Parameterized LR Parsing

Peter Thiemann and Matthias Neubauer [1,2]

Institut für Informatik
Universität Freiburg
Georges-Köhler-Allee 079
D-79110 Freiburg, Germany

Abstract

Common LR parser generators lack abstraction facilities for defining recurring patterns of productions. Although there are generators capable of supporting regular expressions on the right hand side of productions, no generator supports user defined patterns in grammars.

Parameterized LR parsing extends standard LR parsing technology by admitting grammars with parameterized non-terminal symbols. A generator can implement such a grammar in two ways, either by expansion or directly. We develop the theory required for the direct implementation and show that it leads to significantly smaller parse tables and that it has fewer parsing conflicts than the expanded grammar. Attribute evaluation for a parameterized non-terminal is possible in the same way as before, if the semantic functions related to the non-terminal are polymorphic with respect to the parameter.

We have implemented parameterized LR parsing in the context of Essence, a partial-evaluation based LR parser generator for Scheme.

1 Introduction

LR parsing [11] is a powerful tool in the toolbox of the language designer. It provides a parsing algorithm that works in linear time for a wide range of context-free grammars. The theory of LR parsing has been explored in numerous works and it has become a standard part of lectures and textbooks on compiler construction [1]. It also forms the foundation of a whole range of parser generation tools starting with yacc/bison [10,5], the ideas of which have been adapted to virtually any programming language around.

While there has been some evolution of the tools with respect to modularization of parsing actions and integration of the specifications of parsing and

[1] Email: thiemann@informatik.uni-freiburg.de
[2] Email: neubauer@informatik.uni-freiburg.de

This is a preliminary version. The final version will be published in Electronic Notes in Theoretical Computer Science
URL: www.elsevier.nl/locate/entcs

scanning, the actual raw matter, the grammar, remains in its original form in the parser specification. Since large grammars may well run into several hundred productions, grammar maintenance can become a tedious task. Hence, it is surprising that none of the parser generators has a facility for introducing abstractions over grammar rules.

The main attempt to introduce some abbreviation mechanism into LR grammars is the consideration of regular right-hand sides for rules [3]. However, only a few LR parser generators (*e.g.*, [8]) support regular right hand sides or extended BNF directly. Consequently, typical grammars for LR parser generators are full of rule groups that implement common grammatical patterns. Here are some examples with the number of uses of that pattern from a randomly picked grammar.

- lists with separator (seven times)

```
DelimTypeSchemes        : /* empty */
                        | NEDelimTypeSchemes ;
NEDelimTypeSchemes      : TypeScheme
                        | NEDelimTypeSchemes ',' TypeScheme ;
```

- plain lists (five times plus two non-empty lists)

```
TypeVars                : /* empty */
                        | NETypeVars ;
NETypeVars              : TypeVar
                        | NETypeVars TypeVar ;
```

- optional items (two times)

```
Imports                 : /* empty */
                        | Imports import typename ;
```

Often, even the semantic actions coincide or can be made to coincide easily.

Our proposal derives almost directly from these observations. Instead of relying on a fixed set of (regular) operators for use in the right-hand side of grammar rules, we make available an arbitrary, user-definable set of operators in the form of *parameterizable nonterminal symbols*. These nonterminals are used like functions. They can be invoked with actual parameters in the right-hand side of a production. Some care must be taken to restrict the actual parameters suitably, for example, to only one nonterminal or one formal parameter. Without restriction to the actual parameters, we would obtain the notion of a macro grammar which is strictly more powerful than a context-free grammar [4].

With our proposed extension, a grammar writer can write parameterized productions corresponding to the patterns exhibited above once and for all. Sets of parameterized nonterminals might be collected in modules and reused between grammars.

The following example shows parameterized rules for the grammatical patterns identified above.

```
SepList (Sep, Item)     : /* empty */
                        | NESepList (Sep, Item) ;
NESepList (Sep, Item)   : Item
                        | NESepList (Sep, Item) Sep Item ;

List (Item)             : /* empty */
                        | NEList (Item) ;
NEList (Item)           : Item
                        | NEList (Item) Item ;

Option (Item)           : /* empty */
                        | Item ;
```

For the examples above it also makes sense to define generic semantic actions. The only requirement is that these actions are polymorphic with respect to the semantic values of the parameters. Hence, the `SepList (Sep, Item)` and the `List (Item)` might both return a value of type `List Item` and the `Option (Item)` might return a value of type `Maybe Item`. Type-specific semantic actions should take place when returning from parameterized nonterminal instantiated with actual nonterminals.

One way of implementing parameterized LR parsing is by separately specializing parameterized nonterminals with respect to all the nonterminals actually instantiated as parameters separately. This specialization terminates trivially because of our restriction on actual parameters. However, it turns out that developing the theory directly for parameterized grammars yields smaller parse tables (since the parameterized parts can be shared) and sometimes avoids parsing conflicts.

We have extended the LR(k) parser generator Essence [17] to generate parameterized LR(k) parsers. Initial experiments with the implementation are encouraging.

Overview

Before developing the theory of parameterized LR parsing, we make an excursion into formal language theory to introduce the reader to macro grammars and macro languages in Section 2. After defining a suitably restricted notion of macro grammar, Section 3 introduces the basic definitions for parameterized LR parsing. Section 4 defines the parsing algorithm, starting with a nondeterministic specification and then defining the notion of conflict to obtain deterministic parsers. Section 5 is devoted to attribute evaluation. It defines a type system that assigns polymorphic types to parameterized nonterminals. Section 6 describes our implementation of parameterized LR parsing in the context of Essence, a parser generator for Scheme. Finally, Section 7 considers related work and Section 8 concludes and sketches some future work.

