

Greedy regular expression matching

Alain Frisch Luca Cardelli

INRIA

MSRC

2004-05-15

ICALP

The matching problem

The problem

Project the **structure** of a regular expression on a flat sequence.

- $R = (a * | b) *$
 - $w = a_1 a_2 b_1 b_2 a_3$
 - $\Rightarrow v = [1 : [a_1; a_2]; 2 : b_1; 2 : b_2; 1 : [a_3]]$
-
- The result retains the **structure** of the regexp and the **content** of the sequence.
 - Result driven by the syntax of regexps \neq automata.
 - Issues: efficiency, disambiguation.

- Type-directed native representation of values in XDuce-like languages: E.g.:
[int] \rightsquigarrow int
[int int*] \rightsquigarrow struct {int fst; int[] snd;}
- Advantages over uniform representation:
 - More compact representation, less boxing
 - Fast random access
 - Easier to interface/integrate with other language
- Requires coercion between subtypes.
 - Flatten sequences.
 - Project the structure of the new regexp = **matching**.

- Regex packages with structured matching semantics.
- Lexer-parser generators.
- Operation/representation defined by induction on the structure of regexps (e.g.: Hosoya's filters).

- A regexp iterator extension for C#

```
object[] a = new object[] {1,2,3,4,"abc",4,5,"xyz",6,7,false};
applyregexp(a) (
    ( int , int          )*,
    string
)
|
(
    ( int          )*,
    bool
)
)*;
```

- A regexp iterator extension for C#

```
object[] a = new object[] {1,2,3,4,"abc",4,5,"xyz",6,7,false};  
applyregexp(a) (  
    ( { int sum = 0; },  
      ( int x, int y, { sum += x*y; } )*,  
      string s,  
      { System.Console.WriteLine(s + ":" + sum); }  
    )  
    |  
    ( { int sum = 0; },  
      ( int x, { sum += x; } )*,  
      bool b,  
      { System.Console.WriteLine(b + ":" + sum); }  
    )  
  )*;  
~>
```

```
abc:14  
xyz:20  
False:13
```

Key issue: avoiding backtracking

- Consider the regexp:

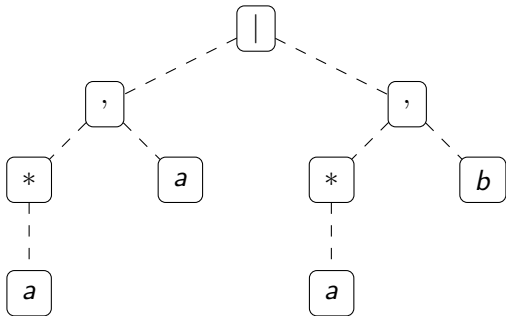
$$R = a^* |(a^*, b, a^*)$$

- To avoid backtracking, and still proceed by induction on the regexp, we need to decide first which branch to take (left or right?)
- Unbounded look-ahead!

An example

- Abstract syntax tree of the regexp.

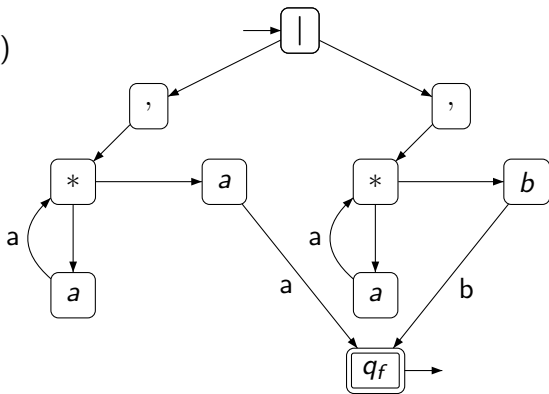
$R = (a^*, a)|(a^*, b)$
 $w = a b$



An example

- Abstract syntax tree of the regexp.
- Build an automaton on top of it.

