DETERMINISTIC TECHNIQUES FOR EFFICIENT NON-DETERMINISTIC PARSERS

Bernard LANG

I.R.I.A. 78 Rocquencourt, FRANCE

Abstract : A general study of parallel non-deterministic parsing and translation à la Earley is developped formally, based on non-deterministic pushdown acceptor-transducers. Several results (complexity and efficiency) are established, some new and other previously proved only in special cases. As an application, we show that for every family of deterministic context-free pushdown parsers (e.g. precedence, LR(k), LL(k), ...) there is a family of general context-free parallel parsers that have the same efficiency in most practical cases (e.g. analysis of programming languages).

0. Introduction

Many practical techniques have been devised to build syntax-directed parsers for context-free (CF) languages. Most of these are based on the pushdown automaton (PDA) model and can be classified as deterministic or non-deterministic.

The deterministic techniques [AU chapter 5] have generally been thoroughly studied and optimized, and yield very efficient parsers. Unfortunately they are necessarily restricted to some subset of the unambiguous grammars.

Here a non-deterministic technique is not one that produces a non-deterministic PDA (NPDA) in the automata theoretic sense; rather it is one that produces an algorithm which, for a given input, simulates all the possible computations of some NPDA. These techniques apply to any CF grammar and may be classified as backtrack and parallel [AU chapters 4.1 and 4.2].

Backtrack parsers are extremely inefficient. Parallel techniques yield very interesting theoretical results on random access machines (at worst time $O(n^3)$, linear time on large classes of grammars). Their generality makes them useful design and research tools for the rapid production of parsers without grammatical constraints [Ir,BPS2]. But despite the efforts made to improve their efficiency [BPS1,Lg], they lag far behind the deterministic techniques in actual computer implementations.

The purpose of this paper is twofold: firstly to unify into one theoretical framework the existing techniques for construction, optimization and evaluation of parallel non-deterministic parsers; secondly to show how the theoretical constructs obtained can be actually implemented, and used to transform well known and optimized deterministic parser generating techniques into parallel non-deterministic ones of the same efficiency, but applicable to all CF grammars.

We shall consider exclusively Earley-type techniques [Ea] which are linear on large classes of grammars. Tabular techniques such as [Yo,Ka] can work in time $O(n^{2,81})$ instead of $O(n^3)$ [Va], but they always require this amount of time. The rest of the paper is divided into seven sections devoted to basic definitions, description of the algorithm, its correctness, its implementation and complexity, efficiency considerations, local determinism and practical application to the construction of CF parsers.

1. Definitions, Notations and Conventions

For a finite alphabet Σ, we denote by Σ^* the free monoïd of words (or strings) over Σ, and by ε_Σ or simply ε the empty word in Σ^* . $\Sigma^+ = \Sigma^* - \{\varepsilon\}$.
$\forall x \in \Sigma^*$, $\forall i,j \in N$, $|x|$ is the length of x, x_i is the i^{th} letter of x, and $x_{i:j}$ is the substring of x from the $(i+1)^{st}$ letter to the j^{th} one. For any set $\mathcal{E}$, $|\mathcal{E}|$ denotes the cardinality of $\mathcal{E}$, and $2^{\mathcal{E}}$ the set of subsets of $\mathcal{E}$.

A *pushdown transducer* (PDT) is an 8-tuple $\mathcal{T}=(Q,\Sigma,\Delta,\Pi,\delta,\overset{o}{q},\$,F)$

where Q is a finite set of states,
Σ,Δ,Π are finite alphabets of respectively input, pushdown and output symbols,
$\overset{o}{q} \in Q$ is the initial state,
$F \subset Q$ is the set of final states,
$\$ \in \Delta$ is the pushdown stack bottom marker,
δ is the transition-output mapping
$$Q \times (\Delta\cup\{\varepsilon\}) \times (\Sigma\cup\{\varepsilon\}) \to 2^{Q\times(\Delta\cup\{\varepsilon\})\times\Pi^*} .$$

$\mathcal{T}$ is *deterministic* iff $\forall(p,A,a)\in Q\times\Delta\times\Sigma$ $|\delta(p,A,a)\cup\delta(p,\varepsilon,a)\cup\delta(p,A,\varepsilon)\cup\delta(p,\varepsilon,\varepsilon)| \le 1$.
Otherwise $\mathcal{T}$ is *non-deterministic*.

A *configuration* of $\mathcal{T}$ is a 4-tuple $\rho=(p,\alpha,x,u)\in Q\times\$\Delta^*\times\Sigma^*\times\Pi^*$ where α is the pushdown stack content with its top on the right, x is the unread input and u is the already emitted output.

A *transition* τ of $\mathcal{T}$ is an element of δ and is denoted by $\tau=\delta(p,A,a)\ni(q,B,z)$. τ is *scanning* (resp. *non-scanning*) iff $a\in\Sigma$, (resp. $a = \varepsilon$).

When $\mathcal{T}$ is in a configuration $\rho=(p,\alpha B,ax,z)$ and there is a transition $\tau=\delta(p,B,a)\ni(q,C,u)$, then $\mathcal{T}$ can *move* to a new configuration $\rho'=(q,\alpha C,x,zu)$. (Note that here, B,C, a and u may be ε and that *the stack grows toward the right*). Such a move is denoted $\rho \underset{\tau}{\vdash} \rho'$ or simply $\rho \vdash \rho'$.

We write $\rho \overset{n}{\vdash} \rho'$ iff ρ' can be computed from ρ in at most n moves $(n\ge0)$, $\rho \overset{*}{\vdash} \rho'$ iff $\exists\ n \ge 0$ such that $\rho \overset{n}{\vdash} \rho'$, and $\rho \overset{+}{\vdash} \rho'$ iff $\rho \overset{*}{\vdash} \rho'$ and $\rho \ne \rho'$. Such sequences of moves are called *computations* of $\mathcal{T}$.