2 Macro Grammars

To define macro grammars properly, we need some standard definitions from universal algebra. A *signature* is a pair $\Gamma = (N, a)$ of a finite set N and an arity function $a : N \to \mathbf{N}$, the set of non-negative integers. The set $T_\Gamma(X)$ of Γ-terms with variables X (a set disjoint from N) is defined inductively by

- $X \subseteq T_\Gamma(X)$
- $(\forall n \in \mathbf{N}, t_1 \dots t_n \in T_\Gamma(X), A \in N)\ a(A) = n \Rightarrow A(t_1, \dots, t_n) \in T_\Gamma(X)$.

Definition 2.1 A *macro grammar* is a quadruple (Γ, Σ, P, S) where $\Gamma = (N, a)$ is a signature with N the *nonterminal symbols*, Σ is a finite set of *terminal symbols*, $P \subseteq N \times T_{\Gamma'}(\Sigma \cup \mathbf{N})$ is a finite set of macro productions, and $S \in N$ with $a(S) = 0$ is the start symbol. The signature Γ' extends Γ by a binary operator $\cdot$ (concatenation) and a constant ε (empty string).

The productions are subject to the following restriction. If $A \to w \in P$ with $a(A) = n$, then $w \in T_{\Gamma'}(\Sigma \cup \{0, \dots, n-1\})$.

A macro grammar generates words over the set of terminal symbols using the following derivation relation $\Rightarrow$ on $T_{\Gamma'}(\Sigma)$.

- If $A \to w \in P$, $a(A) = n$, $t_1 \dots t_n \in T_{\Gamma'}(\Sigma)$,
  then $A(t_1, \dots, t_n) \Rightarrow w[0 \mapsto t_1, \dots, n-1 \mapsto t_n]$, and
- if $f \in \Gamma'$, $a(f) = n$, $t_1 \dots t_n \in T_{\Gamma'}(\Sigma)$, $t_i \Rightarrow t_i'$,
  then $f(t_1, \dots, t_i, \dots, t_n) \Rightarrow f(t_1, \dots, t_I', \dots, t_n)$.

That is, the relation comprises all pairs of terms and it is closed under compatibility. As usual, $\overset{*}{\Rightarrow}$ denotes the reflexive transitive closure of the derivation relation.

A term w is in the language generated by the grammar if $S \overset{*}{\Rightarrow} w$ and $w \in T_{\cdot,\varepsilon}(\Sigma)$, which can be considered as an element of Σ^* in the obvious way.

Usually, the derivation relation is restricted to either substitute nonterminals inside-out (IO) or outside-in (OI).

IO reduction $A(t_1, \dots, t_n) \Rightarrow_{IO} w[0 \mapsto t_1, \dots, n-1 \mapsto t_n]$ only if $t_1, \dots, t_n \in T_{\cdot,\varepsilon}(\Sigma)$ (they do not contain nonterminals) and the relation is closed under compatibility as before.

OI reduction the reduction rule for $\Rightarrow_{OI}$ is the same as for $\Rightarrow$, but compatibility is restricted to $f \in \{\cdot, \varepsilon\}$ (reduction does **not** proceed into argument positions).

The respective languages are called IO- and OI-macro languages. They have been investigated in detail [7] and we recall some of their properties below. [3]

[3] The definition we are giving above is not the one that has been used to obtain the cited results. The original definition considers strings as trees build from monadic operators (the characters) so that standard nonterminals in a context-free grammar are also monadic operators serving as placeholders for trees. In a macro grammar, nonterminals receive *additional* parameters that range over monadic operators. Adding further parameter sets

(i) In general, the language generated from a grammar under IO reduction is different from the language generated under OI reduction. (One corresponds to call-by-value, the other to call-by-name.)

(ii) The classes of IO- and OI-macro languages are incomparable.

(iii) The IO- and OI-macro languages are a strict hierarchy of languages between context-free languages and context-sensitive languages [4].

The macro grammars that we have introduced here only correspond to the first level of the hierarchy mentioned above. However, taken in their full generality, they can describe languages that are not context-free. For example, the following macro grammar generates the set $\{a^n b^n c^n \mid n \in \mathbf{N}\}$:

$$\begin{aligned} S &\to F(\varepsilon, \varepsilon, \varepsilon) \\ F(A, B, C) &\to ABC \\ F(A, B, C) &\to F(aA, bB, cC) \end{aligned}$$

Here, we have taken the liberty of naming the parameters of F instead of using the numbering scheme from the definition.

Since we are interested in making abstractions to help define context-free languages only, we need to restrict the general definition.

Definition 2.2 A *restricted macro grammar* is a macro grammar (Γ, Σ, P, S) where each production has the form $A \to t_1 \cdots t_m$ (for $m \in \mathbf{N}$) such that, for each $1 \leq i \leq m$, either $t_i \in \Sigma$ or $t_i = B(s_0, \ldots, s_{l-1})$ where $l = a(B)$ and, for each $0 \leq j < l$, either $s_j \in N$ with $a(s_j) = 0$, a nullary nonterminal symbol, or $s_j \in \{0, \ldots, a(A) - 1\}$, a parameter symbol.

Each restricted macro grammar determines a context-free grammar by specialization of the nonterminals with respect to their parameters. The resulting grammar is determined by $(N', \Sigma, P', (S, \varepsilon))$ where

$$\begin{aligned} N' &= \{(A, A_0 \ldots A_{n-1}) \mid A, A_0, \ldots, A_{n-1} \in N, a(A) = n, a(A_i) = 0\} \\ &\subseteq N \times N^* \\ P' &= \{(A, A_0 \ldots A_{n-1}) \to |t_1|_{A_0 \ldots A_{n-1}} \cdots |t_m|_{A_0 \ldots A_{n-1}} \mid A \to t_1 \cdots t_m \in P\} \end{aligned}$$

where (for $a \in \Sigma$, $j \in \mathbf{N}$)

$$\begin{aligned} |a|_{A_0 \ldots A_{n-1}} &= a \\ |B(s_0, \ldots, s_{l-1})|_{A_0 \ldots A_{n-1}} &= (B, ||s_0||_{A_0 \ldots A_{n-1}} \ldots ||s_{l-1}||_{A_0 \ldots A_{n-1}}) \\ |j|_{A_0 \ldots A_{n-1}} &= (A_j, \varepsilon) \\ ||C||_{A_0 \ldots A_{n-1}} &= C \\ ||j||_{A_0 \ldots A_{n-1}} &= A_j \end{aligned}$$

leads to higher levels in the mentioned hierarchy.

