International Journal of Foundations of Computer Science

PARSING PARTIALLY ORDERED MULTISETS

TWAN BASTEN
Department of Mathematics and Computing Science, Eindhoven University of Technology,
PO Box 513, NL-5600 MB, Eindhoven, The Netherlands
email: tbasten@win.tue.nl

Received 2 November 1995
Revised 30 October 1996
Communicated by D. P. Bovet

ABSTRACT

A partially ordered multiset or pomset is a generalization of a string in which the total order has been relaxed to a partial order. Strings are often used as a model for sequential computation; pomsets are a natural model for parallel and distributed computation. By viewing pomsets as a generalization of strings, the question is raised whether concepts from language theory can be generalized to pomsets. An important area in the theory of languages is parsing theory. This paper develops the fundamentals of a parsing theory for pomsets, called *PLR* parsing. It is based on the *LR*-parsing technique, which is the most powerful deterministic parsing technique in language theory. The basic algorithm in the class of *PLR* parsing algorithms, the *PLR*(0) algorithm is explained in detail.

Keywords: partial orders – partially ordered multisets – pomset languages – pomset grammars – *LR* parsing – *PLR* parsing – concurrency theory

1. Introduction

Gischer introduces *partially ordered multisets* or *pomsets* as follows[5]:

> "Pomsets have been introduced as a model of concurrency. Since a pomset is a string in which the total order has been relaxed to be a partial order, in this paper we view them as a generalization of strings, ..."

The motivation for writing this paper could not have been formulated in a better way. As explained below, the research presented here originated from a problem in analyzing parallel and distributed programs. However, the presentation in this paper is mostly in a language-theoretical context. The reason being that we believe that one should first develop a correct theory, before trying to use it in actual applications. Thus, theoretical issues are separated from details that are only important in the context of some application.

The theory of strings and their languages has been investigated in great detail (see Ref. [11] among others). An important area of research has been the identification of meaningful classes of languages, such as regular and context free languages, ways to describe these classes of languages, such as regular expressions and grammars, and recognizers for these languages, such as automata and parsers. This paper generalizes some of these concepts from language theory to pomsets.

The main objective of this paper is to develop a parsing technique for pomsets. First, some basic definitions from pomset theory are summarized. The notion of pomset languages is introduced. Concatenation, choice, and Kleene star known from language theory are generalized to pomsets, and two concurrency operators are defined. These operators are used to extend regular expressions to so-called *concurrent regular expressions*. Concurrent regular expressions determine an important class of pomset languages called the *regular pomset languages*. Second, the notion of context-free grammars is generalized to *context-free pomset grammars*, or CFPGs. CFPGs determine another class of pomset languages, the *context-free pomset languages*. The generality of CFPGs, however, prohibits a straightforward adaptation of existing parsing techniques. Hence, the notion of *simple* CFPGs, or SCFPGs, is introduced. The class of languages generated by SCFPGs is strictly smaller than the class of context-free pomset languages, but strictly larger than the regular pomset languages. The format of the production rules of an SCFPG is such that it is relatively straightforward to generalize LR parsing to *pomset* LR parsing, or PLR parsing. The main part of this paper consists of describing the PLR parsing technique and formalizing the most basic algorithm in the class of PLR-parsing algorithms, namely the $PLR(0)$ algorithm.

The motivation for the research presented in this paper comes from the area of analyzing and debugging distributed and parallel programs. A lot of research effort in this area is put into the analysis of causal relationships between events in distributed computations. Researchers in this area generally adopt the partial-order model of parallel and distributed computations introduced by Lamport.[10] A typical question asked when analyzing the causality structure of a computation is whether some *causal pattern*, often called a *behavioral pattern*, occurs in the computation. Such behavioral patterns can be accurately described by pomsets. So algorithms for recognizing pomsets can be used to recognize occurrences of behavioral patterns in a distributed computation.

The automata-theoretic approach to recognizing behavioral patterns has already been investigated in some detail.[7] Unfortunately, the resulting formalism is fairly complex and suffers from some serious drawbacks (see Ref. [2, Section 4.0] for more details). Surprisingly, parsing techniques have not yet been investigated. Hence, this paper introduces a parsing technique for pomsets. This technique is not meant to be a ready-made solution to the problem of detecting behavioral patterns in distributed computations. It is an extension to and a generalization of language theory. It remains for future work to apply this technique to the analysis of distributed programs. Some of the issues that still need to be addressed can be found in Ref. [2, Section 4.5]. In the remainder, some small examples are given to illustrate the use of the concepts of this paper in analyzing causality in distributed computations. For the interested reader, a paper by Schwarz and Mattern[14] contains a good survey of recent results on this topic.

This section ends with a brief overview of the literature on pomsets. A formal definition of pomsets and pomset languages appears as early as the work of Winkowski[15,16] and Grabowski.[6] Both authors introduce pomsets as an algebraic characterization of non-sequential processes. Their main interest is the relationship between pomset languages and Petri nets. The notion of pomsets was introduced independently by Pratt.[12] Pratt is mainly interested in the axiomatizability and decidability of the equational theory of pomsets. Gischer addresses the issue of the axiomatizability of pomset languages and string languages extended with an interleaving, or shuffle, operator.[4,5] Finally, Pratt investigates the use of pomsets and pomset languages as a formalism for specifying parallel and distributed systems.[13]

The remainder of this paper is organized as follows. Section 2 gives a definition of pomsets, pomset languages, concurrent regular expressions, CFPGs, and SCFPGs. Section 3 gives an informal description of the PLR-parsing process. In Section 4, the basic algorithm in the class of PLR-parsing algorithms, the $PLR(0)$ algorithm, is formalized. Section 5 summarizes the results and discusses some open problems and directions for future work. Finally, Appendix A presents a correctness argument for the $PLR(0)$ algorithm.

2. Pomset-Language Theory

This section gives an introduction to the fundamentals of pomsets and their languages. The theory is developed analogously to the theory of string languages. Section 2.1 gives a definition of pomsets, pomset languages, and the class of regular pomset languages. Most of this subsection is based on work of Gischer[4,5] and Pratt.[13] Section 2.2 defines the notions of context-free pomset grammars and context-free pomset languages.

An initial remark is in order. The theory of pomset languages is still in an early stage of its development. This means that the terminology is not yet well established. The terminology used in this paper might differ from what other authors use. However, an attempt is made to develop a consistent and intuitively clear terminology.

2.1. Regular Pomset Languages

Definition 2.1. (lpo and pomset) A 4-tuple $(V, \Sigma, \leq, \mu)$ is a labeled partial order, or lpo, if and only if $(V, \leq)$ is a partial order and μ, the labeling function, is a mapping from the vertex set V to alphabet Σ.

Two labeled partial orders $M_0 = (V_0, \Sigma, \leq_0, \mu_0)$ and $M_1 = (V_1, \Sigma, \leq_1, \mu_1)$ are said to be *isomorphic* if and only if there is a bijective function $\tau : V_0 \longrightarrow V_1$ that preserves ordering and labeling. That is, for any $u, v \in V_0$, $u \leq_0 v \;\Leftrightarrow\; \tau.u \leq_1 \tau.v$ and $\mu_1.(\tau.u) = \mu_0.u$. A partially ordered multiset or pomset is the isomorphism class of some labeled partial order. Square brackets are used to denote pomsets. For example, $[V_0, \Sigma, \leq_0, \mu_0]$ is the pomset corresponding to M_0.

The set of all finite pomsets over alphabet Σ is denoted $\Sigma\ddagger$. This corresponds to the notation used by Pratt.[13] (In his work $\ddagger$ is a closure operator similar to the

Kleene star in language theory.) The vertex set of some labeled partial order or pomset M, is denoted V_M unless it is explicitly defined otherwise. The same naming convention is used for the alphabet, the ordering relation, and the labeling function. The notation ε is used to denote the special pomset $[\emptyset, \Sigma, \emptyset, \emptyset]$, for any alphabet Σ. Note that this corresponds to the notation traditionally used for the empty string. The notation u_A is used to denote the *unit pomset* $[\{x\}, \Sigma, \{(x,x)\}, \{(x,A)\}]$, for any A in some Σ.

Definition 2.2. (Pomset language) A pomset language over Σ is a set of pomsets over Σ.

Note that, since every string is also a pomset, every string language is also a pomset language.

In order to construct new pomsets from other pomsets and, subsequently, new pomset languages from other pomset languages, the following two operators on pomsets are introduced. The first one is sequential composition. This is a generalization of concatenation on strings. Two pomsets are joined such that every vertex of the first one precedes every vertex of the second. The second operator is concurrent composition. Two pomsets are joined without adding any order constraints. An example of these two operators is shown in Figure 1.

Definition 2.3. (Sequential and concurrent composition) Let $M_0 = [V_0, \Sigma_0, \leq_0, \mu_0]$ and $M_1 = [V_1, \Sigma_1, \leq_1, \mu_1]$ be pomsets. Without loss of generality, it is assumed that V_0 and V_1 are disjoint. Pomset $M_2 = [V_2, \Sigma_2, \leq_2, \mu_2]$ is the *sequential composition* of M_0 and M_1, denoted $M_0; M_1$, if and only if $V_2 = V_0 \cup V_1$, $\Sigma_2 = \Sigma_0 \cup \Sigma_1$, $\leq_2 = \leq_0 \cup \leq_1 \cup V_0 \times V_1$, and $\mu_2 = \mu_0 \cup \mu_1$. Pomset M_2 is the *concurrent composition* of M_0 and M_1, denoted $M_0 \& M_1$, if and only if $V_2 = V_0 \cup V_1$, $\Sigma_2 = \Sigma_0 \cup \Sigma_1$, $\leq_2 = \leq_0 \cup \leq_1$, and $\mu_2 = \mu_0 \cup \mu_1$.

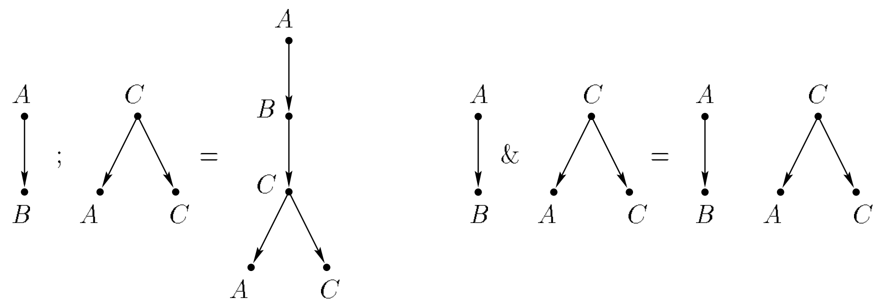


Fig. 1. Sequential and concurrent composition of pomsets.

If no misunderstanding is possible, sequential composition is not explicitly written.

The two operators defined above yield an important class of pomsets, called *series-parallel* pomsets. Gischer[5] defines this class as the smallest set of pomsets that contains the empty pomset ε and all unit pomsets, and is closed under the two composition operators.

The two composition operators defined on pomsets can easily be translated to equivalent operators on pomset languages. The same symbols are used as for pomsets. In addition, three new operators on pomset languages are introduced. Choice is defined as the union of two languages. Two closure operators are defined, one for sequential composition and one for concurrent composition.

Definition 2.4. (Composition, choice, and closure of pomset languages) Let L, L_0, and L_1 be pomset languages over alphabet Σ. Sequential composition $L_0; L_1 = \{M_0 M_1 \mid M_0 \in L_0 \wedge M_1 \in L_1\}$; concurrent composition $L_0 \& L_1 = \{M_0 \& M_1 \mid M_0 \in L_0 \wedge M_1 \in L_1\}$; choice $L_0 + L_1 = L_0 \cup L_1$; sequential closure $L^* = \{\varepsilon, L, LL, LLL, ...\}$; concurrent closure $L\dagger = \{\varepsilon, L, L\&L, L\&L\&L, ...\}$.

In the following, the unary operators have the strongest binding, followed by ;, &, and +.

The set of operators on pomset languages defined so far is sufficient to generalize the notion of regular expressions to *concurrent regular expressions.*

Definition 2.5. (Concurrent regular expressions) The syntax of concurrent regular expressions over an alphabet Σ is defined as follows:

$\alpha \ ::= \ \emptyset \mid A \mid \alpha;\alpha \mid \alpha\&\alpha \mid \alpha + \alpha \mid \alpha^* \mid \alpha\dagger$,

where A is some label in Σ.

As in language theory, where each regular expression defines a unique language, every concurrent regular expression defines a unique pomset language. The mapping from concurrent regular expressions to pomset languages is straightforward and, therefore, omitted.

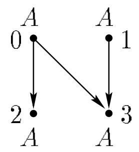


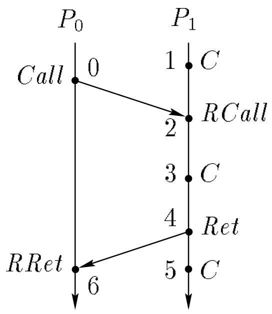
Fig. 2. Graphical representation of the pomset N.