Similar transitive closure notations will be assumed for all other relations defined in this paper.

For a given input x, the *initial configuration* of $\mathcal{T}$ is $\overset{o}{\rho}=(\overset{o}{q},\$,x,\varepsilon)$.
A *final configuration* of $\mathcal{T}$ is one of the form $\rho_f=(f,\$,\varepsilon,w)$ where $f\in F$. When $\overset{o}{\rho} \overset{*}{\vdash} \rho_f$ we say that this is a computation *accepting* the input x ; then w is the *translation* of x in that accepting computation of $\mathcal{T}$. $L(\mathcal{T})$ denotes the set of strings in Σ^* acceptable by $\mathcal{T}$, and for $x\in L(\mathcal{T})$, $\mathcal{T}(x)$ denotes the set of translations of x by $\mathcal{T}$.
If $x\notin L(\mathcal{T})$ then $\mathcal{T}(x) = \emptyset$. One can prove that if $\mathcal{T}$ is deterministic then $|\mathcal{T}(x)|\le1$.

Given $\mathcal{T}$, we will define an algorithm $\mathcal{G}$ which simulates the computations of $\mathcal{T}$. When $x\in\Sigma^*$ is input to $\mathcal{G}$, the output of $\mathcal{G}$ is a *context-free* (CF) *grammar* $\mathcal{G}(x) = (\Gamma,\Pi,S,P)$

where Γ and Π are finite non-terminal and terminal alphabets (sets),

$S \in \Gamma$ is the root (or axiom),

$\mathcal{P}$ is the set of productions (or rules),

i.e. a mapping $\Gamma \overset{\mathcal{P}}{\to} 2^{(\Gamma \cup \Pi)^*}$

A *rule* is denoted $\pi = (U \to \xi)$. Given $\xi = \xi_1 U \xi_3 \in (\Gamma \cup \Pi)^*$, a *direct derivation* through rule $\pi = (U \to \xi_2)$ produces a new string $\xi' = \xi_1\xi_2\xi_3$. We write $\xi \underset{\pi}{\Longrightarrow} \xi'$, or simply $\xi \Longrightarrow \xi'$. There is a *derivation* from ξ to ξ' iff $\xi \overset{*}{\Longrightarrow} \xi'$. The language generated by $\mathcal{G}(x)$ is the set $L(\mathcal{G}(x)) = \{w \in \Pi^* \mid S \overset{*}{\Longrightarrow} w\}$.

We will use the following metavariables (possibly primed, suscripted, etc...) :

- $x \in \Sigma^*$ is the input of $\mathcal{T}$ and $\mathcal{G}$,
- $n = |x|$, - $a \in \Sigma$ (input) ,
- $y,u,v,w,z \in \Pi^*$ (output) , - $p,q,r,s,f \in Q$ (states) ,
- $A,B,C,D \in \Delta$ (stack) , - $\alpha,\beta,\gamma \in \Delta^*$ (stack) ,
- $g,i,j,k \in N$ are indexes,
- ρ and τ denote respectively configurations and transitions of $\mathcal{T}$,
- π denote rules of $\mathcal{G}(x)$,
- $S,U,V,W,Y \in \Gamma$ are non-terminal symbols of $\mathcal{G}(x)$,
  they will also be called items (S excepted) ,
- $\mathcal{S}$ denotes item sets,
- $\zeta,\xi \in (\Gamma \cup \Pi)^*$ (sentential forms in $\mathcal{G}(x)$) .

Comments :

In our definition of a PDT, we have implicitely stated that :

a) only one symbol may be pushed on the stack, and only one symbol popped from it in a single move.

b) a stack symbol that is read is always popped.

c) acceptance is done by final state and empty stack.

In addition, we will assume that the transitions in δ are such that :

d) \$ must and may only appear at the bottom of the stack.

e) the initial state $\overset{\circ}{q}$ appears only in the initial configuration $\overset{\circ}{\rho}$.

This definition of PDT's and acceptance was only chosen to simplify exposition and proofs.

2. The PDT parallel simulator

Given a PDT $\mathcal{T} = (Q,\Sigma,\Delta,\Pi,\delta,\overset{\circ}{q},\$,F)$, we can define its *parallel simulator* $\mathcal{G}$. $\mathcal{G}$ is an algorithm which uses the transition set δ of $\mathcal{T}$ to simulate in parallel all possible computations of $\mathcal{T}$ on a given input $x \in \Sigma^n$.

$\mathcal{G}$ builds successively n+1 item-sets (i.e. "state-sets" in Earley's[Ea] terminology) $\mathcal{S}_i$ while scanning the input $x = x_1 \ldots x_n$. An item in $\mathcal{S}_i$ is a pair of the form $((p,A,i),(q,B,j))$ where $p,q \in Q$, $A,B \in \Delta$ and $0 \le j \le i \le n$.

New items are built from older ones by $\mathcal{G}$ according to the transitions of $\mathcal{T}$.

The predictor, completer and scanner of Earley correspond respectively to pushing popping, and scanning transitions.

The output of $\mathcal{G}$ on input x is a set $\mathcal{P}$ of rules of a CF grammar $\mathcal{G}(x) = (\Gamma,\Pi,S,\mathcal{P})$, where $\Gamma = \{S\} \cup \mathcal{J}$ and $\mathcal{J} = \bigsqcup_{i=0}^{n} \mathcal{J}_i$ (set of all items created by by $\mathcal{G}$).