The following lemma makes the connection precise. The language of the specialized grammar corresponds to the language of the restricted macro grammar under OI-reduction.

Lemma 2.3 *Let $\mathcal{M} = (\Gamma, \Sigma, P, S)$ be a restricted macro grammar and $\mathcal{G} = (N', \Sigma, P', S)$ the corresponding context-free grammar as constructed above.*

$S \overset{*}{\Rightarrow}_{\mathcal{M},OI} w$ iff $(S, \varepsilon) \overset{*}{\Rightarrow}_{\mathcal{G}} w$.

Proof. We show that each derivation step in $\mathcal{M}$ corresponds to a derivation step in $\mathcal{G}$.

Since OI reduction restricts compatibility to the concatenation operator, all terms derivable from S in $\mathcal{M}$ can be written in the form $s_1 \cdots s_m$ where, for each i, either $s_i \in \Sigma$ or $s_i = A^i(A^i_0, \ldots, A^i_{a(A^i)-1})$, for some nullary nonterminals A^i_j. (This can be proven by induction on the number of derivation steps for $S \overset{*}{\Rightarrow}_{\mathcal{M},OI} s_1 \cdots s_m$.)

Hence, the translation function $|\cdots|_{B\ldots}$ from the construction above is applicable to all derivable terms where the subscript is empty since no parameters occur in derivable terms. It remains to see that if $S \overset{*}{\Rightarrow}_{\mathcal{M},OI} \alpha$ and $\alpha \Rightarrow_{\mathcal{M},OI} \alpha'$, then $|\alpha| \Rightarrow_{\mathcal{G}} |\alpha'|$. But this is immediate from the construction of the production set P'. □

Hence, restricted macro grammars with OI reduction define exactly context-free languages. The restriction that actual parameters are either single nonterminals or formal parameters is less severe than it may appear. Alternatively, the actual parameters may be restricted to either a single formal parameter or an arbitrary word over terminals and (nullary) nonterminals. Such a grammar can be transformed to restricted form as defined above by introducing new nullary nonterminals for each word that appears as an actual parameter. However, definition 2.2 is easier to work with formally.

Example 2.4 A restricted macro grammar for a fragment of the grammar of JavaScript [6] expressions serves as running example. The fragment encompasses constants, `c`, array literals enclosed in `[` and `]`, as well as object literals enclosed in `{` and `}`. Array and object literals both contain comma-separated lists modeled with the parameterized nonterminals L and N. Object literals consist of key-value assignments as described by nonterminal A. Terminal symbols are indicated by typewriter font.

$$
\begin{array}{llll}
S \to E & E \to \texttt{c} & E \to \texttt{\{}\ L(C, A)\ \texttt{\}} & E \to \texttt{[}\ L(C, E)\ \texttt{]} \\
C \to \texttt{,} & A \to \texttt{c}\ \texttt{:}\ E & & \\
L \to \varepsilon & L \to N(0, 1) & N \to 1 & N \to N(0, 1)\ 0\ 1
\end{array}
$$

Remark 2.5 The construction from Lemma 2.3 does not rely on the fact that parameters are nullary nonterminals. It works for (a suitable notion of) restricted higher-order OI-macro grammars, as well.

3 Parameterized LR Parsing

Starting from a restricted macro grammar $\mathcal{M}$, we develop the core theory of LR parsing, starting with LR items. For the sake of a name, they are called parameterized LR items, or short PLR items. All definitions are given relative to the arbitrary, but fixed, grammar $\mathcal{M} = (\Gamma, \Sigma, P, S)$ with $\Gamma = (N, a)$. Let further $k \in \mathbf{N}$ be the *lookahead*, *i.e.*, the number of characters used to decide on a parsing action. We assume familiarity with the standard notions of LR parsing theory.

Definition 3.1 A *PLR(k) item* is a triple of a production $A \to t_1 \cdots t_m$, an integer i with $0 \leq i \leq m$, and a string $v \in \Sigma^*$ with length $|v| \leq k$. It is written $[A \to t_1 \cdots t_i \bullet \cdots t_m, v]$, if $i > 0$, or $[A \to \bullet t_1 \cdots t_m, v]$, if $i = 0$.

The standard theory defines a nondeterministic finite automaton with ε transitions with the set of LR items as states. In the presence of parameters, the automaton construction needs to be generalized to a transition graph with an additional kind of arcs in comparison to a finite automaton. The construction of this graph requires—as in the standard case—the computation of k-prefixes of right-hand sides of rules. However, since right-hand sides can now contain parameters, this computation must take place relative to a parameter instance.

Definition 3.2 A *parameter instance* is a tuple of nullary nonterminals $\overline{C} = (C_0, \ldots, C_{m-1})$.

A *parameter instantiation* (for $\overline{C}$) is a tuple $\overline{s} = (s_0, \ldots, s_{l-1})$ of nullary nonterminals or parameter references (*i.e.*, integers in the range $0 \ldots m-1$).

The *application* $\overline{s}(\overline{C})$ of parameter instantiation $\overline{s}$ to parameter instance $\overline{C}$ yields a new parameter instance $\overline{s}(\overline{C}) = \overline{D} = (D_0, \ldots, D_{l-1})$ where $D_i = s_i$, if s_i is a nullary nonterminal, and $D_i = C_j$ if $s_i = j$, a parameter reference.