The class of pomset languages defined by concurrent regular expressions is called the class of *regular pomset languages.* This class is a strict superset of the class of regular string languages. Note that by definition, a regular pomset language only contains series-parallel pomsets. The consequences for the expressive power of concurrent regular expressions can be best explained by means of the following pomset, which was first defined by Grabowski[6] and Gischer.[4] Define N as the pomset $[V_{\mathbf{N}}, \{A\}, \leq_{\mathbf{N}}, \mu_{\mathbf{N}}]$, where $V_{\mathbf{N}} = \{0,1,2,3\}$, $\leq_{\mathbf{N}}$ is the reflexive and transitive closure of $\{(0,2),(0,3),(1,3)\}$, and $\mu_{\mathbf{N}}.v = A$, for all $v \in V_{\mathbf{N}}$. This pomset gets its name from its shape that resembles an uppercase "N" (see Figure 2). A pomset is said to be N-free if and only if it does not have N as a subpomset (possibly after relabeling). It can be used to prove the following theorem. The proof is fairly elaborate and is, therefore, omitted. It can be found in Refs. [4,5]. A similar result is proven in Ref. [6].

Theorem 2.6. *A finite pomset is series-parallel if and only if it is* N*-free.*

Theorem 2.6 implies that a regular pomset language can only contain pomsets that are N-free.

Example 2.7. This section ends with a very simple application of the concepts developed in this section to the analysis of a distributed computation. Consider the following computation which shows a simple remote procedure call (RPC).



The computation consists of two processes P_0 and P_1, and seven events, uniquely identified by the natural numbers zero to six. Events are instantiations of *actions*. Event 0, for example, is an instantiation of action *Call* denoting a procedure call to (remote) process P_1. The other actions have the following meaning: C denotes a local *c*omputation; *RCall* stands for *r*eceive *call*; *Ret* means *ret*urn call and, finally, *RRet* is an abbreviation for *r*eceive *ret*urn. It is not difficult to see that the above pattern of a remote procedure call can be described by the concurrent regular expression: $(Call\&C); RCall; C; Ret; (RRet\&C)$. Note that, despite the asynchronous communication, the computation is N-free.

In this example, the concurrent regular expression describes the entire computation. However, in general, we want to use concurrent regular expressions as a specification language for certain interesting behavioral patterns that may or may not occur in a computation. An occurrence of such a patterns simply is a *sub*pomsets of the entire computation.

2.2. Context-Free Pomset Languages

In this subsection, some relevant definitions concerning context-free grammars are generalized to pomsets. First, the notion of vertex substitution on pomsets is defined. Vertex substitution is necessary to generalize the concept of a derivation to pomset grammars.

Definition 2.8. (Vertex substitution on pomsets[5,13]) Let $M = [V, \Sigma, \leq, \mu]$ be a pomset. A *vertex substitution* is a function f that maps vertices of M to pomsets. Without loss of generality, it is assumed that for every distinct v_0, v_1 in V, the sets $V_{f.v_0}$ and $V_{f.v_1}$ are mutually disjoint. Pomset $M[f]$ is defined as the pomset $[V_f, \Sigma_f, \leq_f, \mu_f]$, where $V_f = (\cup v : v \in V : V_{f.v})$, $\Sigma_f = (\cup v : v \in V : \Sigma_{f.v})$, $\leq_f = (\cup v : v \in V : \leq_{f.v}) \cup (\cup v_0, v_1 : v_0, v_1 \in V \wedge v_0 < v_1 : V_{f.v_0} \times V_{f.v_1})$, and $\mu_f = (\cup v : v \in V : \mu_{f.v})$.

An example of a vertex substitution is shown in Figure 3. For any pomset $[V, \Sigma, \leq, \mu]$, vertex $v \in V$, and pomset M, the vertex substitution $v := M$ is the substitution that assigns M to v and leaves all other vertices in V unchanged. Vertex

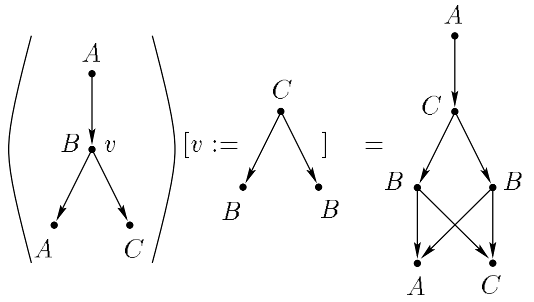


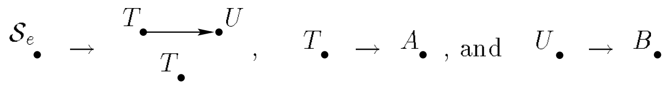
Fig. 3. An example of vertex substitution.

substitution generalizes the process of replacing a non-terminal by a string according to a production rule in a derivation based on a string grammar. The following definition defines context-free pomset grammars.

Definition 2.9. (Context-free pomset grammar, CFPG) A context-free pomset grammar is a 4-tuple $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$, where $\mathcal{N}$ is a finite, non-empty set of *non-terminals*, $\mathcal{T}$ is a finite, non-empty set of *terminals* such that $\mathcal{N}$ and $\mathcal{T}$ are disjoint, $\mathcal{S} \in \mathcal{N}$ is the start symbol, and $\mathcal{R} \subseteq \mathcal{N} \times \mathcal{V}\ddagger$, where $\mathcal{V}$ denotes the union of $\mathcal{N}$ and $\mathcal{T}$, is the set of *production rules*.

The difference between CFPGs and CFGs is that a production rule of a CFPG produces pomsets and a production rule of a CFG produces strings. Since a string is also a pomset, CFPGs are a true generalization of CFGs. A production rule (T, M) of a pomset grammar is written $T \rightarrow M$. Obviously, it is possible to give a graphical representation of a pomset grammar. A simple example of a pomset grammar is given in Example 2.10. Throughout the remainder of this paper, this grammar will be used as a running example.

Example 2.10. Consider the CFPG $\mathcal{G}_e = (\mathcal{N}_e, \mathcal{T}_e, \mathcal{R}_e, \mathcal{S}_e)$ where $\mathcal{N}_e = \{\mathcal{S}_e, T, U\}$, $\mathcal{T}_e = \{A, B\}$, and $\mathcal{R}_e = \{\mathcal{S}_e \rightarrow TU \,\&\, T, T \rightarrow A, U \rightarrow B\}$. The production rules can be represented graphically as follows. The left-hand and right-hand sides of each production rule consist of the graph of the corresponding pomset.



Pomset grammars generate pomset languages.

Definition 2.11. (Pomset language generated by a CFPG) For CFPG $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ the binary relation on $\mathcal{V}\ddagger^2$ *produces in one step*, denoted $\Longrightarrow$, is defined as follows. Let $M_0 = [V_0, \mathcal{V}, \leq_0, \mu_0]$ and M_1 be pomsets in $\mathcal{V}\ddagger$.

$M_0 \Longrightarrow M_1 \;\Leftrightarrow\; (\exists\, v, M : \; v \in V_0 \;\wedge\; \mu_0.v \rightarrow M \in \mathcal{R} : \; M_1 = M_0[v := M])$.

The reflexive, transitive closure of $\Longrightarrow$ is denoted $\stackrel{*}{\Longrightarrow}$. The pomset language generated by pomset M_0 in $\mathcal{V}\ddagger$, denoted $\mathcal{L}.M_0$, is defined as follows:

$\mathcal{L}.M_0 = \{M_1 \in \mathcal{T}\ddagger \mid M_0 \stackrel{*}{\Longrightarrow} M_1\}$.

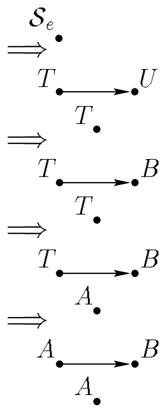
The pomset language generated by grammar $\mathcal{G}$, denoted $\mathcal{L}.\mathcal{G}$, is defined as the pomset language generated by the unit pomset labeled with the start symbol $\mathcal{S}$, i.e., $\mathcal{L}.\mathcal{G} = \mathcal{L}.u_{\mathcal{S}}$.

The class of pomset languages generated by CFPGs is called the class of *context-free pomset languages*. It is strictly larger than the class of regular pomset languages. This result follows from two theorems at the end of this section, where a third class of pomset languages is defined that is strictly larger than the regular pomset languages, but strictly smaller than the context-free pomset languages.

For a pomset M in the pomset language defined by some CFPG, a sequence of steps resulting in M is called a *derivation* for M.

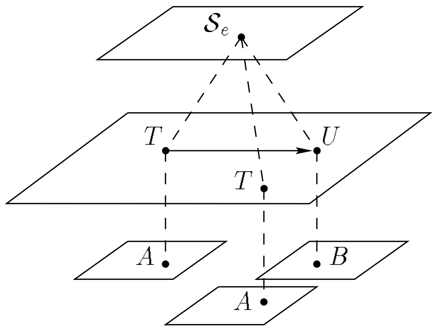
Example 2.12. Consider the CFPG $\mathcal{G}_e$ as defined in Example 2.10. It is easy to verify that $\mathcal{G}_e$ generates the pomset language $\{AB\&A\}$. A possible derivation for the pomset $AB\&A$ is as follows. On the right, the derivation is shown graphically.

$$
\begin{array}{l}
\quad \mathcal{S}_e \\
\Longrightarrow \; \{ \, \mathcal{S}_e \to TU\,\&T \, \} \\
\quad TU\&T \\
\Longrightarrow \; \{ \, U \to B \, \} \\
\quad TB\&T \\
\Longrightarrow \; \{ \, T \to A \, \} \\
\quad TB\&A \\
\Longrightarrow \; \{ \, T \to A \, \} \\
\quad AB\&A
\end{array}
$$



In general, there are many possible derivations for a pomset. In the derivation given above, for example, the last three steps can be done in any order. This means that there are six possible derivations for the pomset $AB\&A$. However, these derivations are not essentially different. Derivations for the same pomset that are not essentially different can be depicted in a so-called *pomset derivation tree* or *pomset parse tree*. Pomset parse trees are the generalization of parse trees in the theory of string languages.

Example 2.13. Consider again the CFPG $\mathcal{G}_e$. The pomset parse tree for the pomset $AB\&A$ is as follows.



The next definition gives two types of derivations that are of particular interest in the context of parsing.

Definition 2.14. (Leftmost and rightmost derivation) Let $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ be a CFPG. Consider a step $M_0 \Longrightarrow M_1$ of some derivation. Assume that $M_0 = [V_0, \mathcal{V}, \leq_0, \mu_0]$ and that $M_1 = M_0[v := M]$, for some $v \in V_0$ and some $M \in \mathcal{V}^\ddagger$. The derivation step is *leftmost* if and only if $(\forall\ v_0 :\ v_0 \in V_0 \wedge v_0 < v :\ \mu_0.v_0 \in \mathcal{T})$. It is *rightmost* if and only if $(\forall\ v_0 :\ v_0 \in V_0 \wedge v < v_0 :\ \mu_0.v_0 \in \mathcal{T})$. A derivation is called leftmost (rightmost) if and only if every step is leftmost (rightmost).

A derivation is leftmost (rightmost) if and only if in each step a vertex labeled with a non-terminal is substituted such that all predecessors (successors) of this vertex are labeled with terminals. This is the natural generalization of leftmost and rightmost derivations known from CFGs. Obviously, for context-free pomset grammars, there exist leftmost and rightmost derivations for every pomset generated by some pomset grammar. The derivation in Example 2.12 is a rightmost derivation. Leftmost and rightmost derivations in the theory of string languages are the basis for two well-known parsing techniques called LL parsing and LR parsing respectively. For any string in a language generated by an (unambiguous) CFG, there always exists a unique leftmost and a unique rightmost derivation. LL-parsing algorithms try to reconstruct the leftmost derivation. LR-parsing algorithms try to reconstruct the rightmost derivation. Due to the concurrency introduced in CFPGs, a leftmost or rightmost derivation for some pomset is not necessarily unique. Steps two and three of the derivation given in Example 2.12 can be reversed without violating the rightmost property. Interchanging the last two steps yields another rightmost derivation.

So far, it has been straightforward to generalize several well-known concepts from language theory to pomset theory. However, since the production rules of CFPGs are allowed to have arbitrary pomsets as their right-hand side, it is difficult to adapt existing parsing algorithms for CFPGs. Therefore, the class of CFPGs is restricted to the subclass of *simple* CFPGs. The right-hand side of a production rule of an SCFPG consists of two concurrent strings. Since the right-hand side of a production rule of a context-free string grammar consists of a single string, it is possible to adapt existing parsing algorithms.

Definition 2.15. (Simple context-free pomset grammar, SCFPG) A CFPG $(\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ is *simple* if and only if $\mathcal{R} \subseteq \mathcal{N} \times (\mathcal{V}^* \& \mathcal{V}^*)$.

