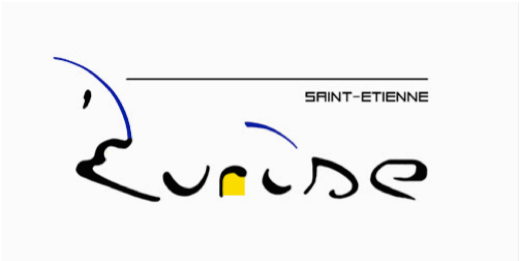
Identification in the limit of substitutable context-free languages

Alexander CLARK
Rémi EYRAUD



Motivation

- Learning languages has a huge array of applications (from natural languages to biology).
- There exist powerful methods to learn regular languages [Oncina et al., 94].
- But learning more complex (and useful) languages, like Context Free Languages, is hard.

State of the art

- Identify in the limit context-free languages is not possible from positive examples [Gold, 64].
- Very simple languages are polynomially identifiable in the limit [Yokomori, 02].
- The learning problem of context-free languages is actively attacked [OMPHALOS competition, 04].

Main result

We introduce a new subclass of context-free languages, based on purely syntactic properties, and give an algorithm that polynomially identifies in the limit this class, from positive examples only.

Key idea

- To deal with the context-free learning problem, we choose to use a purely syntactic approach.
- Substitutability: an old idea [Harris 54]

But:

- No explicit use;
- No adequate formalization.

Outline

- The mathematical formalization of the substitutability
- The class of substitutable languages
- Learning result (algorithm)
- Discussion

Terminology

- *Word:* any finite concatenation of letters of an alphabet Σ.
- *Language:* any set of words (= any subset of Σ*).
- *Grammar:* a grammar is quadruple G=<Σ,V,P,S> where Σ is an alphabet, V a set of non terminals distinct of Σ, P a set of production rules, and S the axiom.
- *Context:* given a substring *u* of a word *w*, a context of *u* is the pair (*l*,*r*) in Σ* such that *w*=*lur*. The set of all contexts of u in a language L is written *C(u).*

  For instance, the substring ab appears in the context (c,b) in the word cabb.

Syntactic congruence

- *The syntactic congruence*:

  u and *v* are syntactically congruent w.r.t. a language L iff for all *l,r* in Σ*, *lur* is in L iff *lvr* is in L ($u \equiv_L v$).

- In term of context, $u \equiv_L v$ iff C(u) = C(v).

- Example: L={bcb,baab,cb,aab},
  $c \equiv_L aa$

Weak substitutability

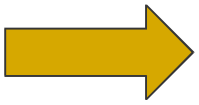
- *The weak substitutability*:

  u and *v* are weakly substitutable w.r.t. a language L, iff there exist *l,r* in Σ*, *lur* is in L iff *lvr* is in L ($u \doteq v$).

- In term of context, $u \doteq v$ iff C(u)∩C(v)≠ø

Substitutability

- The syntactic congruence is the most interesting: *u* and *v* always appear in the same context (then they can be generated by the same non terminal).
- But: from examples, we can only observe the weak substitutability.



We need to unify these notions in order to ensure the syntactic congruence from the observations.

Substitutable languages

- A language L is substitutable iff for all u and v in Σ^*, $u \doteq v$ implies $u \equiv_L v$, i.e. the weak sustitutability implies the syntactic congruence.

- The sets of contexts of two substrings of words of L are either disjoint or identical.

Examples

- Σ^* is substitutable.
- $\{a^n \mid n>0\}$ is substitutable (all contexts of a substring have the form (a^k,a^l)).
- $\{wcw^R \mid w \text{ in } (a,b)^*\}$ is substitutable.
- $\{a^n cb^n \mid n>0\}$ is substitutable

- $\{a^n b^n \mid n>0\}$ is not substitutable.
- $\{a,aa\}$ is not substitutable (a and aa share the context (ϵ,ϵ) but not the context (ϵ,a))

Algorithm: main ideas

- From examples of the language, the algorithm returns a consistent context-free grammar.
- *Substitution graph*: each distinct substring of the learning sample is a node. There is an edge between two nodes if they appear in the same context(s).
- A non terminal N is created for each component C of the graph.

