Acta Informatica (2014) 51:419–447
DOI 10.1007/s00236-014-0201-2

ORIGINAL ARTICLE

On the incremental growth and shrinkage of LR goto-graphs

Walter Cazzola · Edoardo Vacchi

Received: 2 April 2013 / Accepted: 20 May 2014 / Published online: 11 June 2014

Abstract The LR(0) goto-graph is the basis for the construction of parsers for several interesting grammar classes such as LALR and GLR. Early work has shown that even when a grammar is an extension to another, the goto-graph of the first is not necessarily a subgraph of the second. Some authors presented algorithms to grow and shrink these graphs incrementally, but the formal proof of the existence of a particular relation between a given goto-graph and a grown or shrunk counterpart seems to be still missing in literature as of today. In this paper we use the recursive projection of paths of limited length to prove the existence of one such relation, when the sets of productions are in a subset relation. We also use this relation to present two algorithms (Grow and Shrink) that transform the goto-graph of a given grammar into the goto-graph of an extension or a restriction to that grammar. We implemented these algorithms in a dynamically updatable LALR parser generator called DEXTER (the Dynamically EXTEnsible Recognizer) that we are now shipping with our current implementation of the Neverlang framework for programming language development.

1 Introduction

1.1 Motivation and objectives

Modern software development has seen a growing interest in an old practice that has never really gone out of fashion, that is, the habit of developing custom *little languages* [4] (as Unix users like to call them) to solve a particular problem. Advocates of the so-called language-oriented programming [34] consider the implementation of domain-specific languages (DSLs) a great way to communicate with domain experts and validate the correctness of the implementation of a software product [17].

Improving language development in the industry requires dealing with usual concerns in software development: software evolution and component reusability. Just as any other piece

W. Cazzola (✉) · E. Vacchi
Università degli Studi di Milano, Via Comelico 39-41, 20135 Milan, Italy
e-mail: cazzola@di.unimi.it

of software, even a DSL implementation may be required to evolve during the years. In fact, studies pointed out [24,25] that up to 80 % of a software system lifetime is spent on maintenance and evolution activities. With such a high rate, and considering the pace at which business requirements change over the time, it is hard to believe that a DSL implementation could escape from this simple logic. Therefore, it becomes important for a language implementation to be flexible enough to admit the introduction of new features, without a complete overturn of the structures that are already in place. Another important point is DSL component reusability: modern software development is component-based, but today languages are still developed as non-modular entities, using largely static toolchains, with lexers and parsers rebuilt from source each time the grammar changes. The theoretical results that we are presenting, being related to parsers, are all concerned with the syntactic evolution of a language specification, but we are working on a more general language framework that we briefly introduce in the next section.

1.2 The Neverlang framework

The need for a parser for an evolving language is a consequence of our previous experience in the design and implementation of tools for adaptable software systems. In particular, Neverlang [9,10,33] is the framework that we developed to assist the design and implementation of programming languages using a compositional approach. The core idea is that an interpreter or a compiler for a language implementation may be generated by composing together language units, each of which encapsulates different, but related aspects of its syntax and semantics. Each module may be shared between different languages: for instance, C-like programming languages could share many of their grammar specifications, such as the syntax of loop constructs, conditionals and expressions.

Neverlang is a complete toolchain that covers all of the different aspects of the definition of a language, from the generation of its parser, to the implementation of its semantics. In the early stages of its development, the framework worked as a comprehensive code-generating toolchain. However, our final goal is to develop a language implementation system that not only is no more bound to full code re-generation for the smallest changes to the codebase, but that can also be updated *on-the-fly*, by plugging and unplugging features, possibly even at runtime [11,12].

In order to achieve this kind of flexibility, every part of the toolchain should to be developed with this purpose in mind. The work that we are presenting here is the foundation for a parser generator library that we have integrated into the latest version of Neverlang. Our requirements were the following:

– supporting arbitrary new syntax to be plugged into an existing, executable parser;
– supporting arbitrary syntax restrictions by unplugging a module;
– being able to perform such changes on-the-fly, possibly at runtime;
– being able to perform such changes without needing the original input grammar to be available.

These requirements were met by studying a method to perform syntax extensions and restrictions to a language by *updating* its parser. The result was implemented in a library called DEXTER, the *Dynamically EXTEnsible Recognizer*. DEXTER generates LALR(1) parsers that can be updated on-the-fly. The Neverlang framework uses these features to load and unload syntactic modules dynamically, but DEXTER works also as a standalone library. The details of the implementation of DEXTER in Neverlang and further developments of the framework can be found in [11,12].

1.3 State of the art

Several authors have dealt with the problem of evolving the parser of a language from different perspectives. The problem of performing syntax extensions is known to be nontrivial. Traditionally, the problem of parsing a context-free language is a matter of choosing between the LL and the LR family [23], but these techniques alone do not deal with the problem of evolution.

Modern programming languages such as Scala [26] contributed to a new wave of interest in *parser combinator* (e.g., [22]) libraries. Parser combinators are higher order functions that can be composed together to form a parser for a complete language. The focus on composability and component reuse seem in fact a great answer to the requirement for a fast development cycle and continuous evolution. Moreover, because of the way parser combinators are composed, often the resulting program resembles the structure of an EBNF grammar, which makes this technique even more appealing to the casual language developer. [13] used parser combinators in the implementation of its modular language framework. However, parser combinators, PEGs [16], and in general any technique based on recursive descent (e.g., [29,35]) are known not to be trouble-free. For instance, a naïve implementation of an ambiguous context-free grammar requires exponential time and space. Most of these problems can be usually solved using more advanced techniques (e.g., in the first case, memoization), which are however generally less easy for a casual user to grasp. In the top-down family, an early work by [8] uses LL(1) parsers to implement extensible grammars, which makes time linear, but restricts the class of recognized languages. The *rats!* is a packrat parser generator [18] that provides a rich grammar module system; it provides a form of templating to reuse partial input grammar specifications, but only if they are available in their original source form.

Other authors investigated alternative parsing algorithms, to allow more flexibility in grammar definitions. The metafront tool [6] uses a novel parsing algorithm called *specificity parsing*; the main problem with this algorithm is that the class of recognized languages is not fully characterized. On the other hand, Earley's parsing algorithm [15] is well-known, and it is able to handle any (possibly ambiguous) context-free grammar, although at the cost of a non-linear computational complexity [19]. The dynamic nature of the algorithm makes possible to implement *reflective grammars* [31], that is grammars that allow language extension at parse time, similar to those in [8], but not limited to the LL(1) class. The algorithm is also being employed in the SPARK toolkit for DSLs generation in Python [2], but the author reports concerns in term of speed. A proposed solution to this problem (e.g., [3]) involves a pre-computation phase similar to those for LL and LR parsers, but the cost is losing all of the dynamic properties that in fact would make evolution easier. The ANTLR [28] parser generator is more similar to traditional tools such as YACC, but it implements a novel extension to LL(k) called $LL(*)$ which permits arbitrary lookahead, gently falling over backtracking only as a last resort; ambiguities are resolved using *predicated grammars* that resolve nondeterminism in ambiguous grammars using *guards*. ANTLR has been shown to be very competitive in terms of and ease of use, and in terms of performance it is close to GLR (Generalized-LR, [32]) parsers, of the LR family; however, it does have limits: for instance, as opposed to LR parsers, LL(*) cannot handle left-recursive grammars.

On the other hand, LR parsers are known to be an efficient family of bottom-up parsers that is guaranteed to run in linear time for any deterministic context-free language. The main arguments against this family are that LR parsers are not easy to write by hand, and that many notable subclasses of LR, such as LALR, in general are not closed under composition [30]. The first problem can be addressed using an LR parser generator, such as YACC, that takes

(E)BNF grammars as their input. Moreover, there are parsers in the LR family that are known to be able to parse any context-free grammar, such as GLR. Scannerless Generalized LR parsers (GLR) [32] have been showed to be closed under composition, they are able to parse any context-free grammar, and are generally more efficient than Earley for programming languages that are close to LR [7]. However, for nondeterministic grammars, they may generate multiple different parse trees, and therefore execute different actions for the same input text. However, another notable class of the LR family, LALR(1), has been showed to be practical and efficient with respect both to parsing and to composition, by introducing *context-aware scanning* [36]. Both LALR and GLR parsers are based off the same formalism, that is, LR(0) goto-graphs. The conclusion is that studying a method to extend and restrict LR(0) goto-graph leads to a working foundation for reasoning about the other classes. [7] describe an algorithm to compose together different grammar portions and obtain an LR(0) goto-graph. First, the input grammars are translated into ε-NFAs, then they are compose together, and then the result is converted into the LR(0) goto-graph. The algorithm has been implemented in MetaBorg and in the OCaML project `dypgen` [27]. This intermediate representation, however, can be avoided, by applying the updating procedure directly on the LR(0) goto-graph. Early literature on incremental extension of LR(0) goto-graphs [20,21] shows promising results. However, these works do not really account for an actual formal proof of their results, favoring a mostly empirical exposition of the software tools that the authors implemented. In this work, we decided to go for another route. We tried and filled the void by completing these works with a formal proof of the existence of a relation between extended and restricted LR(0) goto-graphs using the technique of reduction over path lengths.

1.4 Technical contribution

Compared to other works, updating the LR(0) goto-graph to reflect changes in the grammar does not only make possible for a language implementation to evolve over time from the syntactic standpoint, but, most importantly, it does so while still relying on a well-known and tested family of parsers. In particular:

- the input grammar is not required to be available as a source file, as opposed to any traditional parser generator such as YACC, Bison or ANTLR;
- parsers can be updated *on-the-fly*; in this regard, the technique might render the LR-family a valid alternative to Earley's algorithm, when implemented in tools similar to Neverlang [3];
- the transformation is applied *directly* to the goto-graph, without intermediate steps (c.f. [7];)

Moreover, although early results in this field do exist [20,21] we believe that the value in our contribution is in the formal results that otherwise would not be found in other literature.

In this paper we consider the case of *growing* and *shrinking* a base grammar $\mathscr{G}$ with a set of productions Q, and describe the process of updating the LR(0) goto-graph. Technical contributions of this work:

- the formal proof of the existence of a particular relation between the goto-graph $\Gamma_{\mathscr{G}}$ of a grammar $\mathscr{G}$ and the goto-graph $\Gamma_{\mathscr{G}'}$ of the modified grammar $\mathscr{G}'$ (whether it has grown or shrunk), and the *description* of such relation; in particular, if $\Gamma_{\mathscr{G}'}$ has grown and V' is its set of vertices, there exist a graph $\hat{\Gamma}$ isomorphic to $\Gamma_{\mathscr{G}}$ such that its vertices are a subset of $\mathscr{P}(V')$.
- the *algorithms* to transform (*update*) the graph of a given grammar into a graph equivalent to that of a corresponding growing or shrinking grammar.

The key idea is that, if we indicate the growing grammar with $\mathscr{G}' = \mathscr{G} \oplus Q$, where $\oplus$ denotes the *growing* operation and Q is a set of rules, then the graph $\Gamma_{\mathscr{G}'}$ intuitively *should include* the graph $\Gamma_{\mathscr{G}}$. If this simple observation held true, then we could *augment* the graph $\Gamma_{\mathscr{G}}$of the initial grammar with new vertices and edges and transform it into a graph that is isomorphic to $\Gamma_{\mathscr{G}'}$; that is, for our intents and purposes, a graph that is completely equivalent to $\Gamma_{\mathscr{G}'}$ to drive the recognition of the language of $\mathscr{G}'$. Conversely, when $\mathscr{G}' = \mathscr{G} \ominus Q$, with $\ominus$ denoting the *shrinking* operation, then graph $\Gamma_{\mathscr{G}'}$ is supposed to be included in $\Gamma_{\mathscr{G}}$, so $\Gamma_{\mathscr{G}}$could be transformed into an equivalent representation of $\Gamma_{\mathscr{G}'}$ by removing vertices and edges. Unfortunately, the reality is more complicated: adding a rule can cause vertices in the graph to be *split* [20,21]. In other words, if $\mathscr{G}$ is a grammar and $\mathscr{G}'$ is a growing grammar, $\Gamma_{\mathscr{G}}$might *not* be a subgraph of $\Gamma_{\mathscr{G}'}$; similar considerations can be made for the shrinking case.

Thus, in this paper, we do not only explore how a graph grows and shrinks with respect to growing and shrinking grammars, but also how the *splitting phenomenon* should be dealt with. The theorems form the foundation for the algorithms presented in the latter part of this document, that we implemented in DEXTER to update on-the-fly a pre-generated parser. For what concerns LALR(1) lookahead sets, we decided to implement the computation in a non-incremental procedure that is called on-demand. In this case we employed the well-known algorithm found in [1].

1.5 Paper outline

In Sect. 2 we describe the theoretical foundations of our work, by briefly recalling common definitions about grammars; we then proceed to introduce the concepts of *growing* and *shrinking* grammar. Section 3 contains all the formal proofs that justify the correctness of the procedures to update a parser that we then describe in Sect. 4. In Sect. 5 we draw the conclusions and describe our plans for future works.

2 Terminology and theoretical background

A context-free grammar $\mathscr{G}$ is a four-tuple $\mathscr{G} = \langle \Sigma, N, S, P \rangle$, where Σ is the set of terminal symbols, N is the set of non-terminal symbols, S is the start symbol (or *axiom*) and P is a set of production rules. The symbol ε represents the empty string. If R is a relation then R^* is the reflexive transitive closure of R. Moreover, when not otherwise specified:

$$S, S', A, B, C, \ldots \in N; \quad X \in (\Sigma \cup N \cup \varepsilon); \quad a, b, c, \ldots \in \Sigma; \quad \alpha, \beta, \gamma, \ldots \in (\Sigma \cup N)^*.$$

We represent the length of a word α as $|\alpha|$. Obviously $|\varepsilon| = 0$. The relation $\Rightarrow$ is defined so that $\alpha A \gamma \Rightarrow \alpha\beta\gamma$ for all $\alpha, \gamma \in (\Sigma \cup N)^*$ and $A \to \beta \in P$. The relation $\Rightarrow^*$ is the transitive, reflexive closure of $\Rightarrow$. A nonterminal A is *useless* if there is no string ω of terminals such that $A \Rightarrow^* \omega$. A nonterminal Z is *unreachable* if there is no derivation $S \Rightarrow^* \alpha Z \beta$, where S is the axiom.

