Graph Transformations

An Introduction
to the
Categorical Approach

Hans J. Schneider
Chapter 3

Derivability in Categories

The study of formal languages is a central theme of theoretical computer science since the very beginning and has been stimulated by the development of programming languages. The definition of ALGOL 60 has demonstrated that the Backus-Naur-Form (BNF) is reasonably adequate for describing the context-free syntax of programming languages [2, 3]. It has been a solid basis to investigate syntax-analysis strategies and to identify subclasses that allow automatically constructing efficient parsers. At the same time, N. Chomsky has looked for characterizing the structure of natural language sentences according to formal mathematical rules [9]. (Of course, if such a set of rules is given, a creative speaker or writer is able to find a sentence that is considered syntactically correct, but is not covered by the rules.)

In both cases, the point is that we need a finite set of rules to characterize an infinite set of strings over a given alphabet. Chomsky’s approach is based on Semi-Thue-Systems providing a mechanism how to derive new strings from given ones. A Semi-Thue-System is a finite set of pairs of strings \((u, v)\), called productions. A production can be applied to any string \(x = wuz\) containing \(u\) as a substring; the application yields a derived string \(y = wuz\), constructed by substituting \(v\) for \(u\). A formal language is defined by the set of all strings that can be generated by a finite sequence of derivation steps that start at a distinguished symbol (start symbol, axiom) and that contain only symbols of a special subset of the alphabet (terminal symbols). Although the definition allows arbitrary strings on the left-hand side of the productions, efficient syntax-analysis algorithms require context-free productions with only one symbol on the left-hand side. This restriction leads to the BNF mentioned above.

The Chomsky grammar is a special case of replacement systems, a well-known concept in various areas of mathematics. E.g., in formal logic, a proof is a sequence of steps replacing formulae one by another according to well-defined transformation rules. In the introduction, we have seen some examples of replacement systems in computer science and its applications. Especially, there are examples not based on strings. Simplifying algebraic terms can still be seen both as a string transformation, but also as a tree transformation. Removing common subexpression, however, leads to graphs that are no longer trees. The same holds true for the database example and the approach to describe concurrent processes, e.g., by actor systems. In formal language theory, we usually define a set of terminal objects that do not allow further derivation
steps. Some examples given in the introduction, illustrate that it may make sense to apply further transformation steps even to a syntactically correct state in order to proceed from one admissible state to another.

Observing various applications where graphs are used to illustrate the underlying concepts has led us to formalize graph transformations [39]. A closer look to that paper shows that the basic definitions do not make use of the special fact that we are interested in graphs. Therefore, we can give the definition in an even more general setting. In the following, we assume categories that have pushouts and we define how to derive objects from objects in such a category.

After defining the notion of derivability, we interpret it in various categories. We proceed step by step considering sets, graphs, labeled graphs, hypergraphs, and graphs with structured alphabet. We draw attention to special effects that do not occur in formal language theory. E.g., we consider the case that the graph to be derived includes the nodes of the left-hand object of the production, but not all of its edges. Another point is that the embedding is not injective.

### 3.1 Definition of Derivation Steps

Definition of derivability in Semi-Thue-Systems essentially makes use of the fact that strings can be concatenated. Looking for an analogous definition deriving graphs from graphs, we must generalize the concatenation. Our solution to this problem is replacing it both on the left-hand side and on the right-hand side by a pushout construction (Definition 2.4.8). Although we are mainly interested in graphs, our definitions of a production and of derivability do not explicitly refer to graphs.

**Definition 3.1.1 (Production):**

A $\mathcal{K}$-production $p = (p^l, p^r)$ is a pair of $\mathcal{K}$-morphisms with common domain $\text{dom}(p^l) = \text{dom}(p^r)$. A $\mathcal{K}$-transformation system $\mathcal{G}$ is a finite set of $\mathcal{K}$-productions.

If $\mathcal{K}$ is the category $\text{Graph}$, we have a graph production and a graph transformation system. Writing $p = (B^l \xleftarrow{p^l} I \xrightarrow{p^r} B^r)$, we can make explicit the objects involved in the production. We call $B^l$ the left-hand object of the production and $B^r$ its right-hand object, whereas the terms left-hand side and right-hand side refer to $p^l$ and $p^r$, respectively. The interface object $I$ is used in generalizing concatenating:

**Definition 3.1.2 (Derivability):**

An object $G^r$ is said to be derivable from an object $G^l$ by the production $p = (B^l \xleftarrow{p^l} I \xrightarrow{p^r} B^r)$, written as $G^l \xrightarrow{p} G^r$ if there is a context object $C$ together with a morphism $g : I \rightarrow C$, called the embedding, such that $G^l$ and $G^r$ are the pushout objects in the following diagram:
We call $G^r$ directly derivable from an object $G^l$ in $G$, written as $G^l \xrightarrow{g} G^r$, if there is a production $p$ in $G$ such that $G^l \xrightarrow{p} G^r$ holds, and finally, we call $G^r$ derivable from $G$, written as $G \xrightarrow{g} G^r$, if there is a sequence $G_0, G_1, \ldots, G_n$ of objects such that $G = G_0 \land (\forall 1 \leq i \leq n)(G_{i-1} \xrightarrow{g} G_i) \land G_n = G^r$. $G^l \xrightarrow{g} G^r$ is a derivation step, and $G \xrightarrow{*} G^r$ is a derivation sequence.

The upper part of the diagram is given by the production. The lower part describes the derivation step. Since the pushout construction is unambiguous, we can construct $G^l$ and $G^r$ if we know the production and the morphism $g$, whereby we also know the context object. But the usual situation is that we know the production and the object $G^l$ together with the handle $g^l$ when starting a derivation step. In this case, we have to find an object $C$ and a morphism $g$ such that the given object $G^l$ is the pushout object of $(p^l, g)$. We discuss how to solve this problem in the next chapter. Then, we shall see that this may be impossible or that the solution may be ambiguous.