Note that ordinary string grammars are also SCFPGs. A pomset language that is generated by an SCFPG is called a *simple context-free pomset language*. The pomset grammar of our running example, $\mathcal{G}_e$, is clearly an SCFPG. The next two theorems show that the class of simple context-free pomset languages is a non-trivial and meaningful class of pomset languages.

Theorem 2.16. *The class of regular pomset languages is a strict subset of the class of simple context-free pomset languages.*

Proof. First, it is shown that every regular pomset language is generated by some SCFPG. The proof is by induction on the structure of concurrent regular expressions. Let Σ be an arbitrary alphabet; let α and β be two regular pomset languages over Σ generated by SCFPGs $\mathcal{G}_0 = (\mathcal{N}_0, \Sigma, \mathcal{R}_0, \mathcal{S}_0)$ and $\mathcal{G}_1 = (\mathcal{N}_1, \Sigma, \mathcal{R}_1, \mathcal{S}_1)$ respectively. Without loss of generality, it may be assumed that $\mathcal{N}_0$ and $\mathcal{N}_1$ are disjoint. Let $\mathcal{S}$ be a new non-terminal.

The empty language: The SCFPG $(\{\mathcal{S}\}, \Sigma, \emptyset, \mathcal{S})$ clearly generates the language $\emptyset$.

Single label languages: For any A in Σ, the regular pomset language A is generated by the SCFPG $(\{\mathcal{S}\}, \{A\}, \{\mathcal{S} \to A\}, \mathcal{S})$.

Sequential composition: Pomset language $\alpha\beta$ is generated by the SCFPG $(\mathcal{N}_0 \cup \mathcal{N}_1 \cup \{\mathcal{S}\}, \Sigma, \mathcal{R}_0 \cup \mathcal{R}_1 \cup \{\mathcal{S} \to \mathcal{S}_0\mathcal{S}_1\}, \mathcal{S})$.

Concurrent composition: Pomset language $\alpha\&\beta$ is generated by the SCFPG $(\mathcal{N}_0 \cup \mathcal{N}_1 \cup \{\mathcal{S}\}, \Sigma, \mathcal{R}_0 \cup \mathcal{R}_1 \cup \{\mathcal{S} \to \mathcal{S}_0 \,\&\, \mathcal{S}_1\}, \mathcal{S})$.

Choice: Pomset language $\alpha + \beta$ is generated by the SCFPG $(\mathcal{N}_0 \cup \mathcal{N}_1 \cup \{\mathcal{S}\}, \Sigma, \mathcal{R}_0 \cup \mathcal{R}_1 \cup \{\mathcal{S} \to \mathcal{S}_0, \mathcal{S} \to \mathcal{S}_1\}, \mathcal{S})$.

Sequential closure: Pomset language α^* is generated by the SCFPG $(\mathcal{N}_0 \cup \{\mathcal{S}\}, \Sigma, \mathcal{R}_0 \cup \{\mathcal{S} \to \varepsilon, \mathcal{S} \to \mathcal{S}_0\mathcal{S}\}, \mathcal{S})$.

Concurrent closure: Pomset language $\alpha\dagger$ is generated by the SCFPG $(\mathcal{N}_0 \cup \{\mathcal{S}\}, \Sigma, \mathcal{R}_0 \cup \{\mathcal{S} \to \varepsilon, \mathcal{S} \to \mathcal{S}_0 \,\&\, \mathcal{S}\}, \mathcal{S})$.

Second, it must be shown that there are simple context-free pomset languages that are not regular. It is not hard to see that each non-regular *string language* that is generated by some context-free string grammar, which is an SCFPG, is a simple context-free but non-regular pomset language. □

Theorem 2.17. *A context-free pomset language is simple if and only if all its elements are* N*-free.*

Proof. First, assume a context-free pomset language is simple. This implies that it is generated by some SCFPG. It follows immediately from the format of the production rules of an SCFPG that the pomset language only contains series-parallel pomsets. Hence, it follows from Theorem 2.6 that all the pomsets in the language are N-free.

Second, assume that $\mathcal{G}$ is a CFPG that generates a pomset language which has only N-free elements. Consequently, we may assume that the right-hand sides of all production rules are N-free. (An N can appear in the right-hand side of a production rule, only if the production rule is never used in a derivation leading to an accepted pomset or if one of its four vertices is labeled with a non-terminal generating only the empty pomset. In both cases, it is straightforward to change the production rules of $\mathcal{G}$ such that they become N-free.) As a result of Theorem 2.6, this means that all right-hand sides of the production rules are series-parallel. Therefore, it is possible to transform $\mathcal{G}$ into an SCFPG that generates the same pomset language as follows. A production rule R of $\mathcal{G}$ may be of any of four forms. If R is of the form $T \to \varepsilon$ or $T \to u_A$, for some terminal or non-terminal A, then R already has the format of a production rule of an SCFPG. If R is of the form $T \to M_0M_1$, where M_0 and M_1 are arbitrary (series-parallel) pomsets, and R is not yet in the desired format, then R can be transformed by introducing two new non-terminals T_0 and T_1 and replacing R by $T \to T_0T_1$, $T_0 \to M_0$, and $T_1 \to M_1$. Of course, the last two production rules may need to be transformed into the desired format themselves before the result is an SCFPG. If R is of the form $T \to M_0 \,\&\, M_1$, then the transformation goes along the same lines. □

Example 2.18. Consider again Example 2.7. The regular pomset language containing the pattern of a remote procedure call can easily be described by an SCFPG. Such an SCFPG can be constructed from the concurrent regular expression in Example 2.7 by following the construction in the proof of Theorem 2.16. This construction is useful for the automatic recognition of occurrences of behavioral patterns specified by concurrent regular expressions. However, it yields in general unnecessarily complex grammars. Therefore, we do not follow this construction in this example. It is not difficult to see that the following SCFPG generates the pattern of an RPC:

$(\{\mathcal{S}, T, U, V\}, \{C, \mathit{Call}, \mathit{RCall}, \mathit{Ret}, \mathit{RRet}\}, \mathcal{R}, \mathcal{S})$,

where

$\mathcal{R} = \{\mathcal{S} \to T; U; V, T \to \mathit{Call} \,\&\, C, U \to \mathit{RCall}; C; \mathit{Ret}, V \to \mathit{RRet} \,\&\, C\}$.

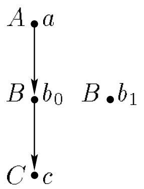
3. An Introduction to *PLR* Parsing

In the area of compiler design, many parsing algorithms have been developed. Well-known algorithms are those based on the *LL*- and *LR*-parsing techniques.[1,3] *LR* parsers are the most powerful and most general class of deterministic parsing algorithms. Therefore, it seems worthwhile investigating whether this technique can be generalized to (simple context-free) pomset languages.

LR parsing was first introduced by Knuth in 1965.[9] The name is derived from the fact that *LR* parsers scan the input from *left* to right and in doing so, try to reconstruct a *rightmost* derivation in reversed order. This section gives an introduction to *PLR* parsing. It shows that the basic operation is similar to the working of *LR* algorithms. In the next subsection, a suitable format of the input to *PLR*-parsing algorithms is discussed. In Section 3.2, the *PLR*-parsing process is described informally. In Section 4, this process is formalized.

3.1. The Input Format

Essentially, a pomset is a partial order plus a labeling function. A partial order can be represented by a set of pairs, consisting of a vertex and its immediate predecessors. Therefore, a pomset can be represented by a set of *triples* consisting of a vertex, its label, and its immediate predecessors. An example is given in Figure 4.



$ABC\&B$

Representation:
$(\emptyset, A, a)(\{a\}, B, b_0)(\emptyset, B, b_1)(\{b_0\}, C, c)$

Fig. 4. A simple representation of pomsets.

A pomset represented by a list of triples can be read easily by parsing algorithms. However, two more conditions must be satisfied. First, it must be guaranteed that before reading a triple, the algorithm has already read all the vertices in the set of predecessors. Second, the algorithm must be able to detect the end of the input.

To fulfill the first condition, the order in which the triples are read must be such that the total ordering imposed on the vertices is a linearization of the pomset.

Definition 3.1. (Linearization of poset, lpo, and pomset) Let $(V, \leq)$ be a poset. The pair $(V, \leq_t)$ is a linearization of $(V, \leq)$ if and only if, for any $u, v \in V$, $u \leq v \;\Rightarrow\; u \leq_t v$ and $\leq_t$ is a total ordering of V. Let $M = (V, \Sigma, \leq, \mu)$ be an lpo. The lpo $(V, \Sigma, \leq_t, \mu)$ is a linearization of M if and only if $(V, \leq_t)$ is a linearization of $(V, \leq)$. The definition for pomsets is analogous to the one for lpos.

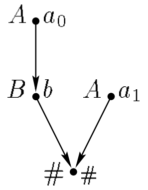
The second condition can be fulfilled by adding a special end marker to the input. Let M be the pomset to be recognized and let $\#$ be a label not in Σ_M. The input to a *PLR*-parsing algorithm is a sequence representing the pomset $M\#$ that satisfies the first condition. The end of such a sequence is marked by a special triple $(\mathit{Maxv}.M, \#, \#)$, where $\#$, labeled with $\#$, is a new vertex not in V_M and where the function *Maxv* is defined as follows.

Definition 3.2. (Maximal vertices of a pomset) The function *Maxv* yields the vertices of a pomset without any successors.

$\mathit{Maxv}.M = \{v_0 \in V_M \mid (\forall\ v_1 :\ v_1 \in V_M :\ v_0 \not< v_1)\}$,

for any pomset M.

Example 3.3. Assume we wish to determine whether the pomset $AB\&A$ belongs to the pomset language generated by $\mathcal{G}_e$, the SCFPG of Example 2.10. The input to the recognition algorithm is a representation of the pomset $(AB\&A)\#$. The order in which the representation is read can be any of the three possibilities below.



Possible Inputs:
$(\emptyset, A, a_1)(\emptyset, A, a_0)(\{a_0\}, B, b)(\{a_1, b\}, \#, \#)$
$(\emptyset, A, a_0)(\emptyset, A, a_1)(\{a_0\}, B, b)(\{a_1, b\}, \#, \#)$
$(\emptyset, A, a_0)(\{a_0\}, B, b)(\emptyset, A, a_1)(\{a_1, b\}, \#, \#)$

3.2. An Informal Description of the PLR-Parsing Process

Just like *LR*-parsing algorithms, a *PLR*-parsing algorithm reads input from left to right. In the previous subsection, a special end marker was introduced to mark the end of the input sequence. A pomset grammar has to be adapted accordingly, before it can be used by a *PLR*-parsing algorithm.

Definition 3.4. (Augmented pomset grammar) If $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ is a pomset grammar, then the corresponding *augmented* pomset grammar $\mathcal{G}' = (\mathcal{N}', \mathcal{T}', \mathcal{R}', \mathcal{S}')$ is the pomset grammar $(\mathcal{N} \cup \{\mathcal{S}'\}, \mathcal{T} \cup \{\#\}, \mathcal{R} \cup \{\mathcal{S}' \rightarrow \mathcal{S}\#\}, \mathcal{S}')$.

Note that this definition is the same as for ordinary grammars. Obviously, some pomset M is an element of the pomset language generated by $\mathcal{G}$ if and only if the pomset $M\#$ is an element of the pomset language generated by the augmented pomset grammar $\mathcal{G}'$. An augmented grammar has the advantage that there is a unique production rule with the start symbol as the left-hand side.

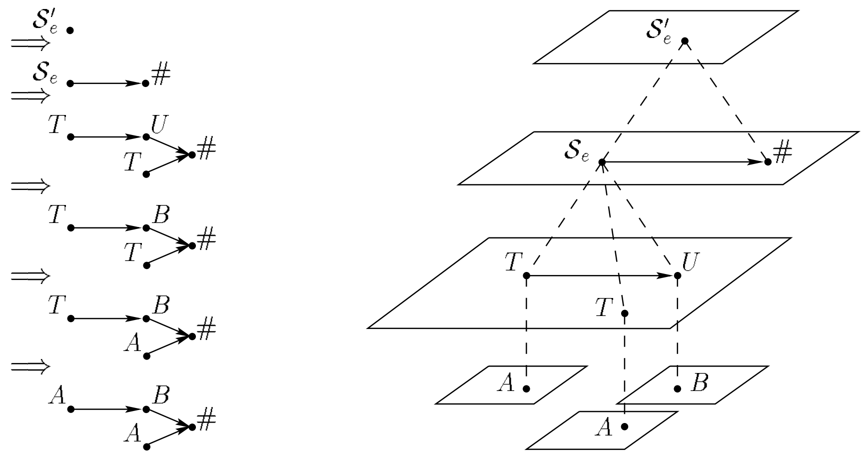
In order to improve understanding of the operation of *PLR* algorithms, it is useful to make a comparison with *LR*-parsing algorithms. *LR*-parsing algorithms are so-called bottom-up parsers. That is, an *LR*-parsing algorithm tries to reconstruct a rightmost derivation in reversed order and, doing so, it essentially builds a parse tree starting from the leaves and working its way up to the root. If the algorithm succeeds, the input is recognized. Otherwise, the algorithm either could not make a decision or recognized an error.