Growing and shrinking grammars. We will now define two operations over grammars called *growth* and *shrinkage*. The result of these operations applied to an input grammar $\mathscr{G}$ and a production $A \to \omega$ is a new (possibly identical) grammar $\mathscr{G}'$.

Definition 1 (*Growth*) Let $\mathscr{G} = \langle \Sigma, N, S, P \rangle$ be a grammar and $A \to \omega$ a production, and let us denote with $\sigma(A \to \omega)$ the set of terminal symbols and with $\nu(A \to \omega)$ the set of non-terminal symbols of the new rule; then grammar

$$\mathscr{G}' = \langle \Sigma \cup \sigma(A \to \omega),\ N \cup \nu(A \to \omega),\ S,\ P \cup \{A \to \omega\}\ \rangle\,.$$

is the *growing* grammar for $\mathscr{G}$, and we will denote it with $\mathscr{G} \oplus \{A \to \omega\}$.

Definition 2 (*Shrinkage*) Let $\mathscr{G} = \langle \Sigma, N, S, P\rangle$ be a grammar and $A \to \omega$ a production; then

$$\mathscr{G}' = \langle \Sigma, N, P \setminus \{A \to \omega\}, S\rangle\,.$$

is the *shrinking* grammar for $\mathscr{G}$, and we will denote it with $\mathscr{G} \ominus \{A \to \omega\}$.

Finally, we make the following generalizations: let Q be a non-empty set of productions such that $Q = \{A_1 \to \omega_1, A_1 \to \omega_2, \ldots, A_n \to \omega_n\}$ then we define

$$\mathscr{G} \oplus Q \triangleq \mathscr{G} \oplus \{A_1 \to \omega_1\} \oplus \{A_1 \to \omega_2\} \oplus \cdots \oplus \{A_n \to \omega_n\}. \qquad (1)$$

If $Q \subseteq P$ we can define $\mathscr{G} \ominus Q$ similarly.

For completeness, we will also mention the *axiom change* operation. Let $\mathscr{G} = \langle \Sigma, N, S, P\rangle$, we define the axiom change operation

$$\rho_{S_{\text{new}}}(\mathscr{G}) \triangleq \langle \Sigma, N, S_{\text{new}}, P\rangle\,.$$

Obviously, if there is no rule $S_{\text{new}} \to \omega \in P$, for some ω, then the grammar $\rho_{S_{\text{new}}}(\mathscr{G})$ generates the empty language. This is an expected effect of the axiom change operation.

2.1 The LR parsing technique

The LR technique (where LR stands for Left-to-Right) has been first described by [23]. It is a *bottom up* parsing technique, because the parse tree is built starting from the leaves, from left to right. The parsing process terminates successfully when all the parts of the input program have been *reduced* to the axiom of the grammar, which constitutes the root of the parse tree.

LR(k) parsers are usually modeled by a deterministic *pushdown automaton*, supplied with a set of states, that scans the input up to k characters ahead of the current one. A *pushdown automaton* is a finite automaton enriched with an auxiliary memory that is organized as a *pushdown stack* of states and symbols

$$I_0 X_1 I_1 X_2 I_2 \ldots X_n I_n$$

with I_j being a state and X_k being a symbol (j, k integers). The *configuration* of the parser is determined by reading a terminal a from the input and peeking the last state I on the stack. The pair (a, I_n) determines how to change the configuration; the possible actions are two: *shift* or *reduce*.

- A *shift action* removes the terminal a from the input and *shifts* a new state on the stack.
- A *reduce action* of a production $A \to X_1 \ldots X_k$ pops $2k$ elements from the stack, resulting in state I_{n-k} being on top of the stack; next, the reduce action pushes A and a new current state on the stack.

The parsing procedure succeeds when only the *starting symbol* (usually denoted with S') is left on the stack. The set of states I_k is determined by applying an iterative procedure. In particular, the construction procedure for the *canonical LR(0) set of states* is the basis for the application of many notable classes of parsing algorithms, such as LALR and GLR. Because the main matter of this article requires a good understanding of the principles behind

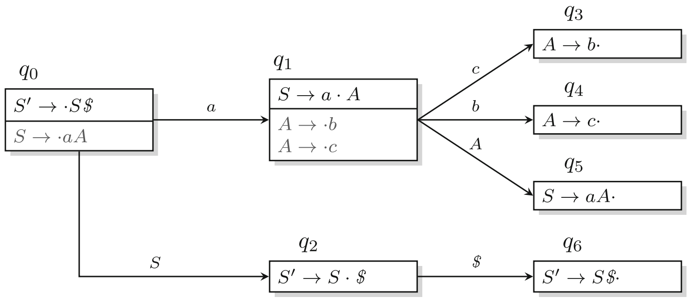

Fig. 1 The goto-graph $\Gamma_{\mathscr{G}}$ of the grammar $\mathscr{G}$ in Example 1. *Black items* are kernel (labels of the vertices), *grayed items* are nonkernel (shown here for completeness)

the construction of these sets, this section is devoted to briefly introduce the reader to the LR technique, that, although well-known, it is no more standard in many computer science curricula. In particular, we will show how to generate the LR(0) *goto-graph* [1], which is the traditional representation of the LR(0) set of states. At the end, we will also show how the goto-graph is used in an LR parser to recognize an input string. We will make use of the following example throughout all of rest of the section.

Example 1 Let $\mathscr{G}_0 = \langle \Sigma, N, P, S \rangle$ be grammar, where the production set is

$$P = \{S \to aA, \quad A \to b, \quad A \to c\} \tag{2}$$

The grammar generates the language $\{ab, ac\}$. The *goto-graph* of this grammar is represented in Fig. 1.

Generation of the canonical LR(0) set of states. The *augmented* grammar of a given grammar $\mathscr{G}_0 = \langle \Sigma, N, S, P \rangle$ is the tuple

$$\mathscr{G} = \left\langle \Sigma \cup \{\$\}, N \cup \{S'\}, S', P \cup \{S' \to S\$\} \right\rangle$$

with $\$ \notin \Sigma$. We will call $S' \to S\$$, where $S' \notin N$, the starting production. For instance, with respect to Example 1, the production set for the augmented grammar $\mathscr{G}$ would be:

$$P = \{S' \to S\$, \quad S \to aA, \quad A \to b, \quad A \to c\} \tag{3}$$

An *LR(0) item* (*item* for short) is a dotted production rule, e.g., $A \to \alpha \cdot X\beta$. A generic item will be denoted by ξ, ξ', ξ'', etc. In the following, we may say that symbol X is "dotted in ξ" if it is preceded by $\cdot$ in the item ξ. We say the dot "$\cdot$" to be leftmost in ξ if $\xi = A \to \cdot\omega$ (for some A, ω) and we say the dot to be *rightmost* if $\xi = A \to \omega\cdot$. A rule with a rightmost "$\cdot$" is said to be a *reduction candidate* or more simply a *candidate*. The item $S' \to \cdot S\$$ will be called "initial item". For convenience we also define the following expressions:

$$\mathsf{next}(A \to \alpha \cdot X\beta) \triangleq A \to \alpha X \cdot \beta \quad \text{and} \quad \mathsf{prev}(A \to \alpha X \cdot \beta) \triangleq A \to \alpha \cdot X\beta$$

with $\mathsf{next}(\xi) = \xi$ when ξ is candidate, and $\mathsf{prev}(\xi) = \xi$ when the dot in ξ is leftmost. Now, let $\mathscr{G} = \langle \Sigma, N, S, P \rangle$ be a grammar and let I be a set of items. The *closure* of I (with respect to $\mathscr{G}$) is defined as the smallest set $\textsc{Closure}_{\mathscr{G}}(I)$ such that:

1. $I \subseteq \textsc{Closure}_{\mathscr{G}}(I)$
2. $B \to \alpha \cdot A\beta \in \textsc{Closure}_{\mathscr{G}}(I),\ A \to \gamma \in \mathscr{G} \implies A \to \cdot\gamma \in \textsc{Closure}_{\mathscr{G}}(I)$

In other words, for all productions $B \to \alpha \cdot A\beta \in I$, then $\textsc{Closure}_{\mathscr{G}}(I)$ is the set obtained by first enriching I with all the items $A \to \cdot\gamma$ (provided $A \to \gamma \in \mathscr{G}$), and then enriching iteratively the obtained set until no more items can be added.

For instance, let us consider the augmented grammar of $\mathscr{G}$ in Example 1, and let $I = \{S' \to \cdot S\$\}$; since nonterminal S is dotted, then the set $\textsc{Closure}_{\mathscr{G}}(I)$ is defined to include also every item of the form $S \to \cdot\gamma$ such that $S \to \omega$ is a production of $\mathscr{G}$. In this case, there is only one such production, that is $S \to aA$. Since the set does not contain any other item with a dotted nonterminal, then $\textsc{Closure}_{\mathscr{G}}(I)$ is the set I_0:

$$I_0 = \{S' \to \cdot S\$,\ S \to \cdot aA\}. \tag{4}$$

A *kernel item* is either the initial item $S' \to \cdot S\$$ of an augmented grammar or an item whose dot is non-leftmost. Any other item is called *nonkernel*. For each set of items I we denote with $K(I)$ the subset of I that contains all of its kernel items. Notice that it is always $\textsc{Closure}_{\mathscr{G}}(K(I)) = I$, thus, for each pair of sets I, J it is $K(I) = K(J)$ if and only if $I = J$.

The particular collection of sets of LR(0) items of a given grammar that is be used to drive the pushdown automaton of an LR parser is called the *canonical* LR(0) collection; we will indicate this collection with the symbol $\mathscr{I}$. The set $\mathscr{I}$ is the result of the iterative application of the $\textsc{Closure}$ function. We will call a set of items I *state* when $I \in \mathscr{I}$. The algorithm to generated the LR(0) set of states $\mathscr{I}$ from an augmented grammar is described in [1]; we represented the algorithm here as the procedure $\textsc{Items}(\mathscr{G})$ (Algorithm 1), with $\mathscr{G}$ being an augmented grammar. The procedure uses the $\textsc{Goto}$ function, defined as follows: let $\mathscr{I}$ be the set of LR(0) states for grammar $\mathscr{G}$, and let $I \in \mathscr{I}$ and $X \in (\Sigma \cup N \cup \{\varepsilon\})$ then

$$\textsc{Goto}_{\mathscr{G}}(I, X) \triangleq \textsc{Closure}_{\mathscr{G}}(\{A \to \alpha X \cdot \beta \mid A \to \alpha \cdot X\beta \in I\}). \tag{5}$$

Because of the way $\textsc{Items}(\mathscr{G})$ is defined, we call the closure of set $I = \{S' \to \cdot S\$\}$ the *initial state* of the canonical LR(0) collection of states, and we usually indicate it with I_0.

Back to Example 1, $\mathscr{I}$ is initialized as $\mathscr{I} =: \{I_0\}$ (4). Then the procedure repeats until no new sets are added to $\mathscr{I}$. During the first run, the only set in $\mathscr{I}$ is I_0 (4); then the innermost loop will iterate over every symbol X of the grammar and compute $\textsc{Goto}_{\mathscr{G}}(I_0, X)$, but the only non-empty sets will be

$$\begin{aligned} I_1 &= \textsc{Goto}_{\mathscr{G}}(I_0, a) = \{S \to a \cdot A, \quad A \to \cdot b, \quad A \to \cdot c\}, \\ I_2 &= \textsc{Goto}_{\mathscr{G}}(I_0, S) = \{S' \to S \cdot \$\}. \end{aligned}$$

Every other set for this run will be empty. The outer **for**-loop will then compute $\textsc{Goto}_{\mathscr{G}}(X, I_1)$ for every symbol X of the grammar; the non-empty sets will be

$$\begin{aligned} I_3 &= \textsc{Goto}_{\mathscr{G}}(I_1, b) = \{A \to b\cdot\}, \\ I_4 &= \textsc{Goto}_{\mathscr{G}}(I_1, c) = \{A \to c\cdot\}. \end{aligned}$$

The procedure continues until no new sets are added to $\mathscr{I}$.

The Goto-Graph. It is quite common to represent the canonical LR(0) set of states $\mathscr{I}$ as a graph. Informally, the set of vertices of this graph corresponds to $\mathscr{I}$, and the set of edges is the set of pairs (I, J), with $I, J \in \mathscr{I}$ and such that $\textsc{Goto}_{\mathscr{G}}(I, X) = J$ for some symbol X. In the next sections we will use the following definition.

Definition 3 (*Goto-Graph*) Let $\mathscr{I}$ be the LR(0) set of states for a grammar $\mathscr{G}$, then the goto-graph is a tuple $\Gamma_{\mathscr{G}} = \langle V, E\rangle$ where V is the set of vertices and E is the set of edges. Each vertex q corresponds to one and only one state $I \in \mathscr{I}$. For each $q \in V$, we denote with $\ell_{\mathscr{G}}(q)$ the kernel $K(I)$ of the corresponding state. Then E is the subset of $V \times V$ such that for all pairs $(p, q) \in E$:

1. $\ell_{\mathscr{G}}(p) = K(I),\ \ell_{\mathscr{G}}(q) = K(J)$ and $I, J \in \mathscr{I}$
2. $\text{GOTO}_{\mathscr{G}}(I, X) = J$ with the symbol $X \in (\Sigma \cup N \cup \{\varepsilon\})$.

