SLR INFERENCE: AN INFERENCE SYSTEM FOR FIXED-MODE LOGIC PROGRAMS, BASED ON SLR PARSING

DAVID A. ROSENBLUETH AND JULIO C. PERALTA

▷ Definite-clause grammars (DCGs) generalize context-free grammars in such a way that Prolog can be used as a parser in the presence of context-sensitive information. Prolog's proof procedure, however, is based on backtracking, which may be a source of inefficiency. Parsers for context-free grammars which use backtracking, for instance, were soon replaced by more efficient methods such as LR parsers. This suggests incorporating the principles underlying LR parsing into a parser for grammars with context-sensitive information. We present a technique which applies a transformation to the program/grammar by adding leaves to the proof/parse trees and placing the contextual information in such leaves. An inference system is then easily obtained from an LR parser, since only the parts dealing with terminals (which appear at the leaves) must be modified. Although our method is restricted to programs with fixed modes, it may be preferable to DCGs under Prolog for some programs. ◁

1. INTRODUCTION

A motivation for the development of Prolog was that of having a programming language for natural-language applications. Hence, it is no coincidence that Prolog programs resemble context-free grammars. This resemblance suggests the possible transfer of results between logic programming and formal-language theory. We study an inference system based on LR parsing meant for "fixed-mode" logic programs, where each argument in a predicate acts either as input or as output of an operation, and input arguments are ground. (This paper is an extended version of [26]: we have added formal proofs, a discussion comparing the search space under

A previous version of this paper was presented at the International Logic Programming Symposium, Ithaca, New York, U.S.A. 1994.

Current address: Department of Computer Science, University of Bristol, Merchant Venturers Building, Woodland Road, Bristol BS8 1UB, U.K.

Address correspondence to David A. Rosenblueth, IIMAS, UNAM, Apdo. 20-726, 01000 México D.F. `drosenbl@servidor.unam.mx`

THE JOURNAL OF LOGIC PROGRAMMING

655 Avenue of the Americas, New York, NY 10010 0743-1066/94/$7.00

the proposed inference techniques with the SLD tree, and more comparisons with related work.)

Prolog's proof procedure can be viewed as a generalization of a simple parser with backtracking. In practical applications, such a parser has been replaced by more sophisticated methods, such as LR parsers [2]. The reasons are that the backtracking parser not only falls easily into nonterminating loops, but is inefficient.

Prolog's proof procedure suffers from the same defects as the parser on which it is based. This phenomenon has prompted the development of other, more sophisticated proof procedures for logic programs. In spite of the resemblance between logic programs and context-free grammars, there are few proof procedures based on parsers [8, 17, 18, 23]. We find this contrast puzzling because logic programs can naturally represent context-free grammars, e.g. using well-known difference-list techniques. The difference-list representation of a context-free grammar associates a production:

$$A \rightarrow B_1 B_2 \ldots B_n$$

with the clause:

$$a(X_0, X_n) \leftarrow b_1(X_0, X_1),\ b_2(X_1, X_2),\ \ldots,\ b_n(X_{n-1}, X_n) \tag{1.1}$$

In addition, this representation has a clause of the form:

$$c'([c|X], X) \leftarrow \tag{1.2}$$

for each terminal c of the context-free grammar.

The problem, then, is how to generalize logic programs which represent context-free grammars with clauses of the form (1.1) to include contextual information. Definite-clause grammars (DCGs) [22] do so with additional argument places. By contrast, we try to keep the "chain" form of clause (1.1) and add the contextual information only to unit clauses (1.2). As we will see, this generalization allows us to adapt an LR parser with minor changes, since only the part dealing with terminals must be modified.

A drawback of our approach, however, is that Prolog programmers normally do not write programs in chain form. Hence, we use a transformation taking a fixed-mode DCG and producing a logic program of the desired form. The resulting program is a logical consequence of an extension of the original program. This transformation essentially adds leaves to the parse/proof trees and places the contextual information in such leaves. (We have already used this transformation in [25], for developing inference systems based on "chart" parsers.) In fact, the original program may not necessarily be a DCG; it may be an arbitrary fixed-mode logic program.

At first sight it might seem that our limitation to fixed-mode logic programs is severe. It can be argued, however, that this is not the case. Drabent [11], for instance, claims that the majority of practical logic programs have fixed modes, and shows examples of programming techniques where multiple modes are used. Fixed-mode logic programs have also been studied in [3, 4, 10].

2. SLR PARSING

This section reviews SLR parsing [2]. We have selected an SLR parser as the basis for our inference system because such a parser illustrates various aspects of LR parsing without being so elaborate as to obscure the presentation of our method. We believe, however, that any LR parser can be converted into an inference system in a similar way.

We use the letter A for nonterminals, the letter B for either terminals or nonterminals, and the letter c for terminals. As usual, Greek letters denote strings, and "\$" is an end marker. By $|\alpha|$ we denote the length of α.

As [2], we assume a context-free grammar in which there is only one production $S' \rightarrow S$ with the start symbol S' on the left-hand side. We denote the set of nonterminals with N, the set of terminals with T, and the set of productions with R. For simplicity, we assume that grammars do not contain epsilon productions.

Before giving a formal explanation of SLR parsing, we will give an intuitive idea [14] behind this parsing method. Suppose that we are constructing a derivation in reverse, that is, from the string generated by the derivation to the start symbol. Assume also that the current string has the form $B_1 \ldots B_i B_{i+1} \alpha \$$, where the suffix $B_{i+1}\alpha\$$ represents the part of the string which we have not yet read. In addition, all possible reductions have been made at the prefix $B_1 \ldots B_i$ of the string so that the right boundary of the handle (substring corresponding to the right-hand side of a production) must be at position B_n for $n \geq i$. If $n > i$, then we must continue reading symbols of the input string (Figure 2.1a, shift), until $n = i$ (Figure 2.1b, reduce), in which case we have found a handle and can reduce by a production.

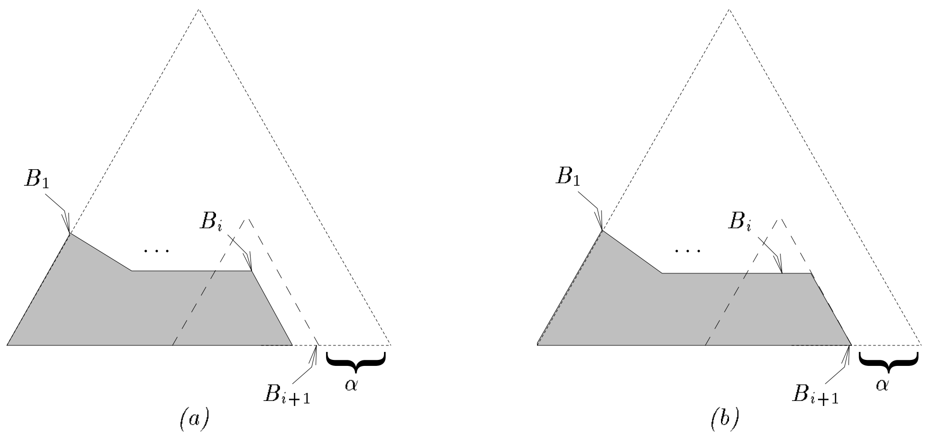

FIGURE 2.1. *Shift and reduce cases.*

Next, we give a definition of an SLR parser. Following [2], we define an *item* as a production in R augmented with the meta-symbol "$\bullet$" occurring on the right-hand side of that production. Given a set R of productions and a set I of items for R, the *closure* of I, $closure(I)$, is the smallest set such that:

1. $q \in closure(I)$ if $q \in I$
2. $(A_1 \rightarrow \bullet\gamma) \in closure(I)$ if $(A_1 \rightarrow \gamma) \in R$ and $(A_0 \rightarrow \alpha \bullet A_1 \beta) \in closure(I)$

We use $\mathcal{I}$ to denote the set of all items for a set of productions, and 2^X to denote the power set of X. A function *goto* from $2^{\mathcal{I}} \times (N \cup T)$ to $2^{\mathcal{I}}$ is defined in [2] as:

$$goto(I, B) = closure(\{A \rightarrow \alpha B \bullet \beta \,:\, (A \rightarrow \alpha \bullet B\beta) \in I\})$$

We also need a function *follow* from N to $2^{T\cup\{\$\}}$, which is defined as:

$$follow(A) = \{c : c \in T \text{ and } S' \stackrel{*}{\Rightarrow} \alpha A c \beta\} \cup \{\$: S' \stackrel{*}{\Rightarrow} \alpha A\}$$

An *LR stack* is an alternating sequence of sets of items and grammar symbols. Let G be a context-free grammar and $\alpha = c_1 c_2 \ldots c_n c_{n+1}$ an input string, where $c_{n+1} = \$$. A *configuration* for G and α is an ordered pair consisting of an LR stack and a suffix of the input string inductively defined as:

1. **(base case)** $(closure(\{S' \rightarrow \bullet S\}),\ c_1 c_2 \ldots c_n c_{n+1})$ is a configuration, which we call an *initial configuration.*
2. **(shift)** If
   (a) $(I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k,\ c_i c_{i+1} \ldots c_n c_{n+1})$ is a configuration,
   (b) there is an item $(A \rightarrow \alpha \bullet c_i \beta) \in I_k$, and
   (c) $goto(I_k, c_i) = J$,

   then $(I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k c_i J,\ c_{i+1} \ldots c_n c_{n+1})$ is also a configuration.
3. **(reduce)** If
   (a) $(I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k,\ c_i \ldots c_n c_{n+1})$ is a configuration,
   (b) there is an item $(A \rightarrow \alpha \bullet) \in I_k$,
   (c) $c_i \in follow(A)$,
   (d) $A \neq S'$, and
   (e) $goto(I_{k-|\alpha|}, A) = J$,

   then $(I_0 B_1 I_1 B_2 I_2 \ldots B_{k-|\alpha|} I_{k-|\alpha|} A J,\ c_i \ldots c_n c_{n+1})$ is also a configuration. □

As observed in [2], the grammar symbols in an LR stack are redundant. In [2], such symbols are used for explanation purposes; for us, they are helpful in the soundness and completeness proofs of our inference system.

Notice that it is possible that for some configurations both the shift and the reduce rules be applicable. In addition, it may be possible to apply the reduce rule using more than one production. These situations are sometimes called *conflicts*. Traditionally, when a conflict arises, either the grammar is changed, or the look ahead of symbols is considered. Lang noted [16], however, that *any context-free grammar can be parsed with an LR parser if we regard a conflict as a nondeterministic choice point.* This fact is important for us, since we will use Prolog's backtracking to handle conflicts.

This description of SLR parsers is useful for proving some of their properties. In a practical SLR parser, however, we would precompute all sets of items that can occur in any configuration, as well as all the values which *follow* and *goto* can have. From these values, the so-called *parsing table* is constructed.

3. CSLR INFERENCE: AN INFERENCE SYSTEM FOR CHAIN PROGRAMS

In this section, we will incorporate "matching" (one-way unification) to the SLR parser to obtain an inference system.

3.1. CSLR inference

We define a *chain program* as a logic program consisting only of clauses of the form (1.1) and $c(t, t') \leftarrow$, where $\text{var}(t') \subseteq \text{var}(t)$, in which no predicate symbol appears both in the head of a nonunit clause and in a unit clause. Here and throughout the paper, var(t) denotes the set of variables occurring in t. We define a *chain goal* as a goal of the form $\leftarrow s(\mathbf{x}, Z)$, where $\mathbf{x}$ is a ground term and Z a variable.

In a chain program the set of predicate symbols appearing in the head of nonunit clauses is disjoint with the set of predicate symbols appearing in unit clauses. This restriction is only meant to facilitate obtaining the inference system from the parser. (In [26] we did not impose this constraint, but we had to make various changes to the parser, apart from the addition of matching; for instance, we had to modify the definition of *follow* with respect to that of [2].) This is not a serious restriction, since it can be satisfied with the addition of predicates.

The similarity between the usual difference-list representation of context-free grammars and chain programs is evident. Only at the level of unit clauses we find that arguments in chain programs are a generalization of the difference-list representation of terminals. However, if we add matching, SLR parsing techniques apply to this case as well. We will show that this extension of SLR parsing provides a sound and complete inference system for this class of program. To make this extension explicit, we will first transform the clauses of a given chain program into context-free production rules by dropping the arguments. Next, we will add matching to the treatment of terminals.