In most cases, we deal with only one transformation system $G$. Then, it is not necessary to mention it explicitly: If the transformation system is obvious, we write $\xrightarrow{}$ instead of $\xrightarrow{g}$.

Our definition of derivability is symmetrical. This means that if $G^r$ is derivable from $G^l$ with production $(p^l, p^r)$, then $G^l$ can be derived from $G^r$ by the inverse production $(p^r, p^l)$. This allows us to describe syntax analysis steps by the same mechanism.

We now illustrate direct derivability considering the simplest case, i.e., the category $\text{Set}$. We start with an example in which all the mappings are injective. We assume that both sides of the production $(p^l, p^r)$ as well as the embedding are injections. Then, the other mappings are injective by Corollary 2.5.5.

**Example 3.1.3:**

Consider the following mappings $p^l, p^r$, and $g$:

\[
\begin{array}{c c c c c c c}
1^l, 2^l, 3^l & \xrightarrow{p^l} & 1, 2 & \xrightarrow{p^r} & 1^r, 2^r, 6^r, 7^r \\
\downarrow{g} & & \downarrow{g} & & \\
1, 2, 4, 5 & & & & \\
\end{array}
\]

We have numbered the elements of the sets in such a way that the mappings are intuitive: $p^l(i) = i^l$, $g(i) = \bar{i}$ and $p^r(i) = \bar{i}$. We can construct both sides of the double pushout using the canonical construction of pushouts (Theorem 2.5.6).

\[
\begin{array}{c c c c c c c}
1^l, 2^l, 3^l & \xrightarrow{p^l} & 1, 2 & \xrightarrow{p^r} & 1^r, 2^r, 6^r, 7^r \\
\downarrow{g^l} & & \downarrow{g} & & \downarrow{g^r} \\
\bar{1}^l, \bar{2}^l, \bar{3}^l, \bar{4}^l, \bar{5}^l & \xrightarrow{\bar{p}^l} & \bar{1}, \bar{2}, \bar{4}, \bar{5} & \xrightarrow{\bar{p}^r} & \bar{1}^r, \bar{2}^r, \bar{4}^r, \bar{5}^r, 6^r, 7^r \\
\end{array}
\]

On the left-hand side, we have to consider the disjoint union of $B^l = \{1^l, 2^l, 3^l\}$ and $C = \{1, 2, 4, 5\}$. Then, we must identify the elements that have a common pre-image in $I = \{1, 2\}$, i.e., $p^l(1) = 1^l$ with $g(1) = \bar{1}$ and $p^l(2) = 2^l$ with
g(2) = 2. For short, we denote the equivalence class \([1^l, 1]\) by \(\bar 1^l\) and \([2^l, 2]\) by \(\bar 2^l\). Thus, the mappings completing the left-hand pushout are given by \(g(i^l) = \bar i^l\) and \(\bar p(i) = \bar i^l\). The right-hand side can be found analogously.

The intuitive meaning of this derivation step is that the elements corresponding to \(1^l, 2^l, 3\) are removed from \(G^l\) and some elements \(1^r, 2^r, 6, 7\) are inserted instead such that \(1^r, 2^r\) take the places of \(1^l, 2^l\) respectively:

\[
\begin{align*}
\{1^l, 2^l, 3\} &\xrightarrow{p^l} \{1, 2\} \xrightarrow{p^r} \{1^r, 2^r, 6, 7\} \\
\{1^l, 2^l, 3, 4, 5\} &\xrightarrow{\bar p^l} \{\bar 1, \bar 2, 4, 5\} \xrightarrow{\bar p^r} \{1^r, 2^r, 4, 5, 6, 7\}
\end{align*}
\]

The context set \(C\) can be seen as consisting of the elements that are not affected by the derivation step and dummy elements \(\bar 1, \bar 2\) defining the correct places of \(1^l, 2^l\) and \(1^r, 2^r\), respectively. Of course, the order of elements is irrelevant in the case of sets. Considering graphs, however, we need these placeholders to ensure the integrity of the graph structure.

Before switching over to giving an example in the category \(\text{Graph}\), we have to consider how to construct the pushout diagram of two graph morphisms. Of course, it would be sufficient to study the coproduct and the coequalizer in \(\text{Graph}\) and then to apply the canonical construction of Theorem 2.5.6. But there is a direct way to find the pushout of graph morphisms based on the fact that the pushout property of a diagram in \(\text{Graph}\) implies that its subdiagrams for edges and nodes must be pushout diagrams in \(\text{Set}\) (Exercise 3.8.1): We construct the pushouts for nodes and edges, separately, and we get the graph structure nearly for free by using the universal property of pushouts. Thus, we can not only reduce the problem to the case of sets, but we also get some more insight into how to take advantage of universal constructions.

**Theorem 3.1.4** (Pushout in \(\text{Graph}\))

The pushout diagram of two graph morphisms \((f, g)\) can be constructed for nodes and edges, separately. This construction yields the graph structure unambiguously.