Algorithm 1: $\text{ITEMS}(\mathscr{G} = \langle \Sigma \cup \{\$\}, N \cup \{S'\}, S', P \cup \{S' \to S\$\}\rangle)$

```
1 $\mathscr{I} := \{\text{CLOSURE}_{\mathscr{G}}(\{S' \to \cdot S\$\})\}$
2 repeat
3     for $I \in \mathscr{I}$ do
4         for $X \in \Sigma \cup N \cup \{\varepsilon\}$ do
5             if $\text{GOTO}_{\mathscr{G}}(I, X)$ is not empty and not in $\mathscr{I}$ then
6                 add $\text{GOTO}_{\mathscr{G}}(I, X)$ to $\mathscr{I}$
7 until no new sets of items are added to $\mathscr{I}$ on a round
8 return $\mathscr{I}$
```

We will also write $\ell_{\mathscr{G}}(p, q)$ to refer to the label of the edge (p, q); if this label is X then $\ell_{\mathscr{G}}(p, q) = X$; if $(p, q) \notin E$ then we might write $\ell_{\mathscr{G}}(p, q) = \perp$. For simplicity, we may also write $\delta(p, X) = q$ when $\text{GOTO}_{\mathscr{G}}(I, X) = J$ and $\delta(p, X) = \perp$ when $\text{GOTO}_{\mathscr{G}}(I, X) = \emptyset$. To denote the edge (p, q) with label X we will also use the notation $p \overset{X}{\to} q$. A sequence $\ell_{\mathscr{G}}(p, q) = X, \ell_{\mathscr{G}}(q, r) = Y$ could be also written $p \overset{X}{\to} q \overset{Y}{\to} r$.

As you may have noticed, because $I = \text{CLOSURE}_{\mathscr{G}}(K(I))$, each vertex $q \in V$ can be labeled with the kernel $K(I)$ of some state I. It follows from the definition of LR(0) states that $\ell_{\mathscr{G}}(q) = \ell_{\mathscr{G}}(p)$ if and only if $q \equiv p$. Finally, since the LR(0) collection of states admits a notion of *initial state*, we can also define a notion of *starting vertex* for the goto-graph.

Definition 4 (*Starting Vertex*) Let be $\Gamma_{\mathscr{G}} = \langle V, E\rangle$ the goto-graph for $\mathscr{G}$ and let be $q_0 \in V$ the vertex such that the starting production $S' \to S\$ \in \ell_{\mathscr{G}}(q_0)$; then q_0 is the *starting vertex* of the goto-graph.

In the following we will always denote the starting vertex of a goto-graph with q_0. In Fig. 1 we represented the goto-graph $\Gamma_{\mathscr{G}}$for the augmented grammar $\mathscr{G}$ of the grammar in Example 1. For convenience, in the picture we also show the nonkernel items of the states (in light gray).

Parsing using LR(0) goto-graphs. For completeness, we will now show how to parse a string for the grammar shown in Example 1 and the goto-graph in Fig. 1. Because of the definition, here and in the following we can assume that for all $I_k \in \mathscr{I}$ there is one vertex $q_k \in V$ (with V being the vertex set of $\Gamma_{\mathscr{G}}$). In particular, in this example, $k = 0, 1, \ldots, 6$.

Consider now the string ab. The initial configuration has I_0 (the initial state) on the stack. Because the first symbol in the input string is a, and because there is a *transition* in the graph between q_0 and q_1, it means that it is possible to *shift* I_1. Thus, the configuration of the stack becomes $I_0\, a\, I_1$. The next symbol in the input is b; since there is a transition $q_1 \overset{b}{\to} q_3$, it is possible to shift I_3 onto the stack, thereby changing the configuration to $I_0\, a\, I_1\, b\, I_3$. In this case, $A \to b\cdot$ is a *reduction candidate*. Therefore, it is possible for the parser to *reduce by* $A \to b$, which translates to popping from the stack every symbol in the left-hand part of the rule and then pushing the head of the production; in this case, state I_3 and symbol b are removed from the stack and A is shifted, leading to configuration $I_0\, a\, I_1\, A$. The process continues until no more shift-reduce actions can be applied. The parser *accepts* the input string, if only S' (the starting symbol) is left on the stack. In this case, ab will be accepted (in fact, it is a word of the language generated by grammar $\mathscr{G}$).

3 Relations between goto-graphs

In this section we will find a relation between the graph of a given grammar $\mathscr{G}$ and the graph of a corresponding growing grammar $\mathscr{G}' = \mathscr{G} \oplus Q$. This relation will help us to define a procedure to *augment* the graph of $\mathscr{G}$ in such a way that the result is equivalent to the graph of the growing grammar $\mathscr{G}'$. Finally, we will also describe the opposite procedure, to obtain the graph of a shrinking grammar $\mathscr{G} \ominus Q$, starting from a given $\mathscr{G}$. Before we carry on with the details, it is useful to introduce a notion of *path*.

Definition 5 (*Path*) Let $\Gamma_{\mathscr{G}} = \langle V, E\rangle$; we call *path* a string $\alpha = X_0X_1 \ldots X_k$ of symbols of $(\Sigma \cup N \cup \{\varepsilon\})$ such that:

$$q_0 \overset{X_0}{\rightarrow} q_1 \overset{X_1}{\rightarrow} q_2 \overset{X_2}{\rightarrow} \cdots \overset{X_{k-2}}{\rightarrow} q_{k-1} \overset{X_{k-1}}{\rightarrow} q_k$$

where q_0 is the starting vertex by convention, and $q_1, q_2, \ldots q_k$ is any sequence of vertices of V for which the condition holds. We can say that path α *reaches* q_k or that q_k *is reachable through* α.

The length of a path is generally the length of the word α, unless the last symbol $X_{k-1} = \varepsilon$; in that case (and only in that case) the length of the path is $|\alpha| + 1$. In fact, because of the way LR(0) states are constructed, if there is one ε on a path, it is always the last symbol of the path: if $p \overset{\varepsilon}{\rightarrow} q$, it is not possible that there is some $\bar{X}$ such that $p \overset{\varepsilon}{\rightarrow} q \overset{\bar{X}}{\rightarrow} r$, as it is generally assumed that rules containing ε are always of the form $Z \rightarrow \varepsilon$, with Z being a nonterminal. Thus, $\ell(q) = \{Z \rightarrow \cdot\varepsilon\}$, which implies that there cannot be any $\bar{X}$ such that $q \overset{\bar{X}}{\rightarrow} r$: in fact, the case $p \overset{\varepsilon}{\rightarrow} q \overset{\bar{X}}{\rightarrow} r$ would only be possible if the grammar contained a rule of the form $Z \rightarrow \omega\varepsilon\bar{X}\omega'$ with $\omega, \omega' \in \Sigma \cup N$; but then the rule would be written as $Z \rightarrow \omega\bar{X}\omega'$.

The degenerate 0-length path is admissible only in those graphs $\Gamma_{\mathscr{G}}$ where $V \equiv \{q_0\}$. For instance, this is the case for a grammar with one sole production of the form $S \rightarrow A$, that is a grammar where A is a useless nonterminal. It is easy to see that in these cases the language generated by $\mathscr{G}$ is empty: this is not to be confused with those grammars $\mathscr{G}$ whose generated language is the sole word ε: in this case there will be at least one path $q_0 \overset{\varepsilon}{\rightarrow} q_1$, for some $q_1 \in V$.

For the sake of simplicity, in the following we will assume grammars not to include productions of the form $Z \rightarrow \varepsilon$, therefore, for all paths α the length of a path will be the length of the word $|\alpha|$.

3.1 Relations between goto-graphs: growing grammars

We shall now consider a grammar $\mathscr{G} = \langle \Sigma, N, P, S\rangle$ and its growing grammar $\mathscr{G}' = \mathscr{G} \oplus Q$. For the sake of simplicity, we will suppose that the production set Q is always a singleton;[1] the results can be easily generalized to the case when the cardinality m of Q (denoted by $|Q|$) is greater than 1 by considering the m singleton sets (one set for each rule of Q) and the chain seen in (1).

[1] We disregard the case when Q is empty since it is explicitly excluded by our definition of growing grammar (see Sect. 2).

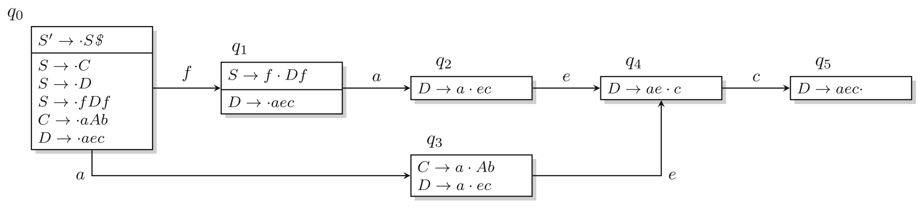

Fig. 2 Representation of a portion of $\Gamma_{\mathscr{G}}$ for the grammar $\mathscr{G}$ in Example 2

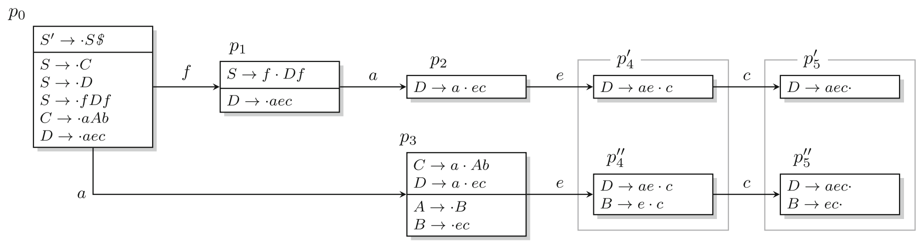

Fig. 3 Representation of a portion of $\Gamma_{\mathscr{G}}$ for the grammar $\mathscr{G}'$ in Example 2

The Graph of $\mathscr{G}'$ as an Augmented $\Gamma_{\mathscr{G}}$. Earlier work (e.g. [21]) has proven by counterexamples that given the goto-graph $\Gamma_{\mathscr{G}}$ for $\mathscr{G}$ and a growing grammar $\mathscr{G}' = \mathscr{G} \oplus Q$ it is not always true that $\Gamma_{\mathscr{G}}$ is a subgraph of $\Gamma_{\mathscr{G}'}$.

In fact, due to what has been called a *splitting phenomenon* [21], one vertex in $\Gamma_{\mathscr{G}}$ might correspond to more than one vertex in $\Gamma_{\mathscr{G}'}$. Consider the next example.

Example 2 (Splitting) The following is the grammar $\mathscr{G}$ presented in [21].

$$S \to C \mid D \mid fDf, \;\; C \to aAb, \;\; D \to aec, \;\; B \to ec \tag{6}$$

We are showing the relevant vertices and edges of this goto-graph in Fig. 2. Please notice that B is an unreachable nonterminal and A is useless. If we consider $\mathscr{G}' = \mathscr{G} \oplus Q$, with $Q = \{A \to B\}$, here B becomes reachable and A is no longer useless. In the new graph $\Gamma_{\mathscr{G}'}$ (Fig. 3) that we can obtain by applying ITEMS($\mathscr{G}'$) (Algorithm 1), we could informally say that some vertices have *split*. In particular, while it is easy to find a bijection between q_0, q_1, q_2, q_3 and p_0, p_1, p_2, p_3, respectively, it is harder to decide whether q_4 is related to p'_4 or p''_4. In fact, in a certain sense we could even say that q_4 is related to *both* p'_4 and p''_4.

The conclusion is that the relation between $\Gamma_{\mathscr{G}}$ and $\Gamma_{\mathscr{G}'}$ is nontrivial, and that, in general, we cannot say that, for any set of productions Q and for any grammar $\mathscr{G}$, $\Gamma_{\mathscr{G}}$ is just a subgraph of $\Gamma_{\mathscr{G} \oplus Q}$. Thus, it is not possible to find a simple mapping $\varphi : V \to V'$, but, more precisely, because of the splitting phenomenon that we just observed it might be possible to find some

$$\varphi : V \to \mathscr{P}(V') \tag{7}$$

In general, if $q \in V$ we will expect $\varphi(q)$ to be a singleton. In other words we usually expect q to correspond to one and only one vertex p of $\Gamma_{\mathscr{G}'}$. For instance, in our example, q_0 corresponds to p_0 and q_1 corresponds to p_1, therefore we could pose $\varphi(q_0) \triangleq \{p_0\}$ and $\varphi(q_1) \triangleq \{p_1\}$. Then, by visually comparing Figs. 2 and 3, it is tempting to conjecture that there might be a way to map q_4 onto the set of $\{p'_4, p''_4\}$, and similarly map q_5 onto $\{p'_5, p''_5\}$. If this were possible, then, these would be the cases when a vertex *has split*. In Sect. 3.1.1 we will find the mapping φ and give a formal definition of *split vertex*.

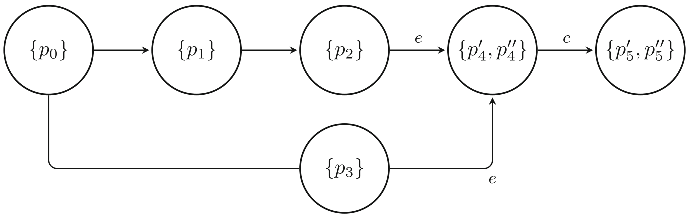

Fig. 4 The portion of graph $\hat{\varGamma}$ that relates the portion of $\varGamma_{\mathscr{G}'}$ in Fig. 3 to the portion of $\varGamma_{\mathscr{G}}$ in Fig. 2

Our final goal is to find some graph $\hat{\varGamma}$ isomorphic to $\varGamma_{\mathscr{G}}$, so that, given $\varGamma_{\mathscr{G}}$ and the set Q of new productions, it is possible to define $\varGamma_{\mathscr{G}'}$ in terms of $\hat{\varGamma}$, a set of new vertices and a set of new edges. Now, let be $\hat{\varGamma} = \left\langle \hat{V}, \hat{E} \right\rangle$: we can define the sets $\hat{V}$ and $\hat{E}$ in terms of the mapping φ; because for each $q \in V$ the image $\varphi(q)$ is a *set* of vertices of V, then the set of vertices for $\hat{\varGamma}$ can be defined as a *collection of sets* of vertices in V; the set of edges $\hat{E}$ would be then a set of pairs of elements of $\hat{V}$. Thus, it would be natural to pose $\hat{V} \subseteq \mathscr{P}(V)$ and $\hat{E} \subseteq \hat{V} \times \hat{V}$; in particular, we want $\hat{V} = \varphi(V)$ and therefore $\hat{E} \subseteq \varphi(V) \times \varphi(V)$; that is, $\hat{E}$ would contain edges between images of V through φ. For instance, let us impose $\varphi(p_4) \triangleq \{p'_4, p''_4\}$ and $\varphi(p_5) \triangleq \{p'_5, p''_5\}$ and let them be vertices in $\hat{\varGamma}$ (Fig. 4). In $\varGamma_{\mathscr{G}'}$ we have $p'_4 \overset{c}{\to} p'_5$ and $p''_4 \overset{c}{\to} p''_5$. Therefore we would like $\hat{\varGamma}$ to contain edge $\varphi(p_4) \overset{c}{\to} \varphi(p_5)$, because this edge is easy to trace back to edge $p_4 \overset{c}{\to} p_5$ in $\varGamma_{\mathscr{G}}$; moreover, edge $(\varphi(p_4), \varphi(p_5))$ should be in $\hat{\varGamma}$ because we know that $p'_4 \overset{c}{\to} p'_5$ and $p''_4 \overset{c}{\to} p''_5$ in $\varGamma_{\mathscr{G}'}$. This could be expressed by a function

$$\psi : E \to \mathscr{P}(E') \tag{8}$$

that maps edges in E onto collection of edges in E', in such a way that edges between vertices like p_4 and p_5 are mapped into the set of edges between p'_4, p'_5 and p''_4, p''_5; in other words, we want that $\psi(\,\varphi(p_4),\ \varphi(p_5)\,) = \{(p'_4, p'_5), (p''_4, p''_5)\}$. In Sect. 3.1.2 we will find this mapping ψ and we will describe the construction for $\hat{\varGamma}$.

