JFP **25**, e10, 39 pages, 2015. © Cambridge University Press 2015
doi:10.1017/S0956796815000131

Efficient parallel and incremental parsing of practical context-free languages

JEAN-PHILIPPE BERNARDY and KOEN CLAESSEN
Chalmers University of Technology & University of Gothenburg, Sweden
(*e-mail:* bernardy@chalmers.se, koen@chalmers.se)

Abstract

We present a divide-and-conquer algorithm for parsing context-free languages efficiently. Our algorithm is an instance of Valiant's (1975; General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.* **10**(2), 308–314), who reduced the problem of parsing to matrix multiplications. We show that, while the conquer step of Valiant's is $O(n^3)$, it improves to $O(\log^2 n)$ under certain conditions satisfied by many useful inputs that occur in practice, and if one uses a sparse representation of matrices. The improvement happens because the multiplications involve an overwhelming majority of empty matrices. This result is relevant to modern computing: divide-and-conquer algorithms with a polylogarithmic conquer step can be parallelized relatively easily.

1 Introduction

Recent years have seen the rise of parallel computer architectures for the masses. Multicore CPUs and GPUs are legion. One would expect functional programs to be a perfect match for these architectures. Indeed, thanks to the absence of side-effects, functional programs are conceptually easy to parallelize. However, functional programmers have traditionally relied heavily on lists as the data-structure of choice. This tradition hinders the adaptation of functional programs to the age of parallelism. Indeed, the very linear structure of lists imposes a sequential treatment of them. In an eloquent 2009 ICFP invited talk, Guy Steele harangued the functional programming crowds to stop using lists and instead use sequences represented as balanced trees. If a computation over them follows the divide-and-conquer skeleton, and uses an associative operator to cheaply combine intermediate results at each node, their fractal structure allows to take advantage of *many processors in parallel*; in fact as many as there are leaves in the tree.

An additional benefit of the structure is its ability to support *incremental computation*. That is, if one remembers the intermediate results of the computation for each node, then after changing a single leaf in the tree, it suffices to recompute the results for the nodes which are on the path from the root to the given leaf. If the tree is balanced, this means that one only has to run the associative operator a few times to update the result after a single incremental change.

Some problems are naturally solved by divide-and-conquer algorithms. This is the case for example for vector operations, which treat each element independently of the others. However, many problems require creativity to discover efficient divide-and-conquer solutions. This is the case for the problem of parsing context-free (CF) languages.

Our aim in this paper is to find a formulation of parsing CF languages which has a divide-and-conquer structure that can lead to both (1) a performance speed-up when implemented in parallel, and (2) a performance gain when doing incremental parsing. Even though many real-life grammars (for example for programming languages) are not CF, most of these grammars can be sufficiently well approximated by a CF grammar (for example by using a pre-processor of the input or a post-processor of the output) for the results in this paper to still be relevant.

Valiant (1975) discovered a divide-and-conquer algorithm for CF recognition. However, given Valiant's assumptions, the cost of the conquer step is cubic. This means that the conquer step dominates the cost of the algorithm: what we gain by running sub-problems in parallel is dwarfed by the cost of what we must run sequentially. Therefore the divide-and-conquer structure does not yield a significant performance benefit.

In this paper we show that one can carefully implement Valiant's algorithm, so that its conquer step is polylogarithmic ($O(\log^2 n)$) on common input patterns, yielding a strict theoretical speed-up of a parallel implementation over a sequential implementation.

Our argument articulates as follows. We first observe that one can use a sparse matrix representation of charts to make Valiant's algorithm perform efficiently on hierarchically organized inputs. While hierarchical inputs are common, simple iteration is perhaps even more common in programming languages. The second part of our argument is to show how one can force an artificial hierarchical handling of iteration.

Outline. In Section 2 we review the divide-and-conquer skeleton, how it adequately abstracts incremental and parallel computation, and its relationship with sequence homomorphisms. In Section 3 we review chart-based CF parsing, and derive Valiant's algorithm from its specification. In Section 4 we characterize a sub-class of CF languages. We argue that this class corresponds to hierarchically organized inputs. We proceed to show that for such languages, the average complexity of the conquer step of the parsing algorithm is $O(log^2 n)$. In Section 5, we describe an extension of CF grammars. This extension remains parsable with Valiant's algorithm. Using this extension, we show how to reduce parse iteration (Kleene's closure) hierarchically. We conclude with a discussion of our results.

2 The divide-and-conquer skeleton

Our aim is to construct a parallel and incremental parsing algorithm. To do so, we need a sufficiently general model of incremental and parallel computation, and choose the *divide-and-conquer skeleton*. We further assume that the input is provided as a sequence of input symbols (taken in a finite alphabet Σ) — strings. Our definition of this skeleton relies on the theory of sequences as initial algebras developed by Bird (1986), which we review here.

Definition 1
A sequence-algebra is a triplet of:

- A carrier type a
- A constant nil of type a
- A ternary operation bin of type $a \to \Sigma \to a \to a$.

which satisfies the associative law:

$$bin\ a\ x\ (bin\ b\ y\ c) = bin\ (bin\ a\ x\ b)\ y\ c \tag{1}$$

One way to define the type of sequences of Σ, written Seq, is as the initial sequence-algebra. Concretely, one naive way to implement Seq is as a list. In actual implementations, sequences will be represented by more complex data structures; perhaps trees featuring dynamic re-balancing such as finger trees (Hinze & Paterson 2006). The associative law (1) guarantees that re-balancing is not observable by user code. We will write Nil and Bin (with capitals) for the operations of the initial sequence-algebra:

$$\begin{aligned} &Nil : Seq \\ &Bin : Seq \to \Sigma \to Seq \to Seq \end{aligned}$$

Assume a function $f : Seq \to A$. The construction of a divide-and-conquer algorithm that computes f can be specified as finding a sequence-algebra $\mathscr{A} = (A, nil_A, bin_A)$ such that f is an homomorphism between Seq and $\mathscr{A}$.

That is, we need a carrier type A, a constant nil_A and a function bin_A such that Equation (1) is satisfied and

$$\begin{aligned} nil_A &= f\ Nil \\ bin_A\,(f\,l)\,x\,(f\,r) &= f\ (Bin\ l\ x\ r) \end{aligned}$$

Given such an algebra $\mathscr{A}$ and a sequence t, one can compute $f\,t$ as the catamorphism of $\mathscr{A}$ applied to t.

Assuming an implementation of Seq as trees, one can obtain a parallel algorithm by spawning a new thread of execution at each node. In an actual implementation, the shape of the tree structure will be dictated by the architecture of the computer running the code. The implementation is free to choose the structure: any choice yields the same result, as guaranteed by the associative law (1).

An incremental algorithm can be obtained by caching the intermediate results in each node. An update at a leaf of the tree needs to run the bin function d times, where d is the depth of the leaf in the tree.

In all the cases considered in the remainder, we never bother to prove the associative law for the bin function that we construct. Indeed, because we consider only values which are generated by the sequence-homomorphism f, associativity holds automatically. In other words, the fact that bin adequately implements f implies associativity.

Lemma 1
Given $f : Seq \to A$, and $bin : A \to \Sigma \to A \to A$ such that

$$bin\ (f\ l)\ x\ (f\ r) = f\ (Bin\ l\ x\ r)$$

then $(A', f\ Nil, bin)$ is a sequence-algebra, where A' is the image of Seq under f.

Proof
The missing associative law is obtained as follows:

$$\begin{aligned}
& bin\ a\ x\ (bin\ b\ y\ c) \\
= \ & \{\text{-by } A' \text{ being the image of } f \text{ -}\} \\
& bin\ (f\ s)\ x\ (bin\ (f\ t)\ y\ (f\ u)) \\
= \ & \{\text{-by assumption on } bin \text{ -}\} \\
& f\ (Bin\ s\ x\ (Bin\ t\ y\ u) \\
= \ & \{\text{-by } Seq \text{ being a sequence-algebra -}\} \\
& f\ (Bin\ (Bin\ s\ x\ t)\ y\ u) \\
= \ & \{\text{-by assumption on } bin \text{ -}\} \\
& bin\ (bin\ (f\ s)\ x\ (f\ t))\ y\ (f\ u) \\
= \ & \{\text{-by definition of } a, b, c \text{ -}\} \\
& bin\ (bin\ a\ x\ b)\ y\ c
\end{aligned}$$

□

2.1 Performance

Crucially, in order for parallelization or incrementalization to yield benefits in terms of performance, the cost of running *bin* must be at most quasi-linear (i.e. $O(n \log^k n)$). Let us analyze why by using the following standard result:

Theorem 1 (*Master Theorem, Cormen et al. 2001*)
Assume a function T_n constrained by the recurrence

$$T_n = aT_{\frac{n}{b}} + g(n)$$

(Such an equation will typically come from a divide-and-conquer algorithm, where a is the number of sub-problems at each recursive step, n/b is the size of each sub-problem, and $g(n)$ is the running time of dividing up the problem space into a parts, and combining the sub-results together.)

If we let $e = \log_b a$ and $g(n) = O(n^c \log^d n)$, then

$$\begin{aligned}
T_n &= O(n^e) && \text{if } c < e \\
T_n &= O(n^c \log^{d+1} n) && \text{if } c = e \\
T_n &= O(n^c \log^d n) && \text{if } c > e
\end{aligned}$$

In our description of sequence homomorphisms we have assumed $b = 2$. In the case of a sequential algorithm, $a = 2$, but in presence of parallelism or incrementality, $a = 1$, because both sub-problems can be run in parallel or because the result of one sub-problem is already computed. In sum $e = 1$ corresponds to the sequential case, while $e = 0$ corresponds to a parallel or incremental case. We can then compute the asymptotic behavior of T_n for each case:

	$e = 1$ (sequential)	$e = 0$ (parallel)	speedup factor
$c = 0$	n	$\log^{d+1} n$	$\frac{n}{log^{d+1} n}$
$0 < c < 1$	n	$n^c \log^d n$	$\frac{n^{1-c}}{log^d n}$
$c = 1$	$n \log^{d+1} n$	$n \log^d n$	$\log n$
$c > 1$	$n^c \log^d n$	$n^c \log^d n$	1

That is, the fastest the conquer step, the bigger gains for parallelization or incrementalization. In particular, a conquer step running in $\Omega(n^{1+\varepsilon})$ yields no asymptotic gain.

2.2 *Summary*

In sum, using a divide-and-conquer skeleton to construct an incremental as well as a parallel algorithm that computes f means finding functions *bin* and *nil* such that:

- $nil = f\ Nil$
- $bin\ (f\ l)\ x\ (f\ r) = f\ (Bin\ l\ x\ r)$
- The complexity of *bin* is quasi-linear (and if possible better)

3 Context free parsing

In this section we review the basics of CF parsing, give a specification of parsing in terms of transitive closure, and review the Cocke–Younger–Kasami algorithm (CYK) and Valiant's algorithm.

3.1 *Conventions and notations*

We assume a CF grammar $\mathscr{G}$, given by a quadruple (Σ, N, P, S), where Σ is a finite set of terminals, N is a finite set of non-terminals of which S is the starting symbol, and P a finite set of production rules.

We furthermore assume an input $w \in \Sigma^*$ — a sequence of terminal symbols of length $|w|$. The input symbol at position i is denoted $w[i]$. A sub-string of w starting at position i (included) and ending at position j (excluded), is denoted $w[i..j]$. Metasyntactic variables standing for arbitrary strings of terminals will have the form $w_1, w_2, \ldots$. The letters $A, B, C, \ldots$, stand for arbitrary non-terminals, while $\alpha, \beta, \ldots$ stand for arbitrary strings (elements of $(\Sigma \cup N)^*$) and t stands for a terminal symbol. Each production rule associates a non-terminal with a string it can generate. We write $A ::= \alpha$ for A generates α.

Definition 2 ($\longrightarrow$)
$\alpha A \beta \longrightarrow \alpha\gamma\beta$ iff. $(A ::= \gamma) \in P$

Definition 3 ($\overset{*}{\longrightarrow}$)
The reflexive and transitive closure of the $\longrightarrow$ relation is written $\overset{*}{\longrightarrow}$.