With a chain program P, we associate a set R_P of productions as follows. We first associate a nonterminal A with each predicate symbol a appearing in the head of a nonunit clause, and we associate a terminal c with each predicate symbol c appearing in a unit clause. In addition, R_P has a production of the form:

$$A \rightarrow B_1 B_2 \ldots B_n \qquad n > 0$$

for each clause in P of the form:

$$a(X_0, X_n) \leftarrow b_1(X_0, X_1),\ b_2(X_1, X_2),\ \ldots,\ b_n(X_{n-1}, X_n) \qquad n > 0$$

Let P be a chain program, $G = \leftarrow s'(\mathbf{x}, Z)$ a chain goal in which s' is a predicate symbol occurring in the head of only one clause of P, and R_P the set of productions associated with P. A *clausal configuration* for $P \cup \{G\}$ is an ordered pair consisting of an LR stack and a ground term, inductively defined as:

1. **(base case)** $(\mathit{closure}(\{S' \rightarrow \cdot S\}),\ \mathbf{x})$ is a clausal configuration, for all ground terms $\mathbf{x}$, which we call an *initial clausal configuration.*
2. **(shift)** If
   (a) $(I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k,\ \mathbf{y})$ is a clausal configuration,
   (b) there is an item $(A \rightarrow \alpha \cdot c\beta) \in I_k$,
   (c) $\mathit{goto}(I_k, c) = J$, and
   (d) there is a clause $(c(t, t') \leftarrow) \in P$ such that $\mathbf{y} = t\theta$ for some substitution θ,

   then $(I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k c J,\ \mathbf{z})$, where $\mathbf{z} = t'\theta$, is also a clausal configuration.
3. **(reduce)** If
   (a) $(I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k,\ \mathbf{y})$ is a clausal configuration,
   (b) there is an item $(A \rightarrow \alpha\, \cdot) \in I_k$,
   (c) i. there is a clause $(c(t, t') \leftarrow) \in P$ such that $\mathbf{y} = t\theta$ for some substitution θ where $c \in \mathit{follow}(A)$; or

ii. $\$ \in follow(A)$,

(d) $A \neq S'$, and

(e) $goto(I_{k-|\alpha|}, A) = J$,

then $(I_0 B_1 I_1 B_2 I_2 \ldots B_{k-|\alpha|} I_{k-|\alpha|} AJ,\ \mathbf{y})$ is also a clausal configuration. □

We call $(I_0 S I_1,\ \mathbf{z})$ a *final clausal configuration*, where $(S' \rightarrow S \bullet) \in I_1$, and $\mathbf{z}$ is a ground term.

Note that the definition of clausal configuration can be viewed as an inference system, which we will call *CSLR inference* (where the "*C*" stands for chain programs). Observe also that, except for conditions (2d) and (3c) a clausal configuration is essentially the same as a configuration. In fact, we first arrived at our inference system fortuitously by implementing an SLR *parser* in Prolog using difference lists, and later discovering that arbitrary ground terms could be used instead of lists.

CSLR inference can be viewed as an instance of a "general resolution scheme" discussed in [10, pages 53–54] by Deransart and Małuszyński. Such schemata construct proof trees of a program by interleaving construction of context-free parse trees with unification of the equations originating from the constructed tree. It is clear that CSLR inference combines construction of the parse trees of the underlying context-free grammar (see Figure 2.1) with unification of the arguments. Because of the chain nature of the program, unification reduces to argument passing, except for the leaves of the tree, where unification reduces to matching. CSLR inference is in fact a "full resolution scheme" as defined in [10, pages 55–56].

3.2. Soundness and completeness of CSLR inference

3.2.1. Soundness of CSLR inference To prove soundness of CSLR inference, we first need to establish a correspondence between clausal configurations and sets of definite clauses.

With an item:

$$q = A \rightarrow B_1 \ldots B_{i-1} \bullet B_i \ldots B_n$$

together with a list $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]$ of ground terms, where $k \geq i-1$, we associate the clause:

$$cl(q, [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]) = \\ a(\mathbf{x}_{k-i+1}, X_n) \leftarrow b_i(\mathbf{x}_k, X_i),\ b_{i+1}(X_i, X_{i+1}),\ \ldots,\ b_n(X_{n-1}, X_n)$$

With a set I of items, together with a list $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]$ of ground terms, we associate the following set of clauses:

$$cl(I, [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]) = \{cl(q, [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]) : q \in I\}$$

With a clausal configuration:

$$C = (I_0 B_1 I_1 B_2 I_2 \ldots B_k I_k,\ \mathbf{x}_k)$$

together with a list $[\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]$ of ground terms, we associate the following set of clauses:

$$cl(C, [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]) = \\ cl(I_0, [\mathbf{x}_0]) \cup cl(I_1, [\mathbf{x}_0, \mathbf{x}_1]) \cup \ldots \cup cl(I_k, [\mathbf{x}_0, \mathbf{x}_1, \ldots, \mathbf{x}_k]) \cup \\ \{b_1(\mathbf{x}_0, \mathbf{x}_1) \leftarrow,\ b_2(\mathbf{x}_1, \mathbf{x}_2) \leftarrow,\ \ldots,\ b_k(\mathbf{x}_{k-1}, \mathbf{x}_k) \leftarrow\}$$

Now we need a property of SLR parsers.

Lemma 3.1. Let $C = (I_0 \ldots B_k I_k, \mathbf{x})$ *be a clausal configuration. If* $(A \rightarrow \alpha \bullet \beta) \in I_i$*, then* $(A \rightarrow \bullet \alpha\beta) \in I_{i-|\alpha|}$ *for all* $i = 0, \ldots, k$.

Proof. By induction on the number of steps in the construction of C. □

Lemma 3.2. Let $C = (I_0 \ldots B_k I_k, \mathbf{x})$ *be a clausal configuration. If* $(A \rightarrow \alpha \bullet \beta) \in I_k$*, then* $\alpha = B_{k-|\alpha|+1} \ldots B_k$.

Proof. By induction on the number of steps in the construction of C, using Lemma 3.1. □

Next, we can establish the soundness of the shift and reduce rules.

Lemma 3.3 Soundness of shift. Let P *be a chain program,* $C_1 = (I_0 \ldots B_k I_k,\ \mathbf{x}_k)$ *a clausal configuration for* $P \cup \{\leftarrow s'(\mathbf{x}_0, Z)\}$*, and* $[\mathbf{x}_0, \ldots, \mathbf{x}_k]$ *a list of ground terms. Assume that conditions (a)–(d) of the shift rule hold and let* $C_2 = (I_0 \ldots B_k I_k c J,\ \mathbf{x}_{k+1})$ *be the clausal configuration resulting from applying the shift rule. Then*

$$P \cup cl(C_1, [\mathbf{x}_0, \ldots, \mathbf{x}_k]) \models cl(C_2, [\mathbf{x}_0, \ldots, \mathbf{x}_k, \mathbf{x}_{k+1}])$$

Proof. We have to prove that:

$$P \cup cl(C_1, [\mathbf{x}_0, \ldots, \mathbf{x}_k]) \models cl(J, [\mathbf{x}_0, \ldots, \mathbf{x}_k, \mathbf{x}_{k+1}]) \cup \{c(\mathbf{x}_k, \mathbf{x}_{k+1}) \leftarrow\}$$

By condition (d), $c(\mathbf{x}_k, \mathbf{x}_{k+1}) \leftarrow$ is an instance of a clause in P. Hence, $P \models \{c(\mathbf{x}_k, \mathbf{x}_{k+1}) \leftarrow\}$.

Using resolution, with input clauses $c(\mathbf{x}_k, \mathbf{x}_{k+1}) \leftarrow$ and clauses of the form $cl(A \rightarrow \alpha \bullet c\beta, [\mathbf{x}_0, \ldots, \mathbf{x}_k])$, where $(A \rightarrow \alpha \bullet c\beta) \in I_k$, we can derive all clauses of the form $cl(A \rightarrow \alpha c \bullet \beta, [\mathbf{x}_0, \ldots, \mathbf{x}_k, \mathbf{x}_{k+1}])$, where $(A \rightarrow \alpha c \bullet \beta) \in J$.

Using instantiation, we can derive the rest of the clauses in $cl(J, [\mathbf{x}_0, \ldots, \mathbf{x}_k, \mathbf{x}_{k+1}])$, which are of the form $cl(A' \rightarrow \bullet \gamma, [\mathbf{x}_0, \ldots, \mathbf{x}_k, \mathbf{x}_{k+1}])$.

We conclude that the lemma holds. □

Lemma 3.4 Soundness of reduce. Let P *be a chain program,* $C_1 = (I_0 \ldots B_k I_k,\ \mathbf{x}_k)$ *a clausal configuration for* $P \cup \{\leftarrow s'(\mathbf{x}_0, Z)\}$*, and* $[\mathbf{x}_0, \ldots, \mathbf{x}_k]$ *a list of ground terms. Assume that conditions (a)–(e) of the reduce rule hold and let* $C_2 = (I_0 \ldots B_{k-|\alpha|} I_{k-|\alpha|} A J,\ \mathbf{x}_k)$ *be the clausal configuration resulting from applying the reduce rule. Then*

$$P \cup cl(C_1, [\mathbf{x}_0, \ldots, \mathbf{x}_k]) \models cl(C_2, [\mathbf{x}_0, \ldots, \mathbf{x}_{k-|\alpha|}, \mathbf{x}_k])$$

Proof. We have to prove that:

$$P \cup cl(C_1, [\mathbf{x}_0, \ldots, \mathbf{x}_k]) \models cl(J, [\mathbf{x}_0, \ldots, \mathbf{x}_{k-|\alpha|}, \mathbf{x}_k]) \cup \{a(\mathbf{x}_{k-|\alpha|}, \mathbf{x}_k) \leftarrow\}$$

By condition (b), there is an item $(A \rightarrow \alpha \bullet) \in I_k$. By Lemma 3.2, $\alpha = B_{k-|\alpha|+1} \ldots B_k$. So, there exists a clause:

$$[a(X_{k-|\alpha|}, X_k) \leftarrow b_{k-|\alpha|+1}(X_{k-|\alpha|}, X_{k-|\alpha|+1}),\ \ldots,\ b_k(X_{k-1}, X_k)] \in P$$

Hence, by instantiation and modus ponens,

$$P \cup \{b_1(\mathbf{x}_0, \mathbf{x}_1) \leftarrow,\ \ldots,\ b_k(\mathbf{x}_{k-1}, \mathbf{x}_k) \leftarrow\} \models a(\mathbf{x}_{k-|\alpha|}, \mathbf{x}_k) \leftarrow$$

Using resolution, with input clauses $a(\mathbf{x}_{k-|\alpha|}, \mathbf{x}_k) \leftarrow$ and clauses of the form

$cl(A' \rightarrow \alpha \bullet A\beta, [\mathbf{x}_0, \ldots, \mathbf{x}_{k-|\alpha|}])$, where $(A' \rightarrow \alpha \bullet A\beta) \in I_{k-|\alpha|}$, we can derive all clauses of the form $cl(A' \rightarrow \alpha A \bullet \beta, [\mathbf{x}_0, \ldots, \mathbf{x}_{k-|\alpha|}, \mathbf{x}_k])$ where $(A' \rightarrow \alpha A \bullet \beta) \in J$.

Using instantiation, we can derive the rest of the clauses in $cl(J, [\mathbf{x}_0, \ldots, \mathbf{x}_{k-|\alpha|}, \mathbf{x}_k])$. The derived clauses are of the form $cl(A'' \rightarrow \bullet \gamma, [\mathbf{x}_0, \ldots, \mathbf{x}_{k-|\alpha|}, \mathbf{x}_k])$.

We conclude that the lemma holds. □

We can now state the soundness of CSLR inference.

Theorem 3.1 Soundness of CSLR inference. Let P be a chain program. If $(I_0 S I_1,\ \mathbf{z})$ is a final clausal configuration for the initial clausal configuration $(I_0,\ \mathbf{x})$, then $P \models s'(\mathbf{x}, \mathbf{z})$, for any ground term $\mathbf{x}$.

PROOF. By induction on the number of steps in the construction of $(I_0 S I_1,\ \mathbf{z})$, using Lemmas 3.3 and 3.4. □

3.2.2. COMPLETENESS OF CSLR INFERENCE Finally, we give a completeness result, which states that the search space contains all correct answers. We will proceed in a manner similar to that used in [10, pages 59–60], and [31].

Let P be a logic program. A *closed proof tree* for P is a finite tree of atoms such that for all nodes:

1. there is a unit clause $(A' \leftarrow) \in P$ and a substitution θ such that $A \equiv A'\theta$ and A has no children, or
2. there is a clause $(A' \leftarrow B_1,\ \ldots,\ B_n) \in P$ and a substitution θ such that $A \equiv A'\theta$ and $B_1\theta,\ \ldots,\ B_n\theta$ are the children of A.

Lemma 3.5 Clark [7], Stärk [31]. Let P be a logic program and A a ground atom. If $P \models A$, then there exists a closed proof tree for P with root A.

Theorem 3.2. Let P be a chain program, and $\mathbf{w}, \mathbf{x}$, and $\mathbf{z}$ be ground terms. Let $C_1 = (I_0 \ldots B_k I_k,\ \mathbf{x})$ be a clausal configuration for $P \cup \{\leftarrow s'(\mathbf{w}, Z)\}$ such that $(A \rightarrow\ \bullet\ B'_1 B'_2 \ldots B'_n) \in I_k$. Let $J = goto(I_{k-n}, A)$. If $P \models a(\mathbf{x}, \mathbf{z})$, then $C_2 = (I_0 \ldots B_k I_k A J,\ \mathbf{z})$ is also a clausal configuration for $P \cup \{\leftarrow s'(\mathbf{w}, Z)\}$.

