Towards Machine Learning of Grammars and Compilers of Programming Languages

Keita Imada and Katsuhiko Nakamura

School of Science and Engineering,
Tokyo Denki University

ECML 2008 Antwerp

Overview

- Syntax Directed Translation Scheme (SDTS): defines translation rules between two Context Free Languages.
- Definite Clause Grammar (DCG) and representation of SDTS by DCG
- Synapse System for grammatical inference of DCG.
- Experiments for learning a grammar and a compiler of a simple programming language from samples of pairs of source programs and object codes

Learning Grammars and Compilers

- Processing of programming languages has been a key technology in computer science.

  Much work in compiler theory: reducing the cost of implementing programming languages and optimizing object codes in compilers.

- Few works on applying machine learning to grammars and compilers.
  - Monsifrot, Boodin and Quiniou (2002) A method of improving optimization of object codes in compilers by decision trees.
  - Stephenson, Martin, O'Relly (2003) Improving optimization heuristic by genetic programming.

SDTS

A *syntax-directed translation scheme* (SDTS) is a system

$$T = (N, \Sigma, \Delta, R, s), \quad \text{where:}$$

Σ and Δ are sets of input and output terminal symbols, respectively; and R is a set of rules of the form

$$p \rightarrow u, v. \quad p \in N, u \in (N \cup \Sigma)^+, v \in (N \cup \Delta)^+,$$

such that the set of nonterminal symbols that occur in u is equal to that in v.

We write $w_1, w_2 \Rightarrow_T x_1, x_2$ for $w_1, x_1 \in (N \cup \Sigma)^*$ and $w_2, x_2 \in (N \cup \Delta)^*$, if there is a rule $(p \rightarrow u, v) \in R$ such that x_1 and x_2 are the results of rewriting w_1 and w_2 by the rules $p \rightarrow u$ and $p \rightarrow v$, respectively.

The SDTS T *translates* a string $w \in \Sigma^+$, into $x \in \Delta^+$ and vice versa, if $(s, s) \Rightarrow_T^* (w, x)$.

- We restrict the form of SDTS to *Chomsky normal form* (CNF), in which every string in the right hand side of the rule has at most two symbols, e.g.

$$p \to p\,q, p\,q. \qquad p \to p\,q, q\,p. \qquad p \to a, b.$$

  It is known that SDTS in CNF is less powerful.
  We use this form because synthesizing rules in CNF is easier.

- The SDTS in our work is extended so that elements of the strings may be not only constants but also terms.

Example of SDTS: Reversing strings

The following SDTS in CNF is to translates any string into its reversal, for example, $aababb$ to $bbabaa$.

$$s \to a, a. \qquad s \to b, b. \qquad s \to a\,s, s\,a. \qquad s \to b\,s, s\,b.$$

This SDTS derives the set of pairs of strings,

$$(a, a), (b, b), (aa, aa), (ab, ba), (ba, ab), (bb, bb), (aaa, aaa), (aab, baa), \cdots.$$

Definite Clause Grammars (DCG)

A *DCG rule*, (or *grammar rule*) is of the form P `-->` $Q_1, Q_2, \cdots, Q_m$, where:

- P is a nonterminal term, which is either a symbol or a complex term of the form $p(T)$ with a *DCG term* T.

  (To simplify the synthesis process, we restrict every nonterminal term to have exactly one DCG term); and

- each of $Q_1, \cdots, Q_m$ is either a constant of the form [a] representing a terminal symbol, or a nonterminal term.

The DCG terms are additional arguments in the Horn clause rules, which are generally used for controlling the derivation and for returning results of the derivation.

The deduction by DCG is similar to that of CFG, except that each of the DCG terms is unified with a corresponding term in the deduction.