Running Example

LS={c;aca;bcb;abcba;aacaa} (palindrome with a center marked)

c bcba abcb

abcba aca acaa ac

bcb ca aaca

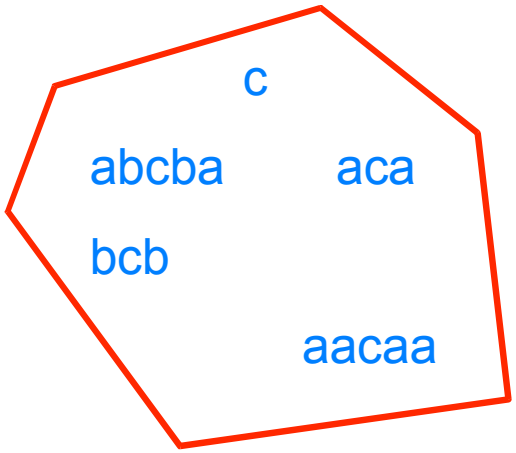
aacaa

a b ab abcb …

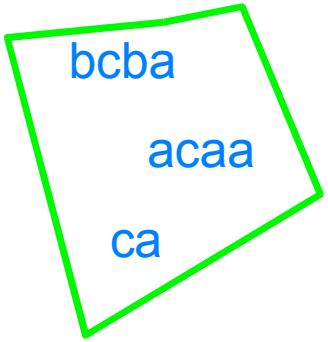
bc ba aac caa

Running example

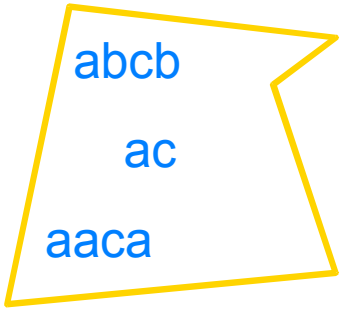
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



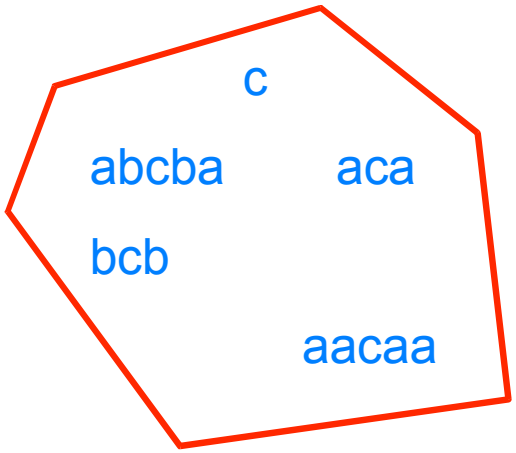
Context (ϵ,a)

a b ab abcb …

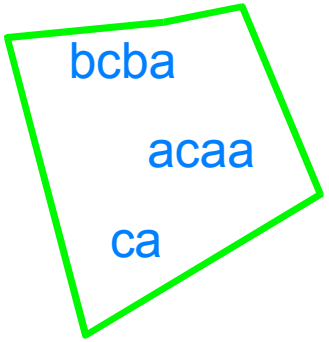
bc ba aac caa

Running example

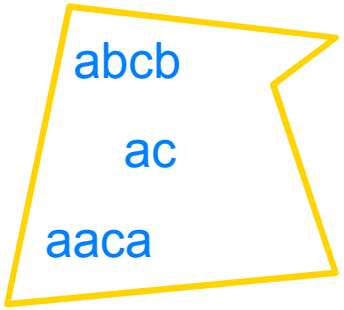
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

First step: create the rules for the letters of the alphabet.