PROOF. The proof is by induction on the number r of nodes of the closed proof tree T with root $a(\mathbf{x}, \mathbf{z})$.

Basis: T has exactly two nodes, because the set of predicate symbols associated with terminals is disjoint with the set of predicate symbols associated with nonterminals. In this case, there exist clauses $(a(X_0, X_1) \leftarrow b'_1(X_0, X_1)) \in P$ and $(b'_1(t, t') \leftarrow) \in P$, where a corresponds to a nonterminal, and b'_1 to a terminal; in addition, $(b'_1(t, t'))\theta \equiv b'_1(\mathbf{x}, \mathbf{z})$. Viewed in terms of CSLR inference, we can shift the symbol B'_1 obtaining the clausal configuration $(I_0 \ldots B_k I_k B'_1 I_{k+1},\ \mathbf{z})$. Because $(A \rightarrow B'_1 \bullet) \in I_{k+1}$, we can then reduce by the production $A \rightarrow B'_1$ and yield the clausal configuration $C_2 = (I_0 \ldots B_k I_k A J,\ \mathbf{z})$.

Inductive step: Next, suppose that the result holds for closed proof trees of $r' < r$ nodes. Let the children of $a(\mathbf{x}, \mathbf{z})$ be $b'_1(\mathbf{x}, \mathbf{y}_1),\ b'_2(\mathbf{y}_1, \mathbf{y}_2),\ \ldots,\ b'_n(\mathbf{y}_{n-1}, \mathbf{z})$ in order from the left.

If b'_1 corresponds to a terminal, we can shift B'_1 to obtain $(I_0 \ldots B_k I_k B'_1 I_{k+1},\ \mathbf{y}_1)$, where $(A \rightarrow B'_1 \bullet B'_2 \ldots B'_n) \in I_{k+1}$. If b'_1 corresponds to a nonterminal, we can apply the induction hypothesis for a closed proof tree with root $b'_1(\mathbf{x}, \mathbf{y}_1)$, which asserts that if $C_1 = (I_0 \ldots B_k I_k, \mathbf{x})$ is a clausal configuration for $P \cup \{\leftarrow s'(\mathbf{w}, Z)\}$ such that $(B'_1 \rightarrow\ \bullet\ B''_1 \ldots) \in I_k$, then $C_2 = (I_0 \ldots B_k I_k B'_1 I_{k+1},\ \mathbf{y}_1)$ is also a clausal configuration for $P \cup \{\leftarrow s'(\mathbf{w}, Z)\}$. Note that also in this case $(A \rightarrow B'_1 \bullet B'_2 \ldots B'_n) \in I_{k+1}$.

We can now construct the clausal configuration:

$$(I_0 \ldots B_k I_k B'_1 I_{k+1} B'_2 I_{k+2} \ldots B'_n I_{k+n},\ \mathbf{z})$$

such that $(A \rightarrow B'_1 B'_2 \ldots B'_n \bullet) \in I_{k+n}$ as follows. For $i = 2, \ldots, n$, we apply the shift rule if b'_i corresponds to a terminal or the induction hypothesis if b'_i corresponds to a nonterminal.

Finally, we can apply the reduction rule, to obtain the clausal configuration $(I_0 \ldots B_k I_k AJ,\ \mathbf{z})$. We conclude that the theorem holds. □

Corollary 3.1 Completeness of CSLR inference. Let P be a chain program. If $P \models s'(\mathbf{x}, \mathbf{z})$, where $\mathbf{x}$ and $\mathbf{z}$ are ground terms, then $(I_0 S I_1,\ \mathbf{z})$ is a final clausal configuration for the initial clausal configuration $(I_0,\ \mathbf{x})$.

4. TRANSFORMING FIXED-MODE PROGRAMS INTO CHAIN FORM

This section deals with our transformation for converting fixed-mode logic programs into chain form.

4.1. Overview of the transformation

In looking for a class of program transformable into chain form, we would like, of course, to find a class as large as possible. "State-oriented" (imperative) programs might give us a guide as to which programs are transformable, for two reasons:

1. Various authors have observed [5, 12, 20, 27] that a state-oriented program can be regarded as a context-free grammar together with an interpretation representing primitive commands as terminals. Such a view is closely related to the concept of chain program.
2. Clark and van Emden have described [6] flowcharts (a kind of state-oriented program) with chain logic programs, in such a way that a flowchart and its associated logic program define the same set of computations.

Hence, state-oriented programs in general, and flowcharts in particular, must have certain properties making them suitable of being represented by chain programs.

A first conspicuous property is that during execution, boxes in a flowchart are "traversed" in only one direction, from their input towards their output. This suggests limiting ourselves to logic programs in which each predicate has only one "mode," i.e. each argument place is used either as input (instantiated) or as output (uninstantiated), but not both. Another property of flowcharts is that the output of a box is never connected to the output of another box. Hence, we will exclude clauses in which a variable occurs as output in more than one subgoal. We will add a third condition, which is only meant to simplify both stating our transformation and proving it correct.

A clause:

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1),\ p_2(t'_1, t_2),\ \ldots,\ p_n(t'_{n-1}, t_n) \qquad n \geq 0$$

is called *fixed-mode* if:

1. $\mathrm{var}(t'_i) \subseteq \mathrm{var}(t_0) \cup \cdots \cup \mathrm{var}(t_i)$, for $i = 0, \ldots, n$;
2. $\mathrm{var}(t_i) \cap \mathrm{var}(t_j) = \emptyset$, for $i, j = 0, \ldots, n$ and $i \neq j$; and

3. each variable occurring in t'_i occurs only once in t'_i, for $i = 0, \ldots, n$, if $n > 0$.

We define a *fixed-mode* program as a logic program consisting only of fixed-mode clauses. A goal is called *fixed-mode* if it is of the form $\leftarrow s(\mathbf{x}, Z)$, where $\mathbf{x}$ is a ground term and Z a variable.

Condition 1 constrains the "flow of data" [4] from the inputs towards the outputs, if subgoals are selected in a left-to-right order. When a subgoal succeeds, condition 2 causes the constructed term to have an effect only on the input of other subgoals. Condition 3 is included with no loss of generality, since it can be easily satisfied as follows. Note first that unit clauses which satisfy condition 1 are already in chain form. Hence there is no need to impose further constraints on such clauses, so that condition 3 only refers to nonunit clauses. Thus, a clause of the form:

$$p(t_0, \langle X, X \rangle) \leftarrow \ldots,\ q(t_i, \langle X \rangle),\ \ldots$$

for example, which has two occurrences of X in t'_n, can be replaced by:

$$p(t_0, \langle X', X'' \rangle) \leftarrow \ldots,\ q(t_i, \langle X \rangle),\ e(\langle X \rangle, \langle X', X'' \rangle),\ \ldots$$
$$e(\langle X \rangle, \langle X, X \rangle) \leftarrow$$

At first sight, fixed-mode programs may appear too constrained for practical purposes, because sometimes the same Prolog predicate can be used in multiple modes. Often, however, obstacles appear when using a predicate in this manner. For instance, an infinite branch to the left of an answer in the SLD tree will prevent Prolog from finding such an answer. Even if all branches are finite, the execution of certain modes may be intolerably inefficient. Other obstacles to the invertibility of logic programs are the use of built-in predicates and the lack of occur check. As a consequence, many practical Prolog programs use each predicate in a single mode. We refer the reader to [3, 4, 10, 11] for arguments in favor of fixed-mode programs.

Example 4.1. We will introduce our transformation through the following fragment of a DCG:

$$expr(p(E, F)) \rightarrow expr(E),\ plus,\ expr(F) \tag{4.1}$$
$$plus \rightarrow [+] \tag{4.2}$$

Clauses (4.1) and (4.2) are a shorthand for:

$$expr(p(E, F), X_0, X_3) \leftarrow expr(E, X_0, X_1),\ plus(X_1, X_2),\ expr(F, X_2, X_3) \tag{4.3}$$
$$plus([+|Xs], Xs) \leftarrow \tag{4.4}$$

Assume a goal of the form $\leftarrow expr(Expr, [a, +, b], Rest)$, in which the second argument place is used as input, and the first and third argument places as output.

As a first step to obtain chain clauses, we rewrite clause (4.3) grouping some input and output argument places as:

$$\begin{aligned} &expr'(\langle X_0 \rangle, \langle X_3, p(E_1, F_3) \rangle) \leftarrow \\ &\quad expr'(\langle X_0 \rangle, \langle X_1, E_1 \rangle),\ plus(\langle X_1 \rangle, \langle X_2 \rangle),\ expr'(\langle X_2 \rangle, \langle X_3, F_3 \rangle) \end{aligned} \tag{4.5}$$

in order to have binary predicates. The angled brackets $\langle\ \rangle$ are used instead of ordinary brackets [ ], for grouping input and output arguments.

The next step in obtaining chain clauses would be to convert the arguments that are not variables into variables. Note that if it were not for the fact that E_1 occurs both in the output of the head and in the output of the first subgoal,

$$\begin{array}{l} expr'(\langle X_0\rangle, \underline{\langle X_3, p(\boxed{E_1}, F_3)\rangle}) \leftarrow \\ \qquad expr'(\langle X_0\rangle, \langle X_1, \boxed{E_1}\rangle),\ plus(\langle X_1\rangle, \langle X_2\rangle),\ expr'(\langle X_2\rangle, \underline{\langle X_3, F_3\rangle}) \end{array} \tag{4.5}$$

we would be able to "fold" the definition of a predicate such as:

$$g(\langle X_3, F_3\rangle, \langle X_3, p(E_1, F_3)\rangle) \leftarrow$$

converting the underlined terms into variables as follows:

$$\begin{array}{l} expr'(\langle X_0\rangle, U_3') \leftarrow \\ \qquad expr'(\langle X_0\rangle, \langle X_1, \boxed{E_1}\rangle),\ plus(\langle X_1\rangle, \langle X_2\rangle),\ expr'(\langle X_2\rangle, U_3),\ g(U_3, U_3') \end{array} \tag{4.6}$$

As observed by Tamaki and Sato [32], in general it is incorrect to fold a definition such as that of g in a clause having a variable such as E_1, which also occurs in the output of the first subgoal. To see this, unfold the definition of g in (4.6), and observe that a generalization of (4.5) is obtained.

For us, the occurrences of E_1 in (4.5) are reminiscent of procedure calls in state-oriented programs, in the sense that the value of E_1 is not "required" by the second or third subgoals, but is used later in the computation to construct the result of the whole clause (the term $\langle X_3, p(E_1, F_3)\rangle$). Procedure calls are sometimes implemented with a *stack* storing the values of the variables required after the call is executed. "Interruptions" are often implemented in this manner. This suggests adding a term to the predicates which plays the role of such a stack. This stack would store the value of E_1 in the first subgoal, until used by the head, at which point this value is recovered with a different name E_3:

$$\begin{array}{l} \widehat{expr}(\langle St_0, X_0\rangle, \langle St_3, X_3, p(\boxed{E_3}, F_3)\rangle) \leftarrow \\ \qquad\qquad \widehat{expr}(\langle St_0, X_0\rangle, \langle St_1, X_1, \boxed{E_1}\rangle), \\ \qquad\qquad \widehat{plus}(\langle [E_1|St_1], X_1\rangle, \langle [E_2|St_2], X_2\rangle), \\ \qquad\qquad \widehat{expr}(\langle [E_2|St_2], X_2\rangle, \langle [E_3|St_3], X_3, F_3\rangle) \end{array} \tag{4.7}$$

With appropriate definitions for $\widehat{expr}$ and $\widehat{plus}$, it is possible to transform (4.5) into (4.7) through fold and unfold steps [24].

We can now fold the following definition of h_3:

$$h_3(\langle [E_3|St_3], X_3, F_3\rangle, \langle St_3, X_3, p(E_3, F_3)\rangle) \leftarrow$$

and get:

$$\begin{array}{l} \widehat{expr}(\langle St_0, X_0\rangle, U_3') \leftarrow \widehat{expr}(\langle St_0, X_0\rangle, \langle St_1, X_1, E_1\rangle), \\ \qquad\qquad\qquad\qquad \widehat{plus}(\langle [E_1|St_1], X_1\rangle, \langle [E_2|St_2], X_2\rangle), \\ \qquad\qquad\qquad\qquad \widehat{expr}(\langle [E_2|St_2], X_2\rangle, U_3), \\ \qquad\qquad\qquad\qquad h_3(U_3, U_3') \end{array}$$

If we take the completed definition [19, page 78] of h_3, then this step preserves all models of the program augmented with the standard equality theory. This can be proved using for example, the result in [29, pages 57, 58].