$R = (a^*, a)|(a^*, b)$
 $w = a b$

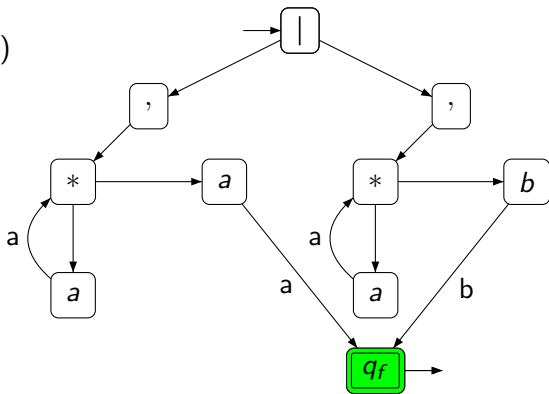


An example

- Abstract syntax tree of the regexp.
- Build an automaton on top of it.
- Scan the input backwards ("subset construction").

$R = (a^*, a)|(a^*, b)$

$w = a b \bullet$

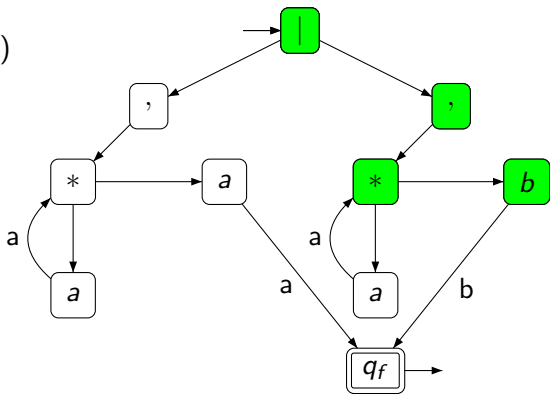


An example

- Abstract syntax tree of the regexp.
- Build an automaton on top of it.
- Scan the input backwards ("subset construction").

$R = (a^*, a)|(a^*, b)$

$w = a \bullet b$

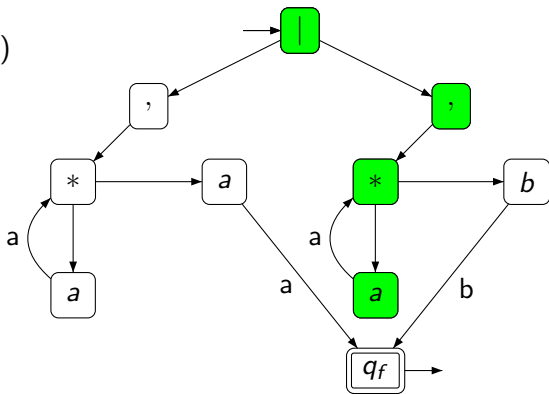


An example

- Abstract syntax tree of the regexp.
- Build an automaton on top of it.
- Scan the input backwards ("subset construction").

$R = (a^*, a)|(a^*, b)$

$w = \bullet a b$



Second pass: the matcher

```
let rec loop = function
  | ε -> ()
  | r1 , r2 -> (loop r1 , loop r2)
  | r1 | r2 -> if ... then (1,loop r1) else (2,loop r2)
  | r* -> if ... then (loop r)::(loop r*) else []
  | c -> (* Consume the token *)
```

- What are the ... ?
- Given by the first pass.
- Disambiguation:
 - first-match for |
 - greedy semantics for *

Non-termination problem

The algorithm always terminates except with a subregex R^* where R is “nullable”.