The well-known construction of a first-k set for the right-hand side of a production needs to be parameterized with respect to an instance.

Definition 3.3 The set of k-prefixes of an right-hand side term with respect to instance $\overline{C}$ is defined (with respect to $\mathcal{M}$) by

$$
\begin{aligned}
\mathit{first}_k^{\overline{C}}(\varepsilon) &= \{\varepsilon\} \\
\mathit{first}_k^{\overline{C}}(\alpha \cdot \beta) &= \{\mathit{prefix}_k(vw) \mid v \in \mathit{first}_k^{\overline{C}}(\alpha), w \in \mathit{first}_k^{\overline{C}}(\beta)\} \\
\mathit{first}_k^{\overline{C}}(a) &= \{a\} \\
\mathit{first}_k^{\overline{C}}(i) &= \bigcup\{\mathit{first}_k^{()}(\delta) \mid C_i \to \delta \in P\} \\
\mathit{first}_k^{\overline{C}}(B(\overline{s})) &= \bigcup\{\mathit{first}_k^{\overline{s}(\overline{C})}(\delta) \mid B \to \delta \in P\}
\end{aligned}
$$

where $\mathit{prefix}_k(a_1 \ldots a_m) = a_1 \ldots a_l$ with $l = \min(k, m)$.

Example 3.4 In Example 2.4, the nonterminals N and L occur in two in-

stances (C, A) and (C, E). Respectively,

$$\mathit{first}_1^{(C,A)}(N) = \{c\} \qquad \mathit{first}_1^{(C,A)}(L) = \{\varepsilon, c\}$$
$$\mathit{first}_1^{(C,E)}(N) = \{c, \texttt{\{}, \texttt{[}\} \qquad \mathit{first}_1^{(C,E)}(L) = \{\varepsilon, c, \texttt{\{}, \texttt{[}\}$$

Definition 3.5 The *PLR(k) transition graph* has as nodes PLR(k) items and arcs labeled with either a terminal symbol, a nonterminal symbol, a parameter, or a parameter instantiation. First, we need two auxiliary notions.

A path $X_0 \dots X_n$ with an arc labeled l_j between X_{j-1} and X_j *has parameter instance* $\overline{C}$ *with respect to* $\overline{B}$ if either

- $n = 0$ and $\overline{C} = \overline{B}$
- $n > 0$ and either
  - $l_n = (\overline{s})$, $X_0 \dots X_{n-1}$ has parameter instance $\overline{D}$ with respect to $\overline{B}$, and $\overline{C} = \overline{s}(\overline{D})$, or
  - l_n is not an instantiation and $X_0 \dots X_{n-1}$ has parameter instance $\overline{C}$ with respect to $\overline{B}$.

A node X *has parameter instance* $\overline{C}$ if there is a nullary nonterminal B, a production $B \to \delta \in P$, a terminal string v, and a path from node $[B \to \bullet\delta, v]$ to X that has parameter instance $\overline{C}$ with respect to $()$.

The transition graph is then defined as the smallest graph G such that the following conditions hold.

- $[S \to \bullet\alpha, \varepsilon]$ is a node of G, if $S \to \alpha \in P$ (S is the start symbol).
- If $X = [A \to \beta \bullet a\gamma, v]$ is a node of G, then $Y = [A \to \beta a \bullet \gamma, v]$ is a node of G and $X \xrightarrow{a} Y$ is a labeled arc in G.
- If $X = [A \to \beta \bullet B(s_1, \dots, s_k)\gamma, v]$ is a node of G (for $k \geq 0$), $B \to \delta \in P$, X has parameter instance $\overline{C}$, and $v' \in \mathit{first}_k^{\overline{C}}(\gamma v)$, then $Y = [B \to \bullet\delta, v']$ is a node of G and $X \xrightarrow{(s_1,\dots,s_k)} Y$ is a labeled arc in G.
- If $X = [A \to \beta \bullet B(s_1, \dots, s_k)\gamma, v]$ is a node of G (for $k \geq 0$), then $Y = [A \to \beta B(s_1, \dots, s_k) \bullet \gamma, v]$ is a node of G and $X \xrightarrow{B} Y$ is a labeled arc in G.
- If $X = [A \to \beta \bullet i\gamma, v]$ is a node of G, then $Y = [A \to \beta i \bullet \gamma, v]$ is a node of G and $X \xrightarrow{i} Y$ is a labeled arc in G.
- If $X = [A \to \beta \bullet i\gamma, v]$ is a node of G, X has parameter instance $\overline{C} = (C_0, \dots, C_{l-1})$, $C_i \to \delta \in P$, and $v' \in \mathit{first}_k^{\overline{C}}(\gamma v)$, then $Y = [C_i \to \bullet\delta, v']$ is a node of G.

Example 3.6 Figure 1 is the transition graph for the example grammar for lookahead $k = 0$. In the figure, terminal symbols are indicated with double quotes. The lookahead part of the items is omitted (since it is always empty). The inclusion of $C \to \bullet$, is caused by $N \to N(0, 1) \bullet 0\ 1$ in instances (C, A) or (C, E). The inclusion of $A \to \bullet\texttt{c}\ :E$ is caused by $N \to \bullet 1$ in instance (C, A).