Similarly, folding the definitions of h_0, h_1, and h_2:

$$h_0(\langle St_0, X_0\rangle, \langle St_0, X_0\rangle) \leftarrow$$

$$h_1(\langle St_1, X_1, E_1\rangle, \langle [E_1|St_1], X_1\rangle) \leftarrow$$
$$h_2(\langle [E_2|St_2], X_2\rangle, \langle [E_2|St_2], X_2\rangle) \leftarrow$$

we obtain:

$$\begin{aligned} \widehat{expr}(U_0, U_3') \leftarrow\; & h_0(U_0, U_0'),\ \widehat{expr}(U_0', U_1), \\ & h_1(U_1, U_1'),\ \widehat{plus}(U_1', U_2), \\ & h_2(U_2, U_2'),\ \widehat{expr}(U_2', U_3),\ h_3(U_3, U_3') \end{aligned} \tag{4.8}$$

which has chain form. Now the contextual information appears only at the added leaves of the parse/proof trees.

Observe that h_0 denotes a subset of the identity relation, so that this predicate does not change the state of the computation. This suggests the possibility of deleting the subgoal with such a predicate symbol and renaming variables so that chain form is preserved. We will show that the program resulting from this deletion is a logical consequence of the completion of the original program, together with the standard equality theory. We will also see that this operation may not be sound in general, and will give a sufficient condition for its soundness.

4.2. Transformation

In practice, it may not be convenient to transform a program with fold and unfold operations. The chain form of a fixed-mode program can be obtained in a more straightforward manner based on the following theorem. A proof sketch appears in [25], and so we do not repeat it here. Such a sketch proof is not difficult, and it essentially follows the same steps we followed for Example 4.1.

Theorem 4.1. Let C be a fixed-mode clause:

$$p_0(t_0, t_n') \leftarrow p_1(t_0', t_1),\ p_2(t_1', t_2),\ \ldots,\ p_n(t_{n-1}', t_n) \qquad n \geq 0$$

and let

$$\Pi_i = (var(t_0) \cup \cdots \cup var(t_{i-1})) \cap (var(t_i') \cup \cdots \cup var(t_n')) \qquad i = 0, \ldots, n+1$$

Then the clause $\hat{C}$:

$$\begin{aligned} \hat{p}_0(U_0, U_n') \leftarrow h_0(U_0, U_0'),\ \hat{p}_1(U_0', U_1),\ h_1(U_1, U_1'),\ \hat{p}_2(U_1', U_2),\ \ldots, \\ h_{n-1}(U_{n-1}, U_{n-1}'),\ \hat{p}_n(U_{n-1}', U_n),\ h_n(U_n, U_n') \end{aligned}$$

is logically implied by C, the standard equality theory, the "iff" version of the function substitutivity axiom for the list-constructor function symbol:

$$[X|Y] = [X'|Y'] \leftrightarrow X = X' \;\&\; Y = Y'$$

the definitions of the $\hat{p}_i$'s:

$$\hat{p}_i(\langle St|X\rangle, \langle St'|Y\rangle) \leftrightarrow St = St' \;\&\; p_i(X, Y) \qquad i = 0, \ldots, n$$

and the completed definitions of:

$$h_i(\langle \Sigma_i|t_i\rangle, \langle \Sigma_{i+1}|t_i'\rangle) \leftarrow \qquad i = 0, \ldots, n$$

where Σ_i is any list of the form $[X_{1,i}, \ldots, X_{d_i,i}|St]$, such that $\{X_{1,i}, \ldots, X_{d_i,i}\} = \Pi_i$, if $\Pi_i \neq \emptyset$, and Σ_i is St if $\Pi_i = \emptyset$, for $i = 0, \ldots, n+1$.

Each predicate with the predicate symbol h_i above will be called an *h-command*. Each subgoal of an h-command will be called an *h-subgoal*. We define the *chain form* $\hat{P}_C$ of a fixed-mode clause C as the clause $\hat{C}$ produced by Theorem 4.1, together with the definitions of the h-commands in definite-clause form (as opposed to their completed definitions). We also define the *chain form* $\hat{P}$ of a fixed-mode program P as the chain form $\hat{P}_C$ of each clause C in P, having no h-command in common with the chain form of other clauses in P.

Let us apply Theorem 4.1 to clause (4.5) of Example 4.1.

$$\overbrace{expr'}^{p_0}(\underbrace{\langle X_0\rangle}_{t_0}, \underbrace{\langle X_3, p(E_1, F_3)\rangle}_{t'_3}) \leftarrow$$

$$\overbrace{expr'}^{p_1}(\underbrace{\langle X_0\rangle}_{t'_0}, \underbrace{\langle X_1, E_1\rangle}_{t_1}),\ \overbrace{plus}^{p_2}(\underbrace{\langle X_1\rangle}_{t'_1}, \underbrace{\langle X_2\rangle}_{t_2}),\ \overbrace{expr'}^{p_3}(\underbrace{\langle X_2\rangle}_{t'_2}, \underbrace{\langle X_3, F_3\rangle}_{t_3})$$

Π_i is the set of variables which receive a substitution in the output of a subgoal to the left of the subgoal with predicate symbol p_i or in the input of the head, and which occur in the input of a subgoal to the right of the subgoal with predicate symbol p_i or in the output of the head.

Hence, Π_i is the set of variables in the stack of the subgoal with predicate symbol p_i.

$$\begin{array}{lcll}
\Pi_0 &=& \emptyset \ \cap\ (\mathrm{var}(t'_0) \cup \mathrm{var}(t'_1) \cup \mathrm{var}(t'_2) \cup \mathrm{var}(t'_3)) &= \emptyset \\
\Pi_1 &=& \mathrm{var}(t_0) \ \cap\ (\mathrm{var}(t'_1) \cup \mathrm{var}(t'_2) \cup \mathrm{var}(t'_3)) &= \emptyset \\
\Pi_2 &=& (\mathrm{var}(t_0) \cup \mathrm{var}(t_1)) \ \cap\ (\mathrm{var}(t'_2) \cup \mathrm{var}(t'_3)) &= \{E_1\} \\
\Pi_3 &=& (\mathrm{var}(t_0) \cup \mathrm{var}(t_1) \cup \mathrm{var}(t_2)) \ \cap\ \mathrm{var}(t'_3) &= \{E_1\} \\
\Pi_4 &=& (\mathrm{var}(t_0) \cup \mathrm{var}(t_1) \cup \mathrm{var}(t_2) \cup \mathrm{var}(t_3)) \ \cap\ \emptyset &= \emptyset
\end{array}$$

So,

$$\Sigma_0 = \Sigma_1 = St, \quad \Sigma_2 = \Sigma_3 = [E_1|St], \quad \Sigma_4 = St$$

The clause in chain form is (4.8). The definitions of the h_i's are as before, up to variable renaming:

$$h_0(\langle St, X_0\rangle, \langle St, X_0\rangle) \leftarrow$$
$$h_1(\langle St, X_1, E_1\rangle, \langle [E_1|St], X_1\rangle) \leftarrow$$
$$h_2(\langle [E_1|St], X_2\rangle, \langle [E_1|St], X_2\rangle) \leftarrow$$
$$h_3(\langle [E_1|St], X_3, F_3\rangle, \langle St, X_3, p(E_1, F_3)\rangle) \leftarrow$$

Theorem 4.1 associates a chain program $\hat{P}$ with a fixed-mode program P in such a way that $\hat{P}$ is a logical consequence of a conservative extension of P. The implication in the other direction also holds. That P is logically implied by a conservative extension of $\hat{P}$ can be seen by resolving the clauses in $\hat{P}$ first with the definitions of the h_i's and the $\hat{p}_i$'s, and then with reflexivity.

5. SOME PROPERTIES OF SLR INFERENCE

Having presented CSLR inference, we will now couple such an inference system with our transformation for converting fixed-mode programs into chain form. This coupling can be regarded as producing a new inference system, because the transformation can be described in terms of fold and unfold operations. So, this new inference system has additional inference rules which are applied (at compile time) before the shift and reduce rules of CSLR inference.

We will first observe that the length of a refutation using CSLR inference on a program resulting from our transformation has a number of steps which is proportional to the length of the corresponding SLD refutation of the original program. This system will be called *hSLR inference*.

Finally, we will remark that under certain conditions, it is possible to remove some subgoals of predicates which manipulate the stack added by the transformation (h-subgoals), thus producing a search space smaller than that of SLD resolution. The removal of such subgoals can also be described in terms of fold and unfold operations, so that another inference system is obtained: *SLR inference*.

5.1. A comparison of search spaces

Example 5.1. As an example, consider the usual definition of *append*, but used for splitting lists. The predicate $ap(\langle Z\rangle, \langle X, Y\rangle)$ is intended to hold if Z can be split into X followed by Y:

$$ap(\langle Z\rangle, \langle [\,], Z\rangle) \leftarrow$$
$$ap(\langle [W|Z]\rangle, \langle [W|X], Y\rangle) \leftarrow ap(\langle Z\rangle, \langle X, Y\rangle)$$

The chain program produced by our transformation is:

$$\widehat{ap}(U_0, U_0') \leftarrow h_0(U_0, U_0')$$
$$\widehat{ap}(U_0, U_1') \leftarrow h_1(U_0, U_0'),\ \widehat{ap}(U_0', U_1),\ h_2(U_1, U_1')$$
$$h_0(\langle St, Z\rangle, \langle St, [\,], Z\rangle) \leftarrow$$
$$h_1(\langle St, [W|Z]\rangle, \langle [W|St], Z\rangle) \leftarrow$$
$$h_2(\langle [W|St], X, Y\rangle, \langle St, [W|X], Y\rangle) \leftarrow$$

The context-free grammar associated with the above chain program is:

(0) $S' \rightarrow A$
(1) $A \rightarrow h_0$
(2) $A \rightarrow h_1 A h_2$

Next, we compute the sets of items:

$$I_0 = \{S' \rightarrow \bullet A,\ \ A \rightarrow \bullet h_0,\ \ A \rightarrow \bullet h_1 A h_2\}$$
$$I_1 = \{S' \rightarrow A \bullet\}$$
$$I_2 = \{A \rightarrow h_1 \bullet A h_2,\ \ A \rightarrow \bullet h_0,\ \ A \rightarrow \bullet h_1 A h_2\}$$
$$I_3 = \{A \rightarrow h_0 \bullet\}$$
$$I_4 = \{A \rightarrow h_1 A \bullet h_2\}$$
$$I_5 = \{A \rightarrow h_1 A h_2 \bullet\}$$

Finally, we obtain the following parsing table (using essentially the notation of [2]):

set of items	action				goto
	h_0	h_1	h_2	\$	A
I_0	s3	s2			1
I_1				acc	
I_2	s3	s2			4
I_3			r1	r1	
I_4			s5		
I_5			r2	r2	

Figure 5.1 shows the search space determined by hSLR inference for the *append* program and the term $\langle [\,], [a, b] \rangle$.

The search space in Figure 5.1 illustrates two aspects. First, although the parsing table has no conflicts, the sequences of clausal configurations have a branching structure. Such a structure appears because the ground term of certain clausal configurations unifies with more than one input of an h-command.

Second, this search space resembles that of SLD resolution for the original program and goal (Figure 5.2). In fact, we will see that: (1) except for some *linear* components (like the one enclosed in a rectangle), the search space of hSLR inference and the search space of SLD resolution with a leftmost computation rule are isomorphic, and (2) such linear components increase the length of each SLD refutation only by a number of nodes proportional to its length.

For instance, by replacing some linear components by a single node each in Figure 5.1, we get a tree which is isomorphic to the SLD tree in Figure 5.2. Now consider the linear component enclosed in a rectangle. Each operation in such a component is caused by an operation applied to an ancestor. In particular, the "r1" operation enclosed in a box is produced by the "s3" operation also enclosed in a box. Both can be viewed as applying the production $A \rightarrow h_0$. Similarly, the encircled "s5" and "r2" are a result of the encircled "s2," which amounts to applying $A \rightarrow h_1 A h_2$.

Thus, apparently hSLR inference does not present any advantage over SLD resolution. In fact, the *size* of the hSLR *search space* may increase considerably with respect to that of SLD resolution. Later, however, we will study the elimination of some h-subgoals in the chain program, which may reduce the size of the search space.

Let P be a chain program G a chain goal. We define a *CSLR tree* for $P \cup \{G\}$ as a tree in which:

1. each node is a clausal configuration for $P \cup \{G\}$,
2. the root is the initial clausal configuration for $P \cup \{G\}$, and
3. each node has one child for each possible application of either the shift or the reduce rule.

Given a fixed-mode program P and a fixed-mode goal G, we define the *hSLR tree* for $P \cup \{G\}$ as the CSLR tree for $\hat{P} \cup \{\hat{G}\}$, where $\hat{P}$ and $\hat{G}$ are obtained from P and G using the transformation of Subsection 4.2. (Figure 5.1 is an example of an hSLR tree.)