**Proof:**

The situation we start from is given by the left-hand diagram. Since \(p = (p_E, p_V)\) and \(g = (g_E, g_V)\) are graph morphisms, the subdiagrams are commutative. We have
depicted only the functions $s_I$, $s_B$, and $s_C$, defining the source nodes of the edges in $I$, $B$, and $C$, respectively. Of course, we have to draw the same picture for the target node functions $t_I$, $t_B$, and $t_C$. The right-hand diagram shows the result of constructing two separate pushout diagrams in $\text{Set}$: $\tilde{g}_E \cdot p_E = \bar{p}_E \cdot g_E$ is the pushout diagram for the edge morphisms and $\tilde{g}_V \cdot p_V = \bar{p}_V \cdot g_V$ the one for the node morphisms. This construction yields the set of nodes and the set of edges of the graph $G$, but not yet the graph structure. We have to find a function $s_G$ assigning a source node to each edge of $G$ and analogously a target node function $t_G$. Furthermore, these functions must make $(\bar{p}_E, \tilde{p}_V)$ and $(\bar{g}_E, \tilde{g}_V)$ graph morphisms. In the right-hand diagram, the outer paths describe a commutative diagram: $(\bar{p}_V \cdot s_C) \cdot g_E = \tilde{p}_V \cdot g_V \cdot s_I = \tilde{g}_V \cdot p_V \cdot s_I = (\tilde{g}_V \cdot s_B) \cdot p_E$. Because of the universal property of the pushout diagram $\tilde{g}_E \cdot p_E = \bar{p}_E \cdot g_E$, there is exactly one morphism $s_G : G_E \to G_V$ such that $\tilde{g}_V \cdot s_B = s_G \cdot \bar{g}_E$ and $\tilde{p}_V \cdot s_C = s_G \cdot \bar{p}_E$. The existence of $s_G$ (together with the analogous construction of $t_G$) makes $G$ a graph, and the equalities ensure that $\bar{p} = (\bar{p}_E, \tilde{p}_V)$ and $\bar{g} = (\bar{g}_E, \tilde{g}_V)$ are graph morphisms:

To complete the proof, we have to show that the universal property also holds in $\text{Graph}$, i.e., if we have graph morphisms $\bar{p}'$ and $\bar{g}'$ with $\bar{p}' \cdot g = \bar{g}' \cdot p$, there must be a unique $u$ making the triangles in the left-hand diagram commutative:

Again, we decompose the diagram into two diagrams considering nodes and edges, separately. (In the picture, we have omitted the arrows $s_I$, $s_B$, $s_C$, and $s_G$ for reasons of perspicuity.) This results in the existence of two set morphisms $u_E$ and $u_V$ making the triangles in the right-hand diagram commutative. Since equality of graph morphisms is defined componentwise, $u = (u_E, u_V)$ also makes the triangles in the left-hand diagram commutative. We still have to show that $u$ is a graph morphism, i.e. $s_G' \cdot u_E = u_V \cdot s_G$. We consider the commutative diagram: $(\tilde{p}_V' \cdot s_C) \cdot g_E = \tilde{p}_V' \cdot g_V' \cdot s_I' = (\tilde{g}_V' \cdot s_B) \cdot p_E$. From this, we get that there is an unambiguous $h$ with $h \cdot \bar{p}_E = \bar{p}_V' \cdot s_C$ and $h \cdot \bar{g}_E = \bar{g}_V' \cdot s_B$, since $\bar{g}_E \cdot p_E = \bar{p}_E \cdot g_E$ is a pushout diagram.
Both morphisms \( s_G \cdot u_E \) and \( u_V \cdot s'_C \) satisfy this condition if we substitute them for \( h \), e.g., \( s_G \cdot u_E \cdot \bar{p}_E = s_G \cdot \bar{p}'_E = \bar{p}_V \cdot s_C \). Therefore, they must be equal. \( \square \)

Analogously, you can prove that \( \mathcal{H}\text{graph} \) has pushouts (Exercise 3.8.2).

We now illustrate this construction by two examples using injective graph morphisms. In the first example, the interface graph consists of nodes only, i.e., it is discrete, in the second, we add an edge. These examples already make clear why we have called the interface object \( \text{gluing graph} \) in our early papers.

**Example 3.1.5:**

In this example, the interface graph \( I \) consists of two nodes 1 and 2. The morphisms \( p \) and \( g \) are defined by \( p_V(i) = i' \) and \( g_V(i) = \bar{i} \). Since there are no edges to be mapped, \( p_E \) and \( g_E \) are empty. By the canonical construction of the pushout diagram, we have to put together the sets of nodes of the graphs \( B \) and \( C \) and to identify the images \( p(i) = i' \) with \( g(i) = \bar{i} \) for all \( i \), giving the nodes \( \bar{i}' \) in graph \( G \). Analogously, we construct the disjoint union of the sets of edges of \( B \) and \( C \). Here, no edges are to be identified, since \( I \) does not contain any edges. Thus, the construction results in two edges from \( \bar{1}' \) to \( \bar{2}' \): one of them is the image of the edge \( 1' \to 2' \), the other the image of \( \bar{1} \to 2 \). To be precise, we had to define which of the parallel edges is the image of the edge in \( B \) and which is the image of the edge in \( C \). The graph structure results from Theorem 3.1.4, e.g., the image of the edge \( \bar{5} \to \bar{1} \) in \( C \) must connect \( \bar{5} \) to \( \bar{1}' \) in \( G \).

In most applications, it is sufficient to consider discrete interface graphs. Nevertheless, there are exceptions.

**Example 3.1.6:**

We slightly modify the previous example by adding an edge between the nodes of \( I \), and we get the diagram on top of the next page. \( p \) and \( g \) are graph morphisms; therefore, we get \( p_E(1 \to 2) = 1' \to 2' \) and \( g(1 \to 2) = \bar{1} \to \bar{2} \). Since these edges in \( B \) and \( C \) are images of the same edge in \( I \), we have to identify them in constructing \( G \). The result is only one edge between \( \bar{1}' \) and \( \bar{2}' \).

\(^{1}\)In simple examples, this notation avoids the introduction of edge identifiers.
Now, we are ready to consider a first example of a derivation step in $\text{Graph}$.