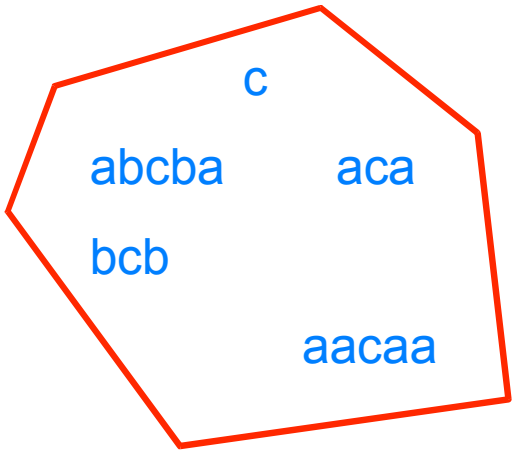
[a] → a

[b] → b

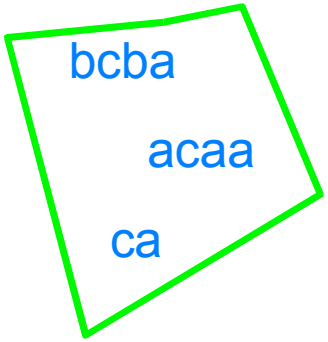
[c] → c

Running example

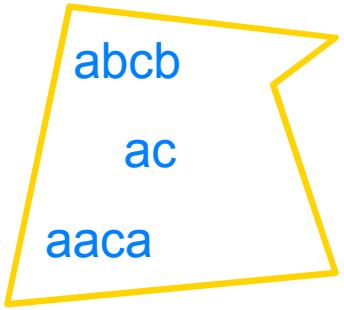
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

Second step: create a non terminal for each component.

[c] (=[aca]=[abcba]=[bcb]=[aacaa])

[ca] (=[bcba]=[acaa])

[ac] (=[abcb]=[aaca])

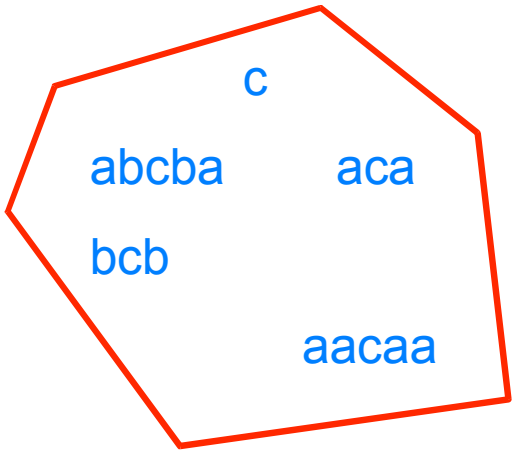
But also: [ab], [abcb], [bc], [aac], [ba]…

Algorithm: constructing the rules

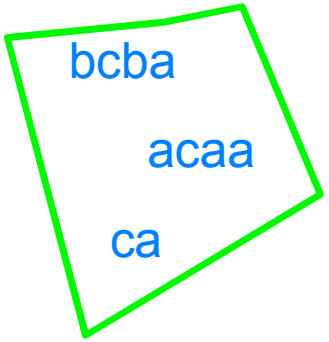
- Notice that the monoid induced by the syntactic congruence is well defined since [uv]=[u][v] for any choice of u and v ([u] represents the congruence class of u).
- We consider this as a production in a grammar: [uv]→[u][v].
- Each component of the graph corresponds to a congruence class.

Running example

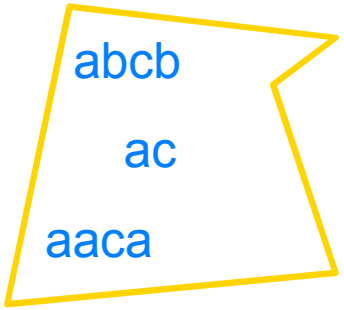
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

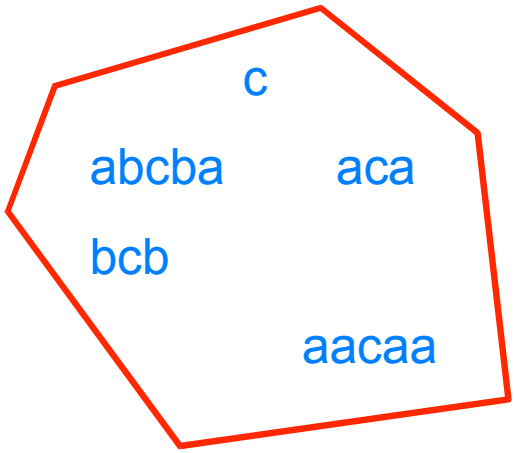
bc ba aac caa

Third step: create the rules for each component.