Example: A DCG for a non-context-free language

A DCG for the language $\{a^n b^n c^n | \, n \geq 1\}$:

```
p(1) --> [a].        p(t(N)) --> [a],p(N).
q(1) --> [b].        q(t(N)) --> [b],q(N).
r(1) --> [c].        r(t(N)) --> [c],r(N).
s(N) --> p(N),q(N),r(N).
```

The Horn clauses transformed from this DCG:

```
p(1,[a|X],X).        p(t(N),[a|X],Y) :- p(N,X,Y).
q(1,[b|X],X).        q(t(N),[b|X],Y) :- q(N,X,Y).
r(1,[c|X],X).        r(t(N),[c|X],X) :- r(N,X,Y).
s(N,X0,X3) :- p(N,X0,X1),q(N,X1,X2),r(N,X2,X3).
```

For the query `?-s(N,[a,a,a,b,b,b,c,c,c],[])`, the Prolog system returns the computation result `N = t(t(1))`.

Transformation of SDTS into DCG

We can transform a SDTS in CNF into a DCG, and into a Prolog program for translating strings by the following relations.

SDTS	DCG
$p \rightarrow q$	$p(X/Y)$ `-->` q
$p \rightarrow a, b$	$p([b\|Y]/Y)$ `-->` $[a]$
$p \rightarrow ar, ra$	$p(X/Y)$ `-->` $[a], r(X/[a\|Y])$
$p \rightarrow qr, qr$	$p(X/Z)$ `-->` $q(X/Y), r(Y/Z)$
$p \rightarrow qr, rq$	$p(X/Z)$ `-->` $q(Y/Z), r(X/Y)$

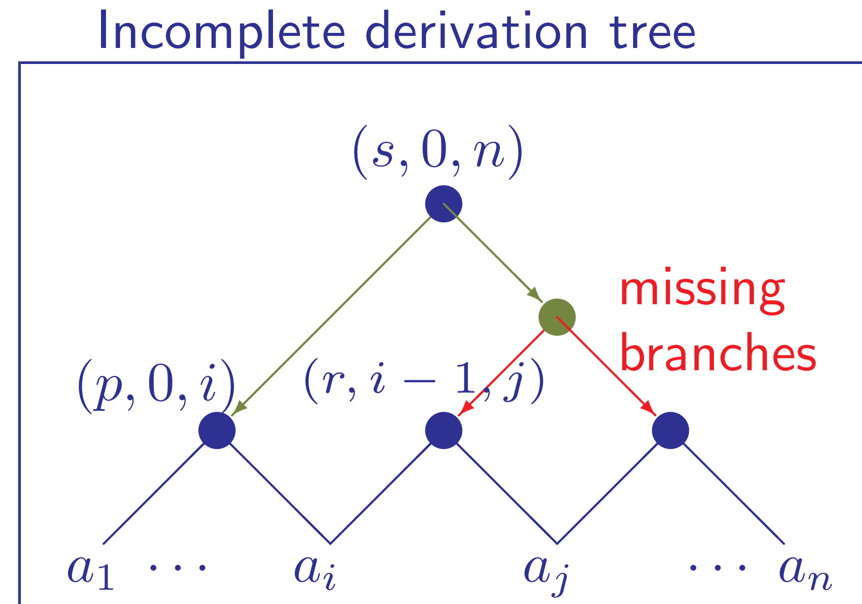
Each pair of the form X/Y represents an output side of a string by a difference list.

By this method, the problem of learning SDTS in CNF can be transformed into that of learning DCG.

Rule Generation by Bridging in *Synapse* System

- *Synapse* is originally developed for inferring CFGs, recently extended to DCGs, from positive and negative sample strings.
- When the system fails to parse a positive sample, it looks for the missing branches in the incomplete derivation trees and generates the rules, which bridges the gap in the trees.

  - Blue lines represent the result of bottom-up parsing.
  - Green arrows represent the existing rule.
  - Red arrows represent the missing branches.



Rule Generation Procedure by Bridging

Procedure $RuleGeneration(w, S)$ (Comment: w: an input string, S: the starting term, Global variable RS holds a set of rules.)