**Example 3.1.7:**

We use the morphism $p$ considered in Example 3.1.5 as the right-hand side of a production and we add a suitable left-hand side:

Analogously to the examples in $\text{Set}$, the morphisms are given by $p'_L(i) = i$ and $p'_R(i) = i'$, whereas $p'_E$ and $p''_E$ are empty. Please note that node $3'$ and node $3''$ have nothing to do with one another, because they are not images of interface nodes, i.e., of the nodes of the interface graph. Using this production together with the embedding $g$ of Example 3.1.5, we get the following derivation step:

The details of constructing $G^l$ are left to the reader.

---

2There are good arguments to denote them differently; on the other hand, the readers should become familiar with the fact that the identifiers are not significant.
We can interpret the production of this example intuitively, saying that it removes an open triangle from a graph and inserts a closed rectangle instead.

The reader should note another point of interest in this example: The nodes $1^l$ and $2^l$ and the edge from $2^l$ to $1^l$ are part of the subgraph matching the left-hand object of the production, but the edge from $1^l$ to $2^l$ is not. In graph theory, a subgraph is usually defined by a subset of nodes and all edges between these nodes. Here, we have not defined applicability of a production by saying that the left-hand object must be a subgraph of the graph to be derived. Instead, there must be a graph morphism embedding the left-hand object into the given graph. The double-pushout approach allows us to transfer edges of $G^l$ such as $1^l \rightarrow 2^l$ to the derived graph as part of the context graph.

### 3.2 Examples in Set and Graph with Noninjective Mappings

In the previous section, we restricted discussion to injective morphisms. This case is of special importance in many application areas, but sometimes, we need noninjective mappings. First, we consider the case of a noninjective embedding, then we allow noninjective left-hand sides.

**Example 3.2.1:**

We use the production $(p^l, p^r)$ of Example 3.1.3, but let the embedding $g$ put together the images of 1 and 2: $g(1) = g(2) = [1, 2]$.

\[
\begin{array}{ccc}
\{1^l, 2^l, 3^l\} & \xrightarrow{p^l} & \{1, 2\} & \xrightarrow{p^r} & \{1^r, 2^r, 6^r, 7^r\} \\
g^l & \downarrow & g & & g^r \\
\{[1^l, 2^l], 3^l, 4^l, 5^l\} & \xleftarrow{\bar{p}^l} & \{[1, 2], 4, 5\} & \xrightarrow{\bar{p}^r} & \{[1^r, 2^r], 4^r, 5^r, 6^r, 7^r\}
\end{array}
\]

Hence the canonical construction identifies $p^l(1) = 1^l$ with $g(1) = [1, 2]$ and $p^l(2) = 2^l$ with the same element $g(2) = [1, 2]$. Therefore, we get $g^l(1^l) = g^l(2^l) = [1^l, 2^l]$ and $\bar{p}^l([1, 2]) = [1^l, 2^l]$.

Again, the intuitive meaning of the production is that the elements corresponding to $1^l, 2^l, 3^l$ are removed from $G^l$ and new elements $1^r, 2^r, 6^r, 7^r$ are inserted instead, where $1^r, 2^r$ take the places of $1^l, 2^l$, respectively. But now, the embedding of the interface set into the context set allows us to apply a production to a set $G^l$ even if we must use some elements of $G^l$ more than once. This effect is of no importance as long as we consider pure sets, but it gains in importance when we switch over to studying graphs and other structured data.

**Example 3.2.2:**

We consider a graph production similar to Example 3.1.7, but we make node 3 an interface node, too, and we choose an embedding $g$ that puts together the nodes
2 and 3. The left-hand object of the production consists of three nodes 1\textsuperscript{l}, 2\textsuperscript{l}, and 3\textsuperscript{l}, and two edges connecting 1\textsuperscript{l} with 2\textsuperscript{l} and 3\textsuperscript{l}, respectively. Since we have chosen an embedding with \( g(2) = g(3) \), we get a \( G\textsuperscript{l} \), in which we have to put together the images of 2\textsuperscript{l} and 3\textsuperscript{l}, too. In consequence of this, two edges connect \( \bar{1} \) with the image node \([\bar{2}, \bar{3}]\). Please note that these edges are not merged into one edge: the pushout construction puts together only edges that have a common pre-image in \( I \), but there are no edges in \( I \).

This example illustrates that a derivation step can be constructed applying a production in such a way that some nodes of the left-hand object coincide in the graph to be derived, and this coincidence is transferred to the result of the derivation step. This is quite different from the definition of derivability in formal language theory, where the left-hand object of the production is a substring of the string to be derived. At the end of the previous section, we have already drawn attention to the fact that the categorical approach to graph transformations allows replacing not only subgraphs isomorphic to the left-hand object of the production, but also other parts that can be the image of \( B\textsuperscript{l} \) under a suitably chosen graph morphism. In that example, we have had edges that are not part of \( B\textsuperscript{l} \), here, we have an example merging nodes with one another. (Of course, it is also possible to merge edges.) Nevertheless, restricting application of productions to injective embeddings \( g \) is an interesting aspect from a theoretical point of view. There are, however, applications taking advantage of noninjective embeddings. (Example 3.7.4 presents such an application.)

An even more complicated situation arises from allowing noninjective mappings in the production. Again, we start with considering the situation in \( \textbf{Set} \), although the effect does not become clear until we study it in \( \textbf{Graph} \).

**Example 3.2.3:**

We consider an example with a noninjective left-hand side \( p\textsuperscript{l} \) of the production: \( p\textsuperscript{l}(1) = p\textsuperscript{l}(2) = [1\textsuperscript{l}, 2\textsuperscript{l}] \), but we use the same embedding as in Example 3.1.3. We have to identify \( p\textsuperscript{l}(1) = [1\textsuperscript{l}, 2\textsuperscript{l}] \) with \( g(1) = \bar{1} \) and \( p\textsuperscript{l}(2) = [1\textsuperscript{l}, 2\textsuperscript{l}] \) with \( g(2) = 2 \).