A *PLR* algorithm is also a bottom-up parser. It also tries to reconstruct a rightmost derivation. During the process a *pomset parse tree* is constructed starting from the leaves. Again, only if the tree can be constructed all the way up to the root, is an input recognized successfully.

LR parsers are members of the family of *Shift-Reduce* parsers. *Shift-Reduce* parsers essentially work as follows. A *stack* contains (terminal and non-terminal) symbols that have already been parsed. Input symbols are read and *shifted* onto the stack until the right-hand side of some production rule is matched. Then, the parser executes a *reduce* action. The symbols that match the right-hand side are removed from the stack and the left-hand side of the production rule, which is the parent node in the parse tree, is pushed on top of the stack so that it can be used for future reduce actions. Every reduce action corresponds to a step in a rightmost derivation for the input. If the input is read completely and the stack contains only the start symbol, then the input is recognized successfully. This essentially describes the operation of any *Shift-Reduce* parser. The details vary for every member of the *Shift-Reduce* family. The stack usually contains some encoding of the symbols already parsed instead of the symbols themselves. Furthermore, every *Shift-Reduce* parser has a different way of determining what the next parse action should be.

The $PLR(0)$-parsing algorithm introduced in the next section is also a *Shift-Reduce* parser. The following example is a visualization of the *PLR*-parsing process.

Example 3.5. The augmented SCFPG $\mathcal{G}'_e$ is the pomset grammar $(\mathcal{N}'_e, \mathcal{T}'_e, \mathcal{R}'_e, \mathcal{S}'_e)$ where $\mathcal{N}'_e = \{\mathcal{S}'_e, \mathcal{S}_e, T, U\}$, $\mathcal{T}'_e = \{A, B, \#\}$, and $\mathcal{R}'_e = \{\mathcal{S}'_e \to \mathcal{S}_e\#, \mathcal{S}_e \to TU \,\&\, T, T \to A, U \to B\}$. The only element of the pomset language generated by $\mathcal{G}'_e$ is the pomset $(AB\&A)\#$. One of its derivations and its pomset parse tree are as follows.



Note that the derivation above is one of three different rightmost derivations for the pomset $(AB\&A)\#$. Every rightmost derivation corresponds to exactly one of the three inputs given in Example 3.3. A *PLR*-parsing algorithm for $\mathcal{G}_e$ that gets the second input reconstructs the derivation above in reversed order as follows. It is easy to see that the particular linearization of the input pomset determines which rightmost derivation is reconstructed. The steps in the derivation below are numbered for future reference. Note that a is used to denote the singleton $\{a\}$.

$$
\begin{array}{ll}
& (\emptyset, A, a_0)(\emptyset, A, a_1)(a_0, B, b)(\{a_1, b\}, \#, \#) \\
\Longleftarrow & \{\ (v): \quad T \rightarrow A\ \} \\
& (\emptyset, T, a_0)(\emptyset, A, a_1)(a_0, B, b)(\{a_1, b\}, \#, \#) \\
\Longleftarrow & \{\ (iv): \quad T \rightarrow A\ \} \\
& (\emptyset, T, a_0)(\emptyset, T, a_1)(a_0, B, b)(\{a_1, b\}, \#, \#) \\
\Longleftarrow & \{\ (iii): \quad U \rightarrow B\ \} \\
& (\emptyset, T, a_0)(\emptyset, T, a_1)(a_0, U, b)(\{a_1, b\}, \#, \#) \\
\Longleftarrow & \{\ (ii): \quad \mathcal{S}_e \rightarrow TU\,\&\,T\ \} \\
& (\emptyset, \mathcal{S}_e, \{a_1, b\})(\{a_1, b\}, \#, \#) \\
\Longleftarrow & \{\ (i): \quad \mathcal{S}'_e \rightarrow \mathcal{S}_e\#\ \} \\
& (\emptyset, \mathcal{S}'_e, \#)
\end{array}
$$

The meaning of the input triples in this derivation is clear. The triples with a *non-terminal* as the second element can be interpreted as follows. The non-terminal is an identifier of a pomset whose immediate predecessors are defined by the first element of the triple and whose set of maximal vertices is the third element of the triple. If a set of triples represents a pomset that matches the right-hand side of a production rule, this set can be replaced by a single triple. Since the input is an unambiguous representation of the pomset $(AB\&A)\#$ and the derivation is rightmost, the process outlined above is unique. The actual parsing process is depicted in Figure 5.

The process consists of nine steps, numbered (0) to (8). They should be read from bottom to top. The state of the parsing process before and after every step is described by the contents of the stack and the remainder of the input. The pictures in the middle represent the stack contents. The sequences on the right are the remainder of the input. Every time an input is read, it is shifted onto the stack. That is, it is added to the figure in the middle. If the right-hand side of a production rule is matched, a reduce action is executed. The matching symbols are removed from the picture and replaced by a triple with the left-hand side of the production rule as its second element and its first and third element as described above. This depicts a reduce action. Every reduce action corresponds to a step in the derivation above. The corresponding number is added in the leftmost column. If the whole figure is compressed in the vertical direction, it strongly resembles the pomset parse tree above.

This example concludes the introduction to the *PLR*-parsing process. In the next section, the *PLR*(0) algorithm is introduced that essentially operates as explained above.

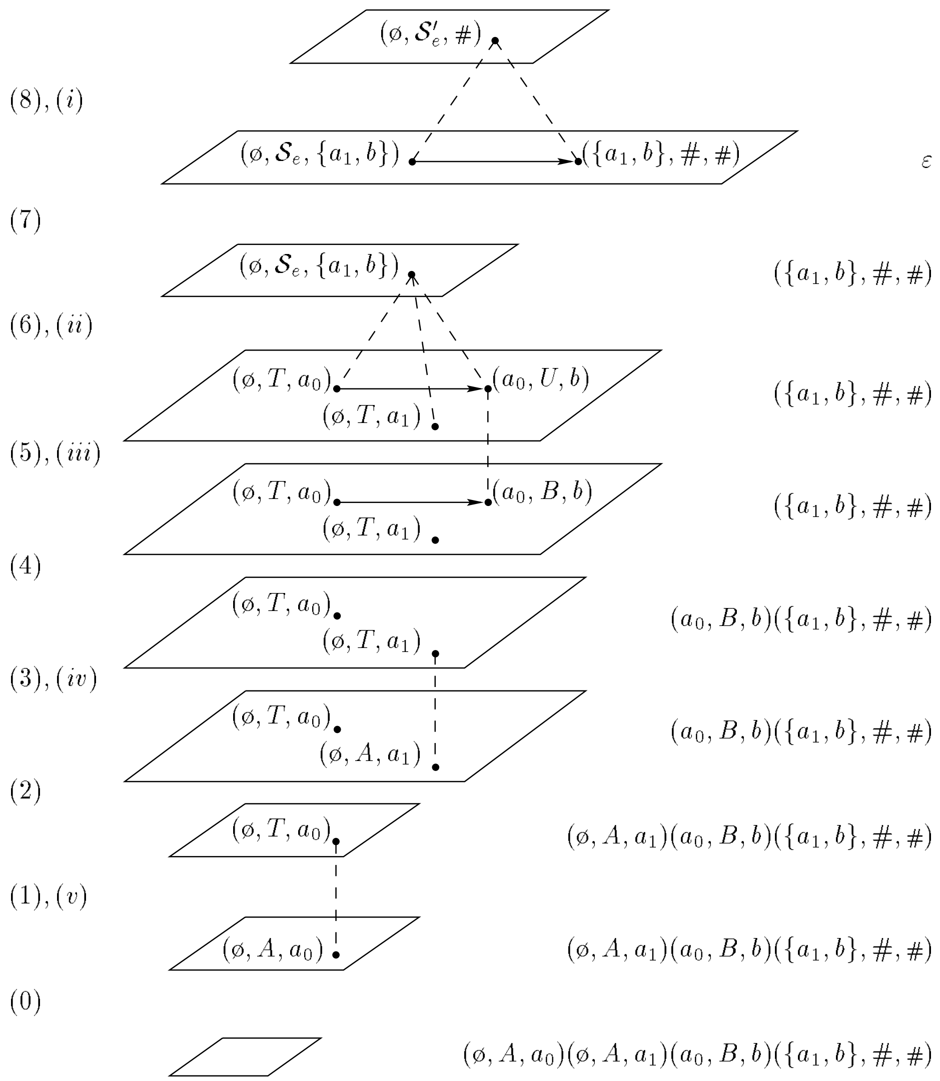


Fig. 5. A visualization of the *PLR*-parsing process.

4. *PLR*(0) Parsing

LR parsers can be characterized by the number of lookahead symbols used to determine parsing actions. If an *LR* parser uses the next k input symbols, it is called an $LR(k)$ parser. $LR(0)$ is the simplest form of *LR* parsing. It does not use lookahead symbols. It is the basis for all the other *LR*-parsing algorithms. Therefore, in this section, $LR(0)$ parsing is generalized to $PLR(0)$ parsing which is the simplest form of *PLR* parsing. In Section 4.1, the process described in the previous section is formalized for simple context-free pomset languages and in case no lookahead symbols are used. In Ref. [2, Appendix B], it is shown that the $PLR(0)$ algorithm correctly decides whether an input belongs to some simple context-free pomset language. An outline of this proof is given in Appendix A. Section 4.2 investigates some grammar constructs for which the $PLR(0)$ algorithm cannot always make a decision. As may be expected from results on $LR(0)$ parsing, there are many such constructs. Therefore, it must be studied how to incorporate lookahead information into the theory of *PLR* parsing. In Section 4.2, we briefly return to this point.

4.1. The PLR(0)-Parsing Algorithm

In this subsection, it is assumed that $\mathcal{G} = (\mathcal{N}, \mathcal{T}, \mathcal{R}, \mathcal{S})$ is an SCFPG and that $M = [V_M, \mathcal{T}, \leq_M, \mu_M]$ is a pomset in $\mathcal{T}\ddagger$. Furthermore, let the sequence $w' = w(Maxv.M, \#, \#)$ be a representation of $M\#$ as defined in the previous section. The notation V_M' is used to denote $V_M \cup \{\#\}$. Finally, the type *input* is defined as $2^{V_M} \times \mathcal{T}' \times V_M'$.

$PLR(0)$ parsing, like *LR*-parsing methods, is based on the idea of *items*. Items encode information about the state of the parse. An item corresponds to a production rule. Since we only consider SCFPGs, it is straightforward to extend the notion of $LR(0)$ items. A $PLR(0)$ item divides each string in the right-hand side of the corresponding production rule into two parts: one part that has been recognized already and one part still to be recognized. To keep track of predecessor information, sets of vertices from the input are added. A more detailed explanation is given below.

Definition 4.1. (*PLR*(0) item) The set of $PLR(0)$ *items*, or *items* for short, is defined as follows:

$$\begin{aligned} item = \{&(W, T, \sigma_0, X, \sigma_1, \varepsilon, \emptyset, \varepsilon) \mid T \to \sigma_0\sigma_1 \in \mathcal{R}' \ \wedge \ W, X \subseteq V_M'\} \ \cup \\ \{&(W, T, \sigma_0, X, \sigma_1, \tau_0, Y, \tau_1) \mid T \to \sigma_0\sigma_1 \,\&\, \tau_0\tau_1 \in \mathcal{R}' \backslash (\mathcal{N}' \times \mathcal{V}'^*) \ \wedge \\ &\qquad\qquad W, X, Y \subseteq V_M'\}. \end{aligned}$$

Before any further explanation is given, some notation is introduced that will be used throughout the remainder of this section. The following notational conventions are used to denote elements of different types.

a, b, c	$: V_M'$	(vertices in the input),
W, X, Y, Z	$: 2^{V_M'}$	(sets of vertices in the input),
T, U	$: \mathcal{N}'$	(non-terminals),
A, B, C	$: \mathcal{T}'$	(terminals),
V	$: \mathcal{V}'$	(terminal or non-terminal),

σ, τ	$: \mathcal{V}'^*$	(strings of terminals and non-terminals),
I, J	$: 2^{item}$	(sets of items),
ι, κ	$: (2^{item})^*$	(sequences of sets of items),
ω	$: input^*$	(sequence of input triples).

Subscripts are used if the notation introduced above does not suffice. Note that lowercase characters are used to denote vertices, uppercase characters are used to denote sets, terminals, and non-terminals, and Greek characters to denote strings or sequences. Another simplification that is used to improve readability is that a singleton is denoted by its single element. Finally, an item $(W, T, \sigma_0, X, \sigma_1, \tau_0, Y, \tau_1)$ is written as $W \cdot T \rightarrow \sigma_0 \cdot X \cdot \sigma_1 \,\&\, \tau_0 \cdot Y \cdot \tau_1$. Since concurrent composition is commutative, items $W \cdot T \rightarrow \sigma_0 \cdot X \cdot \sigma_1 \,\&\, \tau_0 \cdot Y \cdot \tau_1$ and $W \cdot T \rightarrow \tau_0 \cdot Y \cdot \tau_1 \,\&\, \sigma_0 \cdot X \cdot \sigma_1$ are not distinguished. If τ_0 and τ_1 are both equal to ε and thus Y is empty, the item is written as $W \cdot T \rightarrow \sigma_0 \cdot X \cdot \sigma_1$.