The creation by $\mathcal{G}$ of an item $U = ((p,A,i),(q,B,j))$ has the following intuitive meaning (see theorems 1 and 2):

- there are computations of $\mathcal{C}$ on the same input x that reach a configuration ρ' where the state is p, the stack top is A, the last symbol scanned is x_i.
- the next stack symbol is then B ; it was last on the top in a configuration ρ, the state being q and the last symbol scanned x_j.
- any terminal string derivable from U in $\mathcal{G}(x)$ may be output by $\mathcal{C}$ between ρ and ρ' in such a computation.

We now give a precise specification of $\mathcal{G}$ in an Algol-like notation: All *begin*'s and *end*'s are replaced by indentation. We use the additional notation $\mathring{\rho} = (\mathring{q},\$,x,\varepsilon)$, $\mathring{U} = ((\mathring{q},\$,0),(\mathring{q},\$,0))$ and $\mathring{\pi} = (\mathring{U} \rightarrow \varepsilon)$. The input is $x \in \Sigma^n$.

```
A - Initialization :
    $\mathcal{J}_0$ := $\{\mathring{U}\}$;
    $\mathcal{P}$ := $\{\mathring{\pi}\}$;

B - Iteration :
    for 0≤i≤n do

        B.1 - Construction of $\mathcal{J}_i$ :
            for every item U = ((p,A,i),(q,B,j)) in $\mathcal{J}_i$
            for every non-scanning transition $\tau$ in $\delta$
            do

                consider four cases :

                B.1.1.1. : $\tau = \delta(p,\varepsilon,\varepsilon) \ni (r,\varepsilon,z)$
                  then V := ((r,A,i),(q,B,j)) ;
                       $\mathcal{J}_i$ := $\mathcal{J}_i \cup \{V\}$;
                       $\mathcal{P}$ := $\mathcal{P} \cup \{V \rightarrow Uz\}$;

                B.1.1.2. : $\tau$  $\delta(p,\varepsilon,\varepsilon) \ni (r,C,z)$
                  then V := ((r,C,i),(p,A,i))
                       $\mathcal{J}_i$ := $\mathcal{J}_i \cup \{V\}$;
                       $\mathcal{P}$ := $\mathcal{P} \cup \{V \rightarrow z\}$
        B.1.2.1. : $\tau = \delta(p,A,\varepsilon) \ni (r,\varepsilon,z)$
          then for every item $Y = ((q,B,j),(s,D,k))$ in $\mathcal{S}_j$
               do
                    $V := ((r,B,i),(s,D,k))$;
                    $\mathcal{S}_i := \mathcal{S}_i \cup \{V\}$;
                    $P := P \cup \{V \rightarrow YUz\}$;

        B.1.2.2. : $\tau = \delta(p,A,\varepsilon) \ni (r,C,z)$
          then  $V := ((r,C,i),(q,B,j))$;
                $\mathcal{S}_i := \mathcal{S}_i \cup \{V\}$
                $P := P \cup \{V \rightarrow Uz\}$;
        Any other non-scanning transition does not apply to U;
    if i = n then go to step C - Termination ;
  B.2 - Initialization of $\mathcal{S}_{i+1}$ :
    $\mathcal{S}_{i+1} := \emptyset$ ;
    for every item $U = ((p,A,i),(q,B,j))$ in $\mathcal{S}_i$
    for every scanning transition $\tau$ in $\delta$
    do
        ...
        Proceed as in B.1., but add the new items to $\mathcal{S}_{i+1}$ instead of $\mathcal{S}_i$.
        See for example the following case :

        B.2.1.2. : $\tau = \delta(p,\varepsilon,a) \ni (r,C,z)$ with $a = x_{i+1}$
          then  $V := ((r,C,i+1),(p,A,i))$ ;
                $\mathcal{S}_{i+1} := \mathcal{S}_{i+1} \cup \{V\}$;
                $P := P \cup \{V \rightarrow z\}$;
        ...
  End of main loop indexed by i ;
C - Termination :
  for every item $U = ((f,\$,n),(\overset{\circ}{q},\$,0))$ in $\mathcal{S}_n$ such that $f \in F$
  do  $P := P \cup \{S \rightarrow U\}$;
End of $\mathcal{G}$ ;
```

Remarks :

i) The loop statement of step B.1., "for every item U... in $\mathcal{S}_i$..." concerns not only those items already existing when the loop is first entered, but also those items added to $\mathcal{S}_i$ while executing the loop.

ii) There is a similar, but more intricate, situation with the loop statement of step B.1.2.1.,"for every item Y... in $\mathcal{S}_j$...", when j=i. In this case again, Y must span the whole set $\mathcal{S}_i$, including all items created later while executing step B.1. A proper implementation is explicited in section 4. This situation and the problems it raises occur also in all grammar-based Earley-type algorithms, when handling ε-rules.

The previous remarks do not apply to step B.2. as $\mathcal{S}_i$ is already completed.

3. Correctness of the Simulator $\mathcal{G}$.

Definition :

Given two configurations $\rho = (p,\alpha,x_{i:n},u)$ and ρ' of $\mathcal{C}$ we define :

$\rho \;\Box\!\!\overset{n}{\vdash}\; \rho'$ iff there is a computation $\rho \;\Box\!\!\overset{n}{\vdash}\; \rho'$ for $n\geq 0$ and either

i) $\rho = \overset{\circ}{\rho}$

or ii) any ρ''occuring in that computation and distinct from ρ is such that $\rho'' = (r,\alpha\gamma,x_{k:n},uv)$ with $\gamma\in\Delta^{+}$.

In other words, the content of the pushdown existing in configuration ρ is not used in the computation.