Examples: $(a^*, b^*)^*$ $(a * | b^*)^*$

Same problem in the folklore syntax-directed recognizer:

```
let rec loop r k w = match r with
  | ε -> k w
  | r1 , r2 -> loop r1 (loop r2 k) w
  | r1 | r2 -> (loop r1 k w) || (loop r2 k w)
  | r* -> (loop r (loop r* k) w) || (k w)
  | c -> (w <> []) && (hd w = c) && (k (tl w))
```

```
let accept r = loop r ( (=) [] )
```

r : regexp

k : continuation

w : input sequence

$\text{loop } r \text{ k w} = \text{true} \iff w = w_1 @ w_2 \text{ s.t. } (r \text{ matches } w_1) \ \&\& \ (k \ w_2 = \text{true})$

Non-termination problem: solutions

Rewrite regexps to avoid the problematic situation

- E.g.: $(a^*, b^*)^* \rightsquigarrow ((a^*, b^+)|a^+)^*$
- The structure of the regexp is lost: not suitable for the matching problem.
- Interaction with the disambiguation policy ?

Prevent iterations in stars from accepting empty sequences

- In the functional recognizer, replace

```
(loop r (loop r* k) w) || (k w)
```

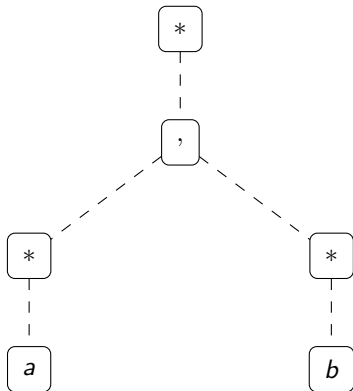
with:

```
(loop r (fun w' -> (w <> w') && (loop r* k w') w) || (k w)
```

- How to adapt our algorithm ?

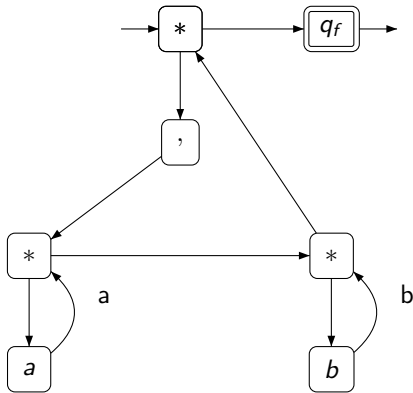
An example

• $R = (a^*, b^*)^*$



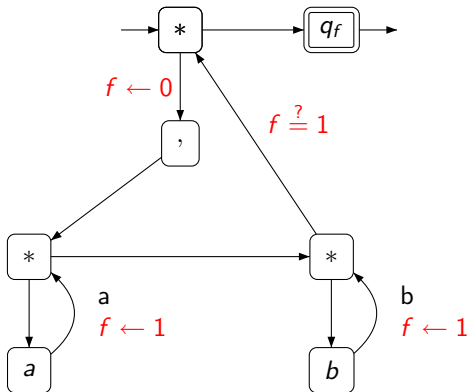
An example

- $R = (a^*, b^*)^*$
- Loop of ε -transitions



An example

- $R = (a^*, b^*)^*$
- Loop of ε -transitions ... **now broken**.
- Still a finite state automaton (states (q, f)).



Second pass: the matcher

```
let rec loop = function
  | ε -> ()
  | r1 , r2 -> (loop r1, loop r2)
  | r1 | r2 -> if ... then (1,loop r1) else (2,loop r2)
  | r* -> if ... then ( f := 0; (loop r)::(loop r*)) else []
  | c -> f := 1; (* Consume the token *)
```

The ... are given by the first pass.

- Keep a tight connection between regexps and automata.
 - Direct construction of the automaton
- Accept problematic regexps, reject problematic matchings.
- Result: linear time (two-passes) matching algorithm, which can be efficiently implemented.
- Abstract specification of the disambiguation policy as an optimization problem (not presented).
- Characterization and study of problematic cases (not presented).

- Evaluate the alternative implementation technique for XML languages.
- Optimizations: the first pass is not always necessary. Use (bounded) look-ahead as long as possible, or a lazy first pass.
- Non-local disambiguation policy, e.g.: longest match semantics.
- Non-regular languages.

Questions ?