An item $W \cdot T \rightarrow \sigma_0 \cdot X \cdot \sigma_1 \,\&\, \tau_0 \cdot Y \cdot \tau_1$ provides information about the state of a parse. The parts σ_0 and τ_0 are the parts that have been matched already. The parts σ_1 and τ_1 are the parts still to be matched. The vertex sets X and Y can be interpreted as the immediate predecessors of σ_1 and τ_1 respectively. An item can be seen as a goal in the parsing process. The next symbol to be matched can be either a terminal or a non-terminal.

If the next symbol to be matched is a terminal, it has to be matched by an input. Thus the sets X and Y specify the possible sets of immediate predecessors of an input triple. Such an item suggests that the next parsing action be a *shift* action.

If the next symbol is a non-terminal, it must be matched by the recognition of a production rule. A production rule $T \rightarrow \sigma \,\&\, \tau$ is *recognized* by an item $W \cdot T \rightarrow \sigma \cdot X \cdot \,\&\, \tau \cdot Y \cdot$. A goal is fulfilled. Such an item suggests that the next action be a *reduce*. The set W can be interpreted as the set of immediate predecessors of the non-terminal T. The union of X and Y is the set of maximal vertices of the pomset that matched the non-terminal T.

If an item is of the format $W \cdot T \rightarrow \cdot W \cdot \sigma \,\&\, \cdot W \cdot \tau$, it is said to *predict* the production rule $T \rightarrow \sigma \,\&\, \tau$. The set W specifies the set of immediate predecessors for the non-terminal T. The item expresses that, in order to recognize T with W as its immediate predecessors, it is necessary to recognize both σ and τ with immediate predecessors W.

Shift-Reduce parsers, and therefore also the $PLR(0)$ algorithm, maintain a stack to store (an encoding of) the state of the parse. The stack of the $PLR(0)$-parsing algorithm contains *sets of items*. By construction, the top of the stack contains a set from which all items can be derived that may be applicable at a given point in the parse. For example, initially, the goal is to recognize the start symbol $\mathcal{S}'$. This means that besides the item $\emptyset \cdot \mathcal{S}' \rightarrow \cdot \emptyset \cdot \mathcal{S}\#$ also all items that predict a production rule with left-hand side $\mathcal{S}$ may be applicable. The following function is helpful in describing the state of a parse.

Definition 4.2. (*PLR*(0) **closure**) For any set of items I, the closure of I, denoted by $\overline{I}$, is the smallest set of items that satisfies the following three conditions:

(i) $I \subseteq \overline{I}$,

(ii) If $W \cdot T \rightarrow \sigma_0 \cdot X \cdot U\sigma_1 \,\&\, \tau_0 \cdot Y \cdot \tau_1 \in \overline{I}$ and $U \rightarrow \sigma \in \mathcal{R}'$, then $X \cdot U \rightarrow \cdot X \cdot \sigma \in \overline{I}$,

(iii) If $W \cdot T \rightarrow \sigma_0 \cdot X \cdot U\sigma_1 \,\&\, \tau_0 \cdot Y \cdot \tau_1 \in \overline{I}$ and $U \rightarrow \sigma \,\&\, \tau \in \mathcal{R}' \backslash (\mathcal{N}' \times \mathcal{V}'^*)$, then $X \cdot U \rightarrow \cdot X \cdot \sigma \,\&\, \cdot X \cdot \tau \in \overline{I}$.

Using this definition, it is sufficient that the top of the stack contains a set of items such that the closure of this set is the current state of the parse. Initially, the top of the stack of a $PLR(0)$ algorithm contains the singleton $\{\emptyset \cdot \mathcal{S}' \rightarrow \cdot \emptyset \cdot \mathcal{S}\#\}$. The initial state is the set of items $\overline{\{\emptyset \cdot \mathcal{S}' \rightarrow \cdot \emptyset \cdot \mathcal{S}\#\}}$.

Example 4.3. The initial state of the $PLR(0)$ algorithm for the SCFPG $\mathcal{G}_e$ is the set of items $\{\mathcal{I}_e,\ \emptyset \cdot \mathcal{S}_e \rightarrow \cdot \emptyset \cdot TU \,\&\, \cdot \emptyset \cdot T,\ \emptyset \cdot T \rightarrow \cdot \emptyset \cdot A\}$, where $\mathcal{I}_e$ is the initial item $\emptyset \cdot \mathcal{S}'_e \rightarrow \cdot \emptyset \cdot \mathcal{S}_e\#$. The stack contains the singleton $\{\mathcal{I}_e\}$.

As explained, the $PLR(0)$ algorithm does not use lookahead symbols to determine the next parse action; it only uses the top of the stack. If the state of a parse contains an item recognizing a production, the parser executes a *reduce* action. If the set of items contains items whose next symbol to match is a terminal, then the next action is a *shift* action. The action function defines a set of possible actions based on only the current state of the parse. A reduce action is defined by the completed item that recognizes a production.

Definition 4.4. ($PLR(0)$-action function) Function $Act : 2^{item} \rightarrow 2^{item \cup \{Shift\}}$ is defined as follows:

$$\begin{aligned} Act.I = \{&W \cdot T \rightarrow \sigma \cdot X \cdot \,\&\, \tau \cdot Y \cdot \mid W \cdot T \rightarrow \sigma \cdot X \cdot \,\&\, \tau \cdot Y \cdot \in \overline{I}\} \cup \\ \{&\mathit{Shift} \mid W \cdot T \rightarrow \sigma_0 \cdot X \cdot A\sigma_1 \,\&\, \tau_0 \cdot Y \cdot \tau_1 \in \overline{I}\}, \end{aligned}$$

for any $I \subseteq item$. Note that in this definition, τ, τ_0, and τ_1 are allowed to be the empty string. Thus, a case analysis as in Definitions 4.1 and 4.2 is not necessary.

Example 4.5. Since the initial state of the $PLR(0)$ algorithm for the grammar $\mathcal{G}_e$ contains the item $\emptyset \cdot T \rightarrow \cdot \emptyset \cdot A$ (see Example 4.3), the first action of the $PLR(0)$ algorithm is a shift.

If at any point during the parse the set of possible actions is empty, an error has occurred; the input is not an element of the pomset language generated by the SCFPG. If at any point during the parse there is more than one possible action, the parser cannot decide what to do next. If more than one item recognizes a production, it is said that a *reduce-reduce* conflict has occurred. If both a shift and a reduce are possible actions, a *shift-reduce* conflict has occurred. A pomset grammar is $PLR(0)$ for an input pomset if and only if at any point during the parse the next action is defined uniquely. The pomset grammar is $PLR(0)$ if and only if it is $PLR(0)$ for any input.

The next step in the formalization of the $PLR(0)$-parsing process is to define the so-called $PLR(0)$-goto relation, or simply goto relation, that determines the next state of the parse. This state is uniquely determined by the current state, which is a set of items, and a triple of type *label*, which is defined as $2^{V_M} \times \mathcal{V}' \times 2^{V'_M}$.

An element of type *label* can be interpreted as a terminal or non-terminal with the set of its immediate predecessors and the set of its maximal vertices. It is either an input triple, in case of a shift action, or a triple determined by a completed item, in case of a reduce action. Note that every input triple determines a unique element of type *label*. Type *label* contains the triples that are introduced informally in the previous section. The goto relation describes the actual execution of a parse action. It moves the matched symbol from the part of the item that still needs to be matched to the part that has already been recognized.

Definition 4.6. **($PLR(0)$-goto relation)** The ternary relation $\cdot - \cdot \rightarrow \cdot : 2^{item} \times label \times 2^{item}$ is defined as follows:

$$I - (X, V, Y) \rightarrow J$$
$$\Leftrightarrow$$
$$J = \{W \cdot T \rightarrow \sigma_0 V \cdot Y \cdot \sigma_1 \,\&\, \tau_0 \cdot Z \cdot \tau_1 \mid W \cdot T \rightarrow \sigma_0 \cdot X \cdot V \sigma_1 \,\&\, \tau_0 \cdot Z \cdot \tau_1 \in \overline{I}\},$$

for any $I, J \subseteq item$ and $(X, V, Y) \in label$.

Example 4.7. If the first input read by the $PLR(0)$-parsing algorithm for the SCFPG $\mathcal{G}_e$ is the triple $(\emptyset, A, a_0)$, then the next state of the parse is defined by $\mathcal{I}_e - (\emptyset, A, a_0) \rightarrow \{\emptyset \cdot T \rightarrow A \cdot a_0 \cdot\}$. The next state is the closure of the singleton $\{\emptyset \cdot T \rightarrow A \cdot a_0 \cdot\}$, which is the singleton itself. At this point, the algorithm has matched non-terminal T. The next action is a reduce.

The parsing process visualized in Figure 5 can now be formalized by the following relation which is defined on *configurations*. A configuration is a pair in $(2^{item})^* \times input^*$. It describes the state of the entire parsing process: the contents of the stack and the remainder of the input.

Definition 4.8. The binary relation $\cdot \vdash \cdot$ on configurations is defined as follows:

$$(\iota I, (Z, A, a)\omega) \vdash (\iota I J, \omega) \;\Leftrightarrow\; Act.I = \{Shift\} \;\wedge\; I - (Z, A, a) \rightarrow J, \text{ and}$$
$$(\iota I \kappa, \omega) \vdash (\iota I J, \omega) \;\Leftrightarrow\; I\kappa = \iota_0 I_0 \;\wedge\; Act.I_0 = \{W \cdot T \rightarrow \sigma \cdot X \cdot \& \tau \cdot Y \cdot\} \;\wedge\; |\kappa| = |\sigma| + |\tau| \;\wedge\; I - (W, T, X \cup Y) \rightarrow J,$$

where the function $|\cdot|$ denotes the length of a string. The reflexive transitive closure of $\vdash$ is denoted $\overset{*}{\vdash}$.

The relation $\vdash$ defines the next configuration of the parsing process for the two cases of shift and reduce actions. In case of a reduce, the number of sets that must be removed from the stack is determined by the number of symbols in the right-hand side of the recognized production. This is not surprising, since every matched symbol causes a change of the parse state which translates to a set of items on the stack.

Only one issue remains. When is an input accepted by the $PLR(0)$ algorithm? That is, what is the pomset language recognized by the $PLR(0)$ algorithm? An input sequence is accepted by the algorithm if and only if, after reading all the input, the production $\mathcal{S}' \rightarrow \mathcal{S}\#$ is recognized and thus the initial goal is fulfilled. The pomset language recognized by the $PLR(0)$ algorithm contains all the inputs accepted. This can be easily formalized using the definition above. Let $\mathcal{I}_0$ be the initial item $\emptyset \cdot \mathcal{S}' \rightarrow \cdot \emptyset \cdot S\#$. Recall that M is a pomset represented by w and that $w' = w(Maxv.M, \#, \#)$.

Definition 4.9. (Language recognized by a $PLR(0)$ parser) The language recognized by a $PLR(0)$ parser for SCFPG $\mathcal{G}$, denoted $\mathcal{L}_{PLR(0)}.\mathcal{G}$, is defined as follows:

$$M \in \mathcal{L}_{PLR(0)}.\mathcal{G} \;\Leftrightarrow\; (\mathcal{I}_0, w') \overset{*}{\vdash} (\iota I, \varepsilon) \;\wedge\; Act.I = \{\emptyset \cdot \mathcal{S}' \rightarrow S\# \cdot \# \cdot\}$$

Definition 4.9 completes the formalization of the $PLR(0)$-parsing process. The following theorem states that an input is accepted by the $PLR(0)$ algorithm if and only if it is an element of the language generated by the SCFPG that is being used, provided of course that the algorithm is able to make a decision. The complete proof is fairly elaborate and can be found in Ref. [2, Appendix B]. As already mentioned, an outline of the proof is given in Appendix A.

Theorem 4.10. *Let $\mathcal{G}$ be an SCFPG and let M be a pomset such that $\mathcal{G}$ is $PLR(0)$ for M.*

$$M \in \mathcal{L}_{PLR(0)}.\mathcal{G} \;\Leftrightarrow\; M \in \mathcal{L}.\mathcal{G}.$$

Let us return to the example developed throughout this paper.