Definition : $\rho \;\Box\!\!\overset{*}{\vdash}\; \rho'$ iff $\exists n\geq 0$ such that $\rho \;\Box\!\!\overset{n}{\vdash}\; \rho'$

Lemma 1 :

$\forall\, p,q\in Q \quad \alpha,\beta\in\Delta^{+} \quad x,x',x''\in\Sigma^{*} \quad u,u',u''\in\Pi^{*}$

If $(p,\alpha,x\,x',u') \overset{n}{\vdash} (q,\beta,x',u'u)$

then $(p,\alpha,x\,x'',u'') \overset{n}{\vdash} (q,\beta,x'',u''u)$

Proof : by induction on n ∎

Lemma 2 :

$\forall\, U\in\mathcal{S} \;\; \exists u\in\Pi^{*}$ such that $U \overset{*}{\Longrightarrow} u$

Proof : by induction on the number of items created before U ∎

Lemma 3 :

If $\rho \;\Box\!\!\overset{*}{\vdash}\; \rho''$, and ρ' occurs in the computation,

then $\rho \;\Box\!\!\overset{*}{\vdash}\; \rho'$

Proof : obvious from the definition of $\Box\!\!\overset{*}{\vdash}$. Note that $\rho' \;\Box\!\!\overset{*}{\vdash}\; \rho''$ may not be true ∎

Lemma 4 :

The relation $\Box\!\!\overset{*}{\vdash}$ is transitive.

Proof : obvious from the definition of $\Box\!\!\overset{*}{\vdash}$ ∎

Lemma 5 :

$\forall\, p,q\in Q \quad \alpha,\alpha',\alpha''\in\Delta^{*} \quad x,x',x''\in\Sigma^{*} \quad u,u',u''\in\Pi^{*}$

if $(p,\alpha',x\,x',u') \;\Box\!\!\overset{n}{\vdash}\; (q,\alpha'\alpha\,,x',u'u)$

then $(p,\alpha'',x\,x'',u'') \;\Box\!\!\overset{n}{\vdash}\; (q,\alpha''\alpha\,,x'',u''u)$

Proof : by induction on n, using lemma 3 ∎

The next two theorems establish a detailed, step by step, correspondance between the computations of $\mathcal{C}$ and $\mathcal{G}$, thus giving a precise meaning to the items used by $\mathcal{G}$. This insight is useful for the proof of further results on the optimization of $\mathcal{G}$. However, weaker theorems (not using $\Box\!\!\overset{*}{\vdash}$) would have been sufficient for the correctness proof (theorem 3).

Theorem 1 :

For a given input $x \in \Sigma^n$

$\forall U = ((p,A,i), (q,B,j))$ in $\mathcal{S}$

$\forall u$ in Π^* such that $U \overset{*}{\Longrightarrow} u$ in the grammar $\mathcal{G}(x)$

$\exists \rho = (q, \alpha B, x_{j:n}, y)$

$\exists \rho' = (p, \beta A, x_{i:n}, yu)$

such that $\overset{\circ}{\rho} \; \Box\!\overset{*}{\longmapsto} \rho \; \Box\!\overset{*}{\longmapsto} \rho'$

and either $A = \$$ or $\beta = \alpha B$

proof : by induction on the length g the derivation $U \overset{g}{\Longrightarrow} u$. The induction step considers the first rule used in the derivation, and studies by cases the steps of $\mathcal{G}$ that may have produced it ∎

Corollary 1 :

$\forall U = ((p,A,i), (q,B,j))$ in $\mathcal{S}$

$A = \$$ implies $B = \$$, $q = \overset{\circ}{q}$, $j = 0$.

Proof : by theorem 1 and lemma 2. In theorem 1, $\beta A = \$$ and $\rho \; \Box\!\overset{*}{\longmapsto} \rho'$ imply by definition of $\Box\!\overset{*}{\longmapsto}$ that $\rho = \overset{\circ}{\rho}$ ∎

Theorem 2 :

Let $\overset{\circ}{\rho} \vdash \rho_1 \vdash \ldots \vdash \rho_g \vdash \ldots$ with $g \geq 0$

be a computation of $\mathcal{C}$ on a given input $x \in \Sigma^n$

Let $\rho_g = (p, \beta A, x_{i:n}, u)$

Then $\exists! \; \rho_{g'} = (q, \alpha B, x_{j:n}, y)$ in the computation such that :

i) $\rho_{g'} \; \Box\!\overset{*}{\longmapsto} \rho_g$ in that computation

ii) $U = ((p,A,i), (q,B,j)) \in \mathcal{S}_i$

iii) $\exists u' \in \Pi^*$ such that $U \overset{*}{\Longrightarrow} u'$ in $\mathcal{G}(x)$ and $u = y\,u'$

iv) either $g' < g$ or $g = g' = 0$

v) either $A = \$$ or $\alpha B = \beta$

Proof : existence and unicity of $\rho_{g'}$ are proved separately. Existence is proved by induction on g. The induction step goes by cases, according to the nature of the computation move preceding ρ_g. Unicity is proved by contradiction ∎

Theorem 3 : Correctness of $\mathcal{G}$.

$\forall x \in \Sigma^* \quad \mathcal{C}(x) = L(\mathcal{G}(x))$

Proof :

Let x be in Σ^n.

$\forall u \in \Pi^*$

$u \in \mathcal{C}(x) \Leftrightarrow \exists f \in F$ such that $\overset{\circ}{\rho} \overset{*}{\longmapsto} (f, \$, \varepsilon, u)$

thus by theorems 1 and 2 :

$\Leftrightarrow \exists f \in F$ and $\exists U = ((f,\$,n), (\overset{\circ}{q},\$,0)) \in \mathcal{S}_n$ such that $U \overset{*}{\Longrightarrow} u$ in $\mathcal{G}(x)$

thus by definition of $\mathcal{G}$, step C :

$\Leftrightarrow \exists U \in \mathcal{S}$ such that $(S \to U) \in \mathcal{P}$ and $U \overset{*}{\Longrightarrow} u$ in $\mathcal{G}(x)$

$\Leftrightarrow u \in L(\mathcal{G}(x))$ ∎

4. Complexity of the algorithm $\mathcal{G}$.

We will now evaluate the complexity of $\mathcal{G}$ relative to an implementation on a random access machine. The implementation used is by no means the most efficient one. We choose it because it can be explained concisely, but still achieves the space and time bounds, and the efficiency characteristics we want to demonstrate.