Step 1 (Initialize variables.)
$D \leftarrow \emptyset$. (D is a set of 3-tuples (β, i, j).)

Step 2: Bottom-up parsing based on CYK algorithm.
If $(S, 0, n) \in D$ then terminate (Success).

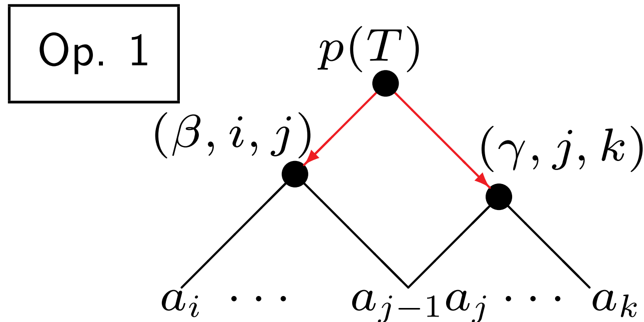
Step 3: (Bridging rule generation)
Call procedure $Bridge(S, 0, n)$. Terminate (Success). (Return RS).

Procedure $Bridge(P, i, k)$ (P : a nonterminal term, i, k : integers.)
Nondetermistically choose one of the bridging operations.

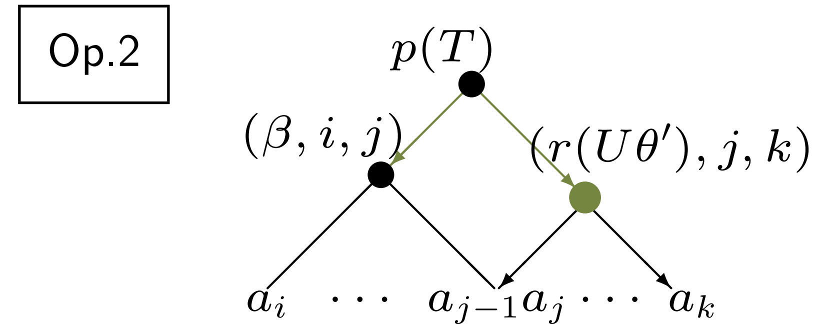
Procedure $AddRule(R)$ (R : a rule.)
Nondereminsitically chose one of the following process 1 or 2.

1. If RS contains a rule R' that unifies with R, delete R' from RS and add the least general generalization of R and R' to RS instead of R'.
2. Otherwise, add R to RS.

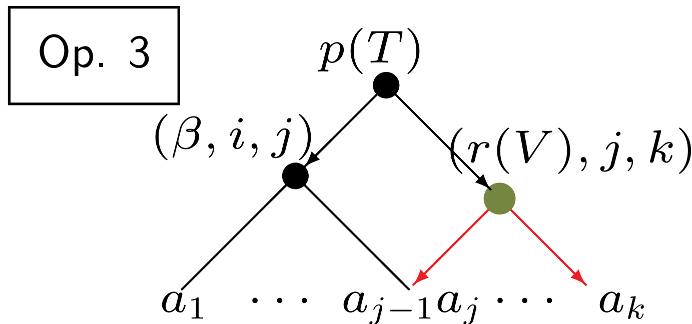
Operators for Type $P(T) \rightarrow \beta\gamma$ Rules



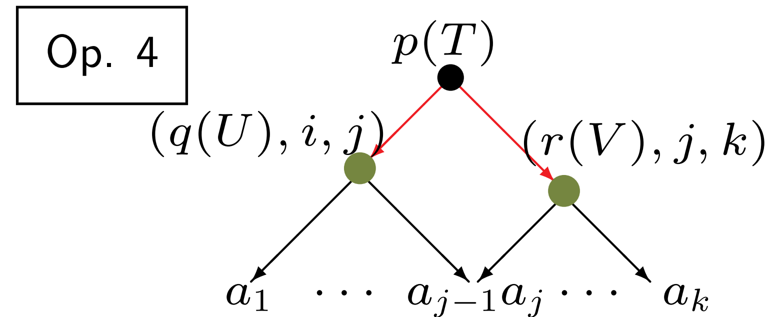
If $(\beta, i, j) \in D$ and $(\gamma, j, k) \in D$,
add $p(T) \rightarrow \beta\gamma$.