Example 4.11. Consider again the SCFPG $\mathcal{G}_e = (\mathcal{N}_e, \mathcal{T}_e, \mathcal{R}_e, \mathcal{S}_e)$ where $\mathcal{N}_e = \{\mathcal{S}_e, T, U\}$, $\mathcal{T}_e = \{A, B\}$, and $\mathcal{R}_e = \{\mathcal{S}_e \rightarrow TU \,\&\, T, T \rightarrow A, U \rightarrow B\}$. The parse of the pomset $AB\&A$ proceeds as follows. It is the formalization of the graphical representation in Figure 5. The numbers refer to the corresponding steps in this figure. Let $\mathcal{I}_e$ be as before. The sets $I_0, \ldots, I_7$ as well as their closures are given below.

$$
\begin{array}{ll}
 & (\mathcal{I}_e,\ (\emptyset, A, a_0)(\emptyset, A, a_1)(a_0, B, b)(\{a_1, b\}, \#, \#)) \\
\vdash & \{(0):\quad Act.\mathcal{I}_e = \{\mathit{Shift}\} \;\wedge\; \mathcal{I}_e - (\emptyset, A, a_0) \rightarrow I_0 \text{ (see below) }\} \\
 & (\mathcal{I}_e I_0,\ (\emptyset, A, a_1)(a_0, B, b)(\{a_1, b\}, \#, \#)) \\
\vdash & \{(1), (v):\ Act.I_0 = \{\emptyset \cdot T \rightarrow A \cdot a_0 \cdot\} \wedge |\, I_0 \,| = |\, A \,| \wedge \mathcal{I}_e - (\emptyset, T, a_0) \rightarrow I_1\} \\
 & (\mathcal{I}_e I_1,\ (\emptyset, A, a_1)(a_0, B, b)(\{a_1, b\}, \#, \#)) \\
\vdash & \{(2):\quad Act.I_1 = \{\mathit{Shift}\} \wedge I_1 - (\emptyset, A, a_1) \rightarrow I_2\} \\
 & (\mathcal{I}_e I_1 I_2,\ (a_0, B, b)(\{a_1, b\}, \#, \#)) \\
\vdash & \{(3), (iv):\ Act.I_2 = \{\emptyset \cdot T \rightarrow A \cdot a_1 \cdot\} \wedge |\, I_2 \,| = |\, A \,| \wedge I_1 - (\emptyset, T, a_0) \rightarrow I_3\} \\
 & (\mathcal{I}_e I_1 I_3,\ (a_0, B, b)(\{a_1, b\}, \#, \#)) \\
\vdash & \{(4):\quad Act.I_3 = \{\mathit{Shift}\} \wedge I_3 - (a_0, B, b) \rightarrow I_4\} \\
 & (\mathcal{I}_e I_1 I_3 I_4,\ (\{a_1, b\}, \#, \#)) \\
\vdash & \{(5), (iii):\ Act.I_4 = \{a_0 \cdot U \rightarrow B \cdot b \cdot\} \wedge |\, I_4 \,| = |\, B \,| \wedge I_3 - (a_0, U, b) \rightarrow I_5\} \\
 & (\mathcal{I}_e I_1 I_3 I_5,\ (\{a_1, b\}, \#, \#)) \\
\vdash & \{(6), (ii):\ Act.I_5 = \{\emptyset \cdot \mathcal{S}_e \rightarrow TU \cdot b \cdot \,\&\, T \cdot a_1 \cdot\} \wedge \\
 & \qquad |\, I_1 I_3 I_5 \,| = |\, TU \,| + |\, T \,| \wedge \mathcal{I}_e - (\emptyset, \mathcal{S}_e, \{a_1, b\}) \rightarrow I_6\} \\
 & (\mathcal{I}_e I_6,\ (\{a_1, b\}, \#, \#)) \\
\vdash & \{(7):\quad Act.I_6 = \{\mathit{Shift}\} \wedge I_6 - (\{a_1, b\}, \#, \#) \rightarrow I_7\} \\
 & (\mathcal{I}_e I_6 I_7,\ \varepsilon)
\end{array}
$$

Since $Act.I_7 = \{\emptyset \cdot \mathcal{S}_e' \rightarrow \mathcal{S}_e\# \cdot \# \cdot\}$, the input is accepted $((8), (i))$.

$$\overline{\mathcal{I}_e} = \mathcal{I}_e \cup \{\emptyset \cdot \mathcal{S}_e \rightarrow \cdot \emptyset \cdot TU \,\&\, \cdot \emptyset \cdot T,\ \emptyset \cdot T \rightarrow \cdot \emptyset \cdot A\}$$
$$I_0 = \overline{I_0} = \{\emptyset \cdot T \rightarrow A \cdot a_0 \cdot\}$$
$$I_1 = \{\emptyset \cdot \mathcal{S}_e \rightarrow T \cdot a_0 \cdot U \,\&\, \cdot \emptyset \cdot T,\ \emptyset \cdot \mathcal{S}_e \rightarrow \cdot \emptyset \cdot TU \,\&\, T \cdot a_0 \cdot\}$$

$\overline{I_1} = I_1 \cup \{a_0 \cdot U \rightarrow\ \cdot a_0 \cdot B,\ \emptyset \cdot T \rightarrow\ \cdot \emptyset \cdot A\}$
$I_2 = \overline{I_2} = \{\emptyset \cdot T \rightarrow A \cdot a_1 \cdot\}$
$I_3 = \{\emptyset \cdot \mathcal{S}_e \rightarrow T \cdot a_0 \cdot U \,\&\, T \cdot a_1 \cdot,\ \emptyset \cdot \mathcal{S}_e \rightarrow T \cdot a_1 \cdot U \,\&\, T \cdot a_0 \cdot\}$
$\overline{I_3} = I_3 \cup \{a_0 \cdot U \rightarrow\ \cdot a_0 \cdot B,\ a_1 \cdot U \rightarrow\ \cdot a_1 \cdot B\}$
$I_4 = \overline{I_4} = \{a_0 \cdot U \rightarrow B \cdot b \cdot\}$
$I_5 = \overline{I_5} = \{\emptyset \cdot \mathcal{S}_e \rightarrow TU \cdot b \cdot \,\&\, T \cdot a_1 \cdot\}$
$I_6 = \overline{I_6} = \{\emptyset \cdot \mathcal{S}'_e \rightarrow \mathcal{S}_e \cdot \{a_1, b\} \cdot \#\}$
$I_7 = \overline{I_7} = \{\emptyset \cdot \mathcal{S}'_e \rightarrow \mathcal{S}_e \# \cdot \# \cdot\}$

It is also possible to describe the $PLR(0)$-parsing algorithm in terms of an abstract programming language. The notation used here is based on the *Guarded Command Language.*[8]

Algorithm 4.12. ($PLR(0)$-parsing algorithm)

proc $PLR(0)Driver$ ($\downarrow \mathcal{G}$: SCFPG; $\uparrow plr0, error, accept$: boolean)
$\{\!\!\{$Pre: The input sequence w' is a representation of pomset $M\#$.
  Post: $plr0 \ \Rightarrow\ \neg error\ \equiv\ accept\ \equiv\ M \in \mathcal{L}_{PLR(0)}.\mathcal{G}$
$\}\!\!\}$
  = |[(P, L, v) : $input$
      ; s : stack of 2^{item}; p : $\mathbb{N}$
    | $s[0], p := \mathcal{I}_0, 0$
      ; $plr0, error, accept$:= true, false, false
      ; $Read(P, L, v)$
      ; do $plr0\ \wedge\ \neg error\ \wedge\ \neg accept$
          $\longrightarrow$ if $|Act.s[p]| > 1$
                  $\longrightarrow$ $\{\!\!\{\mathcal{G}$ is not $PLR(0)$ $\}\!\!\}$ $plr0$:= false
              [] $Act.s[p] = \emptyset$
                  $\longrightarrow$ $\{\!\!\{$syntax error $\}\!\!\}$ $error$:= true
              [] $Act.s[p] = Shift$
                  $\longrightarrow$ $Shift(P, L, v)$
                      ; $Read(P, L, v)$
              [] $Act.s[p] = \emptyset \cdot \mathcal{S}' \rightarrow \mathcal{S}\# \cdot \# \cdot$
                  $\longrightarrow$ $accept$:= true
              [] $Act.s[p] = W \cdot T \rightarrow \sigma \cdot X \cdot \,\&\, \tau \cdot Y \cdot\ \wedge$
                 $Act.s[p] \neq \emptyset \cdot \mathcal{S}' \rightarrow \mathcal{S}\# \cdot \# \cdot$
                  $\longrightarrow$ $Reduce(W \cdot T \rightarrow \sigma \cdot X \cdot \,\&\, \tau \cdot Y \cdot)$
              fi
      od
    ]|

Algorithm 4.13. (Procedure *Shift*)

proc $Shift$ ($\downarrow (P, L, v)$: $input$)
$\{\!\!\{$Pre: $p = Q$
  Post: $p = Q + 1\ \wedge\ s[p-1] - (P, L, v) \rightarrow s[p]$
$\}\!\!\}$
  = |[J : 2^{item}
    | $J := \emptyset$

; for $W \cdot T \rightarrow \sigma_0 \cdot P \cdot L\sigma_1 \,\&\, \tau_0 \cdot Z \cdot \tau_1 \in \overline{s[p]}$
$\longrightarrow\ J := J \cup \{W \cdot T \rightarrow \sigma_0 L \cdot v \cdot \sigma_1 \,\&\, \tau_0 \cdot Z \cdot \tau_1\}$
rof
$\lessdot s[p] - (P, L, v) \rightarrow J \gtrdot$
; $p := p + 1$
; $s[p] := J$
]|

Algorithm 4.14. (Procedure *Reduce***)**
proc *Reduce* $(\downarrow X \cdot U \rightarrow \sigma \cdot Y_0 \cdot \,\&\, \tau \cdot Y_1 \cdot \ :\ item)$
$\lessdot$Pre: $p = Q \ \wedge\ Q \geq |\sigma| + |\tau|$
Post: $p = Q - |\sigma| - |\tau| + 1 \ \wedge\ s[p-1] - (X, U, Y_0 \cup Y_1) \rightarrow s[p]$
$\gtrdot$
= |[J : 2^{item}
| $p := p - |\sigma| - |\tau|$
; $J := \emptyset$
; for $W \cdot T \rightarrow \sigma_0 \cdot X \cdot U\sigma_1 \,\&\, \tau_0 \cdot Z \cdot \tau_1 \in \overline{s[p]}$
$\longrightarrow\ J := J \cup \{W \cdot T \rightarrow \sigma_0 U \cdot Y_0 \cup Y_1 \cdot \sigma_1 \,\&\, \tau_0 \cdot Z \cdot \tau_1\}$
rof
$\lessdot s[p] - (X, U, Y_0 \cup Y_1) \rightarrow J \gtrdot$
; $p := p + 1$
; $s[p] := J$
]|

It might not be immediately clear that Algorithm 4.12 establishes its postcondition. This is caused by the following two simplifications. First, in the fourth alternative of the select statement, the requirement is omitted that all input has been read. Second, in the fifth alternative it is not tested whether the precondition of procedure *Reduce* is guaranteed. The validity of these two simplifications are intermediate results of the proof of Theorem 4.10.

Usually, LR-parsing algorithms are implemented using a simple driver and two *tables*, namely an action table and a goto table. Such a design is chosen because if the grammar is changed, only these two tables need to be recalculated. However, an implementation with tables requires that the action function and goto relation can be calculated given only the grammar. Obviously, this is not possible for the $PLR(0)$-action function and -goto relation. Knowledge of the input is required. Therefore, it is necessary to calculate the action function and goto relation on the fly. This is exactly what the procedures above do.

A final remark about the algorithm is in order. It follows from the construction of the algorithm that it behaves like an ordinary $LR(0)$ algorithm if the pomset language produced by the pomset grammar $\mathcal{G}$ actually is a string language. It is a true generalization of the $LR(0)$-parsing technique.