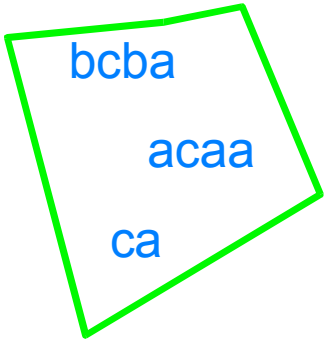
[w] → [u][v] when uv is a member of the component of w.

Running example

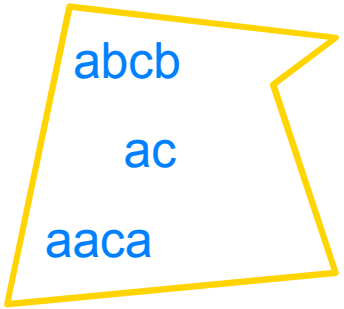
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

Component (ϵ,ϵ):

[c]→[a][bcba], [c]→[ab][cba], c→[abc][ba], [c]→[abcb][a]

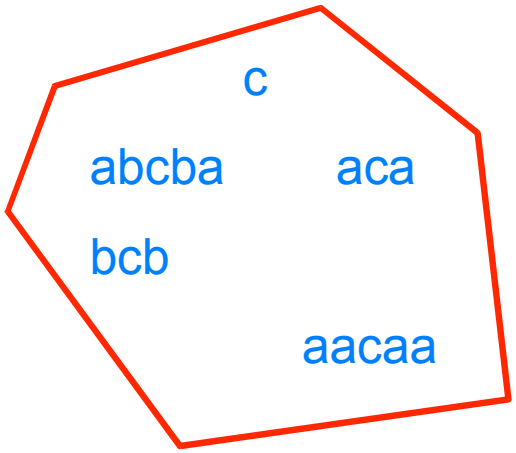
[c]→[a][ca], [c]→[ac][a]

[c]→[b][cb], [c]→[bc][b]

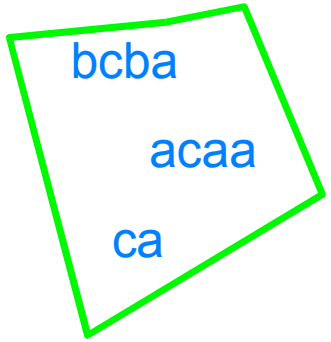
[c]→[a][acaa], [c]→[aa][caa], [c]→[aac][aa], [c]→[aaca][a]

Running example

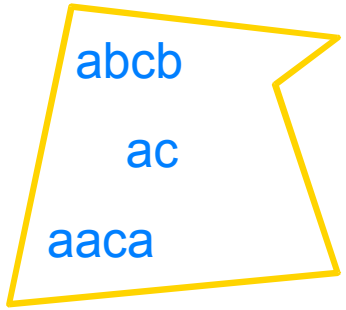
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

Component (ϵ,ϵ):

[c]→[a] [ca], [c]→[ab] [cba], [c]→[abc] [ba], [c]→[ac] [a]

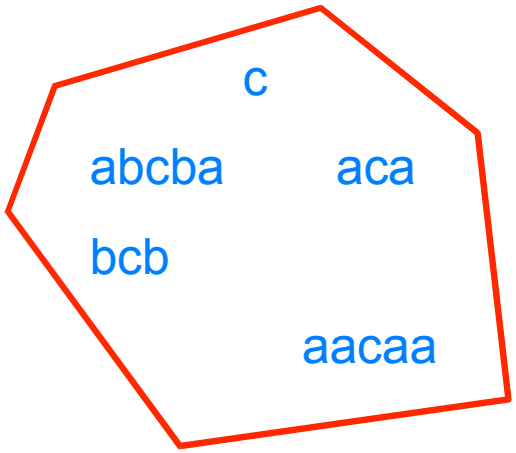
[c]→[a] [ca], [c]→[ac] [a]

[c]→[b] [cb], [c]→[bc] [b]