Definition 4 ($\mathscr{L}$)
The input string w belongs to the language $\mathscr{L}$ iff. $S \overset{*}{\longrightarrow} w$. We say that $\mathscr{G}$ generates $\mathscr{L}$.

3.1.1 Chomsky normal form

The simplest implementation of CYK and Valiant algorithms takes as input a grammar Chomsky Normal Form (Chomsky 1959). In Chomsky Normal Form, hereafter abbreviated CNF, the production rules are restricted to one of the following forms:

$$\begin{aligned} S &::= \epsilon && \text{(nullary)} \\ A &::= t && \text{(unary)} \\ A_0 &::= A_1 A_2 && \text{(binary)} \end{aligned}$$

Any CF grammar $\mathscr{G}$ generating a language $\mathscr{L}$ can be converted to a grammar $\mathscr{G}'$ in CNF defining the same language $\mathscr{L}$. This conversion preserves many useful properties of the input grammar. In particular:

- The size of the grammar does not increase too much: $|\mathscr{G}'| \leqslant |\mathscr{G}|^2$.
- The parse-trees generated by $\mathscr{G}'$ are a binarized version of the parse tree generated from $\mathscr{G}$. This means that from a $\mathscr{G}'$-parse tree one can easily recover a $\mathscr{G}$-parse tree, modulo the following *caveat*.
- The conversion discards unit-rule cycles (such as $A_0 ::= A_1; \quad A_1 ::= A_0$). This is good: such cycles generate infinitely many (equivalent) parse trees, which the user generally wants to ignore anyway.

Hence we will assume from now on a grammar provided in CNF. Moreover, because it is easy to handle the empty string specially, we conventionally exclude it from the input language and thus exclude the nullary rule $S ::= \epsilon$ from the set of productions P. In sum, we assume that P contains only unary and binary production rules. The interested reader is directed to Lange & Leiß (2009) for a pedagogical account of the process of reduction to CNF.

Given a grammar specified as above, the problem of parsing is reduced to finding a binary tree such that each leaf corresponds to a symbol of the input and a suitable unary rule; and each branch corresponds to a suitable binary rule. Essentially, parsing is equivalent to considering all possible bracketings of the input, and verify that they form a valid parse.

3.2 Charts as matrices, parsing as closure

In this section we show how to specify parsing as an equation on matrices. We start by abstracting away from the grammar, via a ring-like structure operating over sets of non-terminals. We define the operations $0, +, \cdot$ and σ as follows.

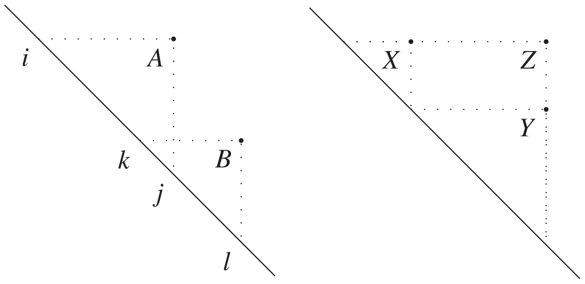


Fig. 1. Example charts. In each chart a point at position (x, y) corresponds to a sub-string starting at x and ending at y. The first parameter x grows downwards and the second one y rightwards. The input string w is represented by the diagonal line. Dots in the upper-right part represent non-terminals. The first chart witnesses $A \overset{*}{\longrightarrow} w[i..j]$ and $B \overset{*}{\longrightarrow} w[k..l]$. An instance of the rule $Z ::= X Y$ is illustrated on the second chart.

Definition 5 ($0, +, \cdot$ *on* $\mathscr{P}(N)$)

$$\begin{aligned} 0 &= \varnothing \\ x + y &= x \cup y \\ x \cdot y &= \{A \mid A_0 \in x, A_1 \in y, A ::= A_0 A_1 \in P\} \\ \sigma_i &= \{A \mid A ::= w[i] \in P\} \end{aligned}$$

The $(\cdot)$ operation characterizes the binary production rules of the grammar, while σ captures the unary ones. (Remember that $w[i]$ denotes the ith input of the string we are parsing, meaning that the input w is fixed for the definition of σ_i.)

We have the following properties: (0,+) forms a commutative monoid (the usual monoid of sets with union); 0 is absorbing for $(\cdot)$; and $(\cdot)$ distributes over $(+)$. However, and crucially, $(\cdot)$ is *not* associative.

$$\begin{gathered} x + 0 = x \\ 0 + x = x \\ (x + y) + z = x + (y + z) \\ x \cdot (y + z) = x \cdot y + x \cdot z \\ x \cdot 0 = 0 \\ 0 \cdot x = 0 \end{gathered}$$

We will then use a matrix of sets of non-terminals C to record which non-terminals can generate a given sub-string. The intention is that $A \in C_{ij}$ iff. $A \overset{*}{\longrightarrow} w[i..j]$. See Figure 1 for an illustration. In parsing terminology, a structure containing intermediate parse results is called a chart. We call the set of charts $\mathscr{C}$.

Definition 6
$\mathscr{C} = \mathscr{P}(N)^{\mathbb{N} \times \mathbb{N}}$

We lift the operations $0, +, \cdot$ from sets of non-terminals to matrices of sets of non-terminals, in the usual manner.

Definition 7 (0, +, · *on* $\mathscr{C}$)

$$\begin{aligned} 0_{ij} &= 0 \\ (A+B)_{ij} &= A_{ij} + B_{ij} \\ (A \cdot B)_{ij} &= \sum_k A_{ik} \cdot B_{kj} \end{aligned}$$

As expected, all the properties carry over to matrices; and associativity is still lacking. The operation σ is used to compute an upper diagonal matrix corresponding to the input w, as follows.

Definition 8 (*Initial matrix*)
The initial matrix, written $I(w)$, is a square matrix of dimension $|w| + 1$[1] such that

$$\begin{aligned} I(w)_{i,i+1} &= \sigma_i \\ I(w)_{i,j} &= 0 && \text{if } j \neq i+1 \end{aligned}$$

Let $W^{(1)} = I(w)$. Note that $W^{(1)}_{i,i+1} = \sigma_i$ contains all the non-terminals which can generate the sub-string $w[i..i+1]$. Let $W^{(2)} = W^{(1)}W^{(1)} + I(w)$. It is easy to see that $W^{(2)}_{i,i+2} = \sigma_i \cdot \sigma_{i+1}$, hence it contains all the non-terminals which can generate the sub-string $w[i..i+2]$. Consider now $W^{(3)} = W^{(2)} \cdot W^{(2)} + I(w)$. We have

$$\begin{aligned} W^{(3)}_{i,i+3} &= W^{(2)}_{i,i+2} \cdot W^{(2)}_{i+2,i+3} + W^{(2)}_{i,i+1} \cdot W^{(2)}_{i+1,i+3} \\ &= (\sigma_i \cdot \sigma_{i+1}) \cdot \sigma_{i+2} + \sigma_i \cdot (\sigma_{i+1} \cdot \sigma_{i+2}) \end{aligned}$$

and

$$\begin{aligned} W^{(3)}_{i,i+4} &= W^{(2)}_{i,i+2} \cdot W^{(2)}_{i+2,i+4} \\ &= (\sigma_i \cdot \sigma_{i+1}) \cdot (\sigma_{i+2} \cdot \sigma_{i+3}) \end{aligned}$$

Hence $W^{(3)}$ contains all possible parsing of three symbols, and all *balanced* parsings of four symbols. By iterating n times, one obtains all the parsings of n symbols. (However, as a hint to our method for efficient parsing, it suffices to repeat the process $\log n + 1$ times to obtain all balanced parsings of n symbols.)

Definition 9 (*Transitive closure*)
If it exists, the transitive closure of a matrix W, written W^+, is the smallest matrix C such that

$$C = C \cdot C + W$$

From the above, we can deduce that $C \supseteq C \cdot C + I(w)$. Consequently, every possible bracketing of the products $I(w) \cdot \cdots \cdot I(w)$ is contained in C, and thus all possible parsings of $w[i..j]$ are found in C_{ij}. Conversely, because C is the smallest matrix

[1] We need one more element in the dimension because we want to talk about points in *between* elements of the input.

satisfying the property, if C_{ij} contains a non-terminal then it must generate $w[i..j]$. Algorithms which parse by computing a chart are known as chart parsers.

The above procedure specifies a recognizer: by constructing $I(w)^+$ one finds if w is parsable, but not the corresponding parse tree. Even though we focus on the recognition problem in this paper, it is straightforward to specify parsers by using matrices of parse trees instead of non-terminals, and adapting the operations accordingly, as we have done in our implementation on top of the parser generator tool BNFC (Forsberg & Ranta 2012).

In order to construct an efficient parallel parser, we must construct a sequence-homomorphism from input strings to charts. Thanks to Lemma 1, it suffices to find an operator *bin* which combines two charts $I(w_1)^+$, $I(w_2)^+$ and a terminal t into a chart $I(w_1 t w_2)^+$.

3.3 Cocke–Younger–Kasami

A straightforward manner to turn the above specification into an algorithm is as follows. Let us first remark that the product of two upper triangular matrices is upper triangular. Hence the closure of an upper triangular matrix must also be upper triangular. Hence, in every chart ever considered, every element at the diagonal and below it equals zero. The output of any algorithm computing the closure of $I(w)$ must satisfy the equation $C = C \cdot C + I(w)$. Expanding it index-wise yields:

$$C_{ij} = I(w)_{ij} + \sum_{k=0}^{n} C_{ik} \cdot C_{kj}$$

Because C is upper triangular, $C_{ik} = 0$ if $k \leqslant i$ and $C_{kj} = 0$ if $k > j$. Hence the sum can be limited to the interval $[i+1..j]$

$$C_{ij} = I(w)_{ij} + \sum_{k=i+1}^{j} C_{ik} \cdot C_{kj}$$

Observing that the summand equals 0 when $j = i + 1$ and $I(w)_{ij} = 0$ otherwise, we distinguish on that condition and obtain the two equations:

$$C_{i,i+1} = \sigma_i \tag{2}$$

$$C_{ij} = \sum_{k=i+1}^{j} C_{ik} \cdot C_{kj} \qquad \text{if } j > i+1 \tag{3}$$

These equations give a method to compute C_{ij} by induction on $j - i$. The equations can be re-interpreted in term of parses and non-terminals as follows. Either

- we parse a single token w_i, and the non-terminals generating it are given directly from unary rules, or
- we parse a longer string. In this case we split it at any intermediate position k, and combine the intermediate results (found in C_{ik} and C_{kj}) in every possible way according to binary rules.

By applying the above rules naively, the computation time is exponential in the length of the input; however by memoizing each intermediate result (for example by using lazy dynamic programming; Allison 1992) the complexity is merely cubic. The resulting dynamic programming algorithm is known as CYK, owing to its independent discoverers: Cocke (1969), Kasami (1965) and Younger (1967).

In the CYK algorithm, any element of the chart is computed only on the basis of elements strictly closer to the diagonal. Hence it can be used to program the combination step of a divide-and-conquer algorithm. The combination of two charts and a terminal $C = bin(A, w[i], B)$, is defined as follows. Elements of C in the upper-left corner are copied from A; elements of the bottom-right corner are copied from B; and elements from the top-right corner are computed using σ_i and the CYK formula (Equation 3).

Even though we have produced a sequence homomorphism, it is not suitable for parallelization: its performance is not good enough. Indeed, the above operator has to compute a matrix of size $n \times m$ from two matrices of size $n \times n$ and $m \times m$, and computing each element takes time linear in $n + m$, which equals the size of the input. The complexity of *bin* is therefore cubic, and as we have seen in Section 2, there is no asymptotic gain to parallelization.

3.4 Valiant

A more subtle way to turn the transitive closure specification into an algorithm is the following. Our task is to find a function $\cdot^+$ which maps a matrix W to its transitive closure $C = W^+$, which implies $C = C \cdot C + W$. As above, we do so by refinement of the definition of transitive closure, but we adopt a divide and conquer approach rather than iterating index-wise.