P_i denotes the set of rules in P whose left part is in $\mathcal{S}_i$. μ denotes the maximum number of transitions applicable to a given configuration of $\mathcal{C}$, or equivalently to a given item of $\mathcal{G}$.

4.1. The storage structure.

a) Every item U is doubly represented by :
   - $\bar{U}$ used for the computation of the algorithm.
   - $\hat{U}$ used exclusively to represent the rules of P obtained.

b) $\hat{U}$ is a linked list of the *right-parts* of the rules having the item U as left-part. A right-part is implemented as an array of three pointers pointing to the "hatted" representation of its constituents, or possibly to NIL.

c) For an item $U = ((p,A,i), (q,B,j))$, $\bar{U}$ is a record containing p,A,i,q,B,j, a pointer to $\hat{U}$ and a pointer to the set $\mathcal{S}_{q,B,j}$ of all items having (q,B,j) as first component; $\mathcal{S}_{q,B,j}$ is implemented as a linked list of its elements.

d) The item set $\mathcal{S}_i$ is also doubly represented by :
   - A linked list $\mathcal{L}_i$ of its elements, inserted as they are produced.
   - A 5-dimensional array $\mathcal{A}_i$ indexed by Δ^2,Q^2 and the integer range [0..i]. For every item $U = ((p,A,i), (q,B,j))$ already produced, the entry $\mathcal{A}_i$ [p,A,q,B,j] points to $\bar{U}$. All other entries are NIL.

e) $\mathcal{E}_i$ is a 2-dimensional array indexed by Q and Δ such that every entry $\mathcal{E}_i$[p,A] points to the item set $\mathcal{S}_{p,A,i}$, or NIL when $\mathcal{S}_{p,A,i} = \emptyset$.

f) A linked list $\mathcal{W}_i$ solves a difficulty of step B.1.2.1. For every item $U = ((p,A,i), (q,B,j))$ produced in step B.1. for the current value of i, and for every popping and non-scanning transition $\tau = \delta(p,A,\varepsilon) \ni (r,\varepsilon,z)$, there is an element in $\mathcal{W}_i$ containing a pointer to $\bar{U}$ and a pointer to τ.

4.2. The algorithm and a preliminary evaluation.

We consider first one iteration of step B. $\mathcal{L}_i$ and $\mathcal{A}_i$ have been initialized with items produced by step B.2. of the previous iteration.

Step B.1. The items in $\mathcal{L}_i$ are processed in order.

a) *Cases* B.1.1.1. , B.1.1.2. *and* B.1.2.2.

When an item $U = ((p,A,i), (q,B,j))$ and a rule π are produced by application of a transition, we can check in constant time with the array $\mathcal{A}_i$ whether U is already in $\mathcal{L}_i$. If it is, we get from $\mathcal{A}_i$ a pointer to $\bar{U}$, hence to $\hat{U}$, and in $\hat{U}$ we insert the additional right part for the rule π.

When U is not in $\mathcal{L}_i$, $\hat{U}$ is built with the right part of π ; then $\bar{U}$ is built and inserted in the list $\mathcal{S}_{p,A,i}$ which is directly accessible througt $\mathcal{E}_i$;

we set $\mathcal{A}_i$ [p,A,q,B,j] to point to $\bar{U}$, and insert $\bar{U}$ at the end of $\mathcal{L}_i$ to be processed later.

In both cases the computation is bounded by a constant time and at least one rule is added to $\mathcal{P}$.

With a proper storage organization of the transition table, we will try to apply to every item only applicable transitions, thus avoiding any extra computation.

Thus, for the three cases considered, the computation is bounded by a time proportional to the number of rules produced.

b) <u>*Case* B.1.2.1.</u>

Let U = ((p,A,i), (q,B,j)) be the item being processed and τ the transition. For every member of $\mathcal{S}_{q,B,j}$ (accessed through the pointer in $\bar{U}$) we compute an item and a new rule, and proceed as above with the same proportionality results.

A difficulty occurs when j = i because it may be the case that not all items in $\mathcal{S}_{q,B,i}$ have yet been produced ; thus :

i) an element ($\bar{U}$,τ) is inserted in $\mathcal{W}_i$ to remember this. The insertion takes for every item U a time bounded by a constant, thus adding to the total time a term at worst proportional to $|\mathcal{S}_i|$;

ii) when a new item V = ((r,C,i), (s,D,k)) is inserted in $\mathcal{L}_i$, we check for every element ($\bar{W}$,τ) in $\mathcal{W}_i$ whether (r,C,i) is the second component of W, i.e. whether V was needed when applying τ to W. When it is the case, we perform this delayed computation, and require as above an amount of time at worst proportional to the number of rules produced.

We must however include the time spent to search $\mathcal{W}_i$. For every search this time is bounded by a constant because the size of $\mathcal{W}_i$ is bounded by $\mu|Q|^2|\Delta|^2$. Hence the total search time is at worst proportional to $|\mathcal{S}_i|$.

c) This all adds to a running time in step B.1. which is bounded by the sum of a term proportional to the number of rules produced, and a term proportional to $|\mathcal{S}_i|$.

<u>*Step* B.2.</u> First $\mathcal{W}_i$, $\mathcal{E}_i$ and $\mathcal{A}_i$ are discarded. The computation then proceeds as in B.1. : it still processes the items in $\mathcal{L}_i$, but inserts new items in $\mathcal{L}_{i+1}$, $\mathcal{A}_{i+1}$ and $\mathcal{E}_{i+1}$. The difficulty of step B.1.2.1. disappears and no array $\mathcal{W}$ is needed. In the end $\mathcal{L}_i$ is discarded.