Let T be an *hSLR tree*. A *reduce-child* of a node N of T is a child of N obtained by applying the reduce rule. A *shift-child* of a node N of T is a child of N obtained by applying the shift rule. An *initial-child* of a node N of T is a shift-child of N

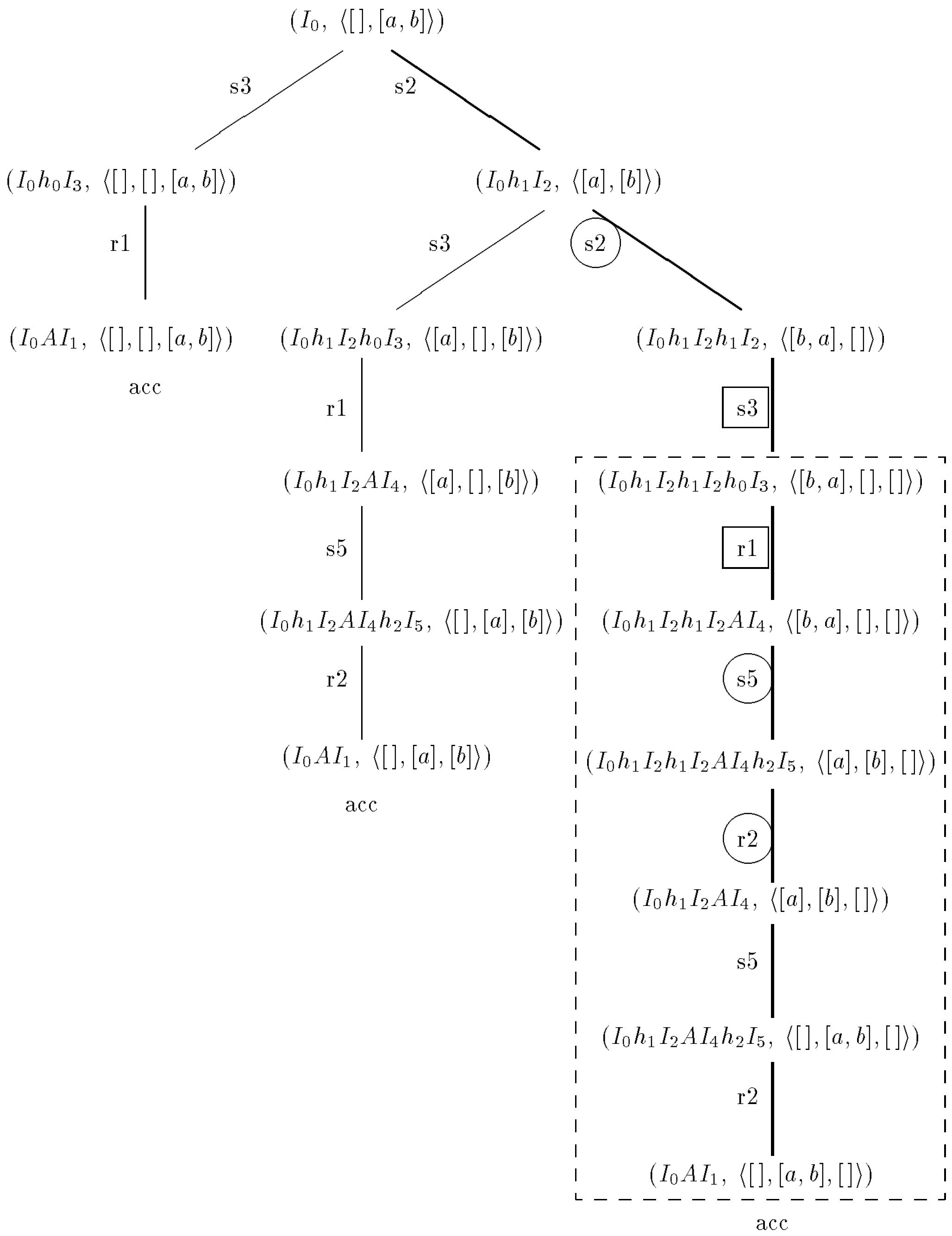

FIGURE 5.1. *The hSLR tree for the append program used for splitting the list* $[a, b]$.

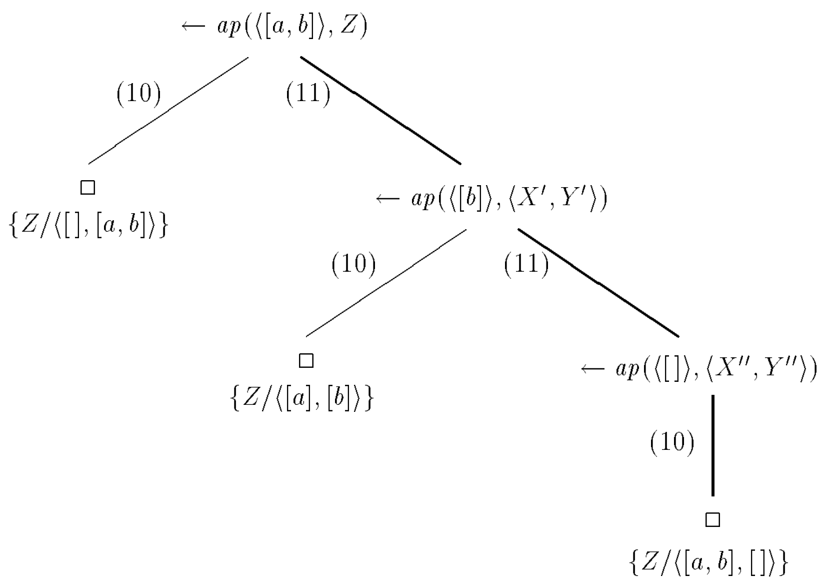

FIGURE 5.2. *The SLD tree for the append program used for splitting the list* $[a,b]$.

obtained by applying the shift rule using the leftmost symbol of a production. (This symbol corresponds to the predicate symbol of an h-command.) A *noninitial-child* of a node N of T is a shift-child of N which is not an initial-child.

Next, we will state a lemma which establishes that only shifts of leftmost symbols of productions may cause branching nodes in an hSLR tree. Hence, the applications of the reduce rule, as well as the applications of the shift rule for symbols other than the leftmost, produce nodes with at most one child.

Lemma 5.1. Let P be a fixed-mode program, G a fixed-mode goal, and T the hSLR tree for $P \cup \{G\}$. Let N be a node of T having either a reduce-child or a noninitial-child. Then N has exactly one child.

PROOF. Observe first that there are no reduce/reduce or shift/reduce conflicts in the parsing table for $R_{\hat{P}}$. A reduce/reduce conflict is caused by a set of items having items $A \rightarrow \alpha B\,\bullet$ and $A' \rightarrow \beta B\,\bullet$. However, all productions in $R_{\hat{P}}$ have distinct rightmost symbols.

In addition, a shift/reduce conflict is caused by a set of items having items $A \rightarrow \alpha B\,\bullet$ and $A' \rightarrow \beta B \bullet \gamma$. However, the rightmost symbol of each production in $R_{\hat{P}}$ does not occur anywhere else.

As a consequence, an application of the reduce rule produces only one child.

Consider now applications of the shift rule (only h's are shifted). A clausal configuration $C = (\ldots I,\ \mathbf{x})$ has more than one shift-child if (i) I has items of the form $A \rightarrow \alpha \bullet h_1 \gamma$ and $A' \rightarrow \beta \bullet h_2 \delta$, where α is a suffix of β, and (ii) $\mathbf{x}$ unifies with the first argument of both h_1 and h_2. However, because the productions in $R_{\hat{P}}$ have no h's in common, α can only be a suffix of β if $\alpha = \beta = \epsilon$. Hence, C may have more than one child only if h_1 and h_2 are predicate symbols of leftmost h-subgoals. $\square$

Let T be an hSLR tree. We define an *R-component* of T as a maximal subtree of T such that: (1) the root C is a clausal configuration with a reduce-child and (2) includes all descendants of C which are either reduce-children or noninitial-children. Note that by Lemma 5.1, an R-component is linear.

We define a *contracted hSLR tree* as a tree obtained from an hSLR tree T such that each R-component of T has been replaced by a single node: the leaf of that R-component.

We are now in a position to establish the following theorem.

Theorem 5.1. Let P be a fixed-mode program and G a fixed-mode goal. Let T be the SLD tree [19, page 55] for $P \cup \{G\}$ in which the leftmost subgoal is selected at each node. Let T' be the contracted hSLR tree for $P \cup \{G\}$. Then T and T' are isomorphic.

Proof. Let N be a node of T corresponding to a node N' of T'. Suppose that the leftmost subgoal of N is $p_i(\mathbf{x}, Y)$. Then N has one child for each clause with predicate symbol p_i in the head, whose first argument unifies with $\mathbf{x}$.

Consider now $N' = (\ldots I,\ \langle \mathbf{st}|\mathbf{x}\rangle)$. Note that T' has no reduce or noninitial children. Then the only way we can apply the shift rule is when a leftmost h-command is used. In this case, I has:

1. one item of the form:

   $\hat{p} \rightarrow h_0\hat{p}_0h_1 \ldots h_i \bullet \hat{p}_ih_{i+1} \ldots$

2. and one item of the form:

   $\hat{p}_i \rightarrow\ \bullet h'_0\hat{r}_0h'_1 \ldots$

   for each clause in $\hat{P}$ with predicate symbol $\hat{p}_i$ in the head.

The transformation of Subsection 4.2 takes a clause with a head having as first argument a term t_0 and produces a clause having as first argument the term $\langle St|t_0\rangle$. Hence, N' will have one shift-child for each clause defining an h-command whose first argument unifies with $\langle \mathbf{st}|\mathbf{x}\rangle$. We conclude that the theorem holds. $\square$

Next, we can bound the length increase of refutations.

Theorem 5.2. Let P be a fixed-mode program and $G = \leftarrow s'(\mathbf{x}, Z)$ a fixed-mode goal. Assume that there is a refutation for $P \cup \{G\}$ of length m which selects the leftmost subgoal at every step. Then a final clausal configuration for $(I_0,\ \mathbf{x})$ can be constructed with $O(m)$ applications of the shift or reduce rules.

Proof. Consider a clause in P of the form:

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1),\ p_2(t'_1, t_2),\ \ldots,\ p_n(t'_{n-1}, t_n) \tag{5.1}$$

so that a goal with a predicate symbol p_0 succeeds if n subgoals succeed. This clause is transformed into a chain clause which is associated with the production:

$$\hat{p}_0 \rightarrow h_0\hat{p}_1h_1\hat{p}_2 \ldots \hat{p}_nh_n$$

Hence, in terms of hSLR inference, from the clausal configuration $(I_0 \ldots B_kI_k,\ \mathbf{y})$, where $(\hat{p}_0 \rightarrow\ \bullet h_0\hat{p}_1h_1\hat{p}_2 \ldots \hat{p}_nh_n) \in I_k$, we need to shift the h_i's, as well as to perform a reduction step, in addition to the application of the $\hat{p}_i$'s ($i \geq 1$). First, we can charge the cost of each shift of h_i to that of applying $\hat{p}_i$, for $i = 1, \ldots, n$.

Similarly, we can charge the cost of shifting h_0 to that of the reduction step. Thus, each SLD resolution step corresponds to at most three steps of CSLR inference: two shifts and one reduction step. $\square$

5.2. *Elimination of some leftmost h-subgoals*

Now, we will logically justify the elimination of certain h-subgoals added by our transformation, which may result in a considerable reduction in the size of the search space.

Theorem 5.3. Let $\hat{P}$ be a chain program obtained from transforming the fixed-mode program P, having a clause C of the form:

$$\hat{p}_0(U_0, U'_n) \leftarrow h_0(U_0, U'_0),\ \hat{p}_1(U'_0, U_1),\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

where h_0 is defined by the clause:

$$h_0(\langle St|t\rangle, \langle St|t\rangle) \leftarrow$$

If $\hat{p}_1$ is defined by:

$$\hat{p}_1(U, V) \leftrightarrow U = \langle St|t'\rangle \ \&\ V = \langle St|t''\rangle \ \&\ \alpha$$

where $t\theta = t'$, then the clause:

$$\hat{p}_0(U_0, U'_n) \leftarrow \hat{p}_1(U_0, U_1),\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

is a logical consequence of C, the definition of h_0, the definition of $\hat{p}_1$, and the standard equality theory.