If W is a 1 by 1 matrix, $W = 0$, and the solution is $C = 0$. Otherwise, let us divide W and C in blocks as follows (for efficiency the blocks should be roughly of the same size; but the reasoning holds for any sizes):

$$W = \begin{bmatrix} A & X \\ 0 & B \end{bmatrix} \qquad\qquad C = \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix}$$

Then the condition $C = C \cdot C + W$ becomes

$$\begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} = \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} \cdot \begin{bmatrix} A' & X' \\ 0 & B' \end{bmatrix} + \begin{bmatrix} A & X \\ 0 & B \end{bmatrix}$$

Applying matrix multiplication and sum block-wise:

$$\begin{aligned} A' &= A'A' + A \\ X' &= A'X' + X'B' + X \\ B' &= B'B' + B \end{aligned}$$

Because A and B are smaller than W (and still upper triangular), we know how to compute A' and B' recursively ($A' = A^+$, $B' = B^+$). There remains to find an algorithm to compute the top-right corner X' of the matrix. That is (renaming variables for convenience) the problem is reduced to finding a recursive function V which maps A, B and X to $Y = V(A, X, B)$, such that $Y = AY + YB + X$. In terms of parsing, the function V combines the chart A of the first part of the input with the chart B of the second part of the input, via a partial chart X concerned only with strings starting in A and ending in B, and produces a full chart Y. Let us divide each matrix in blocks again:

$$Y = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \quad X = \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}$$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}$$

(Again we assume that splitting can be done; the base cases can be obtained by dropping the first rows and/or the second columns in the above splits.) The condition on Y then becomes

$$\begin{aligned} \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} = {} & \begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix} \cdot \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \\ & + \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} \cdot \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix} + \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix} \end{aligned}$$

By applying matrix multiplication and sum block-wise:

$$\begin{aligned} Y_{11} &= A_{11}Y_{11} + A_{12}Y_{21} + Y_{11}B_{11} + 0 + X_{11} \\ Y_{12} &= A_{11}Y_{12} + A_{12}Y_{22} + Y_{11}B_{12} + Y_{12}B_{22} + X_{12} \\ Y_{21} &= 0 + A_{22}Y_{21} + Y_{21}B_{11} + 0 + X_{21} \\ Y_{22} &= 0 + A_{22}Y_{22} + Y_{21}B_{12} + Y_{22}B_{22} + X_{22} \end{aligned}$$

By commutativity of $(+)$ and 0 being its unit:

$$\begin{aligned} Y_{11} &= A_{11}Y_{11} + X_{11} + A_{12}Y_{21} + Y_{11}B_{11} \\ Y_{12} &= A_{11}Y_{12} + X_{12} + A_{12}Y_{22} + Y_{11}B_{12} + Y_{12}B_{22} \\ Y_{21} &= A_{22}Y_{21} + X_{21} + 0 + Y_{21}B_{11} \\ Y_{22} &= A_{22}Y_{22} + X_{22} + Y_{21}B_{12} + Y_{22}B_{22} \end{aligned}$$

Because each of the sub-matrices is smaller and because of the absence of circular dependencies, Y can be computed recursively:

$$\begin{aligned} Y_{21} &= V(A_{22}, X_{21}, B_{11}) \\ Y_{11} &= V(A_{11}, X_{11} + A_{12}Y_{21}, B_{11}) \\ Y_{22} &= V(A_{22}, X_{22} + Y_{21}B_{12}, B_{22}) \\ Y_{12} &= V(A_{11}, X_{12} + A_{12}Y_{22} + Y_{11}B_{12}, B_{22}) \end{aligned}$$

We have ignored the base cases so far because they are straightforward, except for the following point. When computing $V(A, X, B)$ on matrices of dimension 1×1, it is guaranteed that A and B are equal to 0. Indeed, in that case X is just above the diagonal. Therefore A and B are on it and must then be 0. The result matrix is therefore equal to X.

In sum, with the above definitions, we have the following expression for V in the recursive case

$$V\left(\begin{bmatrix} A_{11} & A_{12} \\ 0 & A_{22} \end{bmatrix}, \begin{bmatrix} X_{11} & X_{12} \\ X_{21} & X_{22} \end{bmatrix}, \begin{bmatrix} B_{11} & B_{12} \\ 0 & B_{22} \end{bmatrix}\right) = \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix}.$$

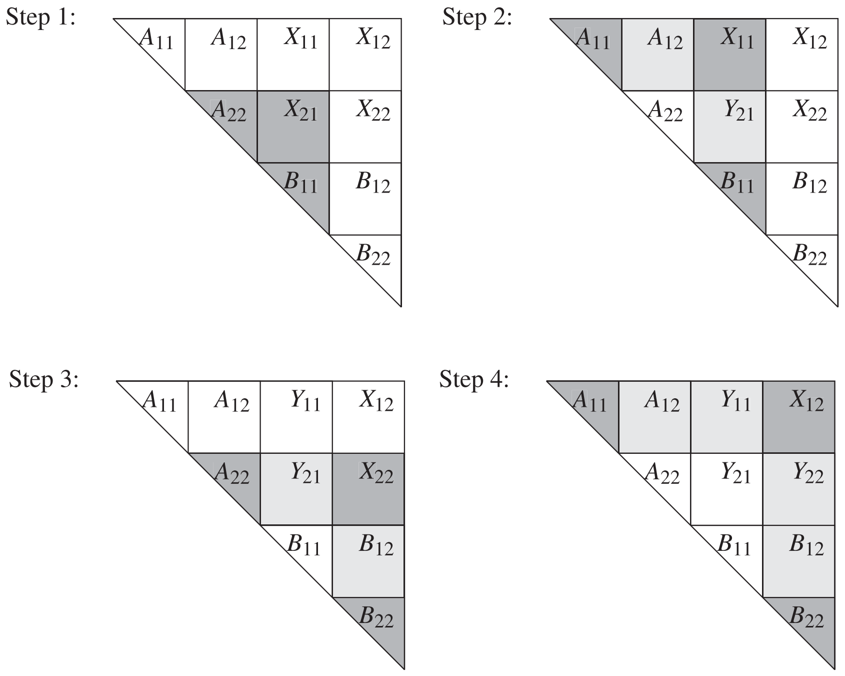

Step 1:
A_{11} A_{12} X_{11} X_{12}
A_{22} X_{21} X_{22}
B_{11} B_{12}
B_{22}
Step 2:
A_{11} A_{12} X_{11} X_{12}
A_{22} Y_{21} X_{22}
B_{11} B_{12}
B_{22}
Step 3:
A_{11} A_{12} Y_{11} X_{12}
A_{22} Y_{21} X_{22}
B_{11} B_{12}
B_{22}
Step 4:
A_{11} A_{12} Y_{11} X_{12}
A_{22} Y_{21} Y_{22}
B_{11} B_{12}
B_{22}

Fig. 2. The recursive step of function V. The charts A and B are already complete. To complete the matrix X, that is, compute $Y = V(A, X, B)$, one splits the matrices and performs four recursive calls. Each recursive call is depicted graphically. In each figure, to complete the dark-gray square, multiply the light-gray rectangles and add them to the dark-gray square, then do a recursive call on the triangular matrix composed of the completed dark-gray square and the triangles.

In the base cases, some or all of the top and/or right sub-matrices are empty and the corresponding recursive calls are omitted. In terms of parsing, initially the partial chart X contains at the bottom-left position a single non-zero element corresponding to the symbol at the interface of A and B. Recursive calls progressively fill this chart, quadrant by quadrant. The above algorithm was first described by Valiant (1975). A graphical summary is shown in Figure 2.

From Valiant's function V, one can construct the *bin* operator (completing the sequence homomorphism) as follows:

$$bin(A, t, B) \;=\; \begin{bmatrix} A & V(A, X, B) \\ 0 & B \end{bmatrix} \qquad \text{where} \quad X \;=\; \begin{bmatrix} 0 & \cdots & \cdots & 0 \\ \vdots & & \iddots & \vdots \\ 0 & 0 & & \vdots \\ \sigma_i & 0 & \cdots & 0 \end{bmatrix}$$

An advantage of Valiant's algorithm over CYK is that it treats whole sub-charts at once, via matrix-level multiplication and addition, while CYK explicitly refers to each element of C individually. In particular, when using a sparse-matrix representation, the multiplication of an empty chart with any other chart is instantaneous. The

```
import Prelude (Eq (..))
class RingLike a where
  zero :: a
  (+) :: a → a → a
  (·) :: a → a → a
data M a = Q (M a) (M a) (M a) (M a) | Z | One a
q Z Z Z Z = Z
q a b c d = Q a b c d
one x = if x ≡ zero then Z else One x
instance (Eq a, RingLike a) ⇒ RingLike (M a) where
  zero = Z
  Z + x = x
  x + Z = x
  One x + One y = one (x + y)
  Q a11 a12 a21 a22 + Q b11 b12 b21 b22
     = q (a11 + b11) (a12 + b12)
         (a21 + b21) (a22 + b22)
  Z · x = Z
  x · Z = Z
  One x · One y = one (x · y)
  Q a11 a12 a21 a22 · Q b11 b12 b21 b22
     = q (a11 · b11 + a12 · b21) (a11 · b12 + a12 · b22)
         (a21 · b11 + a22 · b21) (a21 · b21 + a22 · b22)
v :: (Eq a, RingLike a) ⇒ M a → M a → M a → M a
v a                  Z                      b = Z
v Z                  (One x)                Z = One x
v (Q a11 a12 Z a22)  (Q x11 x12 x21 x22) (Q b11 b12 Z b22)
   = q y11 y12 y21 y22
     where y21 = v a22  x21                          b11
           y11 = v a11 (x11 + a12 · y21            ) b11
           y22 = v a22 (x22 +             y21 · b12) b22
           y12 = v a11 (x12 + a12 · y22 + y11 · b12) b22
```

Fig. 3. Data structure for charts as sparse matrices (M), and implementation of the function V, in Haskell. The tricky parts compared to the mathematical development of Section 3.4 is the handling of empty matrices. Care must be taken to create empty matrices (Z) whenever they contain only zero elements. This is done by using the smart constructors q and *one* in matrix multiplication. The input matrices a and b are empty iff. the matrix x has dimension one. For conciseness, this implementation supports only matrices of size 2^n for some n. It can be extended to matrices of arbitrary dimension in a straightforward manner by adding constructors for row and column matrices, to be used as leaves. An implementation supporting arbitrary matrix dimensions, as well as the optimization explained in Section 7.2 can be found in the BNFC repository:
`https://github.com/BNFC/bnfc/blob/master/source/runtime/Data/Matrix/Quad.hs`

ability to handle this case efficiently is key: in the next section we observe that in many cases, charts are sparse, and composition of charts is efficient.

When using a straightforward representation of sparse matrices as quad-trees, the implementation of Valiant's algorithm is an elegant functional program, as can be seen in Figure 3.

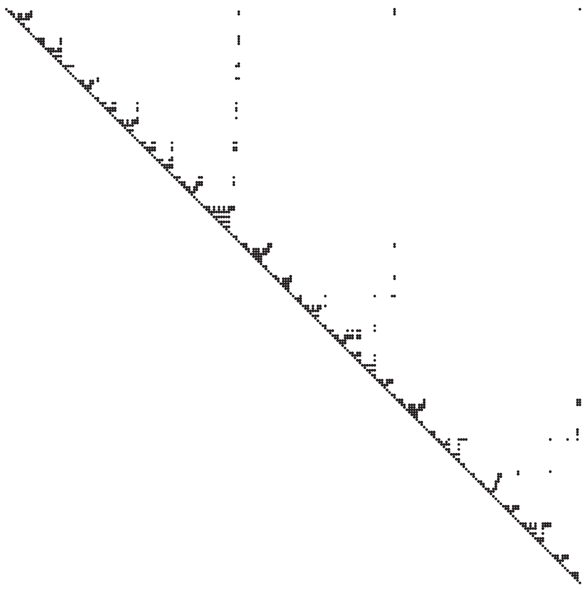


Fig. 4. The chart corresponding to a fragment of a C program. The input program can be found in the appendix. Two remarkable features merit commentary. First, the staircase shapes, which are explained in Section 5.4. Second, some small sub-matrices near the diagonal appear to be dense. These regions correspond to argument lists in the C program, and this iteration structure was implemented by linear recursion rather than our special encoding of Section 5.