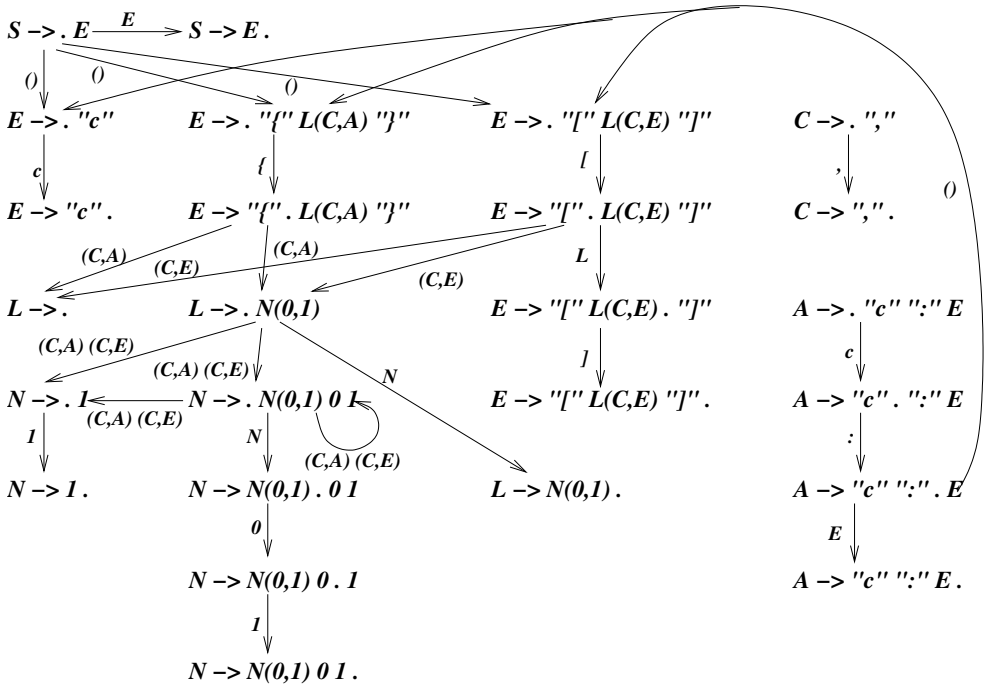

Fig. 1. Transition Graph of Example Grammar

The figure shows that the subgraph for the parameterized nonterminals is shared among their instances.

In traditional LR parsing, the state of a parser is the closure of a set of kernel items. In parameterized LR parsing the state of a parser is a mapping from tuples of nullary nonterminals to a set of PLR(k) items. The notion of closure corresponds to a notion of consistency of a state.

Definition 3.7 A *PLR(k) pre-state* is a mapping that sends a tuple of nullary nonterminals to a set of PLR(k) items.

A PLR(k) pre-state q is *consistent* with respect to transition graph G if, for all $\overline{C} \in dom(q)$, $[A \to \beta \bullet \gamma, v] \in q(\overline{C})$ and $[A \to \beta \bullet \gamma, v] \xrightarrow{(\overline{s})} Y$ in G implies that $Y \in q(\overline{s}(\overline{C}))$.

A *PLR(k) state* is a consistent PLR(k) pre-state.

The function *close* maps a PLR(k) pre-state q to the smallest PLR(k) state containing q. The *close* function performs all instantiation of parameters by enforcing the consistency condition. This happens to be the only place in a PLR parser, where instantiation plays a role!

The *goto* function implements a state transition in the parser. It maps a

PLR(k) pre-state and a grammar symbol to a PLR(k) state.

$$\begin{aligned} goto(q,a) &= close\Big\{ \overline{C} \mapsto \{[A \to \beta a \bullet \gamma, v] \mid [A \to \beta \bullet a\gamma, v] \in q(\overline{C})\} \\ &\qquad \Big|\ \overline{C} \in dom(q)\Big\} \\ goto(q,B) &= close\Big\{ \overline{C} \mapsto \{[A \to \beta B(\overline{s}) \bullet \gamma, v] \mid [A \to \beta \bullet B(\overline{s})\gamma, v] \in q(\overline{C})\} \\ &\qquad\qquad \cup \{[A \to \beta i \bullet \gamma, v] \mid [A \to \beta \bullet i\gamma, v] \in q(\overline{C}), C_i = B\} \\ &\qquad \Big|\ \overline{C} \in dom(q)\Big\} \end{aligned}$$

Example 3.8 Continuing the running example further, the initial state of the parser is the closure of the PLR(0) pre-state $\{() \mapsto \{[S \to \bullet E]\}\}$, that is

$$() \mapsto \{[S \to \bullet E], [E \to \bullet \texttt{c}], [E \to \bullet \texttt{\{}L(C,A)\texttt{\}}], [E \to \bullet \texttt{[}L(C,E)\texttt{]}]\}$$

Applying the *goto* function with terminal `{` as argument yields

$$\begin{aligned} () &\mapsto \{[E \to \texttt{\{} \bullet L(C,A)\texttt{\}}], [A \to \bullet \texttt{c} : E]\} \\ (C,A) &\mapsto \{[L \to \bullet], [L \to \bullet N(0,1)], [N \to \bullet 1], [N \to \bullet N(0,1)\ 0\ 1]\} \end{aligned}$$

For comparison, the transition with terminal `[` yields

$$\begin{aligned} () &\mapsto \{[E \to \texttt{[} \bullet L(C,E)\texttt{]}]\} \\ (C,E) &\mapsto \{[L \to \bullet], [L \to \bullet N(0,1)], [N \to \bullet 1], [N \to \bullet N(0,1)\ 0\ 1]\} \end{aligned}$$

Hence, the parsing state becomes modular with the parameterized part being reusable (but requiring a two-stage mapping to compute *goto*).