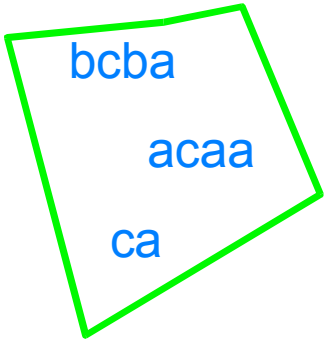
[c]→[a] [ca], [c]→[aa] [caa], [c]→[aac] [aa], [c]→[ac] [a]

Running example

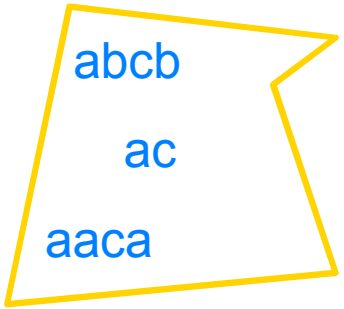
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

Component (ϵ,ϵ):

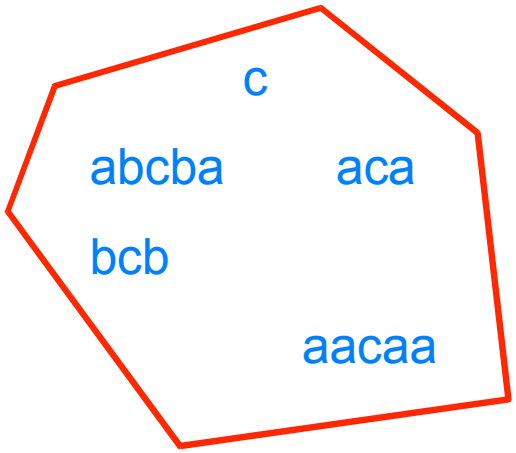
[c]→[a] [ca], [c]→[ac] [a],

[c]→[ab] [cba], [c]→[abc] [ba], [c]→[b] [cb],

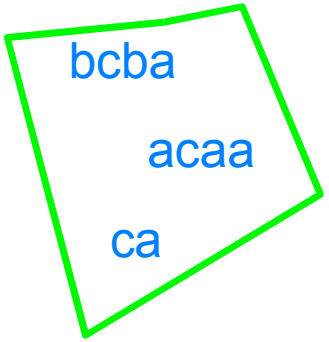
[c]→[bc] [b], [c]→[aa] [caa], [c]→[aac] [aa],

Running example

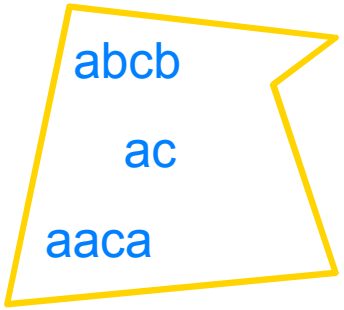
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

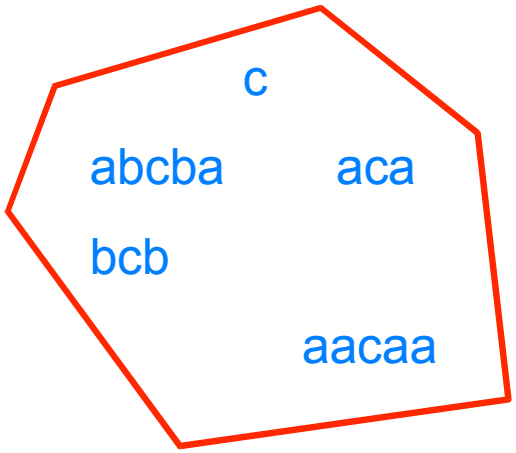
Component (a,ϵ): (final result)

[ca] → [c] [a], [ca] → [ac] [aa]

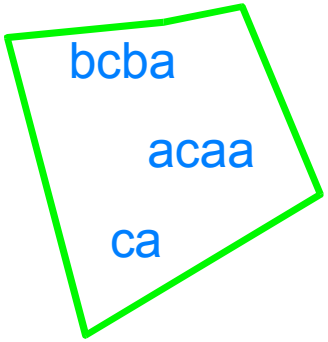
[ca] → [b] [cba], [ca] → [bc] [ba], [ca] → [a] [caa]