4 Sparse matrix assumption and complexity analysis

4.1 Model of the input

In practice, matrices representing charts are expected to be sparse for large inputs, that is, a given sub-string is unlikely to be generated, by any non-terminal. Indeed, in most cases, the sub-string starts in the middle of a construction and ends in the middle of some other, usually unrelated other construction. This effect is illustrated in Figure 4. In the remainder of the paper, we assume that inputs conform to this assumption. Before explaining where it is coming from, we give its formal definition.

Definition 10 (*Assumption*)
There exists a constant α such that, for any input, the distribution of non-zero elements in the chart C corresponding to the input is bounded as follows. For any square sub-chart A of C above the diagonal,

$$\#A \leqslant \left\lceil \alpha \sum_{(i,j)\in \mathrm{dom}(A)} \frac{1}{(j-i)^2} \right\rceil$$

where $\#A$ is the number of non-zero elements in matrix A.

We stress that the assumption involves not a grammar *per se*, but the language itself; i.e. the set of possible input strings we expect to be fed to the algorithm, which may be a strict sub-set of the strings accepted by the grammar. (In fact, for most grammars it is easy to maliciously construct inputs which break our assumption, and for which the parsing algorithm indeed behaves badly performance-wise. We just do not expect these inputs to occur in practice, as we discuss below.)

The above formula merits justification. Before using it to evaluate the complexity of the parsing algorithm, we will build a more precise intuition for it, by examining its consequences.

4.1.1 Intuition based on string length

Let us turn first to the interpretation of the term $\frac{1}{(j-i)^2}$. Recall that a non-terminal in C_{ij} corresponds to a sub-string of size $n = j-i$ in the input. The assumption therefore says that the probability that a sub-string is parsable is inversely proportional to the square of its size. (More precisely, when considering k random sub-strings of size n in a corpus of strings representative of the language, one finds on average that $\frac{\alpha k}{n^2}$ of them correspond to a non-terminal.) That is, by doubling the size of the sub-string considered, it will be four times less likely to be parsable. This corresponds well to intuition. Indeed, in a well-formed input, every single token can be given independent meaning. However, a larger sub-string in the same well-formed input will likely start in a middle of one non-terminal (e.g. a code block or an expression) and end up in the middle of another, unrelated non-terminal. In an input which is organized hierarchically, it takes luck to pick a beginning and an end which match precisely if those are far apart.

4.1.2 Linear inputs

One may argue that most inputs (and in particular computer programs) tend to have a linear top-level structure, rather than a hierarchical one. In the words of a reviewer of an earlier version of this paper: "Longer files have *more* definitions, not *longer* ones". In this case the likelihood of a sub-string corresponding to a definition does not decrease for longer sub-strings.

Can we do anything about this issue? At first sight, the situation seems hopeless: any input which uses nesting in linear proportion to the size of its input will violate our assumption. For example, the lisp program composed of n successive applications of cons does not satisfy our assumption.

$$(\mathsf{cons}\ x\ (\mathsf{cons}\ x\ (\ldots(\mathsf{cons}\ x\ \mathsf{nil})\ldots)))$$

The reason is that the above expression contains a linear number of sub-strings that correspond to a valid list, and each of those sub-strings has length linear in the length of the total string. Our assumption is that the number of sub-strings that correspond to valid inputs should decrease much faster for longer sub-strings.

It appears however that few programs are written in this style, except perhaps for machine-generated ones. Indeed, linear constructions are often present, but they are then supported by special syntax. For example, the above lisp program is invariably written as:

$$(\mathsf{list}\ x\ x \ldots x)$$

In this case, iteration can be specified as such in the grammar (e.g. using Kleene star). We show in Section 5 how to deal with Kleene star, while respecting our assumption.

Because most texts are either hierarchical, linear or a combination of both, we are confident that our results apply to a broad category of inputs.

4.1.3 Experimental corroboration

The assumption we make is not strictly speaking verifiable experimentally, because for any chart there exists an α such that the assumption is verified. However, one can gain confidence in the assumption by plotting the probability of a string to be parsable against its size. One should observe that this probability decreases with the square of the size. In practical terms, given a chart corresponding to a large input, if one observes a drastic cut-off in the density of non-zero elements when departing from a certain distance from the diagonal, then the input is compatible with our assumption. In Figure 4, we show a chart corresponding to a fragment of C code, obtained using our algorithm. This chart, along with all other inputs for which we have run this experiment, exhibits the expected features. The assumption is also confirmed, albeit indirectly, by observing that the cost analysis which depends on it holds in practice (see Section 4.3.4).

4.2 Close and far matrices

In this sub-section we introduce the concepts of close and far matrices, which are useful both to get an intuition of the cost model, and to compute the cost of the algorithm.

For simplicity we consider only inputs of sizes which are powers of 2. This additional assumption implies that we only need to consider square matrices in our analysis.

We first remark that because charts are always divided in the middle, a sub-chart X considered by the algorithm is always square, and at a distance kn to the diagonal, where k is some natural number and n is the size of X. When $k = 0$ we say that X is close to the diagonal and when $k > 0$ we say that X is far from the diagonal. This distinction is crucial, because matrices close to the diagonal have $O(\log n)$ elements in them, whereas matrix far away have a constant number of elements in them. This fact is not obvious, so we devote the present sub-section to its proof.

Definition 11
The *distance to the diagonal* of a sub-chart is $(j - i - 1)$ iff its bottom-most leftmost element has index (i, j) in the complete chart.

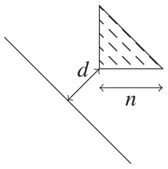
Assume $S(n, d)$ is a square sub-matrix of size n at distance d to the diagonal.

Our assumption puts upper bounds on the number of non-zero elements in $S(n, d)$. In this section, we will compute an asymptotic upper bound of $\#S(n, kn)$, for any k. The strategy is to symbolically evaluate $P(A)$, from which it is easy to infer bounds for $\#A$, where

$$P(A) = \sum_{(i,j)\in \mathrm{dom}(A)} \frac{1}{(j-i)^2}$$

4.2.1 *Triangles*

As a stepping stone, we consider a lower triangle $T(n,d)$, of size n and at distance d to the diagonal, because the above sum is then easy to evaluate symbolically.



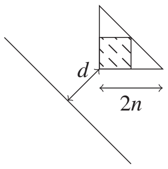
We have:

$$\begin{aligned} P(T(n,d)) &= \sum_{(i,j)\in T(n,d)} \frac{1}{(j-i)^2} \\ &= \sum_{k=1}^{n} \sum_{l=1}^{k} \frac{1}{(d+k)^2} \\ &= \sum_{k=1}^{n} \frac{k}{(d+k)^2} \\ &= \psi^0(d+n+1) - \psi^0(d+1) + \\ &\quad d(\psi^1(d+n+1) - \psi^1(d+1)) \end{aligned}$$

Where ψ is the polygamma function, which is approximated asymptotically by logarithms: $\psi^k(n) \sim \frac{d^k}{dn} \log n$.

4.2.2 *Squares*

From the above result on triangles one can recover a result on squares: a square of size n is a triangle of size $2n$ minus two triangles of size n:



$$P(S(n,d)) = P(T(2n,d)) - 2P(T(n,n+d)) \tag{4}$$

Using the expression for $P\,(T(n,d))$ and the approximation of ψ together with Equation (4), we get

$$\begin{aligned} P\,(S(n,kn)) \sim\ & 2(kn+n)\left(\frac{1}{kn+n+1}-\frac{1}{kn+2n+1}\right) \\ & -kn\left(\frac{1}{kn+1}-\frac{1}{kn+2n+1}\right) \\ & -\log(kn+1)+2\log(kn+n+1)-\log(kn+2n+1) \end{aligned}$$

- if $k \geqslant 1$, we have

  $$\lim_{n\to\infty} P\,(S(n,kn)) = 2\log(k+1)-\log(k+2)-\log(k)$$

  and the limit converges from below. So we the above expression is an asymptotic bound for $P\,(S(n,kn))$.
- if $k=0$, we have

  $$\begin{aligned} P\,(S(n,kn)) &= P\,(S(n,0)) \\ &\sim 2n\left(\frac{1}{1+n}-\frac{1}{1+2n}\right)+2\log(1+n)-\log(1+2n) \\ &\sim \log n \end{aligned}$$

4.2.3 Summary

We therefore have the following upper bounds of a square sub-matrix A:

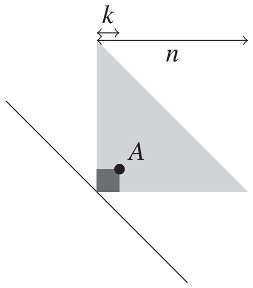
- if A is close to the diagonal, then $\#A \leqslant \lceil \alpha \log n \rceil$
- if A is far from the diagonal (its distance is kn with $k \geqslant 1$), then

  $$\#A \leqslant \lceil \alpha(2\log(k+1)-\log(k+2)-\log(k)) \rceil$$

  Remarkably, this upper bound is independent from the size of A.

4.2.4 Cost model and balancing of trees

We can further support the validity of our assumption using the mathematical tools developed so far. What we can do is to connect the logarithmic amount of non-zero elements in a close matrix with the balancing factor of input trees. Consider the triangle-shaped sub-chart T^n which touches the diagonal and a non-terminal A at distance k from it. We assume all symbols in the triangle but closer to the diagonal combine to form A. If the symbol A can be combined with exactly one other symbol of size βk with $0 < \beta \leqslant 1$, it will yield exactly one symbol at distance $(1+\beta)k$. Inductively we compute that there are $O\left(\frac{\log(n)}{\log(1+\beta)}\right)$ symbols in the triangle, which is compatible with our condition, with $\alpha = 1/\log_2(1+\beta)$.



4.3 Cost estimation

Having described our cost model and built the necessary mathematical tools, we can now proceed with the complexity analysis. We will estimate the cost as the number of elementary multiplications (multiplications on sets of non-terminals) to be performed. This is enough because the number of elementary additions is in the worst case linear in the number of multiplications. All the results of this sub-section assume the distribution of non-zero elements discussed above.

4.3.1 Cost of matrix multiplications

We start by estimating M_n, the cost of the multiplication of two square sub-charts A and B of size n.

Theorem 2
The complexity of sub-chart multiplication M_n is $O(1)$ in average and $O(\log n)$ in the worst case.

Proof
We assume a standard recursive matrix multiplication algorithm that divides both of its argument matrices in four equal-sized sub-matrices. We proceed by case analysis on whether the matrices are close or far from the diagonal. Let us write FF_n for M_n if both matrices are far, CF_n if one is close and one is far, and CC_n if both are close. Let us evaluate each case:

- $FF_n = O(1)$. Indeed, both $\#A$ and $\#B$ are bounded by a constant when A and B are far from the diagonal.
- $CC_n = O(CF_n)$. Indeed, when dividing a matrix close to the diagonal in four equal-sized blocks, only the bottom-left corner is close to the diagonal, the other ones are far away. The recursion for block-wise matrix multiplication then yields $CC_n = 4CF_{\frac{n}{2}} + 4FF_{\frac{n}{2}}$. Because FF_n is $O(1)$, we have that CC_n is $O(CF_{n/2})$. We can then conclude that CC_n also is $O(CF_n)$.
- $CF_n = O(1)$. Let us assume A close and B far away. Let B_{ij} for $i, j \in \{1, 2\}$ be the sub-matrices of the far matrix, B. Because $\#B$ is bound by a constant, we can assume without loss of generality $\#B = 1$. Indeed, after a constant number of recursion steps in the multiplication, there will remain at most a

single element in the far-away matrix B. We can then weigh the cost of each recursive call by $\#B_{ij}$:

$$\begin{aligned} CF_n =& \#B_{11}FF_{\frac{n}{2}} + \#B_{21}FF_{\frac{n}{2}} + \#B_{12}FF_{\frac{n}{2}} + \#B_{22}FF_{\frac{n}{2}} \\ & + \#B_{11}CF_{\frac{n}{2}} + \#B_{21}FF_{\frac{n}{2}} + \#B_{12}CF_{\frac{n}{2}} + \#B_{22}FF_{\frac{n}{2}} \\ =& rCF_{\frac{n}{2}} + O(1) \end{aligned}$$

Where $r = \#B_{11} + \#B_{12}$, and is 1 if the element of the matrix B is in its upper part, and 0 otherwise. In the worst case, $r = 1$, and the solving the recurrence using the Master Theorem (Theorem 1) gives $CF_n = O(\log n)$. In the average case we can assume an even distribution of the non-zero element in B, which implies $r = 1/2$. The solution of the recurrence is therefore $CF_n = O(1)$. □

4.3.2 Cost of the conquer step

We proceed to estimate the running cost V_n of the valiant function V on a matrix of size n.