4 PLR(k) Parsing Algorithm

We adhere to the functional description of LR parsing given by Leermakers [13] and exploited in earlier work by one of the authors [18]. The idea of the functional description is that a parser is considered as a nondeterministic function mapping an input string to a pair of a PLR(k) item and a new string. The specification of a parser is thus

$$([A \to \beta \bullet \gamma, v], w'') \in parse(w) \quad \text{iff} \quad w = w'w'' \wedge \gamma \stackrel{*}{\Rightarrow} w' \wedge prefix_k(w'') = v$$

4.1 Nondeterministic Parsing

In our specific case, the parser function depends on a PLR(k) state and—instead of dealing with nondeterministic functions—it will return a set of pairs of PLR(k) item and unconsumed input. Interestingly, the main parser

function does not change with respect to the previous formulation [17].

$$
\begin{aligned}
&[q]\,(w, c_1 \ldots c_{nactive(q)}) = \\
&\quad \textbf{letrec} \\
&\qquad c_0(X, w) = \textbf{let } q' = goto(q, X) \textbf{ in } [q'](w, c_0 c_1 \ldots c_{nactive(q')}) \\
&\quad \textbf{in} \bigcup (\quad \{c_{|\alpha|}(A, w) \mid [A \to \alpha\bullet, v] \in q, prefix_k(w) = v\} \\
&\qquad\qquad \cup \{c_0(a, w') \mid w = a \cdot w', a \in nextterm(q)\})
\end{aligned}
$$

The parsing function $[q]$ has two parameters, the input string and a list of continuations. Each continuation corresponds to a function that shifts on a nonterminal whenever a reduction occurs. Calling the continuation performs the reduce operation: "pop the right-hand side of the production from the parse stack and then push the state corresponding to the left-hand side." For readers used to the stack-based implementation, it is probably best to regard the list $c_1 \ldots$ as a representation of the topmost items of the parse stack.

The function c_0 performs a shift action by pushing the continuation corresponding to the current state (itself) on the list of continuations and changing the state according to the symbol X.

The auxiliary functions need to be extended to PLR(k) states. The function $nactive : \mathit{PLR\text{-}state} \to \mathbf{N}$ determines the maximum number of continuations required to perform a reduce action in any state reachable from state q. It is the maximum number of symbols to the left of the dot in any item in q.

$$nactive(q) = \max\{|\beta| \mid \overline{C} \in dom(q), [A \to \beta \bullet \gamma, v] \in q(\overline{C})\}$$

The other auxiliary function $nextterm : \mathit{PLR\text{-}state} \to \mathcal{P}(\Sigma)$ yields the set of terminal symbols immediately to the right of the dot in any item in a state.

$$nextterm(q) = \{a \mid \overline{C} \in dom(q), [A \to \beta \bullet a\gamma, v] \in q(\overline{C})\}$$

4.2 Deterministic Parsing

The above, nondeterministic parsing function performs generalized LR parsing [12,21,15] because it explores all parsing alternatives "concurrently." To obtain a deterministic parsing function, states with "parsing conflicts" need to be identified and ruled out. As with standard LR parsing, there are two kinds of conflicts, a shift-reduce conflict or a reduce-reduce conflict.

Definition 4.1 A PLR(k) state q has a *shift-reduce conflict* if there are $\overline{B}, \overline{C} \in dom(q)$ (not necessarily different) such that $[A \to \beta \bullet a\gamma, u] \in q(\overline{B})$ and $[B \to \delta\bullet, v] \in q(\overline{C})$ with $first_k^{\overline{B}}(a\gamma u) = v$.

A PLR(k) state q has a *reduce-reduce conflict* if there are $\overline{B}, \overline{C} \in dom(q)$ (not necessarily different) such that $[A \to \beta\bullet, v] \in q(\overline{B})$ and $[B \to \delta\bullet, v] \in q(\overline{C})$ with $A \to \beta$ different from $B \to \delta$.

A restricted macro grammar is a *PLR(k) grammar* if the states extracted from the grammar's transition graph are all free of conflicts.

While the generalized shift-reduce conflict is identical to the standard notion, the reduce-reduce conflict turns out to be less restrictive. Here is an example that exhibits the difference with respect to the expanded grammar from Lemma 2.3. Let $\mathcal{G}$ be given by the productions

$$S \to L(A), S \to L(B), L \to \varepsilon, L \to 0\ L(0), A \to a, B \to b$$

where the parameterized nonterminal L generates lists. (The example abstracts from a situation where two alternative ways of writing the items of a parameter list.) The expanded grammar has productions

$$S \to L_A, S \to L_B, L_A \to \varepsilon, L_A \to A\ L_A, L_B \to \varepsilon, L_B \to B\ L_B, A \to a, B \to b$$

and the (standard LR) closure of $\{[S \to \bullet L_A], [S \to \bullet L_B]\}$ has a reduce-reduce conflict between $[L_A \to \bullet]$ and $[L_B \to \bullet]$ already (ignoring the shift-reduce conflict of either item with $[A \to \bullet a]$ and $[B \to \bullet b]$):

$$S \to \bullet L_A, S \to \bullet L_B,$$

$$L_A \to \bullet, L_A \to \bullet A\ L_A, L_B \to \bullet, L_B \to \bullet B\ L_B, A \to \bullet a, B \to \bullet b$$

However, the PLR(0) closure of $\{() \mapsto \{[S \to \bullet L(A)], [S \to \bullet L(B)]\}\}$ is

$$\begin{aligned}
() &\mapsto \{[S \to \bullet L(A)], [S \to \bullet L(B)], [A \to \bullet a], [B \to \bullet b]\} \\
(A) &\mapsto \{[L \to \bullet], [L \to \bullet 0\ L(0)]\} \\
(B) &\mapsto \{[L \to \bullet], [L \to \bullet 0\ L(0)]\}
\end{aligned}$$

and—according to our definition—there is no reduce-reduce conflict because the underlying production of the reduce items is $L \to \varepsilon$ in both cases.

Apart from the different definition of conflict, the remaining notions of conflict resolution go through as in the standard case.