Proof. First, we unfold the definition of h_0 in C:

$$\underline{\hat{p}_0(\langle St|t\rangle, U'_n)} \leftarrow \hat{p}_1(\langle St|t\rangle, U_1),\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

Next, we apply predicate substitutivity $p(U, V) \leftarrow U = X,\ V = Y,\ p(X, Y)$ for $\hat{p}_0$ followed by reflexivity:

$$\hat{p}_0(U_0, U'_n) \leftarrow U_0 = \langle St|t\rangle,\ \underline{\hat{p}_1(\langle St|t\rangle, U_1)},\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

Then, we apply predicate substitutivity for $\hat{p}_1$ followed by reflexivity:

$$\hat{p}_0(U_0, U'_n) \leftarrow U_0 = \langle St|t\rangle,\ \underline{\langle St|t\rangle = X},\ \hat{p}_1(X, U_1),\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

We eliminate a subgoal by first applying symmetry and then factoring:

$$\hat{p}_0(U_0, U'_n) \leftarrow U_0 = \langle St|t\rangle,\ \underline{\hat{p}_1(U_0, U_1)},\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

Now, we unfold the definition of $\hat{p}_1$:

$$\hat{p}_0(U_0, U'_n) \leftarrow \underline{U_0 = \langle St|t\rangle,\ U = \langle St|t'\rangle},\ V = \langle St|t''\rangle,\ \alpha,\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

Subsequently, we apply factoring:

$$\hat{p}_0(U_0, U'_n) \leftarrow \underline{U_0 = \langle St|t\theta\rangle,\ V = \langle St|t''\rangle,\ \alpha},\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

Finally, we can fold the definition of $\hat{p}_1$ because $t\theta = t'$:

$$\hat{p}_0(U_0, U'_n) \leftarrow \hat{p}_1(U_0, U_1),\ \ldots,\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

□

This theorem deals with the elimination of h_0 in a clause of the form:

$\hat{p}_0 \rightarrow h_0, \hat{p}_1, h_1, \ldots, h_{n-1}, \hat{p}_n, h_n$

However, it is possible to extend this result for the elimination of the leftmost h-subgoal h_i in a clause of the form:

$\hat{p}_0 \rightarrow \hat{p}_1, \hat{p}_2, \ldots, \hat{p}_{i-1}, \hat{p}_i, h_i, \hat{p}_{i+1}, h_{i+1}, \ldots, h_{n-1}, \hat{p}_n, h_n$

because subgoals with predicate symbols $\hat{p}_i$ succeed without modifying the stack added by the transformation.

Example 5.2. An example illustrating how the search space is reduced by the elimination of leftmost h-subgoals denoting a subset of the identity relation is (in DCG notation):

$$s(A) \rightarrow a(A)$$
$$s(f(A)) \rightarrow b,\ s(A),\ c$$
$$s(g(A)) \rightarrow b,\ s(A)$$
$$a(a', [a'|X], X) \leftarrow$$
$$b([b'|X], X) \leftarrow$$
$$c([c'|X], X) \leftarrow$$

The associated grammar rules are:

$$s \rightarrow h_0\ a\ h_1$$
$$s \rightarrow h_2\ b\ h_3\ s\ h_4\ c\ h_5$$
$$s \rightarrow h_6\ b\ h_7\ s\ h_8$$
$$a \rightarrow h_9$$
$$b \rightarrow h_{10}$$
$$c \rightarrow h_{11}$$

In this case, the subgoals with predicate symbols h_0, h_1, h_2, h_3, h_6, h_7, h_9, h_{10}, and h_{11} can be eliminated (the first six h's by Theorem 5.3, and the other three h's by resolution). With a clausal configuration of the form:

$$(I_0,\ \langle [\,], [\underbrace{b', b', \ldots, b'}_{n}, a']\rangle)$$

the resulting search space has, apart from a successful branch, failed branches of length 1. As a consequence, when adding a search strategy to SLR inference (Section 6), the proof procedure obtained takes a linear time in n to succeed. By contrast, Prolog requires an exponential time in n.

We now explain this example. First note that the grammar with productions:

$$S \rightarrow a \tag{5.2}$$
$$S \rightarrow bSc \tag{5.3}$$
$$S \rightarrow bS \tag{5.4}$$

represented with difference lists, causes Prolog to take an exponential time in n to parse the string $b^n a$. A (correct) top-down parsing of the string uses produc-

$$
\begin{tikzcd}
\textit{hSLR inference} \arrow[r, dashed, no head, "\textit{Thms. 5.1, 5.2}"] \arrow[d, no head, "\textit{Thm. 5.3}"] & \textit{SLD resolution} \\
\textit{SLR inference} \arrow[d, no head, "\textit{Thm. 5.3}"] & \\
\textit{CSLR inference} \arrow[r, dashed, no head] & \textit{nondet. SLR parsing}
\end{tikzcd}
$$

FIGURE 5.3. *Containments between search spaces.*

tion (5.4) n times before using (5.2). Prolog, however, first selects the wrong production (5.3) n times before backtracking and selecting (5.4) only as the final step. Backtracking then removes the last two productions, making now a correct choice for the second last step, but making again a wrong choice for the last one. Thus, Prolog's behavior parallels that of a binary counter as its value goes from 0 to 2^n.

We then added some context-sensitive information, recording which production was used to parse the input string.

Figure 5.3 summarizes the comparison of the search spaces determined by the inference systems described in this paper and that of SLD resolution. Containments depicted with double, dotted lines are meant to hold between inference systems with search spaces which are (essentially) isomorphic (i.e. isomorphic except possibly for R-components). Containments depicted with one line are meant to hold between inference systems with search spaces whose size presumably decreases as we go from top down the figure.

First, if we use CSLR inference on the chain programs produced by our transformation (i.e. hSLR inference), then (1) the size of the refutations is $O(n)$, where n is the length of SLD refutations for the original program, and (2) apart from a possible increase in the length of refutations (R-components), both search spaces are isomorphic. Second, if it is possible to remove some h-subgoals (by Theorem 5.3) then the size of the search space may be reduced. Finally, if the original logic program has chain form, then all h-subgoals added by the transformation can be removed. In this case, CSLR inference determines a search space which is essentially the same as that of SLR parsing (using backtracking in case of conflicts) for the chain program and its associated context-free grammar.

6. A PROOF PROCEDURE

In this section, we will add a search strategy to SLR inference, obtaining a proof procedure.

Once a parsing table is constructed, an ordinary SLR parser uses a program sometimes called *driver* to determine the action to be performed (shift, reduce, accept, or error) for the current configuration. If the table has no conflicts, the search space is linear. Because we regard conflicts as nondeterministic choice points [16], a parsing table with conflicts determines a branching search space. We use Prolog's search strategy to traverse such a space.

Our inference system has yet another source of nondeterminism. A logic program representing a context-free grammar with difference lists contains unit clauses of the form $c'([c|X], X) \leftarrow$. Hence, a list $[c, \ldots]$ representing a string is "transformed" into at most one other list when a unit clause is used. By contrast, a chain program contains unit clauses of the form $c'(t, t') \leftarrow$, and arbitrary ground terms play the role of lists. As a result, there may be more than one way to transform a given ground term using different unit clauses. This is a second source of nondeterminism, which we also handle with Prolog's search strategy.

In the appendix, we give the Prolog code for our driver, which behaves as follows. Given a clausal configuration $(\ldots I_k, \mathbf{x})$ the driver first finds a unit clause of the object program whose input unifies with $\mathbf{x}$. Then the driver uses the predicate symbol of such a clause to determine an action from the parsing table. Unlike the original SLR parser, our proof procedure may find more than one grammar/predicate symbol associated with the same ground term of a (clausal) configuration. This may happen if $\mathbf{x}$ unifies with the input of more than one unit clause. We perform one action and then the other one, through Prolog's backtracking.

In the next examples, our proof procedure may be preferable to DCGs under Prolog. First we give a left-recursive DCG for which Prolog enters into a nonterminating loop, whereas our method does not:

$$expr(A) \rightarrow a(A)$$
$$expr(add(E,T)) \rightarrow expr(E),\ plus,\ expr(T)$$
$$a(int, [int|X], X) \leftarrow$$
$$plus([+|X], X) \leftarrow$$

A goal could be: $\leftarrow expr(E,\ [int, +, int],\ R)$.

Another program is Example 5.2. Using goals of the form:

$$\leftarrow s(A, [\underbrace{b', b', \ldots, b'}_{n}, a'], Z)$$

The original DCG under Prolog takes an exponential time in n to succeed, whereas our method takes a linear time.

7. COMPARISONS WITH OTHER METHODS

There are several inference methods for logic programs, that use variants of LR parsing [13, 21, 28, 30, 34, 35]. In this section, we review some of these methods and compare them with SLR inference.

7.1. A comparison with Nilsson's AID

Nilsson's "alternative implementation of DCGs" (AID) [21] is perhaps closest to the present work. Essentially, Nilsson first generates LR parsing tables ignoring the contextual information in the DCG clauses. He then uses a modified LR parser which considers the contextual information for each clause when the production corresponding to such a clause is reduced. As a consequence, the selection of subgoals having contextual information is delayed until reduction occurs.

Consider for example the following fragment of a DCG:

$$expr(X) \rightarrow expr(Y),\ [+],\ expr(Z),\ \{X\ is\ Y + Z\}$$

The AID method first generates the parsing table for the production:

$$expr \rightarrow expr\ \ +\ \ expr$$

together with the productions corresponding to the rest of the clauses. Then the reduce operation of the parser considers the components of the original clause that were ignored:

$$r(1,\ expr,\ [\_,\_,\_|S],\ S,\ [expr(Y), +, expr(Z)|A],\ [expr(X)|A]) \leftarrow X\ is\ Y + Z$$

The first argument is an index to the original clause. The second argument represents the left-hand side of the production. The third and fourth arguments are the stack of the parser [2], before and after the reduce operation. This stack stores nodes of a finite-state automaton. The last two arguments behave like a stack also, recording the output of the original clause (i.e. the value of the expression).

As observed by Nilsson, because the selection of subgoals is delayed until a reduction occurs, subgoals are not evaluated in the usual order. Consider the following variant of the above example, in which we wish to bound the depth of expressions to 10, for instance:

$$\begin{aligned} expr(N, X) \rightarrow\ & \{N \leq 10,\ N'\ is\ N + 1\}, \\ & expr(N', Y),\ [+],\ expr(N', Z), \\ & \{X\ is\ Y + Z\} \end{aligned}$$

Here, the first argument acts as input. If implemented as inferred from [21], AID would delay the evaluation of $\{N \leq 10,\ N'\ is\ N + 1\}$. As a result, not only the depth of the expression is ignored, but also Prolog selects a subgoal of the form $M \leq 10$, where M is a variable, so that an "instantiation error" occurs. Our proof procedure, by contrast, can handle this clause and in general logic programs in which there are both input and output arguments.

7.2. Parsing for attribute grammars

Abramson [1], as well as Deransart and Małuszyński [10], have investigated connections between attribute grammars [15] and logic programs. These connections result in similarities between parsing methods for attribute grammars, on the one hand, and inference systems for logic programs, on the other hand. We will concentrate on Jones and Madsen's method [13], which is based on LR parsing and generalizes many existing parsers for attribute grammars.

Example 7.1. Consider for example the following attribute grammar for characterizing binary numerals with radix point [10, 15]:

$r_1 : Z \rightarrow N \;.\; N$	$v(Z) = v(N_1) + v(N_2)$	
	$r(N_1) = 0$	$r(N_2) = -l(N_2)$
$r_2 : N \rightarrow N \; B$	$v(N_0) = v(N_1) + v(B)$	$l(N_0) = l(N_1) + 1$
	$r(B) = r(N_0)$	$r(N_1) = r(N_0) + 1$
$r_3 : N \rightarrow \epsilon$	$v(N) = 0$	$l(N) = 0$
$r_4 : B \rightarrow 1$	$v(B) = 2^{r(B)}$	
$r_5 : B \rightarrow 0$	$v(B) = 0$	

"Synthesized" attributes, such as $v(B)$, $v(N)$, $l(N)$, and $v(Z)$ are evaluated from the bottom up in the parse tree. "Inherited" attributes, such as $r(B)$ and $r(N)$, by contrast, are evaluated from the top down.

Jones and Madsen observe [13] that the value of some attributes cannot be computed during left-to-right and bottom-up parsing, but can be computed once the parsing has been performed. An example of such an attribute is $r(N)$, which is inherited and depends on a synthesized attribute $l(N)$ in the second occurrence of N in the production r_1 (equation $r(N_2) = -l(N_2)$).

Following Deransart and Małuszyński [9, page 53], this attribute grammar can be described with the following DCG:

$$z(V_1 + V_2) \rightarrow n(0, L_1, V_1),\ [.],\ n(-L_2, L_2, V_2) \tag{7.1}$$

$$n(R, 0, 0) \rightarrow [\,] \tag{7.2}$$

$$n(R, L_1 + 1, V_1 + V_2) \rightarrow n(R + 1, L_1, V_1),\ b(R, V_2) \tag{7.3}$$

$$b(R, 2^R) \rightarrow [1] \tag{7.4}$$

$$b(R, 0) \rightarrow [0] \tag{7.5}$$

A goal would be: $\leftarrow z(V, [1, 0, ., 0, 1], Rest)$.

An attribute grammar having attributes which cannot be evaluated during left-to-right parsing is translated [10] into a DCG which may not be a fixed-mode program. In this example, for instance, the first argument of the last subgoal $n(-L_2, L_2, V_2)$ in clause (7.1) violates condition 1, and hence this program is not fixed-mode.