Theorem 3
The complexity of the V function is $O(\log n)$ on average and $O(\log^2 n)$ in the worst case.

Proof
We will compute the number of matrix multiplications performed; the worst case complexity is obtained merely by multiplying by a $\log n$ factor.

We assume that we know the resulting chart $Y = V(A, X, B)$. That is, V_n maps Y to the cost of running $V(A, X, B)$. From the definition of Valiant's algorithm, we have the following recurrence:

$$\begin{aligned} V_n(0) &= 0 \\ V_n \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} &= V_{\frac{n}{2}}(Y_{21}) + V_{\frac{n}{2}}(Y_{11}) + V_{\frac{n}{2}}(Y_{22}) + V_{\frac{n}{2}}(Y_{12}) \\ &\quad + M_{\frac{n}{2}}(A_{12}, Y_{21}) + M_{\frac{n}{2}}(Y_{21}, B_{12}) \\ &\quad + M_{\frac{n}{2}}(A_{12}, Y_{22}) + M_{\frac{n}{2}}(Y_{11}, B_{12}) \end{aligned}$$

Because A and B are upper-triangular matrices, the sub-charts A_{12} and B_{12} are close to the diagonal. We distinguish two cases: either Y is close or far from the diagonal of the chart. In the former case we let $V_n = F_n$ and in the latter case $V_n = C_n$.

Y **far.** All sub-matrices of Y are far from the diagonal. The recurrence specializes then to

$$\begin{aligned} F_n(0) &= 0 \\ F_1(Y) &= 1 \\ F_n \begin{bmatrix} Y_{11} & Y_{12} \\ Y_{21} & Y_{22} \end{bmatrix} &= F_{\frac{n}{2}}(Y_{11}) + F_{\frac{n}{2}}(Y_{12}) + F_{\frac{n}{2}}(Y_{21}) + F_{\frac{n}{2}}(Y_{22}) \\ &\quad + O(1) \end{aligned}$$

Because Y has a constant number of non-zero elements, *a fortiori* so has X, therefore most recursive calls will return immediately, and on average only one recursive call needs to be counted. We thus have

$$F_n = F_{\frac{n}{2}} + O(1)$$

Hence we use the Master Theorem with $a = 1, b = 2$ and $g(n) = 1$. We are therefore in the case $c = e$, and obtain $F_n = O(\log n)$.

Y **close.** Out of the four sub-matrices of Y, Y_{21} is close to the diagonal and the other three are far from it. Therefore the recurrence specializes to:

$$\begin{aligned} C_1 &= 1 \\ C_n &= C_{\frac{n}{2}} + 3F_{\frac{n}{2}} + O(1) \\ &= C_{\frac{n}{2}} + O(\log n) \end{aligned}$$

We use the Master Theorem with $a = 1$, $b = 2$ and $g(n) = O(\log n)$. We are in the case $c = e$, and obtain $C_n = O(\log^2 n)$. □

4.3.3 *Total cost*

We can proceed to compute the total cost of our algorithm T_n on an input string of size $n = |w|$. Again, we use the Master Theorem. We divide the input into two parts, so $b = 2$. We assume that the input is already provided as a balanced tree representing the matrix $I(w)$, and so the cost of the divide step is zero. Therefore $g(n)$ is the cost of the conquer step only. This step involves a matrix close to the diagonal, so $g(n) = C_n = O(\log^d n)$, and in turn $c = 0$. The constant d is 2 if one considers the average case or 3 in the worst case.

$$T_n = aT_{\frac{n}{2}} + O(log^d n)$$

- If we assume a sequential execution of the two sub-problems then we have $a = 2$. In turn, $e = 1$ and $T(n) = O(n)$.
- If we assume perfect parallelization of sub-problems, or an incremental situation, where one of the sub-solution can be reused, then $a = 1$. In turn, $e = 0$ and $T(n) = O(\log^{d+1} n)$.

Valiant's evaluation (1974) for V_n is $O(n^\gamma)$, for some γ between 2 and 3 (the exact value depends on the matrix multiplication algorithm used). In his case $c = \gamma$ and $d = 0$, yielding $T(n) = O(n^\gamma)$, whatever the value of a. That is, according to Valiant's analysis, making an incremental or parallel version of his algorithm would lead to no benefit, while our analysis reveals that a big payoff is at hand for inputs that satisfy our assumption.

4.3.4 *Experiments*

In this section, we aim to provide some experimental evidence that supports the timing analyses of the combination operator on which our divide-and-conquer parallel parsing and incremental parsing algorithms rely. We have conducted two sets of experiments, one measuring running times on practical inputs, and one measuring running times to demonstrate worst-case execution time. All timings were obtained using the CRITERION library (O'Sullivan 2013), on an Intel Core 2 at 2.13 GHz. All programs were compiled with GHC 7.6.1.

In the first set, we have measured the performance on a practical language on practical inputs, to confirm that the function is fast enough to use as an incremental parser in an interactive setting. To do so, we have run our BNFC implementation on a C grammar to produce the σ and $(\cdot)$ functions, and tested the running time of the V function on a large C program, extracted from the Linux kernel scheduler (`https://github.com/torvalds/linux/blob/master/kernel/sched/core.c` — preprocessor directives as well as `typedefs` found in it were expanded by hand.) The input was divided into a left part and a right part of equal sizes, and a middle symbol. The complete charts for the left and the right part were computed, then we measured the time of the V function on the charts and the singleton chart containing the middle symbol. After collecting 100 samples, CRITERION reported a mean runtime of 320µs, with a standard deviation of 23µs. This is well within acceptable limits for interactive use: most people cannot perceive a delay less than a millisecond.

In the second set of experiments, we tested the V function on generated inputs of various sizes, to confirm our calculation of the worst-case running time. The grammar is that corresponding to the encoding of t^* (the non-terminal t repeated an arbitrary number of times) using the technique described in Section 5 (which ensures that our assumption is verified with α close to 1). The inputs were a repetition of that terminal symbol. The results are shown in Figure 5. We observe that the measurements, when drawn on a semi-logarithmic scale, fit a quadratic curve; which agrees with the theoretical cost estimation.

Note that these experiments should not be understood as evidence that a parallel implementation actually outperforms a dedicated sequential parsing implementation. To provide a realistic fast parallel implementation, there are many practical issues to be dealt with (such as scheduling processes on processors, memory access patterns, how to chunk up inputs, etc.), which are outside the scope of this paper. These experiments merely measure the time of a single run of the combining operator.

5 Iteration in context-free grammars

5.1 The problem with iteration

While we have worked hard to ensure the efficient handling of the non-associative aspect of CF parsing, we have neglected so far that most CF languages feature regular iteration; that is, associative concatenation rules. Without special treatment, such associative rules cause severe inefficiencies in the algorithm as presented so far.

Iteration is often referred to as Kleene's closure, and is written here as a postfix star (*). In CF grammars, it can be (and usually is) encoded as either as left or right

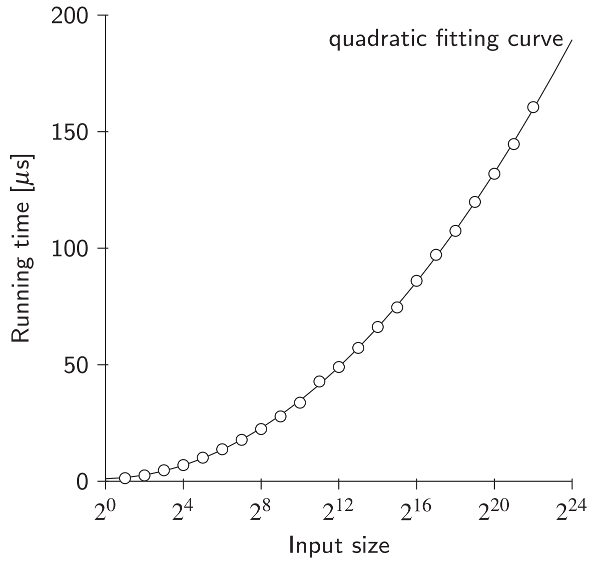


Fig. 5. Running time of the V function in function of the size of the input, using semi-logarithmic scale. The grammar is that corresponding to the encoding of t^* using the technique described in Section 5. The next data point (input size 2^{23}) could not be obtained due to running out of memory. The curve is the graph of a quadratic function which fits the measurements.

recursion. For example a rule $A ::= Y^*$ is typically encoded as follows.

$$A ::= \epsilon$$
$$A ::= AY$$

The problem with this encoding is two-fold. First, inputs consisting mostly of a sequence of Y necessarily violate our assumption on inputs: the depth of the parse tree grows linearly with the size of the input, creating a dense matrix, as discussed in Section 4.1.2.

Second, the generated abstract syntax tree (AST) will necessarily be linear. Consequently, as we have seen in the introduction, this linear shape would preclude fully parallel or incremental processing of the AST by computations consuming it. So, even if we could somehow make parsing efficient using the above encoding of iteration, subsequent passes would still be encumbered. Hence we take the stance that special support for iteration is necessary in any parallel or incremental parser (Wagner & Graham 1998 come to the same conclusion).

5.2 Towards an efficient encoding

Instead of a linear, unary encoding of iterations, one can attempt a binary tree encoding. One might propose the following encoding:

$$A ::= AA$$
$$A ::= Y$$

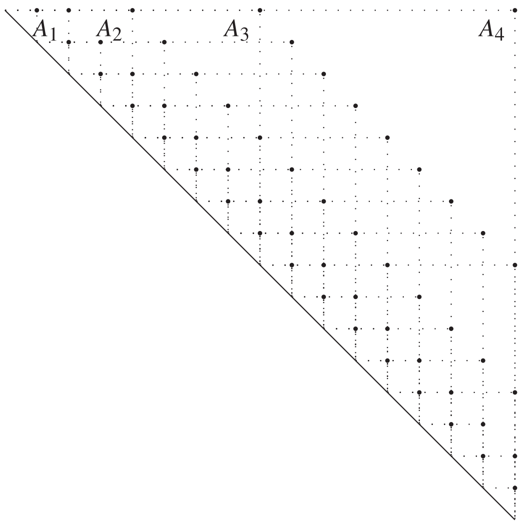


Fig. 6. Example chart for the grammar $A_{k+1} ::= A_k A_k$.

However this encoding accepts all possible associations of sequences of Ys, in particular also linear ones. One might attempt to mend the rules by using a more clever encoding, say:

$$A_{k+1} ::= A_k A_k$$

Ignoring that it codes only lists of size 2^n for some n, our condition on inputs is still violated. Indeed, in a sequence of Y, any sub-sequence of length 2^n for some n would be recognized. This means that there would be a lot of overlap between possible parse trees (see Fig. 6).

In the remainder of the section we describe a way to keep the rule $A ::= AA$, but tweak the parsing algorithm so that for any sequence of Ys only a single association is considered.