A similar complexity analysis yields a time bound proportional to the number of rules produced.

<u>Global results</u>

Summing the above results over all iterations (and including steps A and C), we obtain a total running time bounded by $|\Gamma|T_1 + |\mathcal{P}|T_2$ where T_1 and T_2 are constants.

We note also that $\mathcal{W}_i$, $\mathcal{E}_i$, $\mathcal{A}_i$ and $\mathcal{L}_i$ are used only in iterations i-1 and i, and then discarded. Their space requirements have an upper-bound proportional to $|\mathcal{S}_i|$ for the i^{th} iteration, thus proportional to $|\mathcal{S}|$ for the whole computation. We will ignore them since rules and items require at least as much space.

4.3. <u>The complexity bounds</u>.

a) Maximum number of items

All items in $\mathcal{S}_i$ are of the form ((p,A,i), (q,B,j))
with $A,B \in \Delta$, $p,q \in Q$, $0 \le j \le i$.
Thus $|\mathcal{S}_i| \le |\Delta|^2 \ |Q|^2 \ (i+1)$ and $|\mathcal{S}| \le \sum_{i=0}^{n} |\mathcal{S}_i| = 1/2 \ (n+1)(n+2) \ |\Delta|^2 |Q|^2$

b) Maximum number of rules

Given an item U and an applicable transition τ, $\mathcal{G}$ produces exactly one rule in $\mathcal{P}$, except in cases B.1.2.1. and B.2.2.1. In those two cases (we consider the first one as an example), let U = ((p,A,i), (q,B,j)) and $\tau = \delta \ (p,A,\varepsilon) \ni (r,\varepsilon,z)$. Then the number of rules created is equal to the number of existing items of the form Y = ((q,B,j), (s,D,k)) with $0 \le k \le j$. For a given U, there are at most $(j+1) |\Delta| |Q|$ such items. Thus, with all applicable transitions, the item U produces at most $\mu \ (j+1) |\Delta| |Q|$ rules. Summing over all items, and adding $|Q|$ for the rules created in steps A and C, we obtain :

$$|\mathcal{P}| \le |Q| + \mu \ |\Delta|^3 \ |Q|^3 \ \sum_{i=0}^{n} \ \sum_{j=0}^{i} \ (j+1)$$

$$|\mathcal{P}| \le 1/6 \ (n+1)(n+2)(n+3) \ \mu \ |\Delta|^3 \ |Q|^3 + |Q|$$

c) Summary

The space bounds are quadratic for the items alone, and cubic for the rules when there is an output.

Thus the time bound is $O(n^3)$ in the general case.

Such an amount of time and space is actually required by the following PDT :

$\Sigma = \{ a \}$ $\quad \Delta = \{ A,\$ \}$ $\quad \Pi = \{ 1,2 \}$ $\quad Q = \{ \mathring{q},p,q \}$ $\quad F = \{ q \}$

$\delta(\mathring{q},\varepsilon,\varepsilon) = \{(q,A,\varepsilon)\}$ $\qquad \delta(q,A,a) = \{(q,\varepsilon,2)\}$

$\delta(q,A,\varepsilon) = \{(p,A,1)\}$ $\qquad \delta(p,\varepsilon,\varepsilon) = \{(q,A,\varepsilon)\}$

Of course, the parallel simulator has better performances on large classes of PDTs. In particular, it obviously uses linear time when simulating a deterministic PDT.

5. Efficiency of the implementation

Definitions :

$\forall U = ((\ p,A,i\ ),(\ q,B,j\ ))$

$\forall U' = ((\ p',A',i'\ ),(\ q',B',j'\ ))$

i) $U \mathrel{-\!\!\square} U'$ iff $(\ q,B,j\ ) = (\ p',A',i'\ )$

ii) $U \multimap U'$ iff $(\ U \rightarrow \xi U' \zeta\ ) \in \mathcal{P}$ for some $\xi,\zeta \in (\Gamma \cup \Pi\ )^*$

iii) $U \overset{*}{\dashv} U'$ iff $\exists U'' \in \mathcal{S}$ such that $U \overset{*}{\mathrel{-\!\!\square}} U'' \overset{*}{\multimap} U'$

Lemma 6 :

$U \overset{*}{\multimap} U'$ iff $U \overset{*}{\Longrightarrow} \xi\ U' \zeta$ in $\mathcal{G}(x)$ for some $\xi\zeta \in (\Gamma \cup \Pi\ )^*$

Interpretation :

The implementation of $\mathcal{G}$ is organized so that, for an item $U = ((\ p,A,i\ ),(\ q,B,j\ ))$, $\bar{U}$ points only to $\mathcal{S}_{q,B,j}$ and to $\hat{U}$. The only "barred" items directly accessible from $\bar{U}$ are thus in $\mathcal{S}_{q,B,j}$, i.e. the $\bar{V}$'s such that $U \mathrel{-\!\!\square} V$. Hence, by induction, since "hatted" items do not point to "barred" ones, $\bar{V}$ is accessible from $\bar{U}$ (by following pointers) in the implementation iff $U \overset{*}{\mathrel{-\!\!\square}} V$.

Similarly, $\hat{V}$ is accessible from $\hat{U}$ iff $U \overset{*}{\multimap} V$, and $\hat{V}$ is accessible from $\bar{U}$ iff $U \overset{*}{\dashv} V$.