Finally, in Sect. 3.1.3, we will describe a construction to obtain $\varGamma_{\mathscr{G}'}$ (modulo one isomorphism) by augmenting $\varGamma_{\mathscr{G}}$. In particular we will find a set $\bar{V} \subseteq V$ and a set $\bar{E} \subseteq E$ such that for some sets $\varDelta\bar{V}$ and $\varDelta\bar{E}$, the graph G:

$$G = \left\langle \bar{V} \cup \varDelta\bar{V}, \bar{E} \cup \varDelta\bar{E} \right\rangle$$

is isomorphic to $\varGamma_{\mathscr{G}'}$.

3.1.1 Construction of φ and $\varDelta V$

The mapping φ can be constructed inductively. Our strategy will be the following:

- we will define a family of functions φ_n: each of these functions will map any vertex on a r-length path ($r \leq n$) in $\varGamma_{\mathscr{G}}$ to a collection of vertices of $\varGamma_{\mathscr{G}'}$;
- we will define φ in terms of this family of functions

First, let us define a family of sets $V_n \subseteq V$. Each V_n contains each vertex of V that is reachable in $\varGamma_{\mathscr{G}}$ through every path with length at most n (see Definition 5). For instance, in Fig. 2, $V_0 = \{q_0\}$, $V_1 = \{q_1, q_3\}$, $V_2 = \{q_2, q_4\}$, etc. If the graph is *acyclic*, then there exists a longest finite path of length k in $\varGamma_{\mathscr{G}}$, and we can write:

$$V = \bigcup_{n=0}^{k} V_n \tag{9}$$

where $V_0 \triangleq \{q_0\}$ (by the definition of paths). However, if $\Gamma_{\mathscr{G}}$is *cyclic*, then there are infinite possible paths; therefore (9) becomes:

$$V = \lim_{k \to \infty} \bigcup_{n=0}^{k} V_n \tag{10}$$

We can now define the family of applications:

$$\varphi_n : \bigcup_{r=0}^{n} V_r \to \mathscr{P}(V')$$

Each of these functions maps any vertex q on any r-length path, with $r \leq n$ to a collection of vertices of $\Gamma_{\mathscr{G}'}$. Let us now suppose that $q, q', q_0 \in V$ and $p, p', p_0 \in V'$, where q_0 is the starting vertex for $\Gamma_{\mathscr{G}}$, and p_0 is the starting vertex for $\Gamma_{\mathscr{G}'} = \langle V', E' \rangle$. We can then proceed to construct φ inductively as follows:

$$\begin{aligned} &\varphi_0(q_0) \triangleq \{p_0\} \\ &\varphi_n(q) \triangleq \{p \mid \exists p' \in V', q' \in V : p' \in \varphi_{n-1}(q'), p' \overset{X}{\to} p, q' \overset{X}{\to} q, \text{for some symbol } X\} \end{aligned} \tag{11}$$

In other words, we impose q_0 to map to the singleton set $\{p_0\}$; in fact, since q_0, is the starting vertex, it can never split. Then, the image of $q \in V$ on a r-length path ($r \leq n$) is defined in terms of φ_{n-1} as the collection of all those vertices p such that:

- there is an edge $p' \overset{X}{\to} p$ in $\Gamma_{\mathscr{G}'}$
- p' was in the image of a vertex q' on a path not longer than $n - 1$
- there is an edge $q' \overset{X}{\to} q$ in $\Gamma_{\mathscr{G}}$

For instance, with reference to Fig. 2, $\varphi_1(q_1) = \{p_1\}$ because q_1 can be reached from q_0 through an r-length path, where $r \leq 1$, that is, the 1-length path f, and p_1 can be reached through the same 1-length path f from p_0 (with p_1 in Fig. 2). Also, $\varphi_3(q_4) = \{p'_4, p''_4\}$ (p'_4, p''_4 in Fig. 2); in fact, q_4 can be reached from q_0 through *two* r-length paths from q_0, with $r \leq 3$, and said paths are ae and fae; p'_4 and p''_4 can be reached through those same paths, respectively.

Now, let us consider φ_k for some $k > 0$. By definition, it is:

$$\varphi_k : \bigcup_{r=0}^{k} V_r \to \mathscr{P}(V').$$

If the graph is *acyclic*, then there exist some finite $\bar{k}$ such that $V = V_0 \cup V_1 \cup \cdots \cup V_{\bar{k}}$; thus, we can write:

$$\varphi_{\bar{k}} : V \to \mathscr{P}(V') \tag{12}$$

and pose

$$\varphi \triangleq \varphi_{\bar{k}} \tag{13}$$

When the graph is *cyclic*, paths are infinite in number, but set V is still finite; so, there will still be a finite $\bar{k}$ such that (13) holds. Consequently, even in this case $\varphi \triangleq \varphi_{\bar{k}}$.

Example 3 Consider a generic grammar $\mathscr{G}$ and its goto-graph $\Gamma_{\mathscr{G}}$, of which Fig. 2 is a partial representation. Now consider a growing grammar $\mathscr{G}'$ and its goto-graph $\Gamma_{\mathscr{G}'}$, for which the splitting phenomenon described by [21] takes place. Suppose that the resulting $\Gamma_{\mathscr{G}'}$ is portrayed in Fig. 3. We want to find a correspondence between the set $V = \{q_0, q_1, q_2, q_3, q_4, q_5\}$ of vertices of $\Gamma_{\mathscr{G}}$ and the set $V' = \{p_0, p_1, p_2, p_3, p'_4, p''_4, p'_5, p''_5\}$ of vertices of $\Gamma_{\mathscr{G}'}$. If we consider the construction in this section, the mapping φ is defined as follows: $\varphi(q_i) = \{p_i\}$ for $i = 1, 2, 3$; moreover $\varphi(q_4) = \{p'_4, p''_4\}$ and $\varphi(q_5) = \{p'_5, p''_5\}$.

We shall now prove that the definition of φ is *well-posed*. This would hold true only if we could guarantee that no vertex of $\Gamma_{\mathscr{G}}$ is ever mapped onto an empty set. Otherwise, there would some vertices of $\Gamma_{\mathscr{G}}$ that could not be put in relation with any vertex of $\Gamma_{\mathscr{G}'}$. This can only occur when there is a path in $\Gamma_{\mathscr{G}}$ that *is not* also in $\Gamma_{\mathscr{G}'}$. We shall now prove (Theorem 1) that this can never happen. We will see that φ is well-posed as a simple consequence (Corollary 1).

Theorem 1 *Let be $\mathscr{G}' = \mathscr{G} \oplus Q$; then every path in $\Gamma_{\mathscr{G}}$ is also in $\Gamma_{\mathscr{G}'}$.*

Proof The theorem can be proven by induction over the length n of a path. The 0-length path (Definition 5) is the one where the starting vertex coincides with the last vertex. By definition, it is $q_0 \in V$ and $p_0 \in V'$.

Now, by contradiction, suppose for $n = 1$ that there is one path X in $\Gamma_{\mathscr{G}}$ that is not also in $\Gamma_{\mathscr{G}'}$. Then for some q, $q_0 \xrightarrow{X} q$ but there is no p such that $p_0 \xrightarrow{X} p$. But then $\delta(p_0, X) = \bot$, which would mean that some rule $A \to X\alpha$ is in $\mathscr{G}$ but not in $\mathscr{G}'$: but this is impossible, because $\mathscr{G}' = \mathscr{G} \oplus Q$.

Now consider any n-length path, with $n > 1$. By the inductive hypothesis path $\alpha = X_1X_2 \ldots X_{n-1}$ is both in $\Gamma_{\mathscr{G}}$ and $\Gamma_{\mathscr{G}'}$ and

$$p_0 \xrightarrow{X_1} p_1 \xrightarrow{X_2} \cdots \xrightarrow{X_{n-1}} p_{n-1} \xrightarrow{X_n} p.$$

Then again, by contradiction, let us suppose that there is no p such that $p_{n-1} \xrightarrow{X_n} p$. But then there is an edge $q_{n-1} \xrightarrow{X_n} q$ in $\Gamma_{\mathscr{G}}$ that is not in $\Gamma_{\mathscr{G}'}$, which can only happen if some rule $Z \to \omega_1 X_n \omega_2$ is in $\Gamma_{\mathscr{G}}$, but not in $\Gamma_{\mathscr{G}'}$, which is a contradiction since $\mathscr{G}' = \mathscr{G} \oplus Q$. □

Corollary 1 *The image of any vertex of $\Gamma_{\mathscr{G}}$ is non-empty.*

Proof Because of the inductive definition of φ, the corollary is in turn proven by induction. We posed $\varphi_0(q_0) = \{p_0\}$ by definition (11); then obviously $\varphi(q_0)$ is non-empty.

Now, for $n > 0$, consider a $(n-1)$-length path, and let q be the last vertex of this path. By the inductive hypothesis there is at least one $p \in V'$ such that $p \in \varphi(q)$. Now suppose that $q \xrightarrow{X} q'$. If $\varphi(q')$ were empty, then there would be no $p' \in V'$ such that $p' \in \varphi(q')$. But this would contradict Theorem 1, and therefore $\varphi(q')$ must be non-empty, too. □

The concept of *split vertex* that we introduced in Sect. 3.1 will be now described more formally with the following definition.

Definition 6 (*Split vertices*) If $q \in V$ and $|\varphi(q)| > 1$ we say that q *has split* (in $\Gamma_{\mathscr{G}'}$) or that q is a *split vertex*.

The set of the *split vertices* (Definition 6) will be:

$$V_S \triangleq \left\{q \in V \;\middle|\; |\varphi(q)| > 1\right\} \tag{14}$$

We can also define the set

$$V'_S \triangleq \left\{p \in V' \;\middle|\; \exists q_s \in V_s : p \in \varphi(q_s)\right\} = \bigcup_{q_s \in V_S} \varphi(q_s). \tag{15}$$

The collection ΔV of vertices is the collection of all those vertices of $\Gamma_{\mathscr{G}'}$ that are not an image of any vertex in $\Gamma_{\mathscr{G}}$. In symbols:

$$\Delta V \triangleq \{p \in V' \mid \forall q \in V : p \notin \varphi(q)\} = V' \setminus \bigcup_{q \in V} \varphi(q) \tag{16}$$

Since our initial objective was to define a graph $\hat{\Gamma} = \left\langle \hat{V}, \hat{E} \right\rangle$ that relates $\Gamma_{\mathscr{G}}$ to $\Gamma_{\mathscr{G}'}$, we can now pose the set of vertices $\hat{V} \triangleq \varphi(V)$. The relation $\varphi(V) \subseteq \mathscr{P}(V)$ holds as expected at the beginning.

3.1.2 Construction of ψ and ΔE

In this section we will define the set of edges $\hat{E}$ for $\hat{\Gamma}$.

- We will define the function ψ to relate each edge in $\Gamma_{\mathscr{G}}$ to a (possibly non-singleton) collection of edges of $\Gamma_{\mathscr{G}'}$;
- we will define a set $\hat{E} \subseteq \hat{V} \times \hat{V}$ using ψ;
- we will finally prove this definition to be well-posed.

Let us define $\psi : E \to \mathscr{P}(E')$ by:

$$\psi(e = (q, q')) \triangleq \{(p, p') \in E' \mid p \in \varphi(q), p' \in \varphi(q')\} \tag{17}$$

In other words, the image of $(q, q') \in E$ is the collection of all those edges in E' between a vertex in the image of q and a vertex in the image of q'; that is, that particular subset of $\varphi(q) \times \varphi(q')$ that is contained in E'. Please notice that because of Theorem 1, it is always $\psi(e) \neq \emptyset$, when $e \in E$.

Example 4 With respect to Example 3, $\varphi(q_4) = \{p'_4, p''_4\}$. In $\Gamma_{\mathscr{G}}$there is one edge (q_4, q_5), while in $\Gamma_{\mathscr{G}'}$ there are two edges (p'_4, p'_5) and (p''_4, p''_5). The mapping ψ defines a relation between them all. In fact you can easily verify:

$$\psi(\,(q_4, q_5)\,) = \{(p'_4, p'_5), (p''_4, p''_5)\}.$$

We would like now $\hat{E}$ to be such that when $(q, q') \in E$ also $(\varphi(q), \varphi(q')) \in \hat{E}$. For instance, we would like $(\varphi(q_4), \varphi(q_5)) \in \hat{E}$.

Let us pose $\hat{E}$ as follows:

$$\hat{E} \triangleq \{(\varphi(q), \varphi(q')) \mid \exists p \in \varphi(q), \exists p' \in \varphi(q'), (p, p') \in \psi(\,(q, q')\,)\} \tag{18}$$

That is, for each edge $(\varphi(q), \varphi(q')) \in \hat{E}$ there is at least one edge between an element $p \in \varphi(q)$ and an element $p' \in \varphi(q')$, and that edge $(p, p') \in \psi((q, q'))$.