5.3 *Oracle-sensitive parsing*

5.3.1 *Overview*

Each non-terminal will be paired with a bit indicating whether it should be used either as a left- or right-child in the parse tree. The bit will be chosen by an oracle upon reduction of the non-terminal, so that the tree will be balanced. We write Y^b for the non-terminal Y annotated with bit b. The main rule constructing trees is then written:

$$Y ::= Y^0 Y^1$$

This restricts which trees are constructed, as only Y's labeled with a 0 and Y's labeled with a 1, in that order, are paired up with each other. After parsing with this rule, the chart will contain a sequence of Y^1 (unmatched right children) of growing size followed by a sequence of Y_0 (unmatched left children) of decreasing size, as

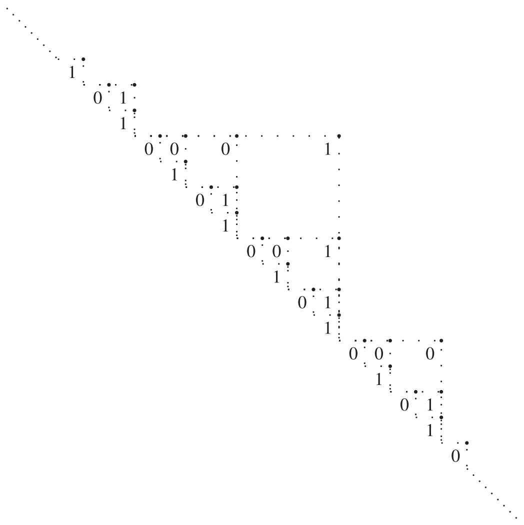


Fig. 7. Matching a list using the oracle-sensitive algorithm. We assume that only one non-terminal Y is involved and thus show only the bit-tags. Considering only the non-terminals which cannot be combined using the rule $Y ::= Y^0 Y^1$, the charts features a sequence of Y^1 (triangles of increasing size with a 1 at the top), followed by a sequence of Y^0 (triangles of decreasing size with a 0 at the top).

depicted in Figure 7. These unmatched nodes will then be collected using special rules, as described below.

For the iteration encoding to work, we need to assume that the iteration in the original grammar occurs in the following form:

$$L ::= C_0 Y * D_0$$

Here, Y is the element that is iterated, and C_0 and D_0 are *delimiters*. For example, in the C language, L may be a code block, Y a statement, and C_0 and D_0 curly braces { and }, respectively.

The extra collecting rules that are needed would be written in our encoding:

$$\begin{array}{ll} C ::= C_0 & \quad D ::= D_0 \\ ::= C\,Y^1 & \quad ::= Y^0 D \end{array}$$

The final list can be then produced by the rule $L ::= CD$. Here, it may seem that the arbitrary left-recursion (resp. right-recursion) in the rule for C (resp. D) may lead us back to the problem where we started. The key difference is that each recursion step in these rules involves a Y that is twice as big as the previous one (as depicted in Figure 7), leading to a logarithmic instead of a linear number of recursion steps (in terms of the number of iterations).

The delimiters C_0 and D_0 are necessary so that only one collection of Y^1 and only one collection of Y^0 are considered; thereby ensuring a good performance. Without delimiters, every combination of sequences of Y^1 and Y^0 would need to be considered.

5.3.2 *Oracle-sensitive grammar formalism*

In general, we extend productions so that non-terminals on a right-hand side are tagged with a bit. Formally, an oracle-sensitive grammar is given by the quadruple (Σ, N, P, S), where each production rule in P has either of the following forms:

- $A ::= B^{b_1} C^{b_2}$
- $A ::= t$, for $t \in \Sigma$

where $b_1, b_2, \ldots$ range over bits.

We allow, as a shorthand, to write non-annotated non-terminals in the right-hand side of a production rule. The production then stands for a pair of productions with either annotation. That is $A ::= \alpha_0 B \alpha_1$ is a shorthand for the pair of rules $A ::= \alpha_0 B^0 \alpha_1$ and $A ::= \alpha_0 B^1 \alpha_1$.

Algorithm. The implementation takes a grammar written using a special construction for iteration and translate it to the above formalism appropriately. The algorithmic part of the parsing procedure remains the same as previously. The part which changes is the operators generating and combining non-terminals, as follows.

Definition 12

$$\begin{aligned} \sigma_i &= \{A^{\mathsf{oracle}(\rho)} \mid A ::= w[i] \in P\} \\ x \cdot y &= \{A^{\mathsf{oracle}(\rho)} \mid B^{b_1} \in x, C^{b_2} \in y, A ::= B^{b_1} C^{b_2} \in P\} \end{aligned}$$

In the above, the oracle function can be considered an effectful function which extracts a bit from the supply of bits ρ. (Each call to the oracle will return a different bit b.) The transitive closure function of $I(w)$ modified to use the above version of the σ and $(\cdot)$ operators is called T_ρ in the remainder of the paper.

Formalization and proof. We proceed to prove that the above implementation indeed recognizes the intended language. But first, we must define the meaning of our extended grammar formalism and show that it corresponds to our needs.

The main issue is that the algorithm behaves non-deterministically, in the sense that the grammar writer does not have access to the bits generated by the oracle. The rest of the section is structured as follows:

1. we define a generation relation restricted to a given source of bits ρ, which represents the oracle;
2. we show that the algorithm decides the above relation for a specific (but intangible) ρ;
3. we narrow the acceptable grammars to those which are oblivious to ρ (describe languages independent of ρ);
4. we provide a toolkit which enables to identify and construct such oblivious grammars;
5. and finally we show that our encoding of iteration preserves obliviousness.

Oracle. We define a new generation relation $\overset{\rho}{\longmapsto}$, indexed by a stream of bits ρ. This stream of bits wholly models the oracle.

The meaning of production rules annotated with bits can then be given. We first define a 1-step generation relation indexed by a single bit.

Definition 13 (*bit-indexed generation*)

- if $(A ::= B^{b_1}C^{b_2}) \in P$, then $w_0 A^b \alpha \overset{b}{\longmapsto} w_0 B^{b_1} C^{b_2} \alpha$
- if $(A ::= x) \in P$, then $w_0 A^b \alpha \overset{b}{\longmapsto} w_0 x \alpha$

Crucially, the rules require the relation to act on the first non-terminal in a string. This forces the bit-stream ρ to be used in a deterministic way. Otherwise, the relation could use each bit of ρ in an arbitrary place, essentially bypassing the instructions of the oracle transmitted via the bit-stream ρ.

Definition 14 (*stream-indexed generation*)
The relation $\alpha \overset{\rho}{\longmapsto} w$ is inductively defined as follows.

- $w \overset{\rho}{\longmapsto} w$
- If $\alpha \overset{b}{\longmapsto} \gamma$ and $\gamma \overset{\rho}{\longmapsto} w$ then $\alpha \overset{b,\rho}{\longmapsto} w$

Algorithm. The algorithm decides the $\overset{\rho}{\longmapsto}$ relation, but only for one particular bit-stream ρ (which the grammar writer has no control over).

Theorem 4
For every ρ,

$$A^b \overset{\rho}{\longmapsto} w_{ij} \quad \text{iff} \quad A^b \in T_\rho(w)_{ij}$$

(Note that the quantification over ρ is outside the iff. operator.)

Proof
By induction on the decomposition structure of the matrix (done by T). Note that ρ changes on every recursion step, this is why we have an explicit quantification over ρ in the formulation of the theorem. □

Obliviousness. Ultimately, we do not want the language defined using our formalism to depend on the actual stream ρ of bits generated by the oracle, because this is out of the control of the grammar writer. That is, if a string is generated using some ρ, it should be generated with every ρ.

We first remark that the set of strings generated by any given tagged non-terminal always depends on ρ. Hence instead we have to consider the strings generated by sets of non-terminals (and in general sets of strings). We thus define the following relations, using Γ, Δ and Ξ to range over sets of strings.

Definition 15

- $\Gamma \overset{\exists}{\longmapsto} w$ iff. $\exists\rho.\exists\alpha \in \Gamma.\ \alpha \overset{\rho}{\longmapsto} w$
- $\Gamma \overset{\forall}{\longmapsto} w$ iff. $\forall\rho.\exists\alpha \in \Gamma.\ \alpha \overset{\rho}{\longmapsto} w$

Definition 16
A set of strings Γ is called *oracle-oblivious* if the set of strings of terminals generated by it is insensitive to non-determinism; that is, for any w_0, if $\Gamma \overset{\exists}{\longmapsto} w_0$ then $\Gamma \overset{\forall}{\longmapsto} w_0$.

Definition 17
We denote by $\tilde{A}$ the set $\{A^0, A^1\}$, and $\Gamma\Delta$ the set $\{\gamma\delta \mid \gamma \in \Gamma, \delta \in \Delta\}$

Definition 18 (well-formed grammar)
An oracle-sensitive grammar (Σ, N, P, S) is well-formed if $\tilde{S}$ is oracle-oblivious.

We can then show that obliviousness fulfills its purpose: the sensitivity to ρ introduced in the algorithm is indeed hidden by obliviousness.

Theorem 5
If $\tilde{A}$ is oracle-oblivious then

$$\tilde{A} \overset{\forall}{\longmapsto} w_{ij} \quad \text{iff} \quad \exists b.\exists \rho. A^b \in T_\rho(w)_{ij}$$

Proof

left-to-right direction. By definition, $\tilde{A} \overset{\forall}{\longmapsto} w_{ij}$ implies in particular that there exists a ρ and a b such that $A^b \overset{\rho}{\longmapsto} w_{ij}$. Theorem 4 yields the desired conclusion.

right-to-left direction. Because of the obliviousness of $\tilde{A}$ it suffices to prove that $\exists b.\exists \rho. A^b \in T_\rho(w)_{ij}$ implies $\exists \rho.\exists b. A^b \overset{\rho}{\longmapsto} w_{ij}$. Again, Theorem 4, in the right-to-left direction, yields the desired conclusion.

□

A kit for well-formed grammars. Given a grammar definition using bit-annotations arbitrarily, it is hard to decide whether it is well formed. Hence we define the following relation, which enables us to reason about obliviousness compositionally.

Definition 19 (transportation of obliviousness)
We write $\Gamma \overset{*}{\Longrightarrow} \Delta$ iff for every w_0,

- if $\Gamma \overset{\exists}{\longmapsto} w_0$ then $\Delta \overset{\exists}{\longmapsto} w_0$.
- if $\Delta \overset{\forall}{\longmapsto} w_0$ then $\Gamma \overset{\forall}{\longmapsto} w_0$.

The usefulness of the above definition is motivated by the following lemma, which states conditions under which obliviousness can be transported from one set to another.

Lemma 2
If $\Gamma \overset{*}{\Longrightarrow} \Delta$ and Δ is oracle oblivious, then so is Γ.

Proof
Assume $\Gamma \overset{\exists}{\longmapsto} w_0$, then we know $\Delta \overset{\exists}{\longmapsto} w_0$. By obliviousness of Δ we then know $\Delta \overset{\forall}{\longmapsto} w_0$. It follows that $\Gamma \overset{\forall}{\longmapsto} w_0$. □

Lemma 3

1. $\overset{*}{\Longrightarrow}$ is reflexive and transitive
2. If $\Gamma \overset{*}{\Longrightarrow} \Delta$ then $\Gamma\Xi \overset{*}{\Longrightarrow} \Delta\Xi$ and $\Xi\Gamma \overset{*}{\Longrightarrow} \Xi\Delta$
3. consider a non-terminal A, and Γ the set of its rhs productions. Then $\tilde{A} \overset{*}{\Longrightarrow} \Gamma$.

Proof
1 and 3 are a direct consequences of the definitions. The proof of 2 is tedious but straightforward, and similar in style to the proof of Lemma 4 and thus omitted. □

The above lemma means that, if productions are written without bit annotations (they generate all possible annotations), then they preserve obliviousness. Hence, a grammar written without annotations is necessarily well formed. Because our encoding of iteration also preserves obliviousness, this in turn means that, if one uses annotations only to encode iteration in the pattern we prescribe, the grammar is then well-formed.

Encoding iteration. As a reminder, we encode $L ::= C_0 Y_0 * D_0$, as

$$\begin{aligned} Y &::= Y_0 \\ &::= Y^0 Y^1 \\ C &::= C_0 \\ &::= C\,Y^1 \\ D &::= D_0 \\ &::= Y^0 D \\ L &::= CD \end{aligned}$$

Theorem 6
$\tilde{L} \overset{*}{\Longrightarrow} \tilde{C}_0 \tilde{Y}_0^* \tilde{D}_0$

Proof
We construct the relation in the following stages, and combine them using transitivity.