5 Attribute Evaluation

As with standard LR(k) parsers, a PLR(k) parser can evaluate all attributes occurrences in a parse tree of an S-attributed grammar during parsing. The novelty is that attributes of parameterized nonterminals should have polymorphic types. To see that, let's reconsider the grammar fragment for comma separated lists from the introduction, equipped with generic semantic actions:

```
SepList (Sep, Item)
        : /* empty */                          { s1 ()   }
        | NESepList (Sep, Item)                { s2 ($1) } ;
NESepList (Sep, Item)
        : Item                                 { n1 ($1) }
        | NESepList (Sep, Item) Sep Item       { n2 ($1, $2, $3) } ;
```

In fact, `Sep` and `Item` may be regarded as type variables so that the generic type for `NESepList (Sep, Item)` has the form $\forall\alpha\beta.\tau$. This observation may be phrased as a type system for semantic actions of parameterized grammars.

It relies on an unspecified, external typing judgment $\Delta \vdash' e : \tau$ which relates a typing environment Δ and an expression e to its type τ. The only assumptions about this type system are that types may contain type variables and $\forall$-quantification is permitted at the top-level.

To derive the type of the right-hand side $w\{e\}$ of a rule with semantic action e, as captured by judgment $\Theta \vdash w\{e\} : \tau$ requires two premises. First, A typing environment Δ is created by the judgment $\Theta \vdash_i w \Rightarrow \Delta$ from the typing environment for nonterminals, Θ, a position in the right-hand side of a production, i, and a right-hand side of a production, w. Second, the right-hand side is the type of the semantic action in typing environment Δ as derived by the external judgment $\Delta \vdash' e : \tau$.

$$\frac{\Theta \vdash_1 w \Rightarrow \Delta \qquad \Delta \vdash' e : \tau}{\Theta \vdash w\{e\} : \tau}$$

The next set of rules specifies the construction of the variable environment for typing the semantic action. It assumes that the action refers to attributes of right-hand side symbols by a positional notation, $\$i$.

$$\Theta \vdash_i \varepsilon \Rightarrow \emptyset \qquad \frac{\Theta \vdash_{i+1} w \Rightarrow \Delta}{\Theta \vdash_i aw \Rightarrow \Delta, \$i : \Sigma} \qquad \frac{\Theta \vdash_{i+1} w \Rightarrow \Delta \qquad \Theta(j) = \tau}{\Theta \vdash_i jw \Rightarrow \Delta, \$i : \tau}$$

$$\frac{\Theta \vdash_{i+1} w \Rightarrow \Delta \qquad \Theta(B) = \forall \alpha_1 \ldots \alpha_m.\tau \qquad (\forall 1 \leq j \leq m)\ \Theta \vdash s_j : \tau_j}{\Theta \vdash_i B(s_1 \ldots s_m)w \Rightarrow \Delta, \$i : \tau[\alpha_j \mapsto \tau_j]}$$

The next rule specifies an auxiliary judgment used to infer the instantiation for a parameterized nonterminal.

$$\frac{\Theta(j) = \tau}{\Theta \vdash j : \tau} \qquad \frac{\Theta(B) = \tau}{\Theta \vdash B : \tau}$$

The final group of rules collects the types from the productions. The first rule collects the types of all right-hand sides, makes sure that their types are all equal, and that the resulting type is polymorphic with respect to the parameters of the left-hand side nonterminal.

$$\frac{\begin{array}{c} a(A) = m \qquad \Theta, 1 : \alpha_1, \ldots, m : \alpha_m \vdash w_i\{e_i\} : \tau \\ \Theta(A) = \forall \alpha_1 \ldots \alpha_m.\tau \qquad (\forall 1 \leq j \leq m)\ \alpha_j \notin \mathit{free}(\Theta, \alpha_1, \ldots, \alpha_{j-1}) \end{array}}{\Theta \vdash A \to w_1\{e_1\} \mid \ldots \mid w_n\{e_n\}}$$

Finally, the productions in a grammar must be mutually consistent.

$$\frac{\mathit{dom}(\Theta) = N \qquad (\forall p \in P)\ \Theta \vdash p}{\Theta \vdash P}$$

6 Implementation

We have implemented parameterized LR parsing in the context of Essence, a partial-evaluation based LR parser generator for Scheme [17].

Essence differs from most other parser generators, as for example bison [5] or yacc [10], both in the way it is built and it is used. Instead of testing and debugging a parser running several edit–generate–compile–test cycles, the user solely works with a generic parser taking both a grammar and an input stream as input to develop the final grammar. Hence, no generation and recompilation is necessary to try out changes to a grammar. In the end, an automatic program transformation called partial evaluation produces a generated parsers from the general parser [16,18]. This guarantees consistency and ensures correctness. Nonetheless users of Essence need not to have any special knowledge about partial evaluation techniques.

Integrating parameterized LR parsing into Essence amounts to adapting its general parser to parameterized LR parsing. A parser generator for PLR parsing results again by applying a partial evaluation framework to the adapted general parser with respect to PLR(k) grammars the same way it is done for the original Essence parser.

The general parser of Essence is a straightforward reformulation of a functional description of general LR parsers in the Scheme programming language [13,18]. The integration of PLR(k) parsing resulted in implementing an additional representation for PLR grammars, for parse items, and in implementing adapted versions of *first*, *goto*, *nactive* and *nextterm*. The rest of the parsing infrastructure of Essence, as for example the main parser function, stays unchanged.

7 Related Work

Parser combinators [20,9] are a highly flexible way of specifying parsers in functional programming languages. In particular, the use of polymorphic functions and parameterized parsers is a natural way of structuring code with parser combinators. In contrast to the present work, they perform predictive or top-down parsing. Recent advances [19] have widened their scope considerably with respect to earlier, inefficient proof-of-concept implementations. The present work makes some of the polymorphism and flexibility that make parser combinators attractive available to the construction of LR parsers.

The syntax definition formalism SDF [22] supports arbitrary context-free grammars and creates GLR parsers [12,21,15] for them. For convenience, right-hand sides may contain an extended set of regular operators. An SDF specification also defines a lexical syntax. SDF includes an abbreviation mechanism which works by expansion.