Now we have potentially all the elements to define $\hat{\Gamma} = \left\langle \hat{V}, \hat{E} \right\rangle$. However, we still need to guarantee that every path in $\Gamma_{\mathscr{G}}$is also in $\hat{\Gamma}$ (Theorem 3). In order to prove this, we need to give define properly the labels of the edges in $\hat{E}$. In particular, we want:

$$\ell(\varphi(q), \varphi(q')) = X \text{ when } q \overset{X}{\to} q', \quad \text{and} \quad p \overset{X}{\to} p', \forall (p, p') \in \psi((q, q')).$$

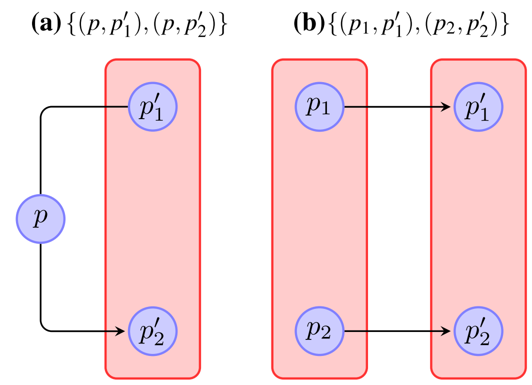
(a) $\{(p, p'_1), (p, p'_2)\}$
(b) $\{(p_1, p'_1), (p_2, p'_2)\}$
p'_1
p
p'_2
p_1
p'_1
p_2
p'_2

Fig. 5 Possible inconsistencies in the definition of $\ell(\varphi(q), \varphi(q'))$

However, given our definition of ψ, we might be concerned that in some situations the choice of that label X is not unique. In particular, if (q, q') is an edge, then the choice for $\ell(\varphi(q), \varphi(q'))$ might not be unique (Fig. 5):

- when $\psi((q, q')) \supseteq \{(p, p'_1), (p, p'_2)\}$; in fact it should be $\ell_{\mathscr{G}'}(p, p'_1) \neq \ell_{\mathscr{G}'}(p, p'_2)$.
  Otherwise the choice between $p \xrightarrow{X} p'_1$ and $p \xrightarrow{X} p'_2$ would be nondeterministic.
- when $\psi((q, q')) \supseteq \{(p_1, p'_1), (p_2, p'_2)\}$; in fact, it *might* be $\ell_{\mathscr{G}'}(p_1, p'_1) \neq \ell_{\mathscr{G}'}(p_2, p'_2)$.

Let us see that neither of these can ever occur by proving the following theorem.

Theorem 2 *If $q \xrightarrow{X} q'$ is in $\Gamma_{\mathscr{G}}$, then, for all $p' \in \varphi(q')$, it is always $p \xrightarrow{X} p'$, for any edge (p, p') of $\Gamma_{\mathscr{G}'}$, with $p \in \varphi(q)$*

Proof Let us call e the generic edge $(q, q') \in E$. Consider the set $\psi(e)$ and the collection

$$\ell(\psi(e)) = \{X \mid p \xrightarrow{X} p', (p, p') \in \psi(e)\}.$$

If any edge to a vertex $p' \in \varphi(q')$ had always the same label X, then $\ell(\psi(e))$ would be the singleton set $\{X\}$. Now, consider a vertex q' of $\Gamma_{\mathscr{G}}$; then, for some $X \in (\Sigma \cup N \cup \{\varepsilon\})$, and because of the definition of the goto-graph (Definition 3):

$$\forall q \text{ s.t. } (q, q') \in E : \ell_{\mathscr{G}}(q, q') = X. \quad (\star)$$

That is, any edge to q' of $\Gamma_{\mathscr{G}}$has the same label X. But then $\ell(\psi(e)) = \{X\}$, because of the definition of φ. □

In the proof, $(\star)$ follows from the definition of goto-graph. In fact, in the goto-graph we have that $q \xrightarrow{X} q'$ (which can also be written as $\delta(q, X) = q'$) if and only if there are two states $I, J \in \mathscr{I}$ such that $\text{GOTO}_{\mathscr{G}}(I, X) = J$ (Sect. 2.1). Because GOTO is defined in terms of CLOSURE, if there is some state $K \in \mathscr{I}$ such that $\text{GOTO}_{\mathscr{G}}(K, Y) = J$ then it must be $Y \equiv X$. Therefore, if there is some some other edge (r, q'), for some $r \in V$, then it is always $\delta(r, X) = q$, that is, $\ell_{\mathscr{G}}(r, q) = X$; but then $(\star)$ holds. For a concrete example of this, consider vertices q_2, q_3, q_4 in Fig. 2; the label of the edges between this nodes is always e.

From Theorem 2, it follows that we can always assign some single label X to $\ell(\varphi(q), \varphi(q'))$, so we can now legitimately write:

$$\varphi(q) \xrightarrow{X} \varphi(q').$$

We can finally prove that the definition of $\hat{E}$ is well-posed.

Theorem 3 *A path is in $\hat{\Gamma}$ if and only if it is in $\Gamma_{\mathscr{G}}$.*

Proof Every path is in $\hat{\Gamma}$ is also in $\Gamma_{\mathscr{G}}$ by construction. Let us then prove that when a path is in $\Gamma_{\mathscr{G}}$ it is also in $\hat{\Gamma}$, by induction on the length of the paths. For the 0-length path, the property is trivially true. Now, consider a path of length $n > 0$ in $\Gamma_{\mathscr{G}}$and $\hat{\Gamma}$, such that q and $\varphi(q)$ is the last vertex on these paths, respectively; then for each $q \overset{X}{\to} q'$ there must also be $\varphi(q) \overset{X}{\to} \varphi(q')$: in fact, let us suppose by contradiction that there is one edge $q \overset{X}{\to} q'$ but there is no edge $\varphi(q) \overset{X}{\to} \varphi(q')$. Then, because of (17) and (18), $\psi((q, q')) = \emptyset$; in other words, $\forall p \in \varphi(q)$ and $\forall p' \in \varphi(q')$ it is $(p, p') \notin E'$. But then there would be paths in $\Gamma_{\mathscr{G}}$that are not also in $\Gamma_{\mathscr{G}'}$, which would contradict Theorem 1. □

We can now define *split edges* and describe another notable set of edges, similarly to what we did previously for vertex sets.

Definition 7 (*Split edges*) If $e \in E$ and $|\psi(e)| > 1$ we say that *e has split* (in $\Gamma_{\mathscr{G}'}$) or that e is a *split edge*.

The *split edges* (Definition 7) will be the set:

$$E_S \triangleq \left\{e \in E \;\middle|\; |\psi(e)| > 1\right\} \tag{19}$$

Finally, the collection ΔE of new edges is the set of all those edges in E' that are not the image of edges in E:

$$\Delta E \triangleq \{e' \in E' \mid \forall e \in E : e \notin \psi(e')\} = E' \setminus \bigcup_{e \in E} \psi(e) \tag{20}$$

3.1.3 Construction of $\Gamma_{\mathscr{G}'}$ from $\Gamma_{\mathscr{G}}$

We will now describe how to obtain a graph $G = \langle V_G, E_G\rangle$ that is isomorphic to $\Gamma_{\mathscr{G}'}$, starting from the given $\Gamma_{\mathscr{G}}$.

Theorem 4 *Let $\Gamma_{\mathscr{G}} = \langle V, E\rangle$ and $\Gamma_{\mathscr{G}'} = \left\langle V', E'\right\rangle$, with $\mathscr{G}' = \mathscr{G} \oplus Q$, for some Q. Then, there is a graph*

$$G = \left\langle V_G = (V \setminus V_S) \cup \Delta\bar{V}, E_G = (E \setminus E_S) \cup \Delta\bar{E}\right\rangle$$

where $\Delta\bar{V} \subseteq V'$, $\Delta\bar{E} \subseteq E'$, and there is an isomorphism $\varphi_G : V_G \to V'$ such that

$$(r, r') \in E_G \iff (\varphi_G(r), \varphi_G(r')) \in E' \tag{21}$$

Proof We first pose $\bar{V} \triangleq V \setminus V_S$ and $\bar{E} \triangleq E \setminus E_S$. We also introduce the mapping $\bar{\varphi} : \bar{V} \to V'$ as a restricted version of φ (13):

$$\bar{\varphi}(q) = p \iff \varphi(q) = \{p\}$$

Please notice that when V_S, E_S [see (14) and (19)] are empty, then $\bar{V} \equiv V$ and $\bar{E} \equiv E$. Now, let us call $\Delta\bar{V} \triangleq V'_S \cup \Delta V$. Similarly, when V'_S is empty $\Delta\bar{V} \equiv \Delta V$. We can then write:

$$V_G \triangleq (V \setminus V_S) \cup (V'_S \cup \Delta V) = \bar{V} \cup \Delta\bar{V} \tag{22}$$

Now, we want to express similarly E_G, that is:

$$E_G \triangleq \bar{E} \cup \Delta\bar{E}$$

$\bar{E}$ has been defined as the set of all those edges of $\Gamma_{\mathscr{G}}$ that are not split. Now, let us describe $\Delta\bar{E}$. The description of this set is not as simple as the one for $\Delta\bar{V}$, since it is supposed to be a collection of pairs that relate vertices of $\bar{V}$ with new vertices in $\Delta\bar{V}$, in a way that makes G isomorphic to $\Gamma_{\mathscr{G}'}$. Let us define $\Delta\bar{E}$ as follows:

$$\Delta\bar{E} \triangleq E_{\text{old}} \cup E_{\text{bdg}} \cup E_{\text{new}} \tag{23}$$

- E_{old} is the set of all those edges that were *not* in $\Gamma_{\mathscr{G}}$ but are between vertices that were *already* in $\Gamma_{\mathscr{G}}$ (modulo φ);
- E_{bdg} is the set of the *bridging* edges; that is, all those edges between vertices of $\Gamma_{\mathscr{G}}$ (modulo φ) and *new* vertices of $\Gamma_{\mathscr{G}'}$ (including split vertices), and vice versa;
- E_{new} is the set of the edges between vertices that are completely new to $\Gamma_{\mathscr{G}'}$ (including split vertices)[2]

In symbols:

$$\begin{aligned}
E_{\text{old}} &\triangleq \{(q, q') \notin E \mid q, q' \in \bar{V}, (\bar{\varphi}(q), \bar{\varphi}(q')) \in \Delta E\}\\
E_{\text{bdg}} &\triangleq \{(q, p) \mid q \in \bar{V}, p \in \Delta\bar{V}, (\bar{\varphi}(q), p) \in E'\}\\
&\quad \cup \{(p, q) \mid p \in \Delta\bar{V}, q \in \bar{V}, (p, \bar{\varphi}(q)) \in E'\}\\
E_{\text{new}} &\triangleq \{(p, p') \mid p, p' \in \Delta\bar{V}, (p, p') \in E'\} \subseteq E'
\end{aligned}$$

We then pose the following isomorphism $\varphi_G : V_G \to V'$:

$$\varphi_G(r) \triangleq \begin{cases} \bar{\varphi}(r), & r \in \bar{V} \\ r, & \text{otherwise} \end{cases}$$

- If $(r, r') \in \bar{E}$ then $(\varphi_G(r), \varphi_G(r')) \equiv (\bar{\varphi}(r), \bar{\varphi}(r')) \in E'$;
- If $(r, r') \in E_{\text{old}}$ then $(\varphi_G(r), \varphi_G(r')) \equiv (\bar{\varphi}(r), \bar{\varphi}(r')) \in \Delta E$;
- If $(r, r') \in E_{\text{bdg}}$ then either:

$$(\varphi_G(r), \varphi_G(r')) \equiv (\bar{\varphi}(r), r') \in \Delta E \text{ or } (\varphi_G(r), \varphi_G(r')) \equiv (r, \bar{\varphi}(r')) \in \Delta E;$$

- If $(r, r') \in E_{\text{new}}$ then $(\varphi_G(r), \varphi_G(r')) \equiv (r, r') \in \Delta E$;

Then, by construction, the (21) holds. □

The previous theorem describes the structure of a graph that isomorphic to $\Gamma_{\mathscr{G}'} = \langle V', E' \rangle$ starting from elements of $\Gamma_{\mathscr{G}} = \langle V, E \rangle$. We called this graph $G = \langle V_G, E_G \rangle$, but, from now on, we will assume the following:

Remark 1 Because of Theorem 4, without loss of generality, we can always assume that it is always $\bar{V} = V \cap V', \bar{E} = E \cap E'$. That is, from now on, we will always assume that $V' \equiv V_G, E' \equiv E_G$. In light of this, we can also assume:

$$\Gamma_{\mathscr{G}'} \equiv G \tag{24}$$

Until now, we put aside any consideration about the labels of the vertices on purpose. We will now see how labels change between a goto-graph $\Gamma_{\mathscr{G}}$ and the graph of its growing grammar $\Gamma_{\mathscr{G}'}$. [20] made similar observations in their early work on lazy construction of LR parsers. We can now restate these observations in light of the previous proofs. Intuitively, with the growth of the graph, labels grow as well. We will proceed by cases, first considering vertices that do not split, and then the case of split vertices.

[2] Therefore $V'_S \subseteq (E_{\text{bdg}} \cup E_{\text{new}})$.

Theorem 5 *(*Labels in $\bar{V}$*) Let $\mathscr{G} = \langle \Sigma, N, S, P\rangle$ and $\mathscr{G}' = \left\langle \Sigma', N', S, P'\right\rangle$ two grammars such that $\mathscr{G}' = \mathscr{G} \oplus Q$ where $Q = \{A \to \omega\}$; let $\Gamma_{\mathscr{G}} = \langle V, E\rangle$ be the subgraph of $\Gamma_{\mathscr{G}'} = \left\langle V', E'\right\rangle$, with $\bar{V} = V \cap V'$, $\bar{E} = E \cap E'$; then, for all $q \in \bar{V}$, it is always $\ell_{\mathscr{G}}(q) \subseteq \ell_{\mathscr{G}'}(q)$.*

Proof Let us again proceed by induction on the length of a path. On the zero-length path, we consider the starting vertex q_0 (Definition 4): because of the definition of ℓ as the kernel of an LR(0) set of items, it is easy to see that $\ell_{\mathscr{G}}(q_0) \equiv \ell_{\mathscr{G}'}(q_0)$, as it will only contain the initial item $S' \to \cdot S\$$. Therefore, it is also true that $\ell_{\mathscr{G}}(q_0) \subseteq \ell_{\mathscr{G}'}(q_0)$.