If $p(T') \rightarrow \beta\, r(U) \in P$ with $T \equiv_\theta T'$,
and $(\beta\theta, i, j) \in D$ with $\beta \equiv_{\theta'} \beta'$,
call $Bridge(r(U\theta'), j, k)$.



If $(\beta, i, j) \in D$, add $p(T) \rightarrow \beta\, r(U)$
and call $Bridge(r(U), j, k)$.



For each j with $i + 2 \leq j \leq k - 2$,
add $(p(T) \rightarrow q(U)\, r(V))$ and call
$Bridge(q(U), i, j), Bridge(r(U), j, k)$.

Search for Rule Sets

- Synapse has inputs of ordered sets of positive and negative samples, and an optional set of initial rules.

  The system searches for any set of rules which derives all the positive strings but no negative string.

- We employ two search strategies.

Global Search (Search for Minimal Rule Sets): The system searches for the minimum set of rules satisfying given sets of positive and negative samples by iterative deepening and backtracking.

Serial Search (Search for Semi-Optimum Rule Sets):
The system searches for the minimum set of rules for each positive sample.

We can generally reduce the computation time at slight expense of increasing the number of rules.

DCGs Synthesized by Synapse.

Grammar	Examples
(a) $\{(a^nb^nc^n) \mid n \geq 1\}$ R: 4, T: 3243, GR: 7×10^5	

`e(t(X))→ e(1), e(X).` `* e(1)→ [a].`
`g(t(X))→ p(X), g(1).` `* f(1)→ [b].`
`p(X)→ f(1), g(X).` `* g(1)→ [c].`
`s(X)→ e(X), p(X).`
	`[a,b,c] - s(1).`
	`[a,a,b,b,c,c] - s(t(1)).`
	`[a,a,a,b,b,b,c,c,c]-s(t(t(1))).`

(b) $\{ww \mid w \in \{a,b\}^+\}$ R: 2, Time: 270, GR: 3×10^5.	

`* p([a])→ [a].` `* p([b])→ [b].`
`p([X|Y])→ p(Y), p([X]).`
`s(X)→ p(X), p(X).`
	`[a,a]-s([a]).`
	`[b,a,b,a]-s([b,a]).`
	`[b,a,a,b,a,a] - s([b,a,a]).`

(c) Simple English grammar (s: singular, p: plural), R: 6, T: 93, GR: 1.3×10^5	

`* article(s)→[a].` `* article(s)→[the].`
`* article(p)→[the].`
`* verb(p)→[have].` `* verb(s)→[has].`
`* be(p)→[are].` `* be(s)→[is].`
`* noun(s)→[pen].` `* noun(p)→[pens].`
`* noun(s)→[this].` `* noun(p)→[these].`
`* noun(s)→[he].` `* noun(p)→[they].`
`p(X)→be(X),p1(X).` `p(X)→verb(X),p1(Y).`
`p(p)→be(p),noun(p).p(s)→verb(s),noun(p).`
`p1(X)→article(X),noun(X).`
`s(X)→noun(X), p(X).`
	`[he, has, pens] - s(s).`
	`[he, has, a, pen] - s(s).`
	`[they, have, a, pen] - s(p).`
	`[this, is, a, pen] - s(s).`
	`[this, is, the, pen] - s(s).`
	`[he, has, the, pens] - s(s).`
	`[they, have, pens] - s(p).`
	`[they, have, the, pens]-s(p).`
	`[these, are, pens]- s(p).`
	`[these, are, the, pens]-s(p).`

Learning Grammar and Simple Compiler

- The source language: extended arithmetic expression containing function calls and the assignment operator $(=)$
- The object language: *SIL* (Simple Intermediate Language). *SIL* is similar to P-code and byte-code, which were originally designed for intermediate languages of Pascal and JAVA compilers, respectively.