It is not surprising that we get the same \( C\textsuperscript{l} \) as in Example 3.2.1. Mapping the

\[ \text{More precisely, we should write } [1\textsuperscript{l}, 2\textsuperscript{l}, [\bar{1}, \bar{2}]] . \]
Interface elements is given by $g_1([1^l, 2^l]) = \bar{1}^l, \bar{2}^l$ and $\bar{p}_1(\bar{1}) = \bar{p}_1(\bar{2}) = [\bar{1}^l, \bar{2}^l]$. On the right-hand side, there are no new aspects since these mappings are identical to the mappings in Example 3.1.3.

Intuitively, we remove the elements corresponding to $[1^l, 2^l], 3^l$ from the set $G^l$ and insert elements corresponding to $1^r, 2^r, 6^r, 7^r$. At first glance, the effect of the derivation step seems to be the same as in Example 3.1.3, but there is a difference: $\bar{1}^r$ and $\bar{2}^r$ together play the role of $[\bar{1}^l, \bar{2}^l]$. This becomes significant when dealing with graphs.

If two nodes in the derived graph take the role of one original node, we may ask for which of the nodes is the source (or target) node of an edge that is connected with the original node.

**Example 3.2.4:**

We consider a graph production splitting a node $[1^l, 2^l]$ on the left-hand side into two nodes $1^r$ and $2^r$ on the right-hand side, and we choose an embedding $g_1$ as follows:

$I$ and $\bar{2}$ are mapped onto the same node $[\bar{1}^l, \bar{2}^l]$ in the left-hand graph $G^l$. Therefore, the images of all the edges that are connected with them, must be connected with $[\bar{1}^l, \bar{2}^l]$, and we get two edges between $\bar{5}^l$ and $[\bar{1}^l, \bar{2}^l]$. On the right-hand side, however, the graph $G^r_1$ reflects the situation we have in the context graph since $p^r$ is injective, and therefore, $\bar{p}_1^r$ is injective, too.

In this example, the production replaces an edge by a triangle where two nodes of the triangle share the role of the edge’s source. In $G^l$, however, node $[\bar{1}^l, \bar{2}^l]$ is both the source of the edge $[\bar{1}^l, \bar{2}^l] \rightarrow \bar{5}^l$ and the target of the edge $\bar{5}^l \rightarrow [\bar{1}^l, \bar{2}^l]$. These edges are part of the context graph $C_1$. The embedding we have chosen transfers the role of the source of the first edge to $\bar{1}$ and the role of the target of the second to $\bar{2}$, and therefore, to $1^r$ and $2^r$, respectively. But, this is not the only solution:
Example 3.2.4 (Cont’d):

We may choose an embedding $g_2 : I \to C_2$ causing both roles to be taken by $\bar{2}$. Again, we define $g_2(1) = \bar{1}$ and $g_2(2) = \bar{2}$, but in $C_2$, both edges are connected to node $\bar{2}$. This situation is transferred to the derived graph since we have injective mappings on the right-hand side of the derivation step. Therefore, the derived graph $G^r_2$ is different from $G^r_1$, although $G^l$ is the same in both derivation steps.

We have to keep in mind that if we have a noninjective production and a well-defined embedding of its left-hand side into a given graph, the right-hand side of the derivation step needs not be defined unambiguously! The result of a derivation step depends on the choice of the context object. In the next chapter, we shall discuss how to construct the context graphs effectively.

The nondeterminism we have observed in Example 3.2.4 is based on how to connect edges to nodes. If we consider the diagrams for both nodes and edges separately, we get identical diagrams. This is not in contradiction with Theorem 3.1.4, since the pushout construction in Graph starts with a special context graph, whereas we have chosen different context graphs.

But, there is another effect causing nondeterminism. Again, we first consider it in Set:

Example 3.2.3 (Cont’d):

We use the same production as before, but choose another embedding $g_2$ such that we get the same $G^l$:

$$
\begin{align*}
\{[1^l,2^l],3^l\} & \xrightarrow{p^l} \{1,2\} & \xrightarrow{p^r} \{1^r,2^r,6^r,7^r\} \\
g^l & \downarrow \quad & g_2 & \downarrow \quad & g^r_2 \\
\{[\bar{1}^l,\bar{2}^l],3^l,4^l,5^l\} & \xleftarrow{\bar{p}^l_2} \{[1,2],4,5\} & \xleftarrow{\bar{p}^r_2} \{[\bar{1}^r,\bar{2}^r],4^r,5^r,6^r,7^r\}
\end{align*}
$$

In the first part of the example, we have achieved $[\bar{1}^l,\bar{2}^l] = g^l(p^l(1)) = g^l(p^l(2)) = \bar{p}_1^l(g_1(1)) = \bar{p}_1^l(g_1(2))$ by a property of the production, namely by $p^l(1) = p^l(2)$. The resulting pushout object, however, is not changed if we additionally define $g_2(1) = g_2(2)$! On the right-hand side, however, we get a $G^r_2$ different from $G^r$. 


Example 3.2.4 (Cont’d):

Now, we apply this effect to the graph example:

In the previous derivation steps, we have put together the interface nodes via $p'$, whereas $g_2$ has kept them separate in the context graph $C_2$. We do not change the pushout object $G'$ if we additionally put the interface nodes involved together in the context graph by a non-injective embedding $g_3$. The pushout object on the right-hand side, however, is changed since the interface nodes are not put together by $p'$. Especially, the edge connecting $1^r$ to $2^r$ on the left-hand side of the production, becomes a loop at node $[\bar{1}^r, \bar{2}^r]$.

This example shows us that three different graphs $G_1'$, $G_2'$, and $G_3'$ are derivable from $G'$ using the same production and the same handle $g'$! It is left to the reader to find the remaining solutions.