Now, for all $q' \in \bar{V}$ on a $(n-1)$-length path, with $n > 0$, let us assume that the inductive hypothesis holds, and consider q such that $(q', q) \in \bar{E}$. Now, because of the ways goto-graphs are constructed (Definition 3):

$$\forall \xi \in \ell_{\mathscr{G}}(q) : \mathsf{prev}(\xi) \in \textsc{Closure}_{\mathscr{G}}(\ell_{\mathscr{G}}(q'))$$
$$\forall \xi' \in \ell_{\mathscr{G}'}(q) : \mathsf{prev}\left(\xi'\right) \in \textsc{Closure}_{\mathscr{G}'}(\ell_{\mathscr{G}'}(q'));$$

but, for the inductive hypothesis $\ell_{\mathscr{G}}(q') \subseteq \ell_{\mathscr{G}'}(q')$: then it is also $\ell_{\mathscr{G}}(q) \subseteq \ell_{\mathscr{G}'}(q)$. □

The theorem above formally proves a simple intuitive observation: if the grammar has grown, then the label of a vertex can only grow; in particular it will grow if the closure changes. Split vertices are a special case, that we treat separately in this remark.

Remark 2 (Labels of the Split Vertices) If $q \in V_S$, for each $p \in \varphi(q)$, $\ell_{\mathscr{G}'}(p)$ is at least the same as $\ell_{\mathscr{G}}(q)$; in particular:

$$\ell_{\mathscr{G}}(q) \subseteq \bigcap_{p\in\varphi(q)} \ell_{\mathscr{G}'}(p) \quad \text{and} \quad \ell_{\mathscr{G}}(q) \subset \bigcup_{p\in\varphi(q)} \ell_{\mathscr{G}'}(p). \quad (\star)$$

Proof Let us first consider all those non-split vertices that lead to the split vertex q for some symbol X; that is, all those q_i such that $q_i \overset{X}{\to} q$ for $i = 0, 1, \ldots, n$, for some n. Then, because of the definition of Goto and Closure (Sect. 2.1):

$$K(\textsc{Goto}_{\mathscr{G}}(\ell_{\mathscr{G}}(q_i), X)) = \ell_{\mathscr{G}}(q), \text{for all } i = 0, 1, \ldots, n$$

But, then this also means that there is a set of items L that is common to all labels $\ell_{\mathscr{G}}(q_i)$, or, in symbols:

$$L \subseteq \bigcap_{i=0,1,\ldots,n} \ell_{\mathscr{G}}(q_i)$$

and this set L is such that $K(\textsc{Goto}_{\mathscr{G}}(L, X)) = \ell_{\mathscr{G}}(q)$. Because every q_i is non-split, we already know that $\ell_{\mathscr{G}}(q_i) \subseteq \ell_{\mathscr{G}'}(q_i)$, then it is also:

$$L \subseteq \bigcap_{i=0,1,\ldots,n} \ell_{\mathscr{G}'}(q_i).$$

But then, by definition of $\varphi(q)$, for each $p \in \varphi(q)$ there is at least one q_i such that $q_i \overset{X}{\to} p$, and it is $\ell_{\mathscr{G}'}(p) \supseteq K(\textsc{Goto}_{\mathscr{G}'}(L, X))$. This proves the first inequality in $(\star)$ for the case when every q_i is non-split. It is easy to see that the second inequality holds as well: in fact, if the relation did not hold true, then the labels of each $p \in \varphi(q)$ would all coincide, but then, by definition of goto-graph $\varphi(q) \equiv \{p\}$ which would mean $q \notin V_S$.

Now, let us consider some edge $p \overset{X}{\to} p'$, where $p \in \varphi(q)$ and $p' \in \varphi(q')$ such that $q \overset{X}{\to} q'$, where q, q' are both split vertices. By induction, for all $p \in \varphi(q)$ there is a set

L such that $L \subseteq \ell_{\mathscr{G}'}(p)$ and such that $L \subseteq \ell_{\mathscr{G}}(q)$: then, for all $p' \in \varphi(q')$ there is a set L' such that $L' \subseteq \ell_{\mathscr{G}'}(p')$ and such that $L' \subseteq \ell_{\mathscr{G}}(q')$, and this set is $L' = \textsc{Goto}_{\mathscr{G}'}(L, X)$. This proves the left-hand inequality in $(\star)$; the right-hand inequality, again, holds as well, otherwise q' would not be a split vertex. □

We can finally enunciate the following theorem.

Theorem 6 (Relation between labels) *Let be $q \in V$, then $\forall p \in \varphi(q)$ it is $\ell_{\mathscr{G}}(q) \subseteq \ell_{\mathscr{G}'}(p)$.*

The proof follows from Theorem 5 and Remark 2.

3.2 Relations between goto-graphs: shrinking grammars

In the previous section we showed how to grow the goto-graph $\Gamma_{\mathscr{G}}$of a given grammar $\mathscr{G}$ by some rule $A \to \omega$ and then obtain the extended graph $\Gamma_{\mathscr{G}\oplus Q}$. In this section we will prove that the graph of a shrinking grammar can be obtained from the graph of the initial grammar in a similar way. The theorem that follows will prove that the operation of growth can be inverted. The graph of a shrinking grammar can be obtained from the initial grammar by removing vertices and edges. Even in this case, split vertices require a special treatment. In the case of the growth operation, we were removing the set V_S and added in the set V'_S, that contained all the edges $\varphi(q)$ such that $q \in V_S$. In this case, we will *remove* all those vertices $p \in V'_S$ that are split in $\Gamma_{\mathscr{G}}$ and then we will add all the vertices in V_S. These operations can always be done, because any grammar $\mathscr{G}'$ can be seen as the growing grammar of some $\mathscr{G} = \mathscr{G}' \ominus Q$, for some Q. In light of Theorem 4 and Remark 1, we can then enunciate the theorem as follows.

Theorem 7 *Let $\Gamma_{\mathscr{G}'} = \langle V', E' \rangle$, then there are $\Delta\bar{V}, V_S, \Delta\bar{E}, E_S$ such that*

$$\Gamma_{\mathscr{G}} = \langle V' \setminus \Delta\bar{V} \cup V_S,\ E' \setminus \Delta\bar{E} \cup E_S \rangle. \tag{25}$$

Proof Let be $\Gamma_{\mathscr{G}'} = \langle V', E' \rangle$. Because of Theorem 4, we know that there are $\bar{V} \subset V$ and $\bar{E} \subset E$ such that

$$\langle (V \setminus V_S) \cup \Delta\bar{V},\ (E \setminus E_S) \cup \Delta\bar{E} \rangle = \Gamma_{\mathscr{G}\oplus Q} = \Gamma_{\mathscr{G}'\ominus Q \oplus Q} = \Gamma_{\mathscr{G}'}$$

for some sets $\Delta\bar{V}$,$\Delta\bar{E}$. The construction of these sets has been described in the correspondent proof. It is obviously $V' = (V \setminus V_S) \cup \Delta\bar{V},\ E' = (E \setminus E_S) \cup \Delta\bar{E}$. We can then derive the following expressions:

$$V = V' \setminus \Delta\bar{V} \cup V_S,\ \ E = E' \setminus \Delta\bar{E} \cup E_S$$

The (25) follows. □

For the sake of completeness, we enunciate the following corollary, about the the vertex labels: in this case they shrink.

Corollary 2 *If q is a vertex of $\Gamma_{\mathscr{G}\ominus Q}$, then $\ell_{\mathscr{G}}(q) \supseteq \ell_{\mathscr{G}\ominus Q}(q)$.*

We omit the proof of the corollary since it trivially follows from Theorems 6 and 7.

We have now sufficient elements to design two procedures to *grow* and *shrink* any graph $\Gamma_{\mathscr{G}}$ of LR(0) states by some rule $A \to \omega$. We will call these procedures $\textsc{Shrink}(\Gamma_{\mathscr{G}}, A \to \omega)$ and $\textsc{Grow}(\Gamma_{\mathscr{G}}, A \to \omega)$, respectively. The first procedure transform $\Gamma_{\mathscr{G}}$ into an isomorph of $\Gamma_{\mathscr{G} \oplus \{A \to \omega\}}$, while the second transforms the graph into an isomorph of $\Gamma_{\mathscr{G} \ominus \{A \to \omega\}}$. We will describe these procedures in Sect. 4.

3.3 Axiom change

In Sect. 2 we briefly mentioned the *axiom change* operation. Although the formal proof will be omitted, because it would be a simple consequence of those that precede, we will sum up the idea here.

Given a grammar $\mathscr{G} = \langle \Sigma, N, P, S \rangle$, $\Gamma_{\mathscr{G}}$ is constructed using the *augmented* grammar $\mathscr{G}' = \left\langle \Sigma \cup \{\$\}, N \cup \{S'\}, S', P \cup \{S' \to S\$\} \right\rangle$, it is therefore possible to transform $\Gamma_{\mathscr{G}}$ into $\Gamma_{\rho_{\mathscr{G}}(S_{\text{new}})}$ by executing the $\textsc{Grow}$ and $\textsc{Shrink}$ procedures in sequence (the order does not matter) on the input graph $\Gamma_{\mathscr{G}}$. In particular, to obtain the graph $\Gamma_{\rho_{\mathscr{G}}(S_{\text{new}})}$ from $\Gamma_{\mathscr{G}}$, given a new axiom S_{new}, it is sufficient to execute the procedure $\textsc{Grow}(\Gamma_{\mathscr{G}}, S' \to S_{\text{new}}\$)$ obtaining a new graph $\Gamma_{\mathscr{G}'}$ and then $\textsc{Shrink}(\Gamma_{\mathscr{G}'}, S' \to S\$)$.

4 Algorithms

In Sect. 3 we showed several theoretical results applying to the LR(0) graph of a given grammar. First, we implied that it is enough to keep track of the kernel items for each state (the "labels" of the graph), since it is always $I = \textsc{Closure}_{\mathscr{G}}(K(I))$. But, most of all, we showed that, given the graph $\Gamma_{\mathscr{G}}$, it is possible to obtain $\Gamma_{\mathscr{G} \oplus Q}$ by

– adding the missing vertices;
– possibly substitute some vertices with the corresponding split vertices;
– adding the new edges along with their labels;
– updating old labels with the new kernel items.

Likewise, $\Gamma_{\mathscr{G} \ominus Q}$ is obtained from $\Gamma_{\mathscr{G}}$ by

– removing the vertices that are not in the graph of the shrinking grammar;
– substitute split vertices with the corresponding non-split vertices;
– removing any edge that is not in $\Gamma_{\mathscr{G} \ominus Q}$;
– updating old labels by removing any missing kernel item.

The axiom change operation can be seen as a combination of the two.

In the following we will present the algorithms to actually produce such results. It is also worth noticing that these algorithms do not require the input grammar to be completely available in order to be performed. In fact, they only depend on the original graph representation $\Gamma_{\mathscr{G}}$ and the particular rule $A \to \omega$ that is being added or removed. This means that an updatable parser generator such as DEXTER does not require the complete input grammar to be available in source form to perform the updates.

Even though it is possible to use the same algorithms to update *any* parser based on the LR(0) graph of states (such as SLR or GLR), our choice for DEXTER was to generate updatable LALR(1) parsers. Therefore, in the last part of this section, we also mention briefly how we compute the LALR lookahead sets. In this case, we opted for a more traditional approach: the lookahead sets are computed in a non-incremental way, using the well-known algorithm due to [1]. Since the computation would be expensive to perform at each growth or shrinkage, it is executed on demand, by calling an ad-hoc API every once in a while; for instance, when a full batch of GROW and SHRINK operations has been completed.

We will outline the proofs of correctness of these algorithms by showing that, given an input graph $\Gamma_{\mathscr{G}}$ and a set $Q = \{A \to X_1 X_2 \dots X_n\}$, then $\Gamma_{\mathscr{G} \oplus Q} = \text{GROW}(\Gamma_{\mathscr{G}}, Q)$. Likewise, we will show that $\text{SHRINK}(\Gamma_{\mathscr{G}}, Q) = \Gamma_{\mathscr{G} \ominus Q}$

4.1 The GROW procedure

In the previous sections we saw that we can obtain a graph that is isomorph to $\Gamma_{\mathscr{G}'} = \langle V', E' \rangle$, where $\mathscr{G}' = \mathscr{G} \oplus Q$ for a given $\mathscr{G}$ and some singleton set of productions Q, by augmenting the graph $\Gamma_{\mathscr{G}}$. In particular, if $\Gamma_{\mathscr{G}} = \langle V, E \rangle$, in Theorem 4, by way of Remark 1, we saw that there are V_S, E_S, and $\Delta\bar{V}$, $\Delta\bar{E}$ such that:

$$\langle V \setminus V_S \cup \Delta\bar{V}, E \setminus E_S \cup \Delta\bar{E} \rangle \equiv \Gamma_{\mathscr{G}'}.$$

Let us suppose that $Q = \{A \to X_1 X_2 \, \dots \, X_n\}$. Then for some vertex r it is:

$$B \to \alpha \cdot A\beta \in \ell_{\mathscr{G}'}(r),$$

and therefore:

$$A \to \cdot X_1 X_2 \, \dots \, X_n \in \text{CLOSURE}_{\mathscr{G}'}(\ell_{\mathscr{G}'}(r)).$$

This implies that there must exist some r' such that

$$A \to X_1 \cdot X_2 \, \dots \, X_n \in \ell_{\mathscr{G}'}(r').$$

and it *must* be $r \xrightarrow{X_1} r'$, or, otherwise stated, $\delta(r, X_1) = r'$. We formalize this requirement with the relation *implies*. We write that $r \rightsquigarrow r'$, and read it as $\ll r$ *implies* $r' \gg$, if $r, r' \in V'$ and

$$\xi \in \ell_{\mathscr{G}'}(r') \implies \mathsf{prev}(\xi) \in \text{CLOSURE}_{\mathscr{G}'}(\ell_{\mathscr{G}'}(r))$$

The relation expresses the fact that when $r \xrightarrow{X_1} r'$, for all items ξ in $\ell_{\mathscr{G}'}(r')$, $\mathsf{prev}(\xi)$ is in the closure of $\ell_{\mathscr{G}'}(r)$. In fact, by definition, trivially $r \rightsquigarrow r'$; therefore this relation shall hold for any other edge that is added to this graph. Split vertices (Definition 6) would then be all those vertices such that, for some edge $(\hat{r}, r')$ in $\Gamma_{\mathscr{G}}$, the relation $\hat{r} \rightsquigarrow r'$ does not hold in $\Gamma_{\mathscr{G}'}$ anymore. For instance, suppose to update the graph in Fig. 3 as described in Example 2. In this case, a new item $B \to \cdot ec$ is added to the label of q_3, which in turn causes the labels for q_4 and q_5 to be updated as well. However, graph $\Gamma_{\mathscr{G}}$ (see Fig. 6) contains an edge $q_2 \xrightarrow{e} q_4$ that is wrong in $\Gamma_{\mathscr{G}'}$ because $q_2 \not\rightsquigarrow q_4$. The GROW procedure is described by Algorithm 2.