Running example

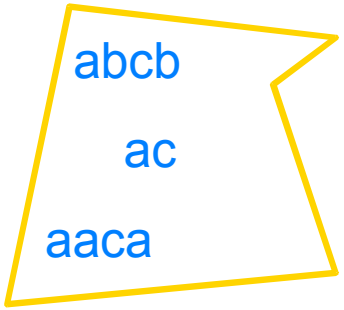
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

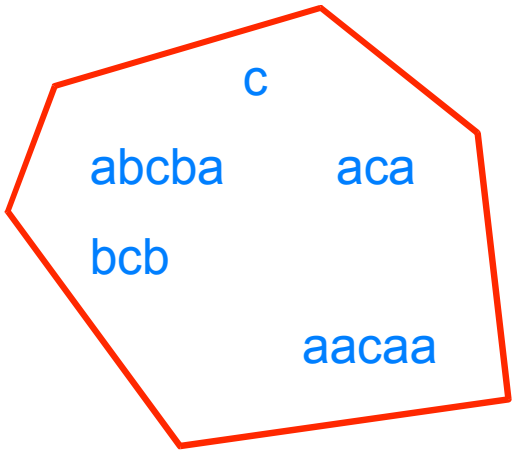
Component (ϵ,a): (final result)

[ac] → [a] [c], [ac] → [aa] [ca]

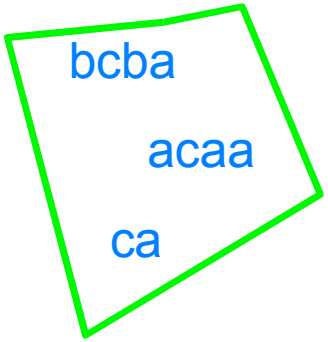
[ac] → [ab] [cb], [ac] → [abc] [b], [ac] → [aac] [a]

Running example

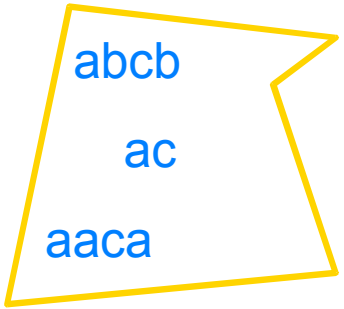
LS={c;aca;bcb;abcba;aacaa}



Empty context (ϵ,ϵ)



Context (a,ϵ)



Context (ϵ,a)

a b ab abcb …

bc ba aac caa

Components of one string:

[ab] → [a][b], [aac] → [a][ac], [aac] → [aa][c],
[aa] → [a][a] …

Running example

LS={c;aca;bcb;abcba;aacaa}

Outputed grammar: G=< {a,b,c}, V, P, [c] >

where P = {

[c] → [a][ca] | [ac][a] | [ab][cba] | [abc][ba] | [b][cb] | [bc][b] | [aa][caa] | [aac][aa] | c

[ca] → [c][a] | [ac][aa] | [b][cba] | [bc][ba] | [a][caa]

[ac] → [a][c] | [aa][ca] | [ab][cb] | [abc][b] | [aac] [a]

[a] → a

[b] → b

[ab] → [a][b]

[aac] → [a][ac] | [aa][c]

[aa] → [a][a]

… }

We can show that this grammar generates the language of palindromes with a center marked.

Learning Result

We prove in the article that the algorithm identifies polynomially in the limit the class of substitutable languages.

The polynomial bounds are on data and computation.

Syntactic monoid

- Given an alphabet Σ, Σ^* is the free monoid generated by Σ.

