

LR法に基づく新しい並列構文解析アルゴリズム

4 F - 3

— 富田法の自然な拡張 —

沼崎 浩明

田中 穂積

(東京工業大学工学部)

1 まえがき

本論文では、富田法を拡張した新しい並列一般化 LR 構文解析アルゴリズムについて述べる。LR 構文解析法を文脈自由文法に適用した富田法 [Tomita 85] は、横型的な戦略によって LR テーブルのコンフリクトを扱っており、並列処理との整合性が良い。既に我々は富田法に基づく並列パーザをの並列論理型言語 GHC を用いて実現した [沼崎 89]。しかし、富田法を並列構文解析に用いるには問題点がある。富田法は複数のスタックを統合する際に、各語に対する shift 動作でプロセスの同期をとる。このため、先に shift 動作を終えたプロセスは、他の全てのプロセスが shift 動作を終えるまで、次の処理に進めない。この戦略は並列性の低下を招く。これを改善するために、我々は複数のプロセスの同期を必要としないスタック統合の方法を並列論理型言語 GHC [Ueda 85] の枠組で実現し、これに基づいた新しい並列一般化 LR 構文解析アルゴリズムを提案する。このアルゴリズムは富田法の自然な拡張であり、計算量は富田法と等しく、並列性は富田法より高い。

LR 法に基づく並列構文解析アルゴリズムの研究として、峯 [峯 89] らの報告がある。このアルゴリズム富田法とは異なった戦略でプロセスの統合を行ない、並列性は富田法より高いが、純粋な CFG を対象としており、文法記号に属性情報を与えた場合を扱っていない。これに対し我々のアルゴリズムは DCG 形式で記述された文法を対象としており、補強項の計算によって意味処理を行なうことができる。

2 富田法について

富田法は LR 構文解析法に基づいており、文法規則から LR テーブルを生成し、パーザの状態と先読みによってこれを参照する。パーザはスタックを持ち、テーブルが指定する shift, reduce 動作によってスタックを操作しながら入力文を解析する。例えば、図 1 の文法に対しては、図 2 のようなテーブルが得られる。

文法規則に曖昧さがある場合、LR テーブルには複数の動作を指定するエントリが生ずる。これをコンフリクトと呼ぶ。パーザがコンフリクトに出会うと、そのエントリ中の各動作に対してプロセスを作り、そのプロセスがそれぞれの処理を実行する。複数のプロセスによる入力文の局所的な解析過程が同じになる場合、各プロセスが持つスタックを統合して、その部分の解析を一つのプロセスで行なう。スタックを統合するために複数のプロセスを同期させる。しかし、これは並列性の低下を招く。

- (1) S → NP, VP.
- (2) S → S, PP.
- (3) NP → NP, PP.
- (4) NP → det, noun.
- (5) NP → pron.
- (6) VP → v, NP.
- (7) PP → p, NP.

図 1: 曖昧性のある文法規則

	det	noun	pron	v	p	\$	NP	PP	VP	S
0	sh1		sh2				4			3
1		sh5								
2				re5	re5	re5				
3					sh6	acc		7		
4				sh8	sh6			10	9	
5				re4	re4	re4				
6	sh1		sh2				11			
7					re2	re2				
8	sh1		sh2				12			
9					re1	re1				
10				re3	re3	re3				
11				re7	sh6/re7	re7		10		
12					sh6/re6	re6		10		

図 2: LR テーブル

3 スタックの扱い

富田法ではグラフ構造化スタックを用いているが、我々は論理型言語での実現の容易さを考慮して、スタックの先頭部分のみを統合した木構造化スタックを用いる。木構造化スタックは、論理型言語のリスト型データで表現できる。例えば、図 1 の文法と図 2 のテーブルを用いて次の入力文を構文解析する時、

I open the door with a key.

(pron) (v) (det) (noun) (p) (det) (noun) (\$)

パーザは、with の shift を完了した時点で、次の二つのスタックを得る。

(1) 先頭: [6, p, 3, s, 0] : 底

(2) 先頭: [6, p, 12, np, 8, v, 4, np, 0] : 底

図 3: 統合可能な二つのスタック

これ以後の二つのスタックを用いた解析は、先頭の要素 '6, p' をポップするまで同じなので、スタックの先頭の部分を一つに統合して木構造化し、それ以後の解析を一つのプロセスで行うことができる。

(3) [6, p, [3, s, 0], [12, np, 8, v, 4, np, 0]]

図 4: 木構造化スタックのリスト表現
この木構造化スタックを以下に図示する。

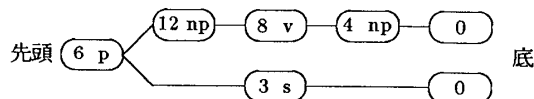


図 5: 木構造化スタック

4 並列構文解析アルゴリズム

我々のアルゴリズムでは、語数 n の入力文に対し親プロセス $P_1, P_2, \dots, P_n, P_S$ を作り、各プロセスに入力語を一つずつ割り当てる。親プロセス P_i は入力ストリームと出力ストリームを持つ。ストリームにはスタックが流れ、次のようなデータ構造で表現される。

$[[Stack_1], [Stack_2], \dots, [Stack_n] | Stream]$

ただし、 $[Stack_i]$ は 3 の (1) のようなスタックか、(3) のような木構造化スタックである。また、 $'Stream'$ は変数である。

各親プロセス P_{i-1} の出力ストリームを P_i の入力ストリームとする。 P_i は割り当てられた入力語の品詞と、スタックトップの状態によって LR テーブルを参照し、スタックを操作する。スタック操作は、必要な回数の reduce 操作の後、一回の shift 操作 (P_S では accept) によって完了する。 P_i はそれによって得られたスタックを出力ストリームに流す。

