, j, True, symlist
    )
```

```
46
47 '''print(rporec('ac', 'abbc', 0, 0, 0, 0, False, ['a', 'b','c']))
48 False
49 print(rporec('abc', 'abbc', 0, 0, 2, 3, False, ['a', 'b','c']))
50 False
51 print(rporec('ab', 'bbbbbbbac', 3, 2, 0, 0, True, ['a', 'b','c']))
52 True'''
53
54 def rpo_original(str1, str2, symlist):
55     if (str1 == ''):
56         return False
57     elif (str2 == ''):
58         return True
59     m = len(str1) - 1
60     n = len(str2) - 1
61     return rporec(str1, str2, m, n, 0, 0, False, symlist)
62
63 rpoo = rpo_original
64
65 '''
66
67 >>> rpoo("abba","baab",['a', 'b'])
68 False
69 >>> rpoo("abab","baab",['a', 'b'])
70 False
71 >>> rpoo('ab', 'bbbbbbbac', ['a', 'b', 'c'])
72 True
73 >>> rpoo('a', 'a', ['a', 'b', 'c'])
74 False
75 >>> rpoo('a', 'bbbbb', ['a', 'b', 'c'])
76 True
77
78 '''
79
80 end = time.time()
81
82
83 print("Implmentation - Process executed in {0} ms".format((end-
       start)*10**3))
84
85
86 #Implementation 2
87 start = time.time()
88 def find_common_suffix(s1, s2):
89
90     suffix = ''
91     for i in range(1, min(len(s1), len(s2))+1):
92         if s1[-i] == s2[-i]:
93             suffix = s1[-i] + suffix
94         else:
95             break
96     return suffix
97
98
99 """
100 >>> find_common_suffix('ababab', 'ccabcab')
101 ab
```

```
>>> find_common_suffix('ababab', 'ccabcb')
b
>>> find_common_suffix('ababab', 'ab')
ab
"""

def remove_common_suffix(s1, s2):

    suffix = find_common_suffix(s1, s2)
    if (suffix == ''):
        return (s1, s2)
    else:
        return (s1[:-len(suffix)], s2[:-len(suffix)])

"""
>>> remove_common_suffix('ababab', 'ccabcab')
('abab', 'ccabc')
>>> remove_common_suffix('ababab', 'ccabcb')
('ababa', 'ccabc')
>>> remove_common_suffix('ababab', 'ab')
('abab', '')
"""


def ParikhVector(str1, chars):
    l = []
    for x in chars:
        l.append(x)
    for i,x in enumerate(l):
        count = len([y for y in str1 if x ==y ])
        l[i] = count
    return l

"""
>>> ParikhVector('asds',['a','s','d'])
[1, 2, 1]
>>> ParikhVector('ads',['a','s','d'])
[1, 1, 1]
>>> ParikhVector('sds',['a','s','d'])
[0, 2, 1]
"""

def rpo(s1, s2, p_list):

    if (s1 == s2):
        return False
    elif len(s1)==0 and len(s2)>0:
        return False
    elif len(s1)>0 and len(s2)==0:
        return True
    else:
      s1,s2 = remove_common_suffix(s1,s2)
    c1 = ParikhVector(s1,p_list)
    c2 = ParikhVector(s2,p_list)
    for i,x in enumerate(c1):
        if x>0 and c2[i]==0:
            return True
```

```
        elif x==0 and c2[i]>0:
            return False
        elif x==c2[i]==0:
            continue
        elif x>c2[i] and c2[i]>0:
            return True
        elif x==c2[i] and c2[i]>0:
                splitted_str1 = s1.split(p_list[i])
                splitted_str2 = s2.split(p_list[i])
                return rpo(splitted_str1[-1], splitted_str2[-1],
    p_list)
        elif x<c2[i] and x>0:
            return False
        else:
            return False

"""
>>> rpo("a","bbbbb",['a', 'b', 'c'])
True
>>> rpo("a","a",['a', 'b', 'c'])
False
>>> rpo("ab","bbbbbbbac",['a', 'b', 'c'])
True
>>> rpo("abba","baab",['a', 'b'])
False
"""

end = time.time()


print("Implementation2 - Process executed in {0} ms".format((end-
    start)*10**3))
```

6 Examples:

6.1 Basic_Theory-Implementation.py

Input & Output :

```
>>> rpo('a', 'bbbbb', ['a', 'b', 'c'])
True
>>> rpo('a', 'a', ['a', 'b', 'c'])
False
>>> rpo('ab', 'bbbbbbbac', ['a', 'b', 'c'])
True
>>> rpo('abba', 'baab', ['a', 'b'])
False
```

6.2 Lemma1_2-Implementations.py

Input & Output :

```
>>> rpo ("a","bbbbb”, ['a', 'b', 'c'])
True
>>> rpo ("a","a”, ['a', 'b', 'c'])
False
>>> rpo ("ab","bbbbbbbac”, ['a', 'b', 'c'])
True
>>> rpo ("abba","baab",['a', 'b'])
False
```