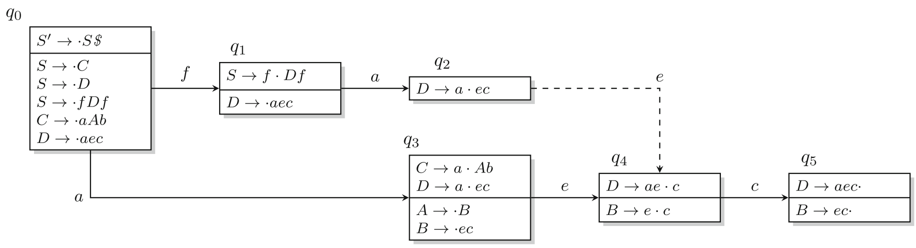

Fig. 6 Dashed, an edge that leads to a split vertex

Algorithm 2: GROW($\Gamma_{\mathscr{G}}$, $A \to X_1 X_2 \cdots X_n$)

```
$V' := V$
$R := \{r \in V' \mid B \to \beta_1 \cdot A\beta_2 \in$ CLOSURE$_{\mathscr{G}'}(\ell(r))\}$
while $\exists r \in R$ unprocessed do
    foreach $r \in R$ do
        $L \subseteq$ CLOSURE$_{\mathscr{G}'}(\ell(r))$ s.t. every item is non-candidate (Sect. 2.1)
        foreach $X$ dotted in an item $\xi \in L$ do
            Find or create $r' : r \rightsquigarrow r'$
            $\delta(r, X) := r'$
            $\ell(r') := \ell(r') \cup \{\mathsf{next}(\xi)\}$ (where $\ell(r') = \emptyset$ if $r'$ is a freshly created vertex)
            if $\exists \hat{r} \in V : \hat{r} \xrightarrow{X} r'$ but $\hat{r} \not\rightsquigarrow r'$ then $R_S =$ SPLIT($r'$)

            $R := R \cup \{r'\} \cup R_S$
```

Algorithm 3: SPLIT(r')

```
$R_S = \emptyset$
foreach $\hat{r} : \hat{r} \xrightarrow{X} r'$ but $\hat{r} \not\rightsquigarrow r'$ do
    Create $r_S$ s.t. $\hat{r} \rightsquigarrow r_S$; $\delta(\hat{r}, X) := r_S$
    foreach $Y : \delta(r', Y) \neq \bot$ do $\delta(r_S, Y) := \delta(r', Y)$

    $R_S := R_S \cup \{r_S\}$
return $R_S$
```

You should see that the set R in the procedure is defined in such a way that, the first time the procedure loops, L always contains also $A \to \cdot X_1 X_2 \ldots X_n$. The procedure is designed so that it will eventually add all of the vertices in $\Delta\bar{V}$ and the edges in $\Delta\bar{E}$ (they are expressed through the updating of the transition function δ). It performs a visit of all the parts of the graph that can be reached starting from the vertices r of R. We will now outline a proof of correctness of this algorithm.

4.1.1 GROW($\Gamma_{\mathscr{G}}$, Q) *computes* $\Gamma_{\mathscr{G}'} = \Gamma_{\mathscr{G} \oplus Q}$.

For what concerns $\bar{V}$ and $\bar{E}$, we can assume they are all preserved as expected, in fact the algorithm never deletes any vertex or edge contained in these sets. Then, the algorithm can be shown to be correct by proving that:

1. it substitutes every split vertex and every split edge with the correspondent couples of new vertices and edges;
2. it generates every new vertex in $\Delta\bar{V}$ and every new edge in $\Delta\bar{E}$;
3. it generates no more vertices than those in $\Delta\bar{V}$ and no more edges than those in $\Delta\bar{E}$.

Please notice that that now $V \subseteq V'$ (Remark 1), and therefore, for all $q \in V$ it is always at least $q \in \varphi(q)$.

New vertices and edges. Let be $r, r' \in \Delta V$, that is, they are not in V'_S. Then $(r, r') \in \Delta\bar{E}$ is created at line 7. In fact:

1. if there is r' such that $r \rightsquigarrow r'$ and $r \in \bar{V}, r' \in V$, then $(r, r') \in E_{\text{old}}$;
2. if there is r' such that $r \rightsquigarrow r'$ and $r \in \bar{V}, r' \notin \bar{V}$, then $(r, r') \in E_{\text{bdg}}$;

3. if there is r' such that $r \rightsquigarrow r'$ and $r, \notin \bar{V}$, $r' \notin \bar{V}$ then $(r, r') \in E_{\text{new}}$; if no such r' exists, then one is created, and therefore it is still $(r, r') \in E_{\text{new}}$ (see Theorem 4)

These observations should apply to split vertices as well: in fact Theorem 4 showed that $E'_S \subseteq E_{\text{bdg}} \cup E_{\text{new}}$.
Split vertices and edges. The algorithm updates the label of a vertex r' at line 9, by adding an item $\mathsf{next}(\xi)$, with $\xi \in L \subseteq \ell(r)$. If item ξ was already in $\ell(r')$, then this label will not change. Otherwise, ξ is based on a grammar rule that was either unreachable or in the set Q. Now, if r' has more than one in-edge, then it might split.

In particular, the algorithm mandates the vertex $r' \in V$ to be *split* when it is reachable from some other $\hat{r} \in V$, but it is $\hat{r} \not\rightsquigarrow r'$. Now, consider (11), which defines φ, and remember that now $V \subseteq V'$ (Remark 1). If $|\varphi(q)| > 1$ then there is certainly one vertex $p \in \varphi(q)$ such that $\ell_{\mathscr{G}'}(p) \supset \ell_{\mathscr{G}}(q)$ (Remark 2), which is therefore an alternate necessary condition for q to be a split vertex. A direct consequence is that, for some vertex $\hat{q} \in V$ such that $\hat{q} \xrightarrow{X} q$, and for said $p \in \varphi(q)$, we have that $\hat{q} \rightsquigarrow q$ but $\hat{q} \not\rightsquigarrow p$. Being this a direct consequence of a necessary condition, it is itself a necessary condition for q to be split. In other words, when the latter holds, $p \in \varphi(q)$. Of course, then $|\varphi(q)| > 1$. In particular for each $\hat{r}$ such that $\hat{r} \not\rightsquigarrow r'$, there must be a $r_S \in \varphi(r')$ such that $\hat{r} \rightsquigarrow r_S$: in fact, the algorithm adds any such vertex to V', and it adds any edge $\hat{r} \xrightarrow{X} r_S$ to E'.
The algorithm generates only vertices in $\Delta\bar{V}$ and edges in $\Delta\bar{E}$. The algorithm generates a new vertex when:

1. for some vertex r there is no vertex r' such that $r \rightsquigarrow r'$ (Algorithm 2, line 7)
2. there is an edge $\hat{r} \xrightarrow{X} r'$ such that $\hat{r} \not\rightsquigarrow r'$ (Algorithm 3, line 3)

In both cases, a new edge between the old vertex and the new vertex is added to the graph. Therefore, new vertices are only added when a new edge is added as well. It follows that we only need to prove that the algorithm generates only edges in $\Delta\bar{E}$. This is easily proven by contradiction. Suppose that the algorithm generates one edge $(r, r') \notin \Delta\bar{E}$; therefore there would exist one edge $r \xrightarrow{X} r'$ in the graph generated by the algorithm that *is not* in the graph $\Gamma_{\mathscr{G}\oplus Q}$; if $r \overset{X}{\not\rightarrow} r'$ then, it follows that $r \not\rightsquigarrow r'$: but this is absurd, because it always holds by construction: in fact, when there is an edge $\hat{r} \xrightarrow{X} r'$ $\hat{r} \not\rightsquigarrow r'$ this is removed from the graph (Algorithm 3).

Conclusion. Since the algorithm GROW($\Gamma_{\mathscr{G}}$, Q) adds to $\Gamma_{\mathscr{G}}$only edges in $\bar{E} \cup \Delta\bar{E}$, we can conclude that it computes $\Gamma_{\mathscr{G}'}$.

4.2 The SHRINK procedure

The SHRINK procedure (Algorithm 4) is defined so that it can ideally undo a previous application of GROW to the graph. If $A \rightarrow X_1X_2 \ldots X_n$ is the input rule, and

$$\xi_0 = A \rightarrow \cdot X_1X_2 \ldots X_n, \quad \xi_1 = A \rightarrow X_1 \cdot X_2 \cdots X_n, \quad \ldots, \quad \xi_n = A \rightarrow X_1X_2 \ldots X_n\cdot$$

then the algorithm visits each $r \in V'$ such that $\xi_i \in \ell(r)$, for all ξ_i, and it disconnects any vertex that should not be reachable from the starting vertex r_0. The last line removes any unreachable vertex from the result. Please notice that a real-world implementation might defer this step to an independent garbage collection procedure,[3] for obvious reasons: unreachable

[3] In fact, this is what happens in our prototype DEXTER.

Algorithm 4: SHRINK($\Gamma_{\mathscr{G}}$, $A \to X_1 X_2 \cdots X_n$)

```
1  $V' := V$
2  $\xi := A \to \cdot X_1 X_2 \cdots X_n$
3  $R = \{r \in V' \mid B \to \beta_1 \cdot A\beta_2 \in \text{CLOSURE}_{\mathscr{G}'}(\ell(r))\}$
4  while $\exists r \in R$ unprocessed do
5      foreach $r \in R$ do
6          $\ell(r) := \ell(r) \setminus \{\xi\}$
7          if $\exists r' : \ell(r) = \ell(r')$ then MERGE($r, r'$)
8
9          else
10             $L = \text{CLOSURE}_{\mathscr{G}'}(\xi) \setminus \text{CLOSURE}_{\mathscr{G}'}(\ell(r))$
11             foreach $X$ dotted in an item of $L$ do $\delta(r, X) := \perp$
12
13     $\xi := \text{next}(\xi)$
14     $R := \{r \in V' \mid \xi \in \ell(r)\}$
15 foreach $r$ unreachable from $r_0$ do $V' := V' \setminus \{r\}$
16
```

Algorithm 5: MERGE(r, r')

```
1  foreach $\hat{r} \in V' : \hat{r} \xrightarrow{X} r$, for some $X$ do $\delta(\hat{r}, X) := r'$
2  foreach $\hat{r} \in V' : r \xrightarrow{X} \hat{r}$, for some $X$ do $\delta(r', X) := \hat{r}$
```

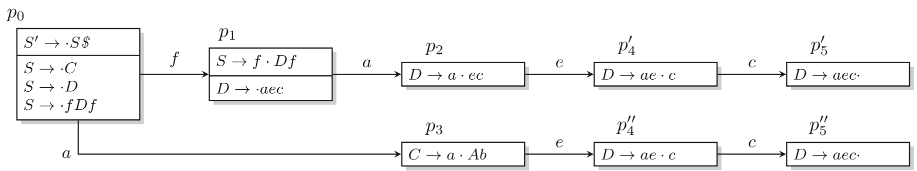

Fig. 7 In this picture p'_4, p''_4 and p'_5, p''_5 have identical labels, and should be merged

vertices will never be visited by a parsing routine, thus they can be retained up until it is necessary to reclaim back some memory. For each node that is visited, the procedure also updates the labels of the remaining nodes.

The MERGE procedure (Algorithm 5) undoes the splitting phenomenon, when it is necessary. For instance, Fig. 7 shows the situation of $\Gamma_{\mathscr{G}'}$ when we apply SHRINK *without* applying MERGE: in this case, we would have $\ell(p'_4) \equiv \ell(p''_4)$ and $\ell(p'_5) \equiv \ell(p''_5)$. This is not acceptable in a goto-graph, because if two labels coincide, then also the vertices should coincide: this stems from the definition of the vertices of the graph as sets of items, and the vertex set as the collection of the LR(0) sets of items (Sect. 2.1). It follows that, for each pair of vertices with the same labels, these vertices should be *merged*, that is, the pair should be substituted by one single vertex, such that:

1. its in-edges are all the in-edges of the vertices that are being merged
2. its out-edges are those of either *one* of the vertices that are being merged. In fact, choosing the out-edges of one vertex over the other is equivalent, because the labels of the subsequent vertices will still equal, pairwise.

Similarly to what we did for the GROW procedure, we will now outline a proof for SHRINK.

4.2.1 SHRINK$(\Gamma_{\mathscr{G}}, Q)$ *computes* $\Gamma_{\mathscr{G}'} = \Gamma_{\mathscr{G}\ominus Q}$

Theorem 7 was proven as a consequence of Theorem 4; it follows that we can see $\Gamma_{\mathscr{G}} = \left\langle V', E'\right\rangle$ as $\Gamma_{\mathscr{G}'\oplus Q}$ (Theorem 7), where $\Gamma_{\mathscr{G}'} = \langle V, E\rangle$. Therefore:

1. there is a set of edges $\Delta\bar{E}$ (described in Theorem 4) that shall be removed from $\Gamma_{\mathscr{G}}$.
2. in particular, there is a subset of vertices of $V'_S \subseteq \Delta\bar{V}$ that should be *merged* back.