It is worth observing that in spite of this violation, this particular example can be transformed into chain form by pushing L_2 onto the stack. First, we group argument places into inputs and outputs. We classify the first argument place of n as input, so that clause (7.1) becomes:

$$\begin{aligned} z'(\langle X_0\rangle, \langle X_3, V_1 + V_2\rangle) \leftarrow\ & n'(\langle X_0, 0\rangle, \langle X_1, L_1, V_1\rangle), \\ & p'(X_1, X_2), \\ & n'(\langle X_2, -L_2\rangle, \langle X_3, L_2, V_2\rangle) \end{aligned}$$

Next, we add the stack, where we push L_2 in addition to the variables indicated by the transformation:

$$\begin{aligned} & \hat{z}(\langle St_0, X_0\rangle, \langle St_3, X_3, V_{1,3} + V_{2,3}\rangle) \leftarrow \\ & \qquad \hat{n}(\langle St_0, X_0, 0\rangle, \langle St_1, X_1, L_1, V_{1,1}\rangle), \\ & \qquad \hat{p}(\langle [V_{1,1}|St_1], X_1\rangle, \langle [V_{1,2}|St_2], X_2\rangle), \\ & \qquad \hat{n}(\langle [\boxed{L_{2,2}}, V_{1,2}|St_2], X_2, -L_{2,2}\rangle, \langle [\boxed{L_{2,3}}, V_{1,3}|St_3], X_3, L_{2,3}, V_{2,3}\rangle) \end{aligned}$$

Finally, we obtain chain form by folding the h predicates:

$$\hat{z} \rightarrow h_0,\ \hat{n},\ h_1,\ \hat{p},\ h_2,\ \hat{n},\ h_3$$

The predicate with symbol h_2 will then produce configurations with a non-ground term, because $L_{2,2}$ is not instantiated when h_2 is shifted. However, if we use full unification instead of matching in CSLR inference (Section 3) —which is what we have done in our driver, since it is written in Prolog—, this modified method will compute the values of binary numbers with the above program.

This example, however, presents another difficulty, which is not overcome by SLR inference, resulting from clause (7.3) being left-recursive. The leftmost h-subgoal of the chain form of clause (7.3) is defined by a clause of the form:

$$h_4(\langle X_0, R\rangle, \langle X_0, R+1\rangle) \leftarrow$$

and does not denote a subset of the identity relation. Thus, such a subgoal cannot be eliminated by Theorem 5.3. By Theorem 5.1, no choice points of SLD resolution (for the original program and goal) are eliminated by SLR inference either. Therefore, SLR inference falls into an infinite loop (after producing the value of the binary numeral) just as Prolog does.

We can now continue with our comparison. First, we will consider the parsing method of Jones and Madsen [13], which we will call AILR parsing (for attribute-influenced LR parsing). We will describe the essence of AILR parsing, and refer the reader to [13] for a more detailed account of this method. AILR parsing creates a data structure called the *expression dag* as the parsing proceeds. The purpose of this structure is to record the necessary information for computing the values of the attributes which could not be computed during parsing.

In addition, instead of the configurations used by ordinary LR parsing, which are of the form:

$$(I_0 B_1 I_1 \ldots B_m I_m,\ c_i c_{i+1} \ldots c_n \$)$$

AILR parsing uses configurations of the form:

$$(I_0 \overline{I}_0 B_1 \overline{B}_1 I_1 \ldots \overline{I}_{m-1} B_m \overline{B}_m I_m,\ c_i c_{i+1} \ldots c_n \$)$$

where:

1. $\overline{I}_j$ is a record containing the values of the inherited attributes of all nonterminals B such that $(A \rightarrow \alpha \bullet B\beta) \in I_j$, and
2. $\overline{B}_j$ is a record containing the values of the synthesized attributes of B_j.

AILR parsing parallels LR parsing, producing a sequence of configurations. The application of the shift and reduce rules, however, is interleaved with the computation of the value of some attributes and the incremental construction of the expression dag.

Let us now consider hSLR inference. For simplicity, assume a DCG which contains, among others, a clause of the form:

$$a(\ldots) \rightarrow x(\ldots)$$

This clause is transformed into the following clause in chain form:

$$\hat{a} \rightarrow h_0,\ \hat{x},\ h_1$$

The command h_0 pushes onto the stack added by the transformation some values needed later in the computation and establishes the input arguments for x. The command h_1, in turn, pops off the stack of the transformation some values and recovers the values of the output arguments for x.

hSLR inference produces then the configuration:

$$(I_0\, h_0\, I_1\, \hat{x}\, I_2\, h_1\, I_3, \quad \mathbf{y})$$

whereas AILR parsing produces the configuration:

$$(S_0\, \overline{S}_0\, X\, \overline{X}\, S_1, \quad y)$$

where:

1. S_0 contains, among other items, $A \rightarrow \bullet X$,
2. $\overline{S}_0$ records the values of the inherited (input) attributes of X,
3. X is the right-hand side of the production without attributes,
4. $\overline{X}$ records the values of the synthesized (output) attributes of X,
5. S_1 contains the item $A \rightarrow X \bullet$, and
6. y is a string.

In this example, the command h_0 corresponds to $\overline{S}_0$ (input arguments), and the command h_1 corresponds to $\overline{X}$ (output arguments).

As a conclusion, the records $\overline{S}_j$ and $\overline{B}_j$ of AILR parsing are reminiscent of our h-commands.

If we use Deransart and Małuszyński's translation [9] of attribute grammars into DCGs, we obtain a logic program which builds a term as its output. By contrast AILR parsing builds an expression dag specifically designed for computing the remaining attribute values. Jones and Madsen [13] have incorporated numerous optimizations in such a data structure. Thus, AILR is probably superior to our method in this respect.

Another conspicuous difference between AILR parsing and SLR inference appears in nondeterministic programs. AILR parsing is meant for obtaining the values of the attributes of the parse tree of a given string. SLR inference, by comparison, can be used not only for parsing an input string but also for computing possibly several answers.

7.3. A comparison with Sato and Tamaki's "success patterns"

Both AID and AILR can be viewed as adapting the complete LR parser. Sato and Tamaki present in [28] an inference method which adapts only the preprocessing stage of such a parser. We will illustrate Sato and Tamaki's "success patterns" [28] through an example, but refer the reader to [28] for a thorough discussion of this method.

Ordinary LR parsing can be regarded as having two stages: a top-down traversal of possible parse trees (encoded in a finite-state automaton), followed by a bottom-up construction of a parse tree (by means of shift and reduce operations). The success-patterns method performs a top-down traversal of proof trees for a logic program P and a goal G, reminiscent of the first stage of LR parsing. Given a goal $G' = G\theta$, the information obtained by such a traversal can then be used to obtain necessary conditions for the success of derivations for $P \cup \{G'\}$.

Consider the following program [28]:

$$P = \{a(X) \leftarrow b(X), d(X), \quad b(X) \leftarrow c(Y), \quad c(1) \leftarrow, \quad c(2) \leftarrow, \quad d(2) \leftarrow\}$$

and the goal $G = \leftarrow a(Z)$. First, the initial set of items:

$$I_0 = \{\leftarrow \bullet\, a(Z), \quad c(1) \leftarrow \bullet, \quad c(2) \leftarrow \bullet, \quad d(2) \leftarrow \bullet\}$$

is computed. Next, this method computes the "downward closure" of I_0 (which resembles the closure of a set of items in ordinary LR parsing):

$$I_0' = I_0 \cup \{a(X) \leftarrow \bullet\, b(X), d(X), \quad b(Y) \leftarrow \bullet\, c(Y)\}$$

and then the "upward closure" of I_0' (which in turn resembles computing the goto function of ordinary LR parsing):

$$\begin{aligned} I_1 = I_0' \cup \{&b(1) \leftarrow c(1)\bullet, \quad b(2) \leftarrow c(2)\bullet, \\ &a(1) \leftarrow b(1), \bullet\, d(1), \quad a(2) \leftarrow b(2), \bullet\, d(2)\} \end{aligned}$$

For example, the item $b(2) \leftarrow c(2)\bullet$ was obtained by operating $c(2) \leftarrow \bullet$ (in I_0) with $b(Y) \leftarrow \bullet\, c(Y)$ (in I_0'). The downward and upward closures are repetitively computed until no more new items can be generated. The resulting set of items is:

$$I = I_1 \cup \{a(2) \leftarrow b(2), d(2)\bullet, \quad \leftarrow a(2)\bullet\}$$

Finally, the authors collect, from I, the items with the dot at the rightmost position which are relevant for the goal.

$$I_{succ} = \{\leftarrow a(2)\bullet, \quad a(2) \leftarrow b(2), d(2)\bullet, \quad c(2) \leftarrow \bullet, \quad d(2) \leftarrow \bullet\}$$

The I_{succ} set has information about successful derivations, and can thus be used to prune the SLD tree. By examining I_{succ} for this example, we know that the goal $\leftarrow a(2)$ succeeds, whereas the goal $\leftarrow a(1)$ fails. In general, the I_{succ} has the following property. Let C be the input clause of a derivation step of a refutation, and θ the corresponding most general unifier. Then there is an item $(C\bullet)\sigma$ in I_{succ} such that $C\theta \equiv C\sigma\gamma$. The items in I_{succ} are more general than we would like, so that this property of I_{succ} is only a *necessary* condition for success.

To ensure termination in the computation of the item set, Sato and Tamaki truncate the term depth to a predefined value k. The value of k is crucial, since too small a k may limit the applicability of the necessary condition, and too large a k may produce an impractically big I_{succ}. By contrast, our method does not need to perform such a truncation.

7.4. A comparison with Yamashita and Nakata's ccfg's

Finally, we consider a formalism devised by Yamashita and Nakata [35], which is amenable to be executed by variants of parsing methods.

A *coupled context-free grammar* (ccfg) is a four-tuple (V, T, P, S), where:

1. V is a finite set of *nonterminals*;
2. T is a finite set of *terminals*, such that $V \cap T = \emptyset$;
3. P is a collection of sets of rules, in which each rule is of the form:

   $$A \rightarrow \alpha$$

   where A is an unsubscripted nonterminal, and α is a string of (i) terminals, (ii) subscripted nonterminals, and (iii) the meta-symbol $\equiv$; and

4. S is a tuple of nonterminals called the *start tuple*.

A tuple of strings:

$(\alpha A_i^j \alpha'', \ \ldots, \ \beta B_i^j \beta'')$

together with an integer k and a set $\{A \to \alpha', \ldots, B \to \beta'\}$ of productions *directly derives* the tuple:

$(\alpha(\alpha')^k \alpha'', \ \ldots, \ \beta(\beta')^k \beta'')$

together with the integer $k+1$. Here $(\ )^k$ is defined as:

$$\begin{aligned}(\epsilon)^k &= \epsilon \\ (A_i \alpha)^k &= A_i^k (\alpha)^k \\ (a\alpha)^k &= a(\alpha)^k\end{aligned}$$

A *derivation* is a finite sequence of derivation steps which starts with the tuple $(S_0^0, \ \ldots, \ T_0^0)$ and integer 1, where $(S, \ \ldots, \ T)$ is the start tuple, and ends with a tuple of strings of terminals possibly containing also $\equiv$. If a derivation ends in a tuple having a string α containing $\equiv$, then α is of the form $\beta \equiv \beta \equiv \cdots \equiv \beta$.

Concatenation of strings is distributive over $\equiv$. For instance, a concatenated with $aN_1^2 \equiv aaN_2^2$ is: $aaN_1^2 \equiv aaaN_2^2$.

Finally, note that a ccfg nondeterministically generates a set of tuples of strings.

Example 7.2. An example of a ccfg for computing Fibonacci numbers is:

$$\{N \to \epsilon, \ F \to a\} \tag{7.6}$$

$$\{N \to a, \ F \to a\} \tag{7.7}$$

$$\{N \to aN_1 \equiv aaN_2, \ F \to F_1 F_2\} \tag{7.8}$$

A derivation which computes the third Fibonacci number is:

$$\begin{aligned} \langle(\underline{N_0^0}, \ \underline{F_0^0}), \ 1\rangle &\underset{(7.8)}{\Rightarrow} \\ \langle(a\underline{N_1^1} \equiv aaN_2^1, \ \underline{F_1^1}F_2^1), \ 2\rangle &\underset{(7.8)}{\Rightarrow} \\ \langle(aa\underline{N_1^2} \equiv aaaN_2^2 \equiv aaN_2^1, \ \underline{F_1^2}F_2^2F_2^1), \ 3\rangle &\underset{(7.7)}{\Rightarrow} \\ \langle(aaa \equiv aaa\underline{N_2^2} \equiv aaN_2^1, \ a\underline{F_2^2}F_2^1), \ 4\rangle &\underset{(7.6)}{\Rightarrow} \\ \langle(aaa \equiv aaa \equiv aa\underline{N_2^1}, \ aa\underline{F_2^1}), \ 5\rangle &\underset{(7.7)}{\Rightarrow} \\ \langle(aaa \equiv aaa \equiv aaa, \ aaa), \ 6\rangle & \end{aligned}$$

Yamashita and Nakata suggest how to use a parser to execute a ccfg. First, each coordinate in the tuples generated by the grammar is identified as being either input or output. Next, a parser is used to parse the input components, while a generator is used to produce the output components. The parser and the generator are coupled in the sense that for each production of the grammar applied by the parser, a corresponding production in the same set of productions is applied by the generator.