入力文が曖昧な時、ある親プロセスは解析の途中でコンフリクトに出会い、そのエントリ中の各処理を一つずつ子プロセスに実行させる。各子プロセスが行なうスタック操作の回数は異なるので、各子プロセスが終了するタイミングは異なる。親プロセスは各子プロセスからスタックを回収し、それを出力ストリームに流すが、先頭の要素が等しいものは統合して木構造化スタックとする。富田法を採用した場合、全ての子プロセスが終了した後で、スタックを出力ストリームに流すことになるが、我々のアルゴリズムは次の点に着目して、一つの子プロセスの処理が終了すると直ちに出力ストリームにそのスタックの情報を送る。

ある子プロセスが、他の子プロセスよりも先に $'shift N'$ 操作 (N は状態番号) を実行し、処理を終えたとする。これによって、他の子プロセスがどのような shift 操作を行なおうとも、隣の親プロセス P_{i+1} が受け取るスタックには $'N'$ を先頭の要素とするスタックが存在することが確定する。従って、親プロセス P_i は全ての子プロセスの終了を待たずに、一つの出力スタックの先頭の状態番号が $'N'$ であるという情報を出力ストリームに送ってよい。ただし、そのスタックの後ろの部分は構造化するかどうか決らないので、未定義とする。

これによって、 P_{i+1} 以後の親プロセスは、 P_i の子プロセスと並列に走ることができる。 P_{i+1} 以後のプロセスが、先送りされたスタックから要素 $'N'$ をポップして、未定義の部分を参照しようとする、そこで初めてプロセスを中断する。

5 スタック及びストリーム操作例

我々のアルゴリズムに従って、スタック及びストリームを操作する例を示す。3 の例文を用いると、入力語 $'with'$ を持つ親プロセス P_5 の中では、図 4 のスタック (1) と (2) が生成される。今、スタック (2) が (1) より先に生成されたとする。この時点で、プロセス P_5 は出力ストリームを次のように具体化する。

$[[6, p | Tail] | Stream]$

図 6: プロセス P_5 の出力ストリーム

ここで、 $'6, p'$ はスタック (2) の先頭の要素である。このストリームは、先頭以外の部分が未定義のスタックを一つ持つ。変数 $'Tail'$ と $'Stream'$ はスタック (1) が生成されるまで未定義のままである。これによって、以下のプロセスが P_5 と並列に走る。

Step	Process	Action	Stream
1	P_6	sh 1	$[[1, \text{det}, 6, p Tail] Stream]$
2	P_7	sh 5	$[[5, \text{noun}, 1, \text{det}, 6, p Tail] Stream]$
3	P_S	re 4	$[[6, p Tail] Stream]$
4	P_S	goto 11	$[[11, \text{np}, 6, p Tail] Stream]$
5	P_S	re 7	$[Tail Stream]$
6	P_S	中断	

図 7: プロセス P_5 と並列実行可能なプロセス

Step 6 の中断はプロセス P_S がスタックの未定義の部分を参照しようとするために起こる。

プロセス P_5 は、スタック (1) が生成された後、ストリームの未定義変数を次のように具体化する。

$Tail = [[12, \text{np}, 8, v, 4, \text{np}, 0], [3, s, 0]]$

$Stream = []$

これによって、図 6 に示したストリームは、図 5 の木構造化スタックを一つ持つことが確定する。図 7 の Step 6 の実行前に、プロセス P_5 がストリームをこのように具体化すれば、Step 6 の中断は生じない。

文法によっては、スタック (1) と (2) の先頭の要素が異なる場合もある。例えば、スタック (1) が次のような場合は、スタック (2) と統合できない。

(1) $[8, p, 3, s, 0]$

このような場合は、プロセス P_5 は出力ストリーム中の未定義変数を、

$Tail = [12, \text{np}, 8, v, 4, \text{np}, 0]$

$Stream = [[8, p, 3, s, 0]]$

のように具体化する。この時、図 6 に示したストリームは、スタック (1) とスタック (2) を別々に持つことになる。

6 結論と今後の課題

本論文では富田法を拡張し、より高い並列性を実現するための新しい並列一般化 LR 構文解析アルゴリズムについて述べた。我々のアルゴリズムでは、富田法で起こるプロセスの無駄な中断がない。また、富田法では、入力語を左から右に逐次的に処理するが、我々のアルゴリズムでは、曖昧な文に対しては、複数の語が並列に処理される。しかも、計算量の点では富田法と全く同等で無駄な解析が少ない。

我々は、より具体的な実験を、64 台のプロセッサを持つ ICOT のマルチ PSI システムを使用して行ない、本アルゴリズムの有用性を確認する予定である。

参考文献

- [沼崎 89] 沼崎浩明, 田村直良, 田中穂積: 並列論理型言語による一般化 LR 構文解析アルゴリズムの実現, 自然言語処理 74-5, PP.33-40 (1989)
- [峯 89] 峯 恒憲, 谷口倫一郎, 雨宮真人: 文脈自由文法の並列構文解析, 情報処理学会自然言語処理研究会研究報告, 73-1, pp.1-8 (1989)
- [Tomita 85] Tomita, M.: *Efficient Parsing for Natural Language*, Kluwer Academic Publishers (1985)
- [Ueda 85] Ueda, K.: *Guarded Horn Clauses*, Proc. The Logic Programming Conference, Lecture Notes in Computer Science, 221 (1985)