Implementation of Regular Approximation of Contex-Free Grammars Through Transformation

Tapani Raiko

May 6, 2003

Abstract

In [1], Mohri and Nederhof presented an algorithm for approximating context-free languages with regular languages. My project work is the implementation of the algorithm using Prolog. The algorithm is tested on the arithmetic expressions example.

1 Introduction

In most real-time applications, general context-free grammars are computationally very demanding for demanding applications (such as real-time speech recognition). This downside can be avoided by approximationg context-free languages with regular languages. They can be further transformed into finite automata for practical reasons. Some context-free languages need to be approximated with regular languages for the transformation. The approximation implemented here is straightforward but not the only option.

2 Transformation

Any grammar can be transformed into a strongly regular grammar approximatively which means that the language generated by the strongly regular grammar is a superset of the original. In our case, all the rules in the strongly regular grammar are right-linear. The size of the resulting grammar is at most twice that of the input grammar. The transformation is defined as follows.

For each nonterminal A:
Introduce a new nonterminal A' (interpreted as "after A").
Add the rule $A' \to \epsilon$. (ϵ is the empty string)
Replace each rule of the form
$A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \dots B_m \alpha_m$,
where α are terminals
by the following set of rules:

E–>E+T
E–>T
T–>T*F
T–>F
F–>(E)
F–>a

E’–>.	T–>T
T’–>.	T’–>*F
F’–>.	F’–>T’
E–>E	T–>F
E’–>+T	F’–>T’
T’–>E’	F–>(E
E–>T	E’–>)F’
T’–>E’	F–>aF’

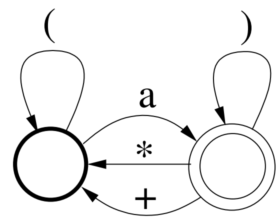

Figure 1: A grammar and its transformation representing arithmetic expressions. E stands for an expression, T for a term and F for a factor. The transformed grammar is equivalent to the finite state machine on the right.

$A \to \alpha_0 B_1$
$B'_1 \to \alpha_1 B_2$
$B'_2 \to \alpha_2 B_3$
$\ldots$
$B'_{m-1} \to \alpha_{m-1} B_m$
$B'_m \to \alpha_m A'$

The nonterminal A' can be interpreted as “after A”. Thus the rule $B'_1 \to \alpha_1 B_2$ could be interpreted as: If we just saw B_1, it is possible to continue with $\alpha_1 B_2$, since it was part of the original rule $A \to \alpha_0 B_1 \alpha_1 B_2 \alpha_2 \ldots B_m \alpha_m$. From this, it is clear that the transformed language is a superset of the original.

3 Implementation and Testing

I implemented the transformation using GNU Prolog 1.2.16[1]. The source code is included as Appendix B. The original grammar (Figure 1) is represented using the dynamic predicate rule/2. The predicate parse/3 implements a simple (and inefficient) parsing of a string using the grammar. The same predicate can be used to generate all parses up to length N. The predicate transform/1 does the transformation explained in Section 2. The rules are asserted to the program itself. If the original grammar has a nonterminal A, the transformed grammar will have the nonterminals $b(A)$ and $e(A)$ corresponding to A and A' accordingly.

An example run is shown in Appendix A. The command test(3) gives all possible parses of expression E up to length 3 using both the original grammar and its transformation. Note that the original language gives four parses: $a+a$, $a*a$, (a) and a whereas the transformed language gives the same ones and four others: $((a$, $(a$, $a))$ and $a)$. Note that a strongly regular grammar does not have the expressive power to check whether the parentheses match or not.

[1] http://pauillac.inria.fr/~diaz/gnu-prolog/

References

[1] M.-J. N. Mehryar Mohri, "Regular approximation of context-free grammars," in *Robustness in Language and Speech Processing* (J.-C. Junqua and G. van Noord, eds.), pp. 251–261, Kluwer Academic Publishers, 2000.

A Example Run

```
 james (8) ./gprolog
GNU Prolog 1.2.16
By Daniel Diaz
Copyright (C) 1999-2002 Daniel Diaz
| ?- consult('~/transform.pl').
compiling /home/praiko/transform.pl for byte code...
/home/praiko/transform.pl compiled, 144 lines read - 11485 bytes written, 187 ms

(23 ms) yes
| ?- test(3).
Parses using the original grammar:
[[a,+,a],[a,*,a],[(,a,)],[a]]

Parses using the transformed grammar:
[[(,(,a],[(,a,)],[(,a],[a,+,a],[a,),)],[a,)],[a,*,a],[a]]

(10 ms) yes
```