Extensions of LR parsing with regular operators on the right-hand sides of productions have been explored by Chapman [2]. He extends the stan-

dard item set construction with new cases for these operators. However, the attached semantic actions are fixed to *e.g.* list construction.

The compiler construction toolkit Eli [8] also constructs bottom-up parsers from grammars with regular right-hand sides. The regular operators are expanded in a preceding grammar transformation. Extended BNF productions are more often supported by LL parser generators [14].

One reviewer mentioned van Wijngaarden (or W-) grammars [23], a Turing-complete parameterized grammar formalism used in the definition of ALGOL 68. Conceptually, W-grammars consist of two-levels. The first level defines context-free languages of interpretations of grammar symbols. These interpretations are used to generate the actual grammar productions by substitution into rule templates. However, W-grammars are a conceptual modeling tool and are not geared at generating efficient recognizers. Rather, they have been designed for describing context-sensitive aspects of programming languages. They lack the conciseness and ease of use of direct parameterization, which is a familiar concept from programming practice.

8 Conclusion and Future Work

Restricted macro grammars are an extension of context-free grammars that enables modular grammar construction from user-definable, parameterized nonterminal symbols. Restricted macro grammars recognize exactly context-free languages and are amenable to linear-time parsing and evaluation of L-attributions using an extension of LR parsing. They are particularly suited for languages that support parametrically polymorphic function because parameterized nonterminals require such functions for specifying the semantic actions. Due to polymorphism, they avoid some reduce-reduce conflicts. Polymorphic attributions may also have applications in conflict avoidance for ordinary LR parsers.

In the present formalization, the lookahead sets for different parameter instances are simply merged. A refined formulation might consider *conditional items* where the lookahead is bound to specific parameter instantiations. This refinement would enable the closure operation to omit unreachable lookahead strings and avoid conflicts between otherwise unrelated instances.

A more direct implementation, factorizing the table construction and the goto function, should be investigated. We hope that this would yield a more effective size reduction of the parse tables, but this is subject to further study.

A formal notion of type soundness for the type system in Section 5 should be defined and type inference for the system should be investigated.

Another avenue for future work would be to work out a notion corresponding to LALR parsing. It might also be worth considering the present framework for parsing OI-macro languages in their full generality. In that case, it would not be possible to precompute the transition graph, rather the graph would evolve according to the input parsed.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[2] N. P. Chapman. *LR parsing: theory and practice*. Cambridge University Press, Cambridge, UK, 1987.

[3] K. Culik and R. Cohen. LR-regular grammars—an extension of LR(k) grammars. *J. Comput. Syst. Sci.*, 7:66–96, 1973.

[4] W. Damm. The IO- and OI-hierarchies. *Theoretical Computer Science*, 20(2):95–207, May 1982.

[5] C. Donnelly and R. Stallman. *Bison—The YACC-compatible Parser Generator*. Free Software Foundation, Boston, MA, Nov. 1995. Part of the Bison distribution.

[6] ECMAScript Language Specification. http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf, Dec. 1999. ECMA International, ECMA-262, 3rd edition.

[7] M. J. Fischer. Grammars with macro-like productions. In *IEEE Conference Record of 9th Annual Symposium on Switching and Automata Theory*, pages 131–142, 1968.

[8] R. W. Gray, V. P. Heuring, S. P. Levi, A. M. Sloane, and W. M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, Feb. 1992.

[9] G. Hutton and E. Meijer. Monadic parsing in Haskell. *Journal of Functional Programming*, 8(4), 1998.

[10] S. C. Johnson. Yacc—yet another compiler compiler. Technical Report 32, AT&T Bell Laboratories, Murray Hill, NJ, 1975.

[11] D. E. Knuth. On the translation of languages from left to right. *Information and Control*, 8:607–639, 1965.

[12] B. Lang. Deterministic techniques for efficient non-deterministic parsers. In *ICALP1974*, pages 255–269, 1974.

[13] R. Leermakers. *The Functional Treatment of Parsing*. Kluwer Academic Publishers, Boston, 1993.

[14] T. J. Parr and R. W. Quong. ANTLR: A predicated-LL(k) parser generator. *Software—Practice & Experience*, 25(7):789–810, July 1995.

[15] J. Rekers. *Parser Generation for Interactive Environments*. PhD thesis, University of Amsterdam, 1992.

[16] M. Sperber and P. Thiemann. The essence of LR parsing. In W. Scherlis, editor, *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation PEPM '95*, pages 146–155, La Jolla, CA, USA, June 1995. ACM Press.

[17] M. Sperber and P. Thiemann. *Essence—User Manual*. Universität Freiburg, Freiburg, Germany, Feb. 1999. Available from ftp://ftp.informatik.uni-freiburg.de/iif/thiemann/essence/.

[18] M. Sperber and P. Thiemann. Generation of LR parsers by partial evaluation. *ACM Transactions on Programming Languages and Systems*, 22(2):224–264, Mar. 2000.

[19] S. D. Swierstra. Fast, error repairing parsing combinators. http://www.cs.uu.nl/groups/ST/Software/UU_Parsing/, Aug. 2003.

[20] S. D. Swierstra and L. Duponcheel. Deterministic, error-correcting combinator parsers. In J. Launchbury, E. Meijer, and T. Sheard, editors, *Advanced Functional Programming*, volume 1129 of *Lecture Notes in Computer Science*, pages 184–207. Springer-Verlag, 1996.

[21] M. Tomita. *Efficient Parsing for Natural Languages*. Kluwer Academic Publishers, 1985.

[22] M. van den Brand and P. Klint. ASF+SDF meta-environment user manual. http://www.cwi.nl/projects/MetaEnv/meta/doc/manual/user-manual.html, July 2002.

[23] A. van Wijngaarden, B. J. Mailloux, J. E. L. Peck, and C. H. A. Koster. Report on the algorithmic language ALGOL 68. *Numerische Mathematik*, 14:79–218, 1969.