Example 4.15. Let us return one last time to our RPC example. We have already seen that behavioral patterns can be specified by means of concurrent regular expressions. Following the construction of Theorem 2.16, a concurrent regular expression can be transformed into an SCFPG which can be used by the $PLR(0)$ algorithm presented in this section. In this example, assume that we want to know whether the computation of Example 2.7 satisfies the pattern of an RPC. We augment the SCFPG given in Example 2.18, yielding the following result:

$(\{\mathcal{S}', \mathcal{S}, T, U, V\}, \{C, \mathit{Call}, \mathit{RCall}, \mathit{Ret}, \mathit{RRet}, \#\}, \mathcal{R}', \mathcal{S}')$,

where

$$\mathcal{R}' = \{\, \mathcal{S}' \to \mathcal{S}; \#, \mathcal{S} \to T; U; V, T \to \mathit{Call} \,\&\, C, U \to \mathit{RCall}; C; \mathit{Ret}, \\ V \to \mathit{RRet} \,\&\, C\}.$$

Assume the input to the parser is the following linearization of the example computation:

$$(\emptyset, \mathit{Call}, 0)(\emptyset, C, 1)(\{0,1\}, \mathit{RCall}, 2)(2, C, 3)(3, \mathit{Ret}, 4)(4, C, 5) \\ (4, \mathit{RRet}, 6)(\{5,6\}, \#, \#).$$

It is not difficult to verify that the input is accepted. Only the first few steps of the parsing process are given in detail. The others are left to the reader to verify.

$$
\begin{array}{ll}
 & (\mathcal{I}_0,\ (\emptyset, \mathit{Call}, 0)(\emptyset, C, 1)(\{0,1\}, \mathit{RCall}, 2)(2, C, 3)(3, \mathit{Ret}, 4)(4, C, 5) \\
 & \qquad (4, \mathit{RRet}, 6)(\{5,6\}, \#, \#)) \\
\vdash & \quad \{\ \mathit{Act}.\mathcal{I}_0 = \{\mathit{Shift}\}\ \wedge\ \mathcal{I}_0 - (\emptyset, \mathit{Call}, 0) \to I_1 \text{ (see below) } \} \\
 & (\mathcal{I}_0 I_1,\ (\emptyset, C, 1)(\{0,1\}, \mathit{RCall}, 2)(2, C, 3)(3, \mathit{Ret}, 4)(4, C, 5) \\
 & \qquad (4, \mathit{RRet}, 6)(\{5,6\}, \#, \#)) \\
\vdash & \quad \{\ \mathit{Act}.I_1 = \{\mathit{Shift}\}\ \wedge\ I_1 - (\emptyset, C, 1) \to I_2\ \} \\
 & (\mathcal{I}_0 I_1 I_2,\ (\{0,1\}, \mathit{RCall}, 2)(2, C, 3)(3, \mathit{Ret}, 4)(4, C, 5)(4, \mathit{RRet}, 6)(\{5,6\}, \#, \#)) \\
\vdash & \quad \{\ \mathit{Act}.I_2 = \{\emptyset \cdot T \to \mathit{Call} \cdot 0 \cdot \,\&\, C \cdot 1 \cdot \}\ \wedge\ |\, I_1 I_2 \,| = |\, \mathit{Call} \,| + |\, C \,|\ \wedge \\
 & \qquad \mathcal{I}_0 - (\emptyset, T, \{0,1\}) \to I_3\ \} \\
 & (\mathcal{I}_0 I_3,\ (\{0,1\}, \mathit{RCall}, 2)(2, C, 3)(3, \mathit{Ret}, 4)(4, C, 5)(4, \mathit{RRet}, 6)(\{5,6\}, \#, \#)) \\
\overset{*}{\vdash} & \quad \{\ \} \\
 & (\mathcal{I}_0 I_4 I_5,\ \varepsilon)
\end{array}
$$

$$
\begin{aligned}
I_1 &= \{\emptyset \cdot T \to \mathit{Call} \cdot 0 \cdot \ \&\ \cdot \emptyset \cdot C\} \\
I_2 &= \{\emptyset \cdot T \to \mathit{Call} \cdot 0 \cdot \ \&\, C \cdot 1 \cdot \} \\
I_3 &= \{\emptyset \cdot \mathcal{S} \to T \cdot \{0,1\} \cdot U; V\} \\
I_4 &= \{\emptyset \cdot \mathcal{S}' \to \mathcal{S} \cdot \{5,6\} \cdot \#\} \\
I_5 &= \{\emptyset \cdot \mathcal{S}' \to \mathcal{S}; \# \cdot \# \cdot \}
\end{aligned}
$$

From I_5 we may conclude that the input is indeed accepted.

As already mentioned, in general, one would like to recognize *occurrences* of patterns in a distributed computation. That is, one must be able to recognize *sub*pomsets in the input stream representing a distributed computation. The generalization of the concepts presented in this paper to recognizing subpomsets is one of the interesting topics for further research.

4.2. Some Constructs that are not PLR(0)

It follows from Theorem 4.10 that *if* the $PLR(0)$ algorithm can decide whether a pomset belongs to the pomset language generated by some pomset grammar, then it decides correctly. The question thus arises *when* can the algorithm make a decision. This section shows a few grammar constructs that are non-$PLR(0)$.

If the language generated by the pomset grammar is a string language, the $PLR(0)$ algorithm behaves like an $LR(0)$ algorithm. Therefore, grammar constructs that are not $LR(0)$ for string grammars are also not $PLR(0)$ for pomset grammars.

Example 4.16. Consider the SCFPG $\mathcal{G} = (\{\mathcal{S}\}, \{A, B\}, \{\mathcal{S} \rightarrow A, \mathcal{S} \rightarrow AB\}, \mathcal{S})$. Note that this SCFPG is also a CFG. It is easy to verify that this grammar generates the string language $A + AB$. It is well known that the non-determinism introduced by the $+$-operator causes the grammar to be non-$LR(0)$. Therefore, it is non-$PLR(0)$ as well. After recognizing a terminal A, the algorithm cannot decide whether it has successfully recognized the input or whether it still has to match the B. A shift-reduce conflict occurs. The only solution to this problem is to read the next input before making a decision.

Another SCFPG that is non-$PLR(0)$ is the pomset grammar $\mathcal{G} = (\{\mathcal{S}\}, \{A\}, \{\mathcal{S} \rightarrow \varepsilon, \mathcal{S} \rightarrow AS\}, \mathcal{S})$. It generates the string language A^*. Obviously, a shift-reduce conflict occurs in the initial state of the parse. Again, the solution is to read the next input before making the decision.

The previous example shows two constructs known from $LR(0)$ parsing that are not $PLR(0)$. However, there is also a construct inherent to pomset grammars that is not $PLR(0)$.

Example 4.17. Consider the SCFPG $\mathcal{G} = (\{\mathcal{S}\}, \{A\}, \{\mathcal{S} \rightarrow \varepsilon, \mathcal{S} \rightarrow A \& S\}, \mathcal{S})$. This pomset grammar generates the language $A\dagger$. Assume the input is pomset A which, obviously, is an element of $A\dagger$. The parse of the augmented grammar $\mathcal{G}'$ starts in configuration $(\mathcal{I}_0, (\emptyset, A, a)(a, \#, \#))$, where $\mathcal{I}_0$ is the initial item. The closure of $\mathcal{I}_0$ is $\mathcal{I}_0 \cup \{\emptyset \cdot \mathcal{S} \rightarrow \ \cdot \emptyset \cdot, \emptyset \cdot \mathcal{S} \rightarrow \ \cdot \emptyset \cdot A \ \& \ \cdot \emptyset \cdot \mathcal{S}\}$. The next action is defined by $Act.\mathcal{I}_0$ which is equal to $\{\mathit{Shift}, \emptyset \cdot \mathcal{S} \rightarrow \ \cdot \emptyset \cdot\}$. Again, a shift-reduce conflict occurs.

The examples given in this subsection show that many pomset-grammar constructs are not $PLR(0)$. As for LR parsing, introducing lookahead information in the theory might solve the problems. However, due to the concurrency inherent to pomsets, there are several ways to include lookahead information in PLR parsing. One possibility is to include lookahead information per item; another possibility is to add lookahead information to each of the strings in the right-hand side of an item. The first possibility is studied in Ref. [2, Section 4.4]. An algorithm using a single lookahead symbol is proposed. It is a very straightforward combination of $PLR(0)$ and $LR(1)$ parsing. However, it should be noted that correctness of the algorithm remains to be shown. The algorithm solves some of the problems identified in this subsection. It can handle the two languages given in Example 4.16. However, language $A\dagger$ of Example 4.17 still cannot be recognized successfully. Another pomset language that cannot be recognized successfully is the language $A \& A^*$. These two examples show that the addition of a single lookahead symbol to items as a whole does not really help for pomset languages exhibiting concurrency between multiple instances of the same terminal. This is an unfortunate conclusion because the main motivation to develop PLR parsing is the possibility to recognize pomsets instead of ordinary strings. For the language $A\dagger$, adding lookahead information to each of the strings in the right-hand sides of items might be a solution; for $A \& A^*$, only adding at least one more lookahead symbol seems to help. A disadvantage of adding lookahead information to each string in the right-hand side of an item is that it is only feasible for SCFPGs. It complicates the generalization of PLR parsing to general CFPGs. In order to generate pomset languages with pomsets containing an N, the pomset N must be allowed in the right-hand side of production rules. For such production rules, the only possibility seems to be to add lookahead information to items as a whole. Summarizing, including lookahead information in PLR-parsing theory remains an interesting challenge.

5. Conclusions and Open Problems

This paper introduces a parsing technique for partially ordered multisets, called PLR parsing. The basic algorithm in the class of PLR-parsing algorithms, namely the $PLR(0)$ algorithm, is explained in detail. The PLR-parsing technique is a natural generalization of the well-known LR-parsing technique.

There are several open problems and directions for future work. Applications of the theory should be investigated. In particular, it should be investigated how the technique can be applied to the area which motivated its development: the area of analyzing causal relationships in parallel and distributed computations. Several unsolved problems remain in this area. One interesting issue is that recognizing causal patterns in a computation means recognizing *sub*pomsets instead of pomsets. This raises questions about the theoretical problems that need to be solved to recognize subpomsets instead of pomsets. Performance might also be an issue. A useful algorithm must be able to detect multiple subpomsets or occurrences of the same subpomset in a single input stream.

An important open problem is the inclusion of lookahead information into the theory. Due to the concurrency inherent to pomsets, there appear to be several ways to include lookahead information. One could maintain lookahead information for each string in the right-hand side of an item. Or one could maintain lookahead information for entire items at once. It is not yet clear what is the proper way.

There is another issue that deserves to be studied. The parsing algorithm presented in this paper can only handle simple context-free pomset languages. This class of pomset languages is a strict subclass of the context-free pomset languages. It is an interesting question how to extend the basic parsing technique presented in this paper to arbitrary context-free pomset languages. This seems possible when production rules of an SCFPG are allowed to have the pomset N as their right-hand side. The definitions of items, the action function, and the goto relation have to be adapted appropriately, as well as the proof of Theorem 4.10. The main reason for not already doing so in this paper is that it would have unnecessarily complicated the resulting theory. Also, it is not clear how the addition of lookahead information and the generalization to production rules allowing the pomset N can be combined in the best way. Generalizing the theory to arbitrary context-free pomset languages would have distracted from the main purpose of this paper, namely developing an elementary, easy to understand, and extendible parsing technique for partially ordered multisets.

Acknowledgments

The author wants to thank Jay Black, Michael Coffin, Thomas Kunz, and John Segers for the many fruitful discussions and their valuable suggestions. He is also grateful to professor Kruseman Aretz and the anonymous referees for their useful comments on earlier drafts of this paper.

References

1. A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques, and Tools* (Addison–Wesley, Reading, Massachusetts, 1988).
2. T. Basten. *Hierarchical Event-Based Behavioral Abstraction in Interactive Distributed Debugging: A Theoretical Approach.* Master's thesis, Eindhoven University of Technology, Dept. of Math. and Comp. Sci., The Netherlands, 1993.
3. C.N. Fischer and R.J. LeBlanc, Jr. *Crafting a Compiler* (The Benjamin/Cummings Publishing Company, Menlo Park, California, 1988).
4. J.L. Gischer. *Partial Orders and the Axiomatic Theory of Shuffle.* Ph.D. Thesis, Technical Report STAN-CS-84-1033, Stanford University, Dept. of Comp. Sci., Stanford, California, 1984.
5. J.L. Gischer. The Equational Theory of Pomsets. *Theoretical Computer Science*, **61**(2,3):199–224, Nov. 1988.
6. J. Grabowski. On Partial Languages. *Fundamenta Informaticae*, **4**(2):427–498, 1981.
7. W. Hseush and G.E. Kaiser. Moddeling Concurrency in Parallel Debugging. *ACM SIGPLAN Notices*, **25**(3):11–20, Mar. 1990. Proceedings of the 2nd ACM/SIGPLAN Symposium on Principles & Practice of Parallel Programming, Seattle, Washington, Mar. 1990.
8. A. Kaldewaij. *Programming: The Derivation of Algorithms* (Prentice–Hall, New York, 1990).
9. D.E. Knuth. On the Translation of Languages from Left to Right. *Information and Control*, **8**(6):607–639, 1965.
10. L. Lamport. Time, Clocks and the Ordering of Events in a Distributed System. *Communications of the ACM*, **21**(7):558–565, 1978.
11. H.R. Lewis and C.H. Papadimitriou. *Elements of the Theory of Computation* (Prentice–Hall, Englewood Cliffs, New Jersey, 1981).
12. V.R. Pratt. On the Composition of Processes. In *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, pages 213–223, Albuquerque, New Mexico, Jan. 1982.
13. V.R. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, **15**(1):33–71, Feb. 1986.
14. R. Schwarz and F. Mattern. Detecting Causal Relationships in Distributed Computations: In Search of the Holy Grail. *Distributed Computing*, **7**(3):149–174, 1994.
15. J. Winkowski. An Algebraic Characterization of the Behaviour of Non-sequential Systems. *Information Processing Letters*, **6**(4):105–109, 1977.
16. J. Winkowski. An Algebraic Approach to Concurrency. In J. Bečvář, editor, *Mathematical Foundations of Computer Science 1979, Proceedings*, volume 74 of *Lecture Notes in Computer Science*, pages 523–532, Olomouc, Czechoslovakia, Sept. 1979. (Springer–Verlag, Berlin, 1979).