1. $\tilde{L}$
2. $\tilde{C}\tilde{D}$
3. $\tilde{C}_0\{Y^1\}^*\{Y^0\}^*\tilde{D}_0$
4. $\tilde{C}_0\tilde{Y}^*\tilde{D}_0$
5. $\tilde{C}_0\tilde{Y}_0^*\tilde{D}_0$

Each transition can be proved, as follows:

- 1–2 is a direct consequence of Lemma 3.
- For 2–3, we need prove $\tilde{C} \overset{*}{\Longrightarrow} \tilde{C}_0\{Y^1\}^*$, (and symmetrically for D). That is, we must prove that $\tilde{C} \overset{*}{\Longrightarrow} \tilde{C}_0\{\underbrace{Y^1 \dots Y^1}_{i} \mid i < n\}$ by induction on n. Both the base case and the induction case are consequences of Lemma 3.

- 3–4 requires special treatment: it depends on the relation

  $$\{Y^1\}^*\{Y^0\}^* \overset{*}{\Longrightarrow} \tilde{Y}^*$$

  Proving it requires two preservation lemmas for every w_0:

  — if $\{Y^1\}^*\{Y^0\}^* \overset{\exists}{\longmapsto} w_0$ then $\tilde{Y}^* \overset{\exists}{\longmapsto} w_0$.

  — if $\tilde{Y}^* \overset{\forall}{\longmapsto} w_0$ then $\{Y^1\}^*\{Y^0\}^* \overset{\forall}{\longmapsto} w_0$.

  The first one is an easy consequence of the ability to choose any possible ρ in the $\overset{\exists}{\longmapsto}$ relation. The second one is the angular stone of our method, and is proved in the following lemma.
- 4–5 is a direct consequence of Lemma 3.

□

Lemma 4
Let $w \in \Sigma^*$ and $\alpha \in \tilde{Y}^*$. If $\alpha \overset{\forall}{\longmapsto} w$ then there exists $\beta \in \{Y^1\}^*$ and $\gamma \in \{Y^0\}^*$ such that $\beta\gamma \overset{\forall}{\longmapsto} w$.

Proof
By induction on the length of α. If α is in the required form, we have the result. If not, then the sub-sequence Y^0Y^1 can be found at least once in α:

$$\alpha = \alpha_0 Y^0 Y^1 \alpha_1$$

We can decompose w into two parts w_0 and w_1 such that

$$\alpha_0 \overset{\forall}{\longmapsto} w_0$$

$$Y^0Y^1\alpha_1 \overset{\forall}{\longmapsto} w_1$$

But, for any b, we have $Y^b\alpha_1 \overset{b}{\longmapsto} Y^0Y^1\alpha_1$. Therefore, $Y^b\alpha_1 \overset{\forall}{\longmapsto} w_1$ and in turn $\alpha_0 Y^b \alpha_1 \overset{\forall}{\longmapsto} w$.

We can then use the induction hypothesis on $\alpha_0 Y^b \alpha_1$ to obtain β and γ satisfying the conditions of the theorem. □

5.4 Performance

The above encoding yields good performance in practice, even with a naive implementation of the oracle providing the stream of bits ρ, which does not produce perfectly balanced trees. Indeed, Figure 7 shows the chart generated from a sample C program. It exhibits the drastic cut-off in non-zero node density formalized in Definition 10, except for a few linear shapes, as one can observe. These are caused by our implementation of the oracle, which is naive. In our implementation, the bit which is generated is a parameter of the function V, and it is flipped (deterministically) for some recursive calls. This means that, inside a given sub-chart, all instances of associative rules either right-associate or left-associate, yielding a linear arrangement of results in the chart. Yet, this strategy for bit generation is the best we have found with respect to observed performance. The reason might be that more even distributions of results in the chart worsens the locality of non-zero data, yielding smaller zero sub-charts.

6 Related work

6.1 Our own previous work

Claessen (2004) wrote a paper titled "parallel parsing processes", which has only tenuous connections with the present work. The paper of 2004 presents a parsing technique based on usual sequential parsers, but where disjunction is represented by processes running concurrently. An advantage of that technique is that the parser processes the input string in chunks that can be discarded as soon as the parser has analyzed them.

Bernardy (2009) has shown how to combine the above idea with the online parsers of Hughes & Swierstra (2003). This makes the resulting parsing algorithm suitable for incremental parsing in an editing environment such as Yi (Bernardy 2008). However the method is brittle, because grammars need to be expressed in a special-purpose formalism, and error-correction must be "baked-in" into the grammar. In contrast, the method presented here accepts grammar in Backus-Naur Form (see Section 7.6); only iterative structures need to be changed to use the special construction of Section 5. One does not have to worry about error recovery because all sub-strings are parsed.

The present work was presented, in a draft version, at ICFP (Bernardy & Claessen 2013). Besides correcting several minor mistakes and improving the presentation, the present version gives a better analysis of the complexity of the parsing algorithm: we show that the algorithm is asymptotically faster by a factor of $\log n$ in the average case.

6.2 Special support for iteration

The assumption we make on inputs, which is tied to the balancing of the parse trees is partially inspired by work by Wagner & Graham (1998). They show that linear parse trees cannot be handled efficiently (in parallel or incrementally), because updating a structure requires time proportional to its depth. Wagner & Graham (1998) then deduce that efficient incremental parsing requires a special purpose support for iteration, as we have done in Section 5.

6.3 General CF parsing

A well-known method for parsing general CF languages is that of Tomita (1986). This method has in common with ours that it achieves linear performance on well-behaved inputs, while degrading gracefully to the best possible performance (cubic) in the worst case.

The main differences between the methods are that Tomita's algorithm processes the input sequentially using the grammar in a top-down fashion, while we process all parts of the input simultaneously, using the grammar in a bottom-up fashion. The condition for well-behaved inputs is thus different for each method. In Tomita's case, the condition is that, at any point during the parsing, the amount of ambiguity is small (bound by a constant), implying that the next action of the parser is most of the time determined by the next symbol in the input. In our case, it is captured by Definition 10, which essentially means that the input should be hierarchical. Tomita's condition does not imply ours: linearly arranged inputs can be deterministic. Checking the other implication is left for future work. It is not easy to conclude: our condition imposes non-local conditions which may or may not restrict non-determinism in a linear processing of the input.

The chief advantage of our method is its divide-and-conquer structure, which means that it can be used in a standard parallel or incremental framework. Tomita inherits essential use of the sequential processing of the input from LR parsing, making his technique not amenable to divide-and-conquer-based parallelization.

6.4 Parallel parsing

There is a wealth of previous work devoted to efficient recognition and parsing of CF languages on abstract parallel machines, so much that a comprehensive survey of the field is out of the scope of this paper. The situation can however be summarized as follows: to the best of our knowledge, before this work, algorithms proposed for parallel parsing either need an unrealistic number of processors, or they target a language class which is too restrictive to be of practical interest.

6.4.1 Too many processors

Sikkel & Nijholt (1997) describe a parallel algorithm (in Section 6.3) which can recognize a string of length n in $O(\log n)$ time, but it requires $O(n^6)$ processors in the worst case.

A line of work involving Rytter gives a dozen of complexity results for various sub-classes of CF and various abstract machines. The most closely related results are perhaps the following.

Chytil *et al.* (1991) present a simple parallel algorithm recognizing unambiguous CF languages on a CREW PRAM in time $\log^2 n$ with only n^3 processors. The similarity with our work is that the authors restrict the languages they accept to a well-behaved sub-set of CF to obtain sensible running time. In our opinion the present work captures better the actual sets of inputs found in the actual practice of CF parsing.

6.4.2 Too restrictive grammars

Rytter & Giancarlo (1987) analyze an algorithm which can parse a bracket grammar in $O(\log n)$ time and $O(n/\log n)$ processors. This is fast and does not use too many processors, but is restricted to languages where the grouping of non-terminals is completely explicit in the input: each production rule starts with an opening bracket and ends with a closing bracket. We also require some bracketing, but only for iteration constructs for which the input may contain a linear number of elements, in terms of the length of the input.

6.5 Automatic parallelization

Gibbons (1996) (following the work of Bird 1986) states that if a function can be expressed both as a leftwards and rightwards function (foldl and foldr), then it can also be expressed as a sequence homomorphism. Morita *et al.* (2007) use this theorem to derive such sequence homomorphism algorithmically. They present a tool which can produce a sequence homomorphism when given functions expressed both as foldl and foldr.

It would be interesting to check if the method could derive an efficient parallel parsing algorithm. As far as we understand, the method might (possibly with extensions) be able to discover the Valiant algorithm from a leftwards and a rightwards CYK algorithm. However, we think that discovering the sparse matrix representation is out of reach; this requires a creative step which is hard to capture in an automatic tool.

Mainstream parsing algorithms (such as LL(k) or LALR(k)) also seem hard to parallelize using an automatic method. First, it is not clear how one can reverse such a parser, because the definition of the algorithm is tightly coupled with direction of parsing (as their name indicates). Second, Morita *et al.* (2007) do not give an upper bound on the efficiency of the generated combination operator (*bin*), but only measure the performance of the generated code on a number of examples. As we understand there may be situations where the method produces an associative operator of linear (or worse) complexity, thereby yielding modest parallelization gains (if any).

6.6 Simultaneous incremental and parallel computation

Burckhardt *et al.* (2011) propose a model of computation which captures both incremental and parallel execution. Their model is based on concurrently running tasks which commit their results atomically upon completion. Our work is instead based on the well-known sequence homomorphism as model of parallel and incremental computation.

7 Discussion

7.1 Destructive updates

We were tempted to solve the problem of iteration by using destructive updates. That is, to make associative rules such as $Y ::= Y\,Y$ consume their arguments. That is, when a Y non-terminal is added to the chart using the above rule, the two Y non-terminals that compose it would be removed. This would have avoided the whole oracle construction. We have attempted this solution, but faced a couple of issues, which will not surprise an audience of functional programmers.

- On the theoretical side, reasoning about parsing with destructive updates of the chart has proven intractable. The generation relation describing which strings are recognized by such a parser is hard to define, let alone reason about. A major difficulty is to combine destructive updates with a notion of non-determinism similar to that described in Section 5. Indeed, the user has no control on which particular consuming rule will fire first, because the order depends on the particulars of the implementation of Valiant's algorithm (the order in which matrix multiplications are run, etc.) and the exact positioning of the sub-strings.

- On the practical side, the presence of updates makes for a more complicated implementation. It would also mean to abandon (so far unexploited) parallel opportunities in the matrix multiplication and the V function.

7.2 *Optimization*

In many grammars, a fair proportion of non-terminals occur only either on the left, or on the right of binary productions. Assume for example that A only ever occurs on the left. It is wasteful in this case to consider A for right-combinations, as does the algorithm we have presented so far.

This optimization is available to many CF parsing algorithms, but it is especially useful to us, because it acts in synergy with the detection of empty matrices. Indeed, by having separate matrices of left-combinable and right-combinable non-terminals, each matrix becomes sparser. This means that some combinations can be discarded *in blocks*, that is, at the level of matrices instead at the level of individual non-terminals.

An additional benefit of this optimization is that it pays for the cost of tagging non-terminals with an extra bit, as we describe in Section 5. Indeed, 0-tagged non-terminals occur only on the left of binary productions, and 1-tagged non-terminals occur only on the right in our encoding of iteration. Therefore this optimization eliminates all the cost of tagging: instead of tagging a non-terminal with a bit, it suffices to insert it only in the relevant matrix.

7.3 *Implementation*

An implementation of the parsing method presented here, including special support for iteration as presented in Section 5 and the optimization presented above, is implemented as a new back-end for the BNFC tool, (Forsberg & Ranta 2012) available in version 2.6, licensed under the GPL (Free Software Foundation 1991). The tool takes a grammar in BNF with annotations for efficient repetition. When running the tool with the option `--cnf`, it produces a Haskell implementation of CNF tables and an instance of the Valiant's algorithm using it. As other BNFC back-ends, our implementation produces full parsers, not mere recognizers.

7.4 *Unexploited parallelism*

The parallelization that we suggest can take advantage of at most a number of processors proportional to the length of the input. When parsing using Valiant's algorithm, there is more parallelism to take advantage of (for example two of the recursive calls in the V function are independent from each other). However, running in parallel all recursive calls to V would require asymptotically more processors than the length of the input. We do believe that this is *not* a reasonable assumption to make when parsing a whole input. However, in the case of incremental parsing, where only a tiny fraction of the input will be re-parsed, one might want to take advantage of such extra parallelism opportunities.