3.3 Labeled Graphs

As we have seen in the introduction, the most interesting cases are labeled graphs and labeled hypergraphs. The pushout diagrams in $\mathcal{L}graph$ and in $\mathcal{LH}graph$ can easily be constructed:

**Theorem 3.3.1 (Pushout in $\mathcal{L}graph$):**

The pushout diagram in $\mathcal{L}graph$ is uniquely defined by the construction in $\mathcal{Graph}$.

**Proof:**

We construct the pushout diagram of the underlying graph morphisms in $\mathcal{Graph}$ as shown in Theorem 3.1.4. Furthermore, we know the labeling functions of the given graphs: $l_{VI} : I_V \to L_V$, $l_{VB} : B_V \to L_V$, and $l_{VC} : C_V \to L_V$, and the analogous functions on edges. Since $f$ and $g$ are morphisms in $\mathcal{L}graph$, we have $l_{VB} \cdot f_V = l_{VI} = l_{VC} \cdot g_V$. By the universal property of the pushout diagram $\tilde{g}_V \cdot f_V = \tilde{f}_V \cdot g_V$, we get
an unambiguous labeling $l_{VG} : G_V \to L_V$ making commutative the small triangles:

\[ l_{VB} = l_{VG} \cdot \bar{g} \quad \text{and} \quad l_{VC} = l_{VG} \cdot \bar{f}. \]

(1)

Since the same result holds true for edges, the construction not only yields a labeled graph $G$, but also shows that $\bar{g}$ and $\bar{f}$ are $L$-graph morphisms, i.e., the diagram we have constructed as a pushout diagram in $\mathcal{Graph}$ also is a commutative diagram in $\mathcal{Lgraph}$. Is it a pushout diagram, too? To show this, we assume another commutative diagram in $\mathcal{Lgraph}$ based on $f$ and $g$, e.g., $\bar{g}' \cdot f = \bar{f}' \cdot g$. We know from the construction in Theorem 3.1.4, that there is a unique graph morphism $u = (u_E, u_V) : G \to G'$ such that $\bar{g}'$ and $\bar{f}'$ can be factorized in the usual way:

\[ \bar{f}'_V = u_V \cdot \bar{f}_V \quad \text{and} \quad \bar{g}'_V = u_V \cdot \bar{g}_V. \]

(2)

Again, we can restrict discussion to nodes, since the same arguments hold true for edges. $\bar{g}'$ and $\bar{f}'$ are $L$-graph morphisms, i.e.,

\[ l_{VC} = l_{VG'} \cdot \bar{f}'_V \quad \text{and} \quad l_{VB} = l_{VG'} \cdot \bar{g}'_V, \]

and therefore, $l_{VG'} \cdot \bar{f}'_V \cdot g_V = l_{VG'} \cdot \bar{g}'_V \cdot f_V$. This assures the existence of a unique $h$ with $l_{VG'} \cdot \bar{f}'_V = h \cdot \bar{f}_V$ and $l_{VG'} \cdot \bar{g}'_V = h \cdot \bar{g}_V$. On the one hand, $l_{VG}$ is such an $h$. This follows from (1) and (3). On the other hand, $l_{VG'} \cdot u_V$ also is such an $h$ because of (2). This means that we have $h = l_{VG} = l_{VG'} \cdot u_V$, and that $u$ is a morphism in $\mathcal{Lgraph}$. It is unambiguous, since another $u'$ with the same properties would lead to an ambiguity in $\mathcal{Graph}$. \hfill \Box

This theorem provides us with a convenient way to construct derivation steps in $\mathcal{Lgraph}$: We start with a derivation step in $\mathcal{Graph}$. Then, we label the resulting graph in such a way that the graph morphisms are $L$-graph morphisms, and we get a derivation step in $\mathcal{Lgraph}$ immediately.

**Example 3.3.2:**

Let us consider Example 3.2.2 augmented by suitable labels. The embedding $g$ with $g(2) = g(3)$ requires that the nodes 2 and 3 are labeled identically: $l_{VK}(2) = l_{VK}(3) = b$. As a consequence of this, we have the same situation in the left-hand and in the right-hand objects of the production: $l_{VB'}(2) = l_{VB'}(3) = l_{VB'}(1) = l_{VB'}(3) = b$. (More precisely, we had to distinguish between $1^l$ and $1^r$, etc., but the nodes can not be confused with one another.) Intuitively, we construct $G'$ as
the disjoint union of $B^l$ and $C$ putting together the nodes that have a common pre-image in $I$. We can extend this intuitive description: If an element, i.e., a node or edge, of $G^l$ has a pre-image in $B^l$, we take the label from there; if it has a pre-image in $C$, this pre-image defines the label. If both cases occur, there is no ambiguity, since there must be a common pre-image in $I$, and the elements in $B^l$ and $C$, therefore, bear the same label.

Now, we are able to formally treat the example from computational linguistics, which we have intuitively discussed in the introduction (Example 1.2.4).

**Example 3.3.3:**

We represent a sentence by an ordered sequence of edges, and we consider the production describing that a verb and a noun phrase constitute a verb phrase. We illustrate the application of this production by considering the phrase “verb - determiner - noun”, where the noun phrase consists of the determiner and the noun:

This detail, indicated in the figure by node 3 and the edges $1 \rightarrow 3$ and $3 \rightarrow 2$, is part of the context graph. Therefore, it is transferred from the left-hand side to the right-hand side without change. Of course, a noun phrase need not consist of a determiner and a noun. An alternative is a proper noun. In this case, we get the following derivation step:
This example again illustrates the effect we have already mentioned after Example 3.1.7: The image of the left-hand object of the production is the graph $g^l[B^l] \subseteq G^l$, but this is not a subgraph in the strict sense, since the edge labeled with $pn$ connects two nodes of $G^l$ that are nodes of $g^l[B^l]$, although the edge is not. This becomes clearer if we remember another observation: In most cases, we can remove the edges from the interface graph without changing the possible derivation steps:

**Example 3.3.3 (Cont’d):**

We remove the edges from the interface graph in the production of our last example. We want to apply the production to the same left-hand graph $G^l$. To get this graph as the pushout object, we have to remove these edges from the context graph $C$, too. (Otherwise, these edges would occur twice in $G^l$.)