Theorem 4 (resp.5, and 6)

$\forall U = ((\ p,A,i\ ),(\ q,B,j\ ))$

$\forall U = ((\ p',A',i'\ ),(\ q',B',j'\ ))$

If $U \overset{*}{\mathrel{-\!\!\square}} U'$ (resp. $U \overset{*}{\multimap} U'$, and $U \overset{*}{\dashv} U'$)

then $\exists \rho_1 = (\ q',\alpha'B',x_{j':n}\ ,y\ )$

$\exists \rho_2 = (\ p',\beta'A',x_{i':n}\ ,yu\ )$

$\exists \rho_3 = (\ p,\beta A,x_{i:n}\ ,yuw\ )$

such that $\mathring{\rho} \mathrel{\square\!\!\overset{*}{\vdash}} \rho_1 \mathrel{\square\!\!\overset{*}{\vdash}} \rho_2 \mathrel{\square\!\!\overset{*}{\vdash}} \rho_3$

(resp. $\mathring{\rho} \mathrel{\square\!\!\overset{*}{\vdash}} \rho_1 \mathrel{\square\!\!\overset{*}{\vdash}} \rho_2 \overset{*}{\vdash} \rho_3$ for both 5 and 6)

Proofs :

- theorem 4 : the proof goes by induction on the length g of the pointing sequence $U \overset{g}{\mathrel{-\!\!\square}} U'$, using theorem 1 and lemma 5 ■
- theorem 5 : by lemma 6, we can replace $U \overset{*}{\multimap} U'$ by $U \overset{*}{\Longrightarrow} \xi U' \zeta$; the proof is done by induction on the length of the derivation ; the induction step considers the first rule used in the derivation and studies by cases the steps of $\mathcal{G}$ that may have produced it ■
- theorem 6 : the proof follows from theorems 4 and 5, by lemma 5 ■

Interpretation

First we make the following remarks :

a) In our implementation of $\mathcal{G}$, at the beginning of iteration i of the main loop, the only directly accessible items are the "barred" ones already inserted in $\mathcal{L}_i$ at step B.2. of iteration i-1. All other "barred" or "hatted" items are accessible only indirectly through those in $\mathcal{L}_i$.

b) Theorem 1 (with lemma 2) states that, to every item U corresponds at least one possible computation of $\mathcal{C}$ simulated by $\mathcal{G}$.

c) A computation of $\mathcal{C}$ may fail to accept the input either by not reaching a final configuration after scanning the whole input, or by reaching a configuration where no transition is applicable even though the input may be uncompletely scanned.

If all computations corresponding to an item U fail before scanning x_i, then theorem 6 states that $\hat{U}$ and, therefore $\bar{U}$ are not accessible from any item in $\mathcal{L}_i$, i.e. that they are not accessible at all by the program implementing $\mathcal{G}$ after iteration i-1. Thus, with a "garbage collector", it is possible to reuse the storage formerly occupied by $\hat{U}$ and $\bar{U}$. U is a useless symbol in $\mathcal{G}(x)$ (i.e. it does not occur in any string derivable from the axiom S), and any such useless symbol of $\mathcal{G}(x)$ is similarly eliminated. Combined with lemma 2, this shows that we get in fact a *reduced* form of $\mathcal{G}(x)$ (i.e. all non-terminals are used is some terminal derivation). Another consequence is that we can reduce the storage requirements of $\mathcal{G}$ by speeding up the detection of failing computations for $\mathcal{C}$.

These properties, would not hold without the double representation of items which gives an interpretation to theorem 6. Similarly theorem 4 states, in our implementation, that the storage used by $\bar{U}$ is freed as soon as U is no more necessary for $\mathcal{G}$ to simulate computations of $\mathcal{C}$. Indeed, by theorem 1, an item U = ((p,A,i) (...)) corresponds to a symbol A pushed on the stack in each of the computations of $\mathcal{C}$ associated to U ; by theorem 4, $\bar{U}$ is freed as soon as $\mathcal{G}$ has simulated all these computations to a point where they all have popped that symbol A.

Comments on Real Implementations :

A real implementation should follow the lines we have indicated. Without detailing the improvements to be made, we may mention that :

i) arrays should be replaced by linked lists or hash-coded tables, especially when their length is bounded by a constant.

ii) for an item U = ((p,A,i),(q,B,j)), $\bar{U}$ may be implemented as a record containing only a pointer to $\hat{U}$ and a pointer to $\mathcal{S}_{q,B,j}$; $\mathcal{S}_{q,B,j}$ is then itself implemented as a record containing B and a pointer to the linked list of the items in $\mathcal{S}_{q,B,j}$.

Then, with a very pessimistic (for $\mathcal{G}$) evaluation of a fully deterministic computation, $\mathcal{G}$ takes only about 5 times as much space as $\mathcal{C}$ does. Time ratios are on the same order of magnitude.

6. Local determinism

Equivalence definitions :

i) $\forall \rho' = (p,\alpha A,x',u)$ and $\rho'' = (q,\beta B,x'',v)$
$\rho' \sim \rho''$ iff $p = q$, $A = B$ and $x' = x''$.

ii) Then, for a given input x, an equivalence class of configurations is called a *mode* and is represented by a triple (p,A,i) such that all configurations in this mode are of the form $(p,\alpha A,x_{i:n},u)$ for some $\alpha \in \Delta^*$ and $u \in \Pi^*$. Note that *an item is a pair of modes*.

iii) Two items U and V are said *equivalent* ($U \sim V$) iff they have the same first mode. Again, an equivalence class can be represented by the common first mode.

Definitions about local determinism :

iv) We say that a configuration, or an item, is *uat* iff it has a unique applicable transition.

v) For a given input x, a configuration $\rho = (p,\alpha A,x_{i:n},u)$ is *locally deterministic* (ld) iff in every computation that scan x_i there occurs a configuration ρ' such that $\rho' \sim \rho$, and ρ is *uat*.

vi) The mode (p,A,i) of ρ is ld iff ρ is ld.

vii) An item is ld iff its first mode is ld.

viii) A computation $\rho \overset{*}{\vdash} \rho'$ is ld iff every configuration occurring in it is ld (except that ρ' may not be *uat*).