B Source Code

```
% Write out all parses of an arithmetic expression with the
% length restricted to N. Use first the original and then the
% transformed grammar.
test(N) :-
  findall(X, parse(['E'],X,N), Parses1),
  write('Parses using the original grammar:'),nl,
  write(Parses1), nl, nl,
  transform(_),
  findall(Y, parse([b('E')],Y,N), Parses2),
  remove_duplicates(Parses2, Parses2b),
  write('Parses using the transformed grammar:'),nl,
  write(Parses2b).

:- dynamic(rule/2).

% The original grammar
rule('E',['E','+','T']).
```

```
rule('E',['T']).
rule('T',['T','*','F']).
rule('T',['F']).
rule('F',['(','E',')']).
rule('F',['a']).

% Is X a nonterminal symbol?
nonterminal(X) :-
  rule(X,_).
% Is X a terminal symbol?
terminal(X) :-
  \+ nonterminal(X).

% parse(StringAbs,StringGround,MaxLength).
% applies rules to the abstract string producing the ground string in the end.
% maxlength is the maximum length for the string.

% done?
parse([],[],_) :- !.
% is the string already too long?
parse(AString,_,MaxLength) :-
  my_length(AString, Length),
  Length > MaxLength, !,
  fail.
% found a terminal, move to the rest.
parse([A|ARest],[G|GRest],MaxLength) :-
  terminal(A),
  A = G,
  ML1 is MaxLength - 1,
  parse(ARest,GRest,ML1).
% found a nonterminal, apply a rule.
parse([A|ARest],GString,MaxLength) :-
  rule(A,SubString),
  append(SubString, ARest, AStringNew),
  parse(AStringNew,GString,MaxLength).


% transform finds the transformed rules and asserts them to the program
transform(NewRules) :-
  retractall(rule(b(_),_)),
  retractall(rule(e(_),_)),
  findall(rule(NonT,Str),
  rule(NonT,Str),
  OldRules),
  transform(OldRules,NewRules,[]),
  assertall(NewRules).
```

```
% transform( OldRules, NewRulesRest, RejectThese)
% the rejection list is just for removing duplicates.
transform([],[],_).
transform([OldRule|ORest],OutNewRules,Reject) :-
  transform_rule(OldRule,TransformSet),
  cleanup_rules(TransformSet, Accepted, Reject),
  append(Reject, Accepted, Reject2),
  transform(ORest, InNewRules, Reject2),
  append(InNewRules, Accepted, OutNewRules).

% transforms a single rule. Returns a list of rules
transform_rule(rule(X,String), [rule(e(X),[])|TransRules]) :-
  new_rules(b(X), [], String, e(X), TransRules).

% new_rules( NonTerminal, Terminals, StringRest, FinalNonTerminal, ResultingRules)
% the original rule string is now empty
new_rules(NTer, Terminals, [], FinalNonTerminal, [rule(NTer,RuleString)]) :- !,
  append(Terminals, [FinalNonTerminal], RuleString).
% found a terminal. Move it to the terminal list from the rule string
new_rules(NTer, Terminals, [Sym|Symbols], FinalNonTerminal, Rules) :-
  terminal(Sym), !,
  append(Terminals, [Sym], Terminals2),
  new_rules(NTer, Terminals2, Symbols, FinalNonTerminal, Rules).
% found a nonterminal. Make a new rule.
new_rules(NTer, Terminals, [Sym|Symbols], FinalNonTerminal,
  [rule(NTer,NewRuleString)|Rules]) :-
  append(Terminals, [b(Sym)], NewRuleString),
  new_rules(e(Sym), [], Symbols, FinalNonTerminal, Rules).

% end of the main part. some utilities for removing duplicates etc. follow:

% identity_rule is a rule like T->T which makes no sense.
identity_rule(rule(X,[X])).

% cleanup_rules(OriginalSet,Result,RemoveThese).
% for removing duplicates and identity rules.
cleanup_rules([],[],_).
cleanup_rules([X|XRest],Result,Remove) :-
  member(X,Remove), !,
  cleanup_rules(XRest,Result,Remove).
cleanup_rules([X|XRest],Result,Remove) :-
  identity_rule(X), !,
  cleanup_rules(XRest,Result,Remove).
cleanup_rules([X|XRest],[X|YRest],Remove) :-
  cleanup_rules(XRest,YRest,Remove).
```

```
% assert all members of a list
assertall([]).
assertall([X|Xs]) :-
  asserta(X),
  assertall(Xs).

% my_length/2 is like normal length/2 except that it does not count
% any members of the form e(_).
% Other symbols will always produce at least one terminal.
my_length([], 0).
my_length([e(_)|Rest],N) :- !,
  my_length(Rest,N).
my_length([_|Rest],N1) :-
  my_length(Rest,N),
  N1 is N + 1.

% removes duplicates from input list (arg 1) and gives the result as arg 2
remove_duplicates([],[]).
remove_duplicates([X|Rest1],Rest2) :-
  member(X,Rest1), !,
  remove_duplicates(Rest1,Rest2).
remove_duplicates([X|Rest1],[X|Rest2]) :-
  remove_duplicates(Rest1,Rest2).
```