7.5 Unexploited incrementality

We have suggested that the incremental version of the parser should run the V function $O(\log n)$ times when changing one symbol in the input. In fact, it might be possible to use a better implementation of the chart data structure, which would support an incremental update with a single run of the V function. Indeed, when changing a single symbol of the input, only the part of the chart which depends on that symbol (the square whose bottom-left corner is the symbol in question) needs to be recomputed. This improved re-use of results is left for future work.

7.6 Chomsky normal-form

Even though we assume that we transform the grammar to CNF for ease of presentation, this is not actually the best form to use in an implementation. In fact, it is better to convert the grammar to 2NF (where productions have at most two symbols) and derive the operations $(\cdot)$ and σ using a slightly modified algorithm, using the method described by Lange & Leiß (2009), as we have done in our implementation.

The conversion from Backus-Naur Form (BNF) to CNF (or 2NF) involves a division of long productions into binary ones. This is usually done by chaining the binary rules linearly. If the productions of the input grammar are long, this impacts negatively the performance of our algorithm, which performs best on balanced inputs. Fortunately it is not difficult to divide long productions into a balanced tree of binary rules.

The CNF grammar is suitable not only for recognition of languages, but also for parsing: the parse trees obtained by the converted grammar are essentially a binarization of the trees obtained by the grammar in BNF. The aspect which cannot be preserved by the conversion is the presence of cycles of unit rules. However, the elimination of such cycles can only be seen as a benefit: they introduce an unbounded amount of ambiguity in the grammar, and are a symptom of a mistake in the grammar specification.

7.7 A new class of languages

The assumption we make on the input (depending on a constant α), defines implicitly a new class of languages. The class lies between regular and CF languages. We call the class α-balanced context-free languages, or BCF(α). The use of the parameter α contrasts with that of the parameter k in classes such as LL(k) or LR(k). While LL(k) or LR(k) restricts the form that a CF grammar can take, BCF(α) does not. Instead, it restricts the strings of the languages.

We have found that for a given grammar, programs are written with a shallow nesting structure, instead of a deep one (with the exception of regular iteration) and hence we have anecdotal evidence that any given programming language is a member of BCF(α), if we consider the language as the set of strings actually written in it by programmers. Together with observation that the parsing problem for BCF(α) has lower computational complexity than that of general-CF languages, this makes BCF(α) worthy of study.

In fact, because the assumption we make is not one which is enforced by usual CF grammars, but we still observe it to hold in practice, it must mean that the assumption is self-imposed by the writers of these inputs, namely programmers. This is not too surprising, as our assumption can be violated only by programs which exhibit an amount of nesting comparable to the total length of the input. As folklore goes, programmers are adverse to deeply-nested constructions. Indeed, understanding a program with n levels of nesting requires to remember n levels of context. The link between the ability for a computer to efficiently parse an input in parallel and incrementally and for a human to do so is intriguing, and we hope that the present paper sheds an interesting light on it.

7.8 Generalization

The body of the paper does not depend on the particulars of CF recognition: we abstract over it via an arbitrary association operator. This means that other applications can be devised. A natural extension is to support CF *parsing*, as we have done in our implementation. More exotic extensions are also possible. A first example would be to support symbol tables, which are for example necessary for proper parsing of C. In this extension, non-terminals would be associated with two symbol sets, one that they assume comes from the environment and one which they provide to the environment. The combination operator would reconcile these two sets. A second example is stochastic parsing. Here, a probability would be associated with each non-terminal and production rule, and the association operator would simply multiply the probabilities.

In fact, our method can be seen as a general way to turn a non-associative operator into an associative one by computing all possible associations. The efficiency is recovered by the ability to filter out most of the results; either because the original operator discards them, or because there is (possibly hidden) associativity which can be taken advantage of.

Yet another generalization of Valiant's algorithm produces a parser for Boolean grammars, as recently shown by Okhotin (2014). Boolean grammars allow to define the generation of non-terminals not only by union of production rules, but also intersection and complement. They can characterize non-CF languages, such as $\{a^n b^n c^n \mid n \in \mathbb{N}\}$. In this case, the ring-like structure that we have used is not sufficient: one must apply a Boolean function to all possible combination of non-terminals before obtaining the parses of a given sub-string.

7.9 The old as new

It strikes us that a parsing algorithm published in 1975 finds an application in the area of parallelization for computer architectures of the 2010 decade. Further, Valiant gives no indication that the algorithm described should find any practical parsing application. As it seems, he aims only to tie the complexity of CF recognition to that of matrix multiplication (via the transitive closure operation).

Indeed, in the case of parsing (in contrast to mere recognition), subtraction of matrices is not defined. Hence one cannot use the efficient Strassen algorithm (Strassen 1969) for multiplication, and in turn the complexity of general CF parsing using Valiant's method is cubic, and fails to beat the simpler CYK algorithm.

Our contribution is to recognize that Valiant's algorithm performs well for parsing practical inputs, given a special handling of iteration and a sparse matrix representation (even when using the naive matrix multiplication algorithm). If we also account for the ease of making parallel and incremental implementations of the algorithm thanks to its divide and conquer structure, we must classify Valiant's algorithm as a practical method of parsing.

In fact, Valiant's algorithm offers such a combination of simplicity and performance that we believe it deserves a prominent place in textbooks, on par with LALR algorithms.

8 Conclusions

At the start of this work, we set out to find an associative operator with sublinear complexity that could be used to implement a divide-and-conquer algorithm for parsing. The goal was to obtain a parallelizable parsing algorithm that would double as an incremental parsing algorithm. We managed to find such an operator, but the desired complexity only holds under certain assumptions that luckily do seem to hold in practice. The conditions hold when the recursive nesting depth of a program text only grows, say logarithmically in terms of the total length of the program. An unanticipated result of our work is thus the definition of a new class of languages. We were also forced to come up with a special way of dealing with iteration (frequently occurring in grammars) so it would not break this practical assumption.

Acknowledgments

The proof-method used in the presentation of Valiant's algorithm was suggested by Patrik Jansson. Engaging discussions about the complexity of Valiant algorithm were conducted with Devdatt Dubhashi. Peter Ljunglöf pointed us to some most relevant related work. Thomas Bååth Sjöblom, Darius Blasband, Peter Ljunglöf, as well as anonymous reviewers, gave useful feedback on drafts of this paper. This work has been partially funded by the Swedish Foundation for Strategic Research, under grant RAWFP.

References

Allison, L. (1992) Lazy dynamic-programming can be eager. *Inform. Process. Lett.* **43**(4), 207–212.

Bernardy, J.-P. (2008) Yi: An editor in Haskell for Haskell. In Proceedings of the 1st ACM SIGPLAN Symposium on Haskell. ACM, pp. 61–62.

Bernardy, J.-P. (2009) Lazy functional incremental parsing. In Proceedings of the 2nd ACM SIGPLAN Symposium on Haskell. ACM, pp. 49–60.

Bernardy, J.-P. and Claessen, K. (2013) Efficient divide-and-conquer parsing of practical context-free languages. In Proceedings of the 18th ACM SIGPLAN International Conference on Funct. Programming, pp. 111–122.

Bird, R. (1986) *An Introduction to the Theory of Lists*. Programming Research Group, Oxford University Comp. Laboratory.

Burckhardt, S., Leijen, D., Sadowski, C., Yi, J. & Ball, T. (2011) Two for the price of one: A model for parallel and incremental computation. In Proceedings of the 2011 ACM International Conference on Object Oriented Programming Systems Languages and Applications. ACM, pp. 427–444.

Chomsky, N. (1959) On certain formal properties of grammars. *Inform. Control* **2**(2), 137–167.

Chytil, M., Crochemore, M., Monien, B. & Rytter, W. (1991) On the parallel recognition of unambiguous context-free languages. *Theor. Comput. Sci.* **81**(2), 311–316.

Claessen, K. (2004) Parallel parsing processes. *J. Funct. Program.* **14**(6), 741–757.

Cocke, J. (1969) *Programming Languages and their Compilers: Preliminary Notes*. Courant Institute of Mathematical Sci., New York University.

Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2001) *Introduction to Algorithms*, 2nd ed. MIT press.

Forsberg, M. & Ranta, A. *BNFC Quick reference*, chapter Appendix A, London: College Publications, pp. 175–192.

Free Software Foundation. (1991) Gnu general public license.

Gibbons, J. (1996) The third homomorphism theorem. *J. Funct. Program.* **6**(4), 657–665.

Hinze, R. & Paterson, R. (2006) Finger trees: A simple general-purpose data structure. *J. Funct. Program.* **16**(2), 197–218.

Hughes, R. J. M. & Swierstra, S. D. (2003) Polish parsers, step by step. In Proceedings of the Eighth ACM SIGPLAN International Conference on Funct. Programming. ACM, pp. 239–248.

Kasami, T. (1965) *An Efficient Recognition and Syntax Analysis Algorithm for Context-Free Languages*. Technical Report, DTIC Document.

Lange, M. and Leiß, H. (2009) To CNF or not to CNF? An efficient yet presentable version of the CYK algorithm. *Inform. Didactica 8,* 2008–2010.

Morita, K., Morihata, A., Matsuzaki, K., Hu, Z. & Takeichi, M. (2007) Automatic inversion generates divide-and-conquer parallel programs. *ACM SIGPLAN Not.* **42**(6), 146–155.

Okhotin, A. (2014) Parsing by matrix multiplication generalized to boolean grammars. *Theor. Comput. Sci.* **516**(0), 101–120.

O'Sullivan, B. (2013) The Criterion benchmarking library.

Rytter, W. and Giancarlo, R. (1987) Optimal parallel parsing of bracket languages. *Theor. Comput. Sci.* **53**(2), 295–306.

Sikkel, K. and Nijholt, A. (1997) *Parsing of Context-Free Languages*. Berlin: Springer-Verlag, pp. 61–100.

Strassen, V. (1969) Gaussian elimination is not optimal. *Numer. Math.* **13**, 354–356. DOI: 10.1007/BF02165411.

Tomita, M. (1986) *Efficient Parsing for Natural Language*. Dordrecht: Kluwer Academic Publishers.

Valiant, L. (1975) General context-free recognition in less than cubic time. *J. Comput. Syst. Sci.* **10**(2), 308–314.

Wagner, T. A. and Graham, S. L. (1998) Efficient and flexible incremental parsing. *ACM Trans. Program. Lang. Syst.* **20**(5), 980–1013.

Younger, D. (1967) Recognition and parsing of context-free languages in time n^3. *Inform. Control* **10**(2), 189–208.

Appendix: C program fragment

```
__BEGIN_PROGRAM

void start_bandwidth_timer(struct hrtimer period_timer , int period)
{
  unsigned long delta;
  int soft, hard, now;

  for (;;) {
    if (hrtimer_active(period_timer))
      break;

    now = hrtimer_cb_get_time(period_timer);
    hrtimer_forward(period_timer, now, period);

    soft = hrtimer_get_softexpires(period_timer);
    hard = hrtimer_get_expires(period_timer);
    delta = into_ns(ktime_sub(hard, soft));
    hrtimer_start_range_ns(period_timer, soft, delta,
                                   HRTIMER_MODE_ABS_PINNED, 0);
  }
}


static void update_rq_clock_task(struct rq *rq, long delta);

void update_rq_clock(struct rq *rq)
{
  long delta;

  if (rq->skip_clock_update > 0)
    return;

  delta = sched_clock_cpu(cpu_of(rq)) - rq->clock;
  rq->clock += delta;
  update_rq_clock_task(rq, delta);
}

static int sched_feat_show(struct seq_file * m, void v)
{
  int i;

  for (i = 0; i < SCHED_FEAT_NR; i++) {
    if (!(sysctl_sched_features & (1 << i)))
      seq_puts(m, "NO_");
    seq_printf(m, "%s ", sched_feat_names[i]);
  }
  seq_puts(m, "\n");

  return 0;
}

__END_PROGRAM
```

Fragment of a C program corresponding to the chart in Figure 4. It is excerpt by hand from the linux kernel scheduler (beginning of the file `https://github.com/torvalds/linux/blob/master/kernel/sched/core.c`).