Remark : Equivalent transitions, or items, have the same set of applicable transitions.

Lemma 7 :If an item is ld, then it is *uat*.

Interpretation

Intuitively, these definitions mean that, for a given input x, $\mathscr{C}$ has to perform identical deterministic moves in the locally deterministic parts of all its possible computations. $\mathscr{G}$ simulates all these computations in parallel ; when they are ld, $\mathscr{G}$ has only the unique common deterministic computation to simulate. This deterministic computation can be done more efficiently by $\mathscr{C}$.

Thus it is essential to know when $\mathscr{G}$ is simulating ld computations,i.e. when the created items are ld. We do not have a simple characterization of all ld items, but the following sufficient conditions will do for most practical purposes.

Theorem 7 :

If, at the end of iteration (i-1) of the main loop of $\mathscr{G}$, the item-set $\mathscr{S}_i$ has been initialized with equivalent *uat* items, then the items in $\mathscr{S}_i$ are ld.

Proof : using theorem 1 and 2, we show that the common first mode of items in $\mathscr{S}_i$ is the unique equivalence class of the configurations following the scanning of x_i in every computation of $\mathscr{C}$ ∎

Theorem 8 :

Let $\mathcal{S}'$ be an equivalence class of ld items. Let $\mathcal{S}''$ be the set of items obtained by applying the unique applicable transition to the items in $\mathcal{S}'$. If all items in $\mathcal{S}''$ are equivalent and *uat*, then they are ld.

Proof : by theorem 1, there is a class $\mathcal{C}$, corresponding to $\mathcal{S}'$, of equivalent ld configurations occuring in every computation of $\mathcal{C}$. We then show by contradiction, using theorem 2, that the common first mode of the items in $\mathcal{S}''$ is the equivalence class of all configurations following immediatly those in $\mathcal{C}$ ∎

Thus we can modify $\mathcal{G}$ in such a way that it generally detects ld items. In such a case, $\mathcal{G}$ passes control to $\mathcal{C}$. $\mathcal{C}$ performs the computation, *efficiently because deterministically*, until it encounters a non-deterministic "situation". Then $\mathcal{C}$ gives control and results (output) back to $\mathcal{G}$.

This construction will be detailed in a further paper.

Local determinism can be put to other uses, such as the early detection and localisation of ambiguities (i.e. multiple translations for the same input). This may be of interest when parsing syntactically extensible programming languages.

7. Practical Applications

This work applies to the construction of very efficient parallel non-deterministic parsers for all context-free languages.

If we consider the currently available techniques for the construction of deterministic parsers. We notice that, when they fail to work, it is usually because of the occurrence of a conflict in some precedence table, or of some inadequate state in an automaton [DR], etc... Rather than viewing these occurrences as causes for failure, we can consider that we have obtained a parser that is non-deterministic in its "inadequate" parts. Whence a method for building an efficient general context-free parser :

- From the given grammar build a pushdown parser $\mathcal{C}$, using some well known and efficient parsing technique Θ .
- If some "inadequate" feature occurs, consider it as non-determinism.
- Construct a parallel simulator $\mathcal{G}$ for the non-deterministic parser considered for example as a NPDT that translate the input into its canonical parse (i.e. the sequence of the indexes of the rules used to parse the input from left to right).

In most practical cases (programming languages), there is little non-determinism in $\mathcal{C}$ and local determinism plays an important role. On the often large parts of the computation that are locally deterministic, $\mathcal{G}$ is as efficient as any deterministic parser obtaind with the technique Θ . On the rest of the computation, the efficiency remains within good limits if there is not "too much non-determinism" encountered. Local determinism insures also an early detection of ambiguities. In addition, all tools (such as error recovery devices) developped around the technique Θ should be adaptable to $\mathcal{G}$.

Aknowledgements : Part of this research has been conducted at the Center for Research in Computing Technology, Harvard University.

References

[AU] A H O, A.V., and ULLMAN, J.D., *The theory of parsing, translation, and compiling.* Prentice-Hall Inc. Englewood Cliffs, N.J., (1972).

[BPS1] BOUCKAERT, M. PIROTTE, A.,and SNELLING, M., Efficient parsing algorithms for general context-free grammars", *Information Sciences* (to appear).

[BPS2] ————————, "SOFT : a tool for writing software", IEE Conference on Software Engineering for Telecommunication Switching Systems, Colchester (1973), Proceedings published by IEE (London).

[DR] DEREMER, F.L., *Practical translators for LR (k) languages,* Ph.D. Thesis, MIT , Cambridge, Mass. (1969).

[Ea] EARLEY,J. *An efficient context-free parsing algorithm,* Thesis, Dept. of Computer Science, Carnegie-Mellon University (1968).

[Ir] IRONS, E.T. "Experiment with an extensible language", *Comm. ACM 13, 1,* Jan. 1970, pp. 31-40.

[Ka] KASAMI, J. *An efficient recognition and syntax analysis algorithm for context-free languages,* Report of University of Hawaii, (1965).

[Lg] LANG, B. "Parallel non-deterministic bottom-up parsing", Abstract published in the Proceeding of the Int. Symp. on Extensible Languages, Grenoble (1971) *SIGPLAN Notices 6-12,* December 1971.

[Va] VALIANT, L.G. "*General context-free recognition in less than cubic time*", Report of Carnegie-Mellon University (1974).

[Yo] YOUNGER,D.H. Recognition and parsing of context-free languages in time n^3. *Inf. and Control,10* 189-208 (1967).