These authors give an example of a ccfg that could be executed using an LR parser. In such an example, there is no occurrence of $\equiv$ and it is plausible that such a parser could be adapted as claimed. In a ccfg in which $\equiv$ occurs, however, it is not clear how to modify a parser for execution, since not only string concatenation is distributive over $\equiv$, but the same string must appear in between different occurrences of $\equiv$ for a ccfg derivation to succeed.

8. CONCLUDING REMARKS

This work was motivated by similarities between logic programs and context-free grammars. These similarities suggest the possibility of developing inference systems based on parsers. Because it is not obvious how to obtain such inference systems, we considered chain logic programs, which have clauses of the form:

$$a_0(X_0, X_n) \leftarrow b_1(X_0, X_1),\ b_2(X_1, X_2),\ \ldots,\ b_n(X_{n-1}, X_n)$$
$$\text{or}\ \ a_1(t, t') \leftarrow$$

where $\mathrm{var}(t') \subseteq \mathrm{var}(t)$, and goals of the form $\leftarrow s(\mathbf{x}, Z)$, where $\mathbf{x}$ is a ground term and Z a variable. We saw that by incorporating a match operation (one way unification) to the parts of an SLR parser dealing with terminals, we get an inference system for chain programs. Conflicts in the parsing table can be treated as nondeterministic choice points [16], which we handle with a driver written in Prolog.

Normally, logic programs do not have chain form. Hence, we used a transformation taking a fixed-mode logic program and producing a chain program, which we had previously developed [25] for inference systems based on chart parsers. We convert each clause:

$$p_0(t_0, t'_n) \leftarrow p_1(t'_0, t_1),\ p_2(t'_1, t_2),\ \ldots,\ p_n(t'_{n-1}, t_n)$$

into:

$$\hat{p}_0(U_0, U'_n) \leftarrow h_0(U_0, U'_0),\ \hat{p}_1(U'_0, U_1),\ h_1(U_1, U'_1),\ \hat{p}_2(U'_1, U_2),\ \ldots,$$
$$h_{n-1}(U_{n-1}, U'_{n-1}),\ \hat{p}_n(U'_{n-1}, U_n),\ h_n(U_n, U'_n)$$

(together with unit clauses defining the h_i's). As a result, our modified SLR parser becomes an inference system for fixed-mode logic programs.

The addition of the h_i predicates, however, creates new nondeterministic choice points, and eliminates potential advantages that could have resulted with the obtained inference system. The reason is that the resulting search space is essentially isomorphic to that of SLD resolution for the original program.

Such advantages may reappear once we observe that some of the introduced subgoals of predicates h_i denoting subsets of the identity relation can be eliminated.

Although we can handle arbitrary fixed-mode logic programs, we have mainly seen advantages of our technique over SLD resolution in grammatical examples. A reason could be perhaps that the context-sensitive information of such examples is not "so much" as to prevent the elimination of some of the introduced h-subgoals. Thus, we have illustrated SLR inference with grammatical applications. We gave an example making DCGs under Prolog enter into a nonterminating loop, for which our proof procedure halts. We also gave a program causing DCGs under Prolog to take an exponential time in the length of the input to succeed, for which our proof procedure requires a linear time.

We based our exposition on an SLR parser [2]. We believe, however, that variations around LR parsing, such as LALR, LR(k), and even Tomita's parser [33], could be used alternatively. Depending on the parser on which an inference system is based, different search spaces will be obtained.

Here and in [25], we modified parsers by (1) adding unification to the treatment of the leaves of the proof/parse trees and (2) applying the resulting inference system to programs transformed into chain form. The idea of adapting other parsers in a similar way remains to be explored.

Acknowledgments

Our work benefited from motivating discussions with Carlos Velarde. Warren Greiff's careful reading of an earlier version of this paper was most helpful. Fernando Magariños helped us with LaTeX. We are also grateful to the referees of both ILPS '94 and The Journal of Logic Programming, whose comments helped improve the presentation of these results. We acknowledge the facilities provided by IIMAS, UNAM. This work was supported by grant IN301192 of DGAPA, UNAM.

APPENDIX: DRIVER FOR CHAIN PROGRAMS

The appendix shows the Prolog code of a driver for SLR inference. We handle nondeterminism with Prolog's backtracking, but for simplicity, we have assumed that the parsing table has no reduce/reduce conflicts. (A driver handling such conflicts would be considerably longer.)

```
% example DCG
% expr(A) --> a(A).
% expr(add(E,T)) --> expr(E), plus, expr(T).
% a(int,[int|X],X).
% plus([+|X],X).

% rule(0,expr1,[expr]).
% rule(1,expr,[expr,h1,plus,h2,expr,h3]).
% rule(2,expr,[a]).

% next_state(plus,[St,[+|X]],[St,X]).
% next_state(a,[St,[int|X]],[St,X,int]).
% next_state(h1,[St,X,E],[[E|St],X]).
% next_state(h2,[[E|St],X],[[E|St],X]).
% next_state(h3,[[E|St],X,F],[St,X,p(E,F)]).

% action(7,fin,r(1)). action(7,h1,r(1)).   action(7,h3,r(1)).
% action(6,h1,s(3)).  action(6,h3,s(7)).   action(5,a,s(1)).
% action(4,h2,s(5)).  action(3,plus,s(4)). action(2,fin,acc).
% action(2,h1,s(3)).  action(1,fin,r(2)).  action(1,h1,r(2)).
% action(1,h3,r(2)).  action(0,a,s(1)).
% goto(0,expr,2).     goto(5,expr,6).
```

```
% ?- move( [0], [[],[int,+,int,+,int]], S ). example goal

move( [Node|Stack], State, State2 ) :-
     next_state( Command, State, State1 ),
     action( Node, Command, s(Node1) ),
     move( [Node1,Node|Stack], State1, State2 ).
move( [Node|Stack], State, State1 ) :-
     action1( Node, State, Rule ),
     rule( Rule, Head, Body ),
     reduce( [Node|Stack], Head, Body, Stack1 ),
     move( Stack1, State, State1 ).
move( [Node|Stack], State, State ) :-
     action( Node, fin, acc ).

reduce( [Node|Stack], Head, [], [Goto,Node|Stack] ) :-
     goto( Node, Head, Goto ).
reduce( [Node|Stack], Head, [NonTerm|NonTerms], Stack1 ) :-
     reduce( Stack, Head, NonTerms, Stack1 ).

action1( Node, State, Rule ) :- next_state( Command, State, State1 ),
                                action( Node, Command, r(Rule) ), !.
action1( Node, State, Rule ) :- action( Node, fin, r(Rule) ).
```

REFERENCES

1. Harvey Abramson. Definite clause translation grammars. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 233–240, Atlantic City, U.S.A., 1984.
2. Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers —Principles, Techniques, and Tools*. Addison-Wesley, 1986.
3. Krzysztof R. Apt. Declarative programming in Prolog. In *Proceedings of the 1993 International Symposium on Logic Programming*, pages 12–35, Vancouver, Canada, 1993. MIT Press.
4. Krzysztof R. Apt and Elena Marchiori. Reasoning about Prolog programs: from modes through types to assertions. *Formal Aspects of Computing*, 6A:743–764, 1994.
5. Edward Ashcroft, Zohar Manna, and Amir Pnueli. Decidable properties of monadic functional schemas. *Journal of the ACM*, 20(3):489–499, 1973.
6. Keith L. Clark and M.H. van Emden. Consequence verification of flowcharts. *IEEE Transactions on Software Engineering*, SE-7(1):52–60, January 1981.
7. Keith Leonard Clark. Predicate logic as a computational formalism. Technical Report 79/59, Dept. of Computing and Control, Imperial College, 1979.
8. Eric Villemonte de la Clergerie and Bernard Lang. LPDA: Another look at tabulation in logic programming. In Pascal Van Hentenryck, editor, *Proceedings of the 11th International Logic Programming Symposium*, pages 471–486. MIT Press, 1994.
9. Pierre Deransart and Jan Małuszyński. What kind of grammars are logic programs? In Patrick Saint-Dizier and Stan Szpakowicz, editors, *Logic and Logic Grammars for Language Processing*, pages 28–55. Ellis Horwood, 1992.

10. Pierre Deransart and Jan Małuszyński. *A Grammatical View of Logic Programming.* MIT Press, 1993.
11. Włodzimierz Drabent. Do logic programs resemble programs in conventional languages? In *Proceedings of the 1987 IEEE Symposium on Logic Programming*, pages 389–396, San Francisco, USA, 1987. IEEE Press.
12. Joost Engelfriet. *Simple Program Schemes and Formal Languages.* Springer-Verlag, 1974. Lecture Notes in Computer Science No. 20.
13. Neil D. Jones and Michael Madsen. Attibute-influenced LR parsing. In *Semantics Directed Compiler Generation*, pages 393–407, 1980. Lecture Notes in Computer Science 94.
14. Donald E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.
15. Donald E. Knuth. Semantics of context-free languages. *Mathematical Systems Theory*, 2(2):127–145, 1968.
16. Bernard Lang. Deterministic techniques for efficient non-deterministic parsers. In J. Loeckx, editor, *Proceedings of the 2nd Colloquium on Automata*, pages 255–269. Springer-Verlag, 1974.
17. Bernard Lang. Complete evaluation of Horn clauses: An automata theoretic approach. Technical Report 913, INRIA, November 1988.
18. Bernard Lang. Datalog automata. In *Proceedings of the 3rd. International Conference on Data and Knowledge Bases*, pages 389–404, 1988.
19. J.W. Lloyd. *Foundations of Logic Programming.* Springer-Verlag, 2nd edition, 1987.
20. A. Mazurkiewicz. Recursive algorithms and formal languages. *Bull. Acad. Polon. Sci., Sér. Sci., Math., Astr. Phys.*, 20:799–803, 1972.
21. Ulf Nilsson. AID: an alternative implementation of DCGs. *New Generation Computing*, 4:383–399, 1986.
22. Fernando C.N. Pereira and David H.D. Warren. Definite clause grammars for language analysis —A survey of the formalism and a comparison with augmented transition networks. *Artificial Intelligence*, 13:231–278, 1980.
23. Fernando C.N. Pereira and David H.D. Warren. Parsing as deduction. In *21st Annual Meeting of the Association for Computational Linguistics*, pages 137–144, Cambridge, Massachusetts, 1983. Association for Computational Linguistics.
24. David A. Rosenblueth. Fixed-mode logic programs as state-oriented programs. Technical Report Preimpreso No. 2, IIMAS, UNAM, 1991.
25. David A. Rosenblueth. Chart parsers as inference systems for fixed-mode logic programs. *New Generation Computing*, 14(4):429–458, 1996.
26. David A. Rosenblueth and Julio C. Peralta. LR inference: Inference systems for fixed-mode logic programs, based on LR parsing. In Maurice Bruynooghe, editor, *Proceedings of the International Logic Programming Symposium*, pages 439–453, Ithaca, N.Y., U.S.A., 1994.
27. J.D. Rutledge. On Ianov's program schemata. *Journal of the ACM*, 11(1):1–9, 1964.
28. Taisuke Sato and Hisao Tamaki. Enumeration of success patterns in logic programs. *Theoretical Computer Science*, 34:227–240, 1984.
29. Joseph R. Shoenfield. *Mathematical Logic.* Addison-Wesley, 1967.
30. Edward P. Stabler. Deterministic and bottom-up parsing in Prolog. In *Proceedings of the 1st National Conference on Artificial Intelligence*, pages 383–396, 1983.
31. Robert F. Stärk. A direct proof for the completeness of SLD-resolution. In E. Börger, H. Kleine Büning, and M.M. Richter, editors, *Computation Theory and Logic 89*, pages 382–383. Springer-Verlag, 1990.
32. Hisao Tamaki and Taisuke Sato. Unfold/fold transformation of logic programs. In *Proceedings of the Second International Logic Programming Conference*, pages

127–138, 1984.

33. Masaru Tomita. An efficient augmented-context-free parsing algorithm. *Computational Linguistics*, 13(1–2):31–46, 1987.
34. Kuniaki Uehara, Ryo Ochitani, Osamu Kakusho, and Junichi Toyoda. A bottom-up parser based on predicate logic: A survey of the formalism and its implementation techique. In *Proceedings of the 1984 International Symposium on Logic Programming*, pages 220–227, Atlantic City, U.S.A., 1984.
35. Yoshiyuki Yamashita and Ikuo Nakata. Coupled context-free grammar as a programming paradigm. In *Proceedings of the International Workshop on Programming Languages Implementation and Logic Programming PLILP '88*, pages 132–145, Orléans, France, 1988.