The reader should compare the structure of the proof of Theorem 3.3.1 (Pushout in $L_{graph}$) with that of Theorem 3.1.4 (Pushout in $G_{raph}$). In both cases, we construct the pushout diagram in a category by (1) constructing the pushout diagram in an underlying category and (2) “embedding” the result into the category under consideration. Then, we have to (3) show that the morphisms we get in the underlying category together with the embedding are morphisms in the new category, too, and finally (4) the new diagram must satisfy the universal property of pushouts. Steps 3 and 4 do not construct new things; we have to show only properties of elements already constructed in steps 1 and 2. These proofs always take advantage of the pushout
property of the diagrams constructed in the underlying category. In the proof of Theorem 3.1.4, embedding means connecting edges with their source and target nodes; in Theorem 3.3.1, it means defining the labels. In Chapter . . . , we shall give a general construction freeing us from the necessity to consider the details of the proof in each case. Therefore, we can omit to explicitly prove the following theorem:

**Theorem 3.3.4 (Pushout in \( \mathcal{LHgraph} \))**:

The pushout diagram in \( \mathcal{LHgraph} \) is uniquely defined by the construction in \( \mathcal{Hgraph} \).

An important application of labeled hypergraphs is describing computations. Each hyperedge represents an operation, and the nodes it visits define the parameters and the result(s). We consider the specification of the data type *queue* as an example.

**Example 3.3.5 (Specification of data type *queue*):**

The data type *queue* usually provides the operations *init*, *appd*, *remove* and the value returning functions *empty*, *next*:

- \( \text{init} : \text{queue} \)
- \( \text{appd} : \text{queue} \times \text{elmt} \rightarrow \text{queue} \)
- \( \text{remove} : \text{queue} \rightarrow \text{queue} \)
- \( \text{empty} : \text{queue} \rightarrow \text{bool} \)
- \( \text{next} : \text{queue} \rightarrow \text{elmt} \)

Then, the semantics of the first-in-first-out queue (FIFO queue) is given by the following equations:

\[
\begin{align*}
(1) \quad \text{empty}(\text{init}) &= \text{true} \\
(2) \quad \text{empty}(\text{appd}(q,e)) &= \text{false} \\
(3) \quad \text{next}(\text{appd}(\text{init},e)) &= e \\
(4) \quad \text{next}(\text{appd}(\text{appd}(q,e),e')) &= \text{next}(\text{appd}(q,e)) \\
(5) \quad \text{remove}(\text{appd}(\text{init},e)) &= \text{init} \\
(6) \quad \text{remove}(\text{appd}(\text{appd}(q,e),e')) &= \text{appd}(\text{remove}(\text{appd}(q,e)),e')
\end{align*}
\]

We can interpret these equations as productions in the category of labeled hypergraphs. We illustrate this translation by looking at Equation (6). The left-hand side and the right-hand side of the equation are translated into labeled hypergraphs as already explained in Example 2.2.10. The nodes are labeled with the types of the operands they represent. The node numbers are added as exponents, as long as they are necessary to indicate the morphisms, i.e., we mention only the numbers of the interface nodes. The variables \( q, e, \) and \( e' \) used in the equation to denote the parameter positions do not occur explicitly in the production. Their part is taken by the interface nodes 3, 4, and 2, respectively. Interface node 1 denotes the position of the result of the computation. The interface graph consists of these four nodes.

The example shows that the translation of data type specifications into graph productions is rather straightforward. Before giving an example of how computations correspond to graph derivations, we want to discuss two special situations: parameters that can be eliminated from the computation, and parameters that coincide on the right-hand side.
Example 3.3.5 (Cont’d):
In Equation (5) of the example, we have a variable $e$ on the left-hand side which is no longer used on the right-hand side. We can translate the left-hand side as before. The parameter $e$ is represented by the interface node 2 and the result by interface node 1. Of course, these two nodes must be found in the interface graph. But what happens on the right-hand side? In the equation, the right-hand side consists only of the operation $init$, which has no parameter and a result of type $queue$. Therefore, we get a hyperedge visiting only one node, namely the interface node 1 representing the result. In order to get a correct morphism, we need a node $p_5^r(2)$ on the right-hand side, but this node is no longer needed if
we consider the computation. It is an isolated node, nicely illustrating that this parameter is a matter of garbage collection.

Equation (3) results in a noninjective mapping $p_3^r$ on the right-hand side of the production. Again, we can translate the left-hand side straightforwardly; the result is represented by interface node 1 and the parameter $e$ by interface node 2. On the right-hand side, however, the parameter and the result coincide:

![Diagram](image)

Equation (3) of Example 3.3.5 as a production

We have no problem applying a production in a derivation step if its right-hand side is noninjective. The pushout construction is unambiguous even in this case. A problem arises only if we want to apply this production in the inverse direction (syntax analysis), since the noninjective part then moves to the left-hand side and the problems discussed in Section 3.2 arise.