  These languages are based on the inverse Polish notation and stack machines.
- For reducing the computation time, we divide the learning of translation into two steps, and use incremental learning.

  First step: arithmetic expressions with $+, -, *, /$.
  Second step: extended arithmetic expressions with $=$ and function calls.

Instructions in SIL

- `load(T,V)` pushes the value of variable (symbolic address) `V` of type `T`, which is either `i` or `f` (float), to the top of the stack.
- `push(T,C)` pushes a constant `C` of type `T` to the top of the stack.
- `store(T,V)` stores the value at the top of the stack to variable (symbolic address) `V` of type `T`.
- `fadd, fsubt, fmult, fdivide`. These float type arithmetic operations are applied to two values on the stack and return the result at the top of the stack instead of the two values.
- `call(n,S)` calls function `S` with `n` parameters, which are placed at the top of the stack. The function returns the value at the top of the stack.

Samples for Step One

Positive samples (arithmetic expression : object code)

```
a : push(f,a).                           ( a ) : push(f,a).
a + a : push(f,a),push(f,a),fadd.        a * a : push(f,a),push(f,a),fmult.
a / a : push(f,a),push(f,a),fdivide.     a - a : push(f,a),push(f,a),fsubt.
a + a + a : push(f,a),push(f,a),fadd,push(f,a),fadd.
a * a + a : push(f,a),push(f,a),fmult,push(f,a),fadd.
( a ) + a : push(f,a),push(f,a),fadd.
a + a * a : push(f,a),push(f,a),push(f,a),fmult,fadd.
a * a * a : push(f,a),push(f,a),fmult,push(f,a),fmult.
```

Negative samples

```
( a : X.   a ) : X.   ( + a : X.   ( * a : X.   + a : X.   * a : X.
a + : X.     a * : X.    (restriction on on the source language)
a + a : push(f,a),fadd,push(f,a).  a * a : push(f,a),fmult,push(f,a).
a + a + a : push(f,a),push(f,a),push(f,a),fadd,fadd.
a * a + a : push(f,a),push(f,a),push(f,a),fmult,fadd.
a + a * a : push(f,a),push(f,a),fadd,push(f,a),fmult.
```

Initial Rules

(a) Initial rules for Step One

```
n([push(f,a)|Y]/Y) --> a.        n([push(f,b)|Y]/Y) --> b.
n([load(f,x)|Y]/Y) --> x.        v([store(f,x)|Y]/Y) --> x.
n([load(f,y)|Y]/Y) --> y.        v([store(f,y)|Y]/Y) --> y.
op1([fadd|Y]/Y) --> +.           op1([fsubt|Y]/Y) --> -.
op2([fmult|Y]/Y) --> *.          op2([fdivid|Y]/Y) --> /.
lp(Y/Y) --> '('.                 rp(Y/Y) -->')'.
```

(b) Initial rules for Step Two

```
op3(Y/Y) --> =.                  s(X/Y) --> s1.
fn([call(1,sin)|Y]/Y) --> sin.   fn([call(1,cos)|Y]/Y) --> cos.
```

Relations between:
the operators and operands in the source languages and
the basic instructions in the object codes.