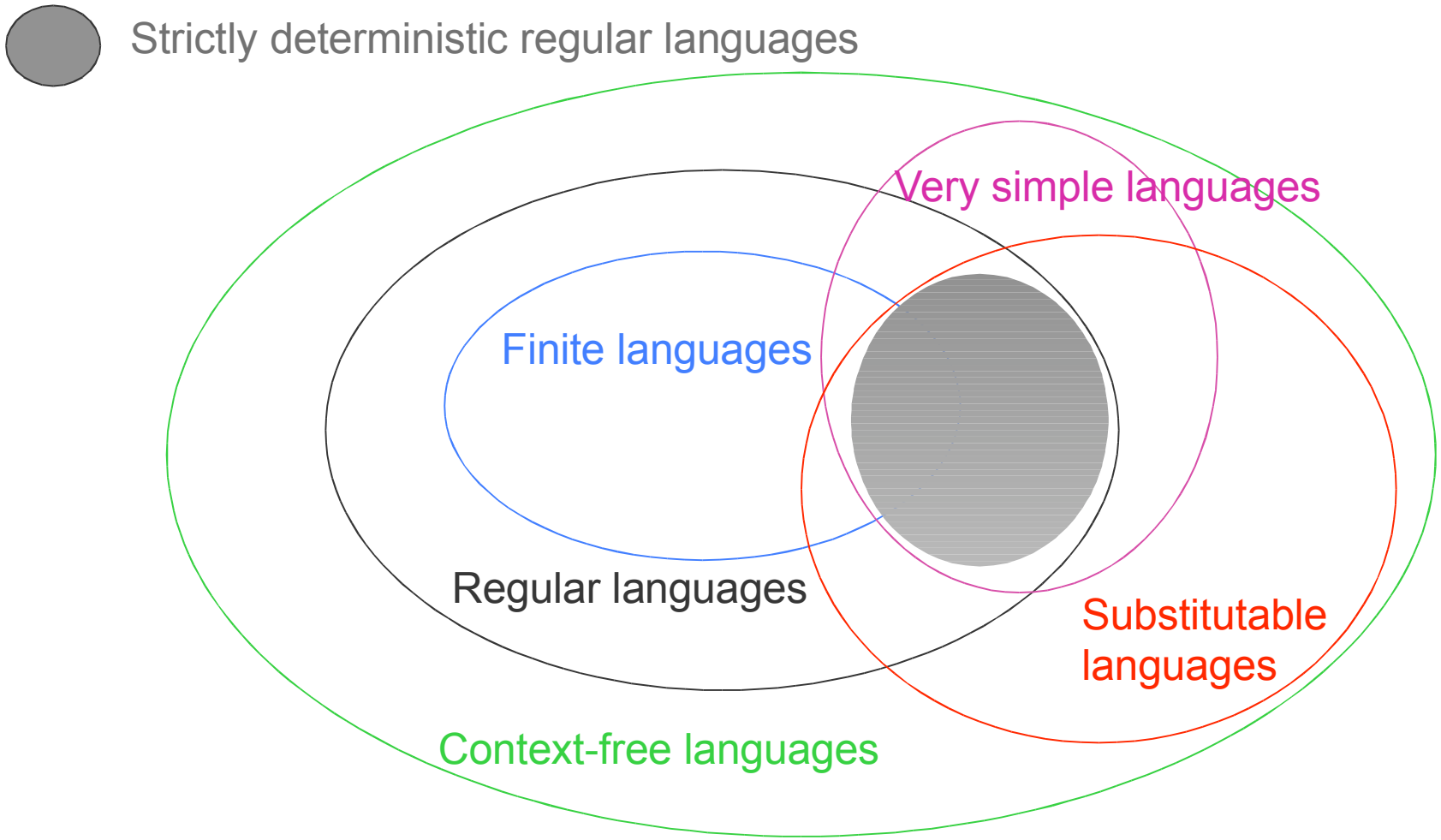
  The syntactic congruence $\equiv_L$ w.r.t. a language L defines the *syntactic monoid on L* as the quotient of Σ^* by $\equiv_L$.

- The algorithm identifies the (needed) elements of this monoid.

A new approach for CF

- Other approaches often rely on finding constituents (units of the structure of the target grammar). Thus they are strongly linked with the representation of the language.
- Our approach is purely syntactic and thus does not depend on the representation of the language (i.e. the property is on the structure of the language, not on the one of its representation).

Graphic comparison between classes



Substitutable context-free and reversible regular languages

- A regular language is reversible if whenever uw and vw are in the language then (ux is in the language iff vx is in the language) [Angluin, 82].
- Substitutable languages are the (context-free) exact analogue of reversible languages (regular).

Future work

- Decidability of the substitutable property of any given language.
- PAC-learning result.
- Comparing substitutable languages with other classes.

Open problems

- Our approach uses the simplest possible test for congruence. But one can use any other test and if the test works, the algorithm will work.
- Extension to bigger class.
- Using the same approach but with counter-examples.