Removed vertices and edges. For obvious reasons, the algorithm is expected to act symmetrically to the GROW procedure. At line 5 of the algorithm, the label of a vertex is updated by removing the item ξ, which is an dotted item derived from the input rule $A \to X_1X_2\ldots X_n$. The algorithm visits every $r \in V'$ such that $\xi \in \ell(r)$, for all ξ (that is, for every possible dot position in the input rule), and it disconnects any vertex that shall not be reachable from the starting vertex r_0. In other words, let be $r, r' \in \Delta V$ (let us not consider V'_S for now); then each pair $r, r' \in (E_{\text{old}} \cup E_{\text{bdg}})$ is removed at line 10. In fact, the set

$$L = \text{CLOSURE}_{\mathscr{G}'}(\xi) \setminus \text{CLOSURE}_{\mathscr{G}'}(\ell(r))$$

contains every item that enables a transition that shall *not* be retained in δ. Now let be r' the vertex such that $\delta(r, X) = r'$ for all X dotted in L; then either:

1. $(r, r') \in E_{\text{old}}$, that is, $r, r' \in V$
2. $(r, r') \in E_{\text{bdg}}$, that is, $r \in V, r \in V'$

The last line of the algorithm "garbage collects" every node unreachable from r_0, effectively removing every $(r, r') \in E_{\text{new}}$, that is, such that $r, r' \in \Delta V$.

Split vertices and edges. Obviously, if $\Gamma_{\mathscr{G}} = \left\langle V', E'\right\rangle$ can be seen as $\Gamma_{\mathscr{G}'\oplus Q}$, where $\Gamma_{\mathscr{G}'} = \Gamma_{\mathscr{G}\ominus Q} = \langle V, E\rangle$, then $\mathscr{P}(V'_S)$ is the collection of sets of vertices of V' that shall be mapped onto vertices of V_S. In particular, we know that $\varphi(r_S) = R_S$, with $R_S \in V'_S$, $r_S \in V_S$. Therefore, for all R_S for which the relation holds, we expect the procedure to find every $r \in R_S$ and substitute them all with one r_S. By definition of LR(0) goto-graph it is obvious that there cannot be $p, p' \in \varphi(q)$ such that $\ell(p) = \ell(p')$ otherwise $p \equiv p'$.

The removal of an item might occasionally result in two (or more) vertices of V' with the same label on two (or more) distinct paths. This means that their paths must all exist in $\Gamma_{\mathscr{G}'}$, even though these vertices cannot coexist in an LR(0) goto-graph. Then, for all pairs $r, r' \in V'$ such that $\ell(r) \equiv \ell(r')$, it must be $r \equiv r'$, that is, the path to r' and the path r shall lead to the same vertex $r_S \equiv r' \equiv r$, which is in fact an alternate definition of split vertex, when $V \subseteq V'$.

Conclusion. We showed that every edge in $\Delta\bar{E}$ is removed from E' and that any vertex in V'_S is transformed to a corresponding vertex in V_S, therefore we can conclude that SHRINK$(\Gamma_{\mathscr{G}}, Q)$ computes $\Gamma_{\mathscr{G}'}$.

It is interesting to notice the different approach taken by [21], where the SHRINK procedure is an exact inverse of GROW. The drawback is that, in this case, the procedure is likely to decimate non-ill states; in fact, as [21] suggests, it is necessary to call the equivalent of our GROW procedure afterwards, in order to restore states that may have been erroneously deleted.

Our procedure takes instead a conservative approach, by keeping most states and only then merging duplicates. Given the low incidence (in our experience and in [21]) of the splitting phenomenon, we are overall convinced that our approach is an improvement.

4.3 DEXTER and LALR lookahead sets

As we briefly mentioned in Sect. 1.1, DEXTER is a LALR(1) parser generator. The LALR technique extends the LR(0) sets of items with *lookahead* sets to solve reduce-reduce conflicts. The description of the parsing technique and the algorithms to compute these sets has been extensively discussed in several works, such as [1,5,14].

We decided not to incrementally update LALR lookahead sets: as we deemed that the performing entire computation would become rather expensive. In our prototype, we defined an UPDATELOOKAHEAD procedure that recomputes the lookahead sets on demand. The idea is that this procedure should be called every once in a while, after a batch of subsequent GROW or SHRINK applications. We will omit any implementation detail, since we adopted the well-known algorithm reported in [1]. Moreover, when the grammar does not generate any reduce-reduce conflict, the system can operate using an alternate lightweight internal algorithm that avoids the lookahead computation. We omit here an extensive description of this algorithm, since it is only a slightly modified LR(0) parsing routine that imposes a default choice in case of a shift-reduce conflict.

The version of the algorithms that we implemented *does not* deal with conflict resolution. Therefore, in our prototype, ambiguous grammars still lead to a nondeterministic LALR(1) parser. In this case, our implementation raises an error to indicate the type of conflict, which is then to be resolved by the user (possibly, by refactoring the grammar). In our experience with implementing languages using the Neverlang framework, of which DEXTER is now a core part, we did not run into many issues involving grammar conflicts. This brought us to the conclusion that DEXTER is enough for our purposes, at least for the time being. However, it has been showed that pairing context-aware scanning [36] with LALR(1) addresses many of the shortcomings of the LALR(1) family, with respect to composition issues. Since this would be a very simple extension to our prototype, we plan to look into this solution in future iterations. Moreover, because the GROW and SHRINK algorithms apply to the LR(0) goto-graph, it would be interesting to experiment with GLR parsers as well.

5 Conclusions and future work

Even though intuition suggests that there must be a particular relation between the LR(0) graph of a given grammar and the graph of a correspondent growing (or shrinking) grammar, we could not find in literature the proof that this relation must always exist. In this paper we presented these theoretical proofs (Sect. 3), and we showed how it is possible to obtain the (equivalent of a) graph of a growing or shrinking grammar by applying an algorithm to the graph of the initial grammar. The GROW procedure and the SHRINK procedure described in Sect. 4 do not depend on anything but the graph structure and the rule that is being added or removed. These procedures have been successfully implemented in a prototype called DEXTER that is now integrated in the Neverlang framework for modular language development. Our framework stresses the compositional aspect of developing a domain-specific language by conceiving any language as the result of the composition of several *slices*, each of which groups together related syntactical and semantic features. The plan is turning Neverlang into a development system that exploits modularity to reduce the need for recompilations of a language toolset to a minimum, possibly enabling runtime extensibility with new syntactic and semantic features. The DEXTER library, the proofs and the algorithms that we presented here have been all developed with these requirements in mind.

Performance-wise, early testing of our Java prototype has shown promising results. On a test machine, a Windows 7 ×64 PC with a Core 2 Duo P8400, 2.26 GHz per core, 4 GB RAM, and JDK 6, adding or removing one rule takes 1/100 s on average. Parse time varies in function of the grammar, but, for our intents, we found it quite reasonable: we implemented simple grammars such as LISP and JSON and stress-tested DEXTER with big inputs (around 130K characters). DEXTER outperformed Scala's parser combinators and PetitParser's grammars, resulting from two to about four times quicker. In general, we still expected a traditional parser generator to win over DEXTER, but we saw that the gap is not as big as one would expect. For instance, in the case of JSON, our prototype is only two times slower than ANTLR. We also implemented a fairly bigger parser, the C89 grammar (220 productions): the initialization phase took less than 2 s, 1.5 on average; 100 lines of C were parsed in 55 ms on average. We plan to test DEXTER further and publish a complete summary of our results in a follow-up paper.

Acknowledgments This work has been partially supported by the MIUR project CINA: Compositionality, Interaction, Negotiation, Autonomicity for the future ICT society.

References

1. Aho, A.V., Sethi, R., Ullman, J.D.: Compilers: Principles, Techniques, and Tools. Addison Wesley, Reading, Massachusetts (1986)
2. Aycock, J.: The design and implementation of SPARK, a toolkit for implementing domain-specific languages. J. Comput. Inf. Technol. **10**(1), 55–66 (2004). special Issue on Domain Specific Languages
3. Aycock, J., Horspool, R.N.: Directly-executable Earley parsing. In: Wilhelm, R., (ed.) Proceedings of the 10th International Conference on Compiler Construction (CC'01), LNCS, vol. 2027, pp. 229–243. Springer, Genova, Italy (2001)
4. Bentley, J.: Programming pearls: little languages. Commun. ACM **29**(8), 711–721 (1986)
5. Bermudez, M.E., Logothetis, G.: Simple computation of LALR(1) lookahead sets. Inf. Process. Lett. **31**(5), 233–238 (1989)
6. Brabrand, C., Schwartzbach, M.I.: The metafront system: safe and extensible parsing and transformation. Sci. Comput. Program. **68**, 2–20 (2007)
7. Bravenboer, M., Visser, E.: Parse table composition: separate compilation and binary extensibility of grammars. In: Software Language Engineering, LNCS, vol. 5452, pp. 74–94. Springer (2009)
8. Cardelli, L., Matthes, F., Abadi, M.: Extensible Syntax with Lexical Scoping. Technical Report SRC-RR-121, DEC Systems Research Center, Palo Alto, CA, USA (1994)
9. Cazzola, W.: Domain-Specific Languages in Few Steps: The Neverlang Approach. In: Gschwind, T,, De Paoli, F., Gruhn, V., Book, M. (eds.) Proceedings of the 11th International Conference on Software Composition (SC'12), Lecture Notes in Computer Science, vol. 7306, pp. 162–177. Springer, Prague, Czech Republic (2012)
10. Cazzola, W., Speziale, I.: Sectional domain specific languages. In: Proceedings of the 4th Domain Specific Aspect-Oriented Languages (DSAL'09), pp. 11–14. ACM, Charlottesville, Virginia, USA (2009)
11. Cazzola, W., Vacchi, E.: DEXTER and Neverlang: a union towards dynamicity. In: Jul, E., Rogers, I., Zendra, O. (eds.) Proceedings of the 7th Workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages, Programs and Systems (ICOOOLPS'12). ACM, Beijing, China (2012)
12. Cazzola, W., Vacchi, E.: Neverlang 2: componentised language development for the JVM. In: Binder, W., Bodden, E., Löwe, W. (eds.) Proceedings of the 12th International Conference on Software Composition (SC'13), Lecture Notes in Computer Science, vol. 8088, pp. 17–32. Springer, Budapest, Hungary (2013)
13. Cleenewerck, T.: Modularizing Language Constructs: A Reflective Approach. PhD thesis, Vrije Universiteit Brussel, Brussel, Belgium (2007)
14. DeRemer, F., Pennello, T.J.: Efficient computation of LALR(1) look-ahead sets. ACM Trans. Program. Lang. Syst. **4**(4), 615–649 (1982)
15. Earley, J.: An efficient context-free parsing algorithm. Commun. ACM **13**(2), 94–102 (1970)
16. Ford, B.: Parsing expression grammars: a recognition-based syntactic foundation. In: Jones, N.D., Leroy, X. (eds.) Proceedings of the 31st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'04), pp. 111–122. ACM, Venice, Italy (2004)

17. Ghosh, D.: DSL for the uninitiated. Commun. ACM **54**(7), 44–50 (2011)
18. Grimm, R.: Practical Packrat Parsing. Technical Report TR2004-854, New York University, New York, NY, USA (2004)
19. Grune, D., Jacobs, C.J.H.: Parsing techniques. In: Monographs in Computer Science, 2nd edn. Springer (2008)
20. Heering, J., Klint, P., Rekers, J.: Incremental generation of parsers. IEEE Trans. Softw. Eng. **16**(12), 1344–1351 (1990)
21. Horspool, R.N.: Incremental generation of LR parsers. J. Comput. Lang. **15**(4), 205–223 (1990)
22. Hutton, G.: High-order functions for parsing. J. Funct. Program. **2**(3), 323–343 (1992)
23. Knuth, D.E.: On the translation of languages from left to right. Inf. Control **8**(6), 607–639 (1965)
24. Lehman, M.M., Fernández-Ramil, J.C., Kahen, G.: A Paradigm for the Behavioural Modelling of Software Processes using System Dynamics. Technical Report 2001/8, Imperial College, Department of Computing, London, United Kingdom (2001)
25. Mens, T., Wermelinger, M.: Separation of concerns for software evolution. J. Maint. Evolut. **14**(5), 311–315 (2002)
26. Moors, A., Piessens, F., Odersky, M.: Parser Combinators in Scala. CW Report 491, Katholieke Universiteit Leuven, Leuven, Belgium (2008)
27. Onzon, E.: `dypgen` User's Manual . Available at http://dypgen.free.fr/dypgen-doc.pdf (2012)
28. Parr, T., Fisher, K.: LL(*): the foundation of the ANTLR parser generator. In: Padua, D. (ed.) Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'11), pp. 425–436. ACM, San José, California (2011)
29. Renggli, L., Ducasse, S., Gîrba, T., Nierstrasz, O.: Practical dynamic grammars for dynamic languages. In: Proceedings of the 4th Workshop on Dynamic Languages and Applications (DYLA'10), Málaga, Spain (2010)
30. Schwerdfeger, A.C., van Wyk, E.R.: Verifiable parse table composition for deterministic parsing. In: van den Brand, M.G.J., Gasevic, D., Gray, J.G. (eds.) Proceedings of the 2nd International Conference on Software Language Engineering (SLE'09), LNCS, vol. 5969, pp. 184–203. Springer, Dublin, Ireland (2009)
31. Stansifer, P., Wand, M.: Parsing reflective grammars. In: Brabrand, C., van Wyk, E. (eds.) Proceedings of the 11th Workshop on Language Descriptions, Tools and Applications (LDTA'11), pp. 10:1–10:7. ACM, Saarbrucken, Germany (2011)
32. Tomita, M.: Efficient Parsing for Natural Language: A Fast Algorithm for Practical Systems. Kluwer Academic Publishers, Berlin (1985)
33. Vacchi, E., Olivares, D.M., Shaqiri, A., Cazzola, W.: Neverlang 2: a framework for modular language implementation. In: Proceedings of the 13th International Conference on Modularity (Modularity'14), pp. 23–26. ACM, Lugano, Switzerland (2014)
34. Ward, M.P.: Language oriented programming. Softw. Concept Tools **15**(4), 147–161 (1994)
35. Warth, A.: Experimenting with Programming Languages. PhD thesis, University of California at Los Angeles, Los Angeles, CA, USA (2009)
36. van Wyk, E., Schwerdfeger, A.: Context-aware scanning for parsing extensible languages. In: Consel, C., Lawall, J.L. (eds.) Proceedings of the 6th International Conference on Generative Programming and Component Engineering (GPCE'07), pp. 63–72. ACM, Salzburg, Austria (2007)