Samples for Step Two

Positive samples for Step 2

```
x = a : push(f,a), store(f,x).
sin ( a ) : push(a),call(1,sin).
x = a + b : push(f,a), push(f,b),fadd, store(f,x).
x = y = a : push(f,a), store(f,y),store(f,x).
sin ( a + b ) :  push(f,a),push(f,b),fadd,call(1,sin).
sin ( a ) + b : push(f,a),call(1,sin),push(f,b),fadd.
sin ( a ) * cos ( b ) : push(f,a),call(1,sin), push(f,b),
           call(1,cos),fmul.
sin ( cos ( b ) ) : push(f,b),call(1,cos),call(1,sin).
```

Negative samples for Step 2

```
sin a : X.           sin + a : X.         sin * a : X.         sin a ) : X.
sin ( a ) a : X.     sin ) : X.           sin sin : X.         sin + : X.
sin * : X.           = a a ) : X.         = a : X.             ( = a : X.
( a = a : X.         a = x : X.           = x ( a ) : X.       a = b b : X.
x = a + b : push(f,a),store(f,x),push(f,b),fadd.
```

The Results of Learning

Step 1: Synapse synthesized eight DCG rules after generating 3.4×10^6 rules in approximately 2000 sec.
(We used a Xeon processor with 3.6 GHz clock and SWI-Prolog for Linux.)

Among the positive samples, only the first nine samples were directly used for generating the grammar and the other positive samples are used for checking of the translation.

Step 2: By incremental learning, Synapse synthesized the four additional rules after generating 1.8×10^7 rules in 4400 sec.

Only the first four positive samples were directly used for generating the grammar.

Synthesized Rules

(a) Rules for Arithmetic Expressions Synthesized in Step 1

DCG	SDTS
`s1(X/Y) --> e(X/Y).`	s $\rightarrow$ e, e
`s1(X/Z) --> s1(X/Y), f(Y/Z).`	s $\rightarrow$ s f, f p
`f(X/Z) --> op1(Y/Z), e(X/Y).`	p $\rightarrow$ op1 e, e op1
`e(X/Y) --> n(X/Y).`	e $\rightarrow$ n, n
`e(X/Z) --> e(X/Y), g(Y/Z).`	e $\rightarrow$ e g, e g
`g(X/Z) --> op2(Y/Z), n(X/Y).`	r $\rightarrow$ op2 n, n op2
`n(X/Z) --> lp(X/Y), p(Y/Z).`	s $\rightarrow$ lp p, lp p
`p(X/Z) --> s1(X/Y), rp(Y/Z).`	p $\rightarrow$ s rp, s rp

(b) Rules for Function Calls and "=" Operator Synthesized in Step 2

`n(X/Z) --> fn(Y/Z), q(X/Y).`	n $\rightarrow$ fn q, q fn
`q(X/Z) --> lp(X/Y), p(Y/Z).`	n $\rightarrow$ lp p, lp p
`s(X/Z) --> v(Y/Z), r(X/Y).`	s $\rightarrow$ v r, r v
`r(X/Z) --> op3(Y/Z), s(X/Y).`	q $\rightarrow$ op3 s, op3 s

- We converted the synthesized DCG rules to usual Prolog rules and tested the compiler.

  We needed to remove the left recursion by folding some of the clauses before executing the compiler in Prolog.

- The synthesized SDTS contains an unambiguous CFG for the extended arithmetic expressions, which specifies that:

  1. The precedence of the operators $*$ and / is higher than that of + and −, which is higher than the assignment operator "="; and
  2. All arithmetic operators are left associative.
     The operator "=" is right associative as in C.

Concluding Remarks

- We showed an approach for machine learning of grammars and compilers of programming languages based on grammatical inference of DCG and SDTS, and showed the experimental results.

- Although we showed learning of only a portion of compiling process of the existing language, the learning system synthesized an essential part of the compiler from samples of translation.

- This approach can be used for producing the grammar of a new language and at the same time implement the language from the samples of source programs and object codes.

Current Works and Future Problems

We are currently working on improving and extending the methods of learning DCGs and compilers.

Other future subjects include:

- Theoretical analysis of learning DCG and SDTS.
- Clarifying the limitations of our methods in learning grammars and compilers.
- Applying our approach for learning DCG to syntactic pattern recognition.
- Applying our approach for learning DCG to general ILP, and inversely applying methods in ILP to learning DCG and SDTS.

Thank you.