Appendix A: Correctness of $PLR(0)$ Parsing

This section gives an outline of a correctness proof of the $PLR(0)$ algorithm. It shows that a pomset for which a given SCFPG is $PLR(0)$ is accepted by the $PLR(0)$ algorithm if and only if it is an element of the pomset language generated by the SCFPG. Important intermediate results in the proof are given; their proofs are omitted. For the details of these proofs see Ref. [2, Appendix B].

Let $\mathcal{G}$ be an SCFPG. Furthermore, let $M\#$ be the input pomset represented by the sequence $w' = w(Maxv.M, \#, \#)$. The same notational conventions are used as in Section 4. We can now formulate the proof requirements as follows. First, we must show that $M \in \mathcal{L}_{PLR(0)}.\mathcal{G} \;\Rightarrow\; M \in \mathcal{L}.\mathcal{G}$. Second, given $\mathcal{G}$ is $PLR(0)$, it must be shown that $M \in \mathcal{L}.\mathcal{G} \;\Rightarrow\; M \in \mathcal{L}_{PLR(0)}.\mathcal{G}$.

In the first and most difficult part of the proof, we construct a derivation from a successful parse by extracting the arguments of the goto relation causing the state changes in the parse. These arguments are triples of type *label* and form a derivation in reversed order as shown in Example 3.5. In the second part of the proof, we construct a parse from a derivation by means of an inductive argument.

The proof often uses derivations in terms of triples of type *label* as shown in Example 3.5 instead of derivations as defined in Definition 2.11. However, it should be clear that both representations of a derivation are interchangeable.

Two sets of items occur often in the entire proof and are hence abbreviated. Let $\mathcal{I}_1$ and $\mathcal{I}_2$ be the two sets of items defined as $\mathcal{I}_0 - (\emptyset, \mathcal{S}, Maxv.M) \rightarrow \mathcal{I}_1$ and $\mathcal{I}_1 - (Maxv.M, \#, \#) \rightarrow \mathcal{I}_2$. Recall that $\mathcal{I}_0$ is the initial item $\emptyset \cdot \mathcal{S}' \rightarrow \cdot \emptyset \cdot \mathcal{S}\#$.

For the first part of the proof, assume M is an element of $\mathcal{L}_{PLR(0)}.\mathcal{G}$. It follows from Definition 4.9 of the pomset language recognized by a $PLR(0)$ parser that there exist $\iota_0, \ldots, \iota_n$ and $\omega_0, \ldots, \omega_n$, for some $n \in \mathbb{N}$, that satisfy $(\iota_0, \omega_0) \vdash \ldots \vdash (\iota_n, \omega_n)$, $\iota_0 = \mathcal{I}_0$, $\omega_0 = w'$, and $\iota_n = \iota I$, for some ι and I, such that $Act.I = \{\emptyset \cdot \mathcal{S}' \rightarrow \mathcal{S}\# \cdot \# \cdot\}$. According to the definition of $\vdash$, none of the sequences ι_i is empty, which justifies the introduction of $m_i \in \mathbb{N}$ and I^i_j, for $0 \leq i \leq n$ and $0 \leq j \leq m_i$, such that $\iota_i = I^i_0 \ldots I^i_{m_i}$. In the remainder, it is assumed that i ranges over $[0..n]$ and j over $[0..m_i]$ unless explicitly stated otherwise. We must show that the steps $(\iota_0, \omega_0) \vdash \ldots \vdash (\iota_n, \omega_n)$ define a derivation for pomset M.

The following property is a basic result which can be proven by induction. The first part states that every sequence ι_i starts with the singleton containing the initial item and that this item does not occur anywhere else in the sequence. The second part says that each state change in any ι_i is caused by a triple of type *label*.

Property A.1.

(i) *For all i, $I^i_0 = \mathcal{I}_0$ and, for all $j \neq 0$, $\mathcal{I}_0 \notin I^i_j$.*

(ii) *For all i, there exist $X^i_1, V^i_1, Y^i_1, \ldots, X^i_{m_i}, V^i_{m_i}, Y^i_{m_i}$ such that*
$I^i_0 - (X^i_1, V^i_1, Y^i_1) \rightarrow I^i_1 \ldots I^i_{m_i-1} - (X^i_{m_i}, V^i_{m_i}, Y^i_{m_i}) \rightarrow I^i_{m_i}$.

In the remainder of the proof, it is assumed that X^i_j, V^i_j, and Y^i_j are defined as in this property. The following property shows the uniqueness of the sets of items $\mathcal{I}_0, \mathcal{I}_1$, and $\mathcal{I}_2$. It can be proven by applying the various definitions.

Property A.2.

(i) $\varnothing \cdot \mathcal{S}' \rightarrow \mathcal{S} \cdot \mathit{Maxv}.M \cdot \# \in I^i_j \;\Leftrightarrow\; I^i_j = \mathcal{I}_1$.

(ii) $\varnothing \cdot \mathcal{S}' \rightarrow \mathcal{S}\# \cdot \# \cdot \in I^i_j \;\Leftrightarrow\; I^i_j = \mathcal{I}_2$.

(iii) $(\exists X, V, Y : I^i_j - (X, V, Y) \rightarrow \mathcal{I}_1) \Leftrightarrow I^i_j = \mathcal{I}_0$.

(iv) $(\exists X, V, Y : I^i_j - (X, V, Y) \rightarrow \mathcal{I}_2) \Leftrightarrow I^i_j = \mathcal{I}_1$.

The next result is a simple consequence of the previous properties.

Property A.3. $\iota_n = \mathcal{I}_0\mathcal{I}_1\mathcal{I}_2$

This property states that a parse always ends in the same final configuration. Note that since $\mathcal{I}_2$ can only be reached by reading the input $(\mathit{Maxv}.M, \#, \#)$, it follows that $\mathcal{I}_2$ is on top of the stack if and only if the remaining input is empty. This validates the first simplification of the $PLR(0)$ algorithm discussed in Section 4.1.

The next function is essential for the construction of a derivation. For each state change in a parse, it extracts the triple causing the change.

Definition A.4. (Function ℓ)
The function ℓ maps sets of items to triples of type *label* or ε as follows:

$$\ell.I^i_j = \begin{cases} \varepsilon, & \text{for } j = 0, \\ (X^i_j, V^i_j, Y^i_j), & \text{otherwise.} \end{cases}$$

Function ℓ is easily generalized to sequences of sets of items as follows:

$$\ell.\varepsilon = \varepsilon, \;\; \ell.(\iota I^i_j) = \ell.\iota\ell.I^i_j.$$

Before we can actually construct a derivation, one more result is needed. It shows the correctness of a reduce action.

Property A.5. *Let i be such that the* completed *item $W \cdot T \rightarrow \sigma \cdot Z_0 \cdot \& \tau \cdot Z_1 \cdot$ not equal to $\varnothing \cdot \mathcal{S}' \rightarrow \mathcal{S}\# \cdot \# \cdot$ is an element of $\overline{I^i_{m_i}}$. For k equal to $|\sigma| + |\tau|$,*

(i) $\ell.(I^i_{m_i-k+1} \ldots I^i_{m_i}) = (W, \sigma\&\tau, Z_0 \cup Z_1)$,

(ii) $\ell.I^{i+1}_{m_{i+1}} = (W, T, Z_0 \cup Z_1)$.

In Property A.5(i), triple $(W, \sigma\&\tau, Z_0 \cup Z_1)$ is an abbreviation of a sequence of triples of type *label* representing pomset $\sigma\&\tau$ with predecessors W and maximal vertices $Z_0 \cup Z_1$ respectively. Property A.5 shows that a reduce action is well-defined. If the next parse action is a reduce, the stack contains at least $k+1$ sets of items, where k is defined as in the premise of the property. It also shows that after removing k sets the next state of the parse is well-defined and has the expected label. Note that Property A.5(i) validates the second simplification of the $PLR(0)$ algorithm mentioned in Section 4.1. Property A.5 can be used to prove the following result, which is the basis for the construction of a derivation for pomset M.

Property A.6.

(i) *For some i, where $0 \le i < n$,*
$(\iota_i, \omega\omega_{i+1}) \vdash (\iota_{i+1}, \omega_{i+1}) \;\Rightarrow\; \ell.\iota_{i+1} \overset{*}{\Longrightarrow} \ell.\iota_i\omega$.

(ii) *For any* k, l, *where* $0 \leq k \leq l \leq n$,
$(\iota_k, \omega\omega_l) \overset{*}{\vdash} (\iota_l, \omega_l) \;\Rightarrow\; \ell.\iota_l \overset{*}{\Longrightarrow} \ell.\iota_k\omega$.

Property A.6(i) shows that a single parse step corresponds to zero or more steps in a (reversed) derivation. It can be shown that a shift action does not generate any derivation steps and that a reduce action corresponds to exactly one (rightmost) step in a derivation. Property A.6(ii) generalizes (i) to an arbitrary number of parse steps.

At this point, it is not very difficult anymore to show that pomset M is generated by $\mathcal{G}$. Since we assumed that M is an element of $\mathcal{L}_{PLR(0)}.\mathcal{G}$, Property A.3 shows that $(\mathcal{I}_0, w') \overset{*}{\vdash} (\mathcal{I}_0\mathcal{I}_1\mathcal{I}_2, \varepsilon)$. It follows from Property A.6 that $\ell.(\mathcal{I}_0\mathcal{I}_1\mathcal{I}_2) \overset{*}{\Longrightarrow} \ell.\mathcal{I}_0 w'$. Since $\ell.\mathcal{I}_0 = \varepsilon$, $\ell.(\mathcal{I}_0\mathcal{I}_1\mathcal{I}_2) = (\emptyset, \mathcal{S}\#, \#)$, and w' is a representation of $M\#$, it follows that $\mathcal{S}\# \overset{*}{\Longrightarrow} M\#$, or $\mathcal{S} \overset{*}{\Longrightarrow} M$. Hence, we may conclude that $M \in \mathcal{L}.\mathcal{G}$, which completes the first part of the proof.

Theorem A.7. $\quad \mathcal{L}_{PLR(0)}.\mathcal{G} \subseteq \mathcal{L}.\mathcal{G}$.

For the second half of the proof, assume that $M \in \mathcal{L}.G$ and that $\mathcal{G}$ is $PLR(0)$ for M. It follows that, for $n \in \mathbb{N}$ and $\gamma_0, \ldots, \gamma_n \in label^*$, there is a rightmost derivation $\gamma_0 \Longrightarrow \ldots \Longrightarrow \gamma_n$, where $\gamma_0 = (\emptyset, \mathcal{S}', \#)$ and $\gamma_n = w'$. Introduce $m_i \in \mathbb{N}$, where $0 \leq i \leq n$, and X^i_j, V^i_j, Y^i_j, where $0 \leq j \leq m_i$, such that $\gamma_i = (X^i_0, V^i_0, Y^i_0) \ldots (X^i_{m_i}, V^i_{m_i}, Y^i_{m_i})$. In the following, the notation $\rho.i$, where $0 \leq i < |\rho|$, is used to denote element i of sequence ρ. The notation $\rho[i..j)$ denotes the subsequence $\rho.i \ldots \rho.(j-1)$; $\rho[i..j]$ denotes $\rho.i \ldots \rho.j$.

Property A.8. *For* i *and* j, *where* $0 \leq i \leq n$ *and* $0 \leq j \leq m_i$, *and* p, q, *where* $0 \leq p \leq q \leq |w'|$,

$(X^i_j, V^i_j, Y^i_j) \overset{*}{\Longrightarrow} w'[p..q) \;\wedge\; I - (X^i_j, V^i_j, Y^i_j) \rightarrow J$
$\Rightarrow$
$(\iota I, w'[p..m_n]) \overset{*}{\vdash} (\iota IJ, w'[q..m_n])$.

This property shows that each part of the derivation for pomset M can be mimicked by the parser. It can be proven by induction to the length of the derivation. The proof uses the fact that the above derivation is *rightmost*. Property A.8 is sufficient to complete the second half of the proof. Since M is an element of $\mathcal{L}.\mathcal{G}$, it follows that $(\emptyset, \mathcal{S}, Maxv.M) \overset{*}{\Longrightarrow} w$. Property A.8 and the definitions of $\mathcal{I}_0$ and $\mathcal{I}_1$ yield that $(\mathcal{I}_0, w(Maxv.M, \#, \#)) \overset{*}{\vdash} (\mathcal{I}_0\mathcal{I}_1, (Maxv.M, \#, \#))$. Since $\mathcal{G}$ is $PLR(0)$, it follows from the definition of $\mathcal{I}_2$ that $(\mathcal{I}_0\mathcal{I}_1, (Maxv.M, \#, \#)) \vdash (\mathcal{I}_0\mathcal{I}_1\mathcal{I}_2, \varepsilon)$ and thus $M \in \mathcal{L}_{PLR(0)}.\mathcal{G}$.

Theorem A.9. *Let* $\mathcal{G}$ *be* $PLR(0)$.
$\mathcal{L}.\mathcal{G} \subseteq \mathcal{L}_{PLR(0)}.\mathcal{G}$.

Theorems A.7 and A.9 together prove Theorem 4.10.