**Example 3.3.5 (Cont’d):**

We complete the specification example considering a short derivation. We start with the term $\text{next(\text{remove(appd(appd(init, y), x)))}}$ which can be reduced to $x$. This reduction is described by the derivation given on the opposite page. Equations (6), (5), and (3) are applied in this derivation. To simplify the figure (on the next page), we have omitted the explicit identification of all the nodes and hyperedges. This leads to a problem: In the first graph, the topmost node represents the “result” of the expression. In the last graph, however, we have two “topmost” nodes. If we had explicit identifiers, we would know which of these nodes represents the result. Of course, it is possible to follow the node via the context graphs of the derivation steps. In the figure, we choose a trick making this process intuitive: We introduce a special hyperedge, which is labeled with $\text{res}$ and which visits only the node representing the result. This hyperedge is forwarded from graph to graph by the derivation steps as part of the context graphs, and it becomes clear which of the nodes in the last graph represents the result.
Example of a derivation using the productions of Example 3.3.5

3.4 Graph Grammars

After introducing the notion of graph derivations and studying some examples how to transform graphs by applying graph productions, we can generalize Chomsky's approach to formalizing the notion of a grammar. The main point is to apply productions until some kind of normal form is reached. For this, Chomsky has distinguished terminal symbols from nonterminal ones. Productions are applied until the string does no longer contain nonterminal symbols. We can easily transfer this idea to graph grammars:

**Definition 3.4.1** (Graph grammar):
A graph grammar is given by a quadruple \( G = (L, T, P, S) \) with \( P \) being a finite set of graph productions in a category of labeled (hyper-)graphs using \( L \) as the labeling alphabet. \( T \subseteq L \) is called the terminal alphabet, \( S \) is the starting graph.

Please note that \( L = (L_E, L_V) \) consists of labeling alphabets for (hyper-)edges and nodes, which may be equal, but in general, are not. \( T \subseteq L \) is to be interpreted componentwise, i.e., \( T_E \subseteq L_E \) and \( T_V \subseteq L_V \). Another difference between our definition and Chomsky's is that we allow a starting graph instead of a starting symbol. But this is only a matter of taste: We can add a production the right-hand object of which is \( S \) and whose left-hand object consists of only one node labeled with a special starting symbol. Alternatively, we can use a graph with exactly one edge labeled with the starting symbol. Our solution to allow an arbitrary starting graph avoids the problem of deciding whether we start with a nonterminally labeled edge or with a nonterminally labeled node.

**Definition 3.4.2** (Graph language):
If \( G \) is a graph grammar, then the set
\[
L(G) := \{ G \mid S \xrightarrow[G]{} G \land l_{E_G}[E_G] \subseteq T_E \land l_{V_G}[V_G] \subseteq T_V \}
\]
is called the language of \( G \).

The language of a grammar includes all the graphs that are derivable from the starting graph, but do not contain any nonterminally labeled node or edge.

Example 3.3.5 can be seen as a graph grammar with empty, next, and remove being the nonterminal edge labels. A derivation stops after removing these labels from the graph. The graph language defined by this grammar is the set of normal forms consisting only of edges labeled with init, appd, the special label res, and labels originating from the definition of the elements. Defining abstract data types, one can distinguish between constructors and derived function symbols. The derived functions are defined in terms of the constructors. The terminal symbols of the graph grammar correspond to the constructors in the data type terminology. The productions correspond to the equations of the axiomatic data type specification. They are used to remove the nonterminal symbols, i.e., to replace the derived function symbols by their definition in terms of the constructors.

Even if we restrict discussion to finite graphs, the derivability relation \( \rightarrow \) is undecidable in the general case. This immediately follows from its undecidability in the
case of unrestricted Chomsky grammars since we can represent strings by graphs as we have seen in Example 3.3.3. If derivability were decidable for graph grammars, we would have a decision procedure for Chomsky grammars, too. Direct derivability, however, is decidable in the case of finite graphs as it is in the case of strings [75]. Similar to the Chomsky hierarchy, we can define restrictions that ensure decidability. An interesting restriction corresponds to Chomsky's contextfree grammars: The left-hand object of each production consists of exactly one hyperedge and the nodes it visits. These graph grammars, the so-called hyperedge replacement grammars, have properties very similar to the contextfree grammars. A detailed study has been given by Habel [46]. Here we resume the example already introduced as Example 1.2.1 in the introduction.

Example 3.4.3 (Well-structured programs):

Well-structured programs are characterized by the fact that each statement and each group of statements has exactly one entry point and one exit point, and they can be constructed by stepwise refinement, i.e., a statement can be replaced by a sequence of statements, an alternative, or a loop. We describe each of these refinement steps by one production:

\[
stmt \rightarrow s_1 \quad \text{with} \quad t_1 \rightarrow stmt
\]

In these productions, the interface graphs are trivial; in each production, it consists of two isolated nodes 1 and 2. Therefore, we can save space using the Backus-Naur notation [2, 66]: We combine the productions with the same left-hand object separating the right-hand objects by a vertical bar and putting the ::= sign between the left-hand object and the right-hand objects. stmt (statement) and cond (condition) are the nonterminal hyperedge labels. We have omitted the productions replacing these labels by terminal ones such as assignment; they do not change the structure of the resulting graphs. Furthermore, we have omitted the node labels, since they are not significant. (We can label all the nodes identically.)

Comparing this example with Example 3.3.5, the reader will remark that we have changed the direction of arrows. Here, they point from source nodes to hyperedges and from hyperedges to target nodes indicating the control flow, whereas in the other example, we have used the data flow direction. Purists do not like such variations, but we prefer illustrations meeting the reader’s intuition. The reader should note that the formal definition is not affected by how we illustrate it, and he or she should learn to distinguish between definition and illustration.
The first part of the example refines only the statements. The graph production approach, however, allows us to refine the conditions in all the same way.

**Example 3.4.3 (Cont’d):**

Hyperedges labeled with `cond` visit three nodes. Therefore, the interface graph consists of three isolated nodes:

```
\[ stmt \quad ::= \]
```

The first production refines a condition by stating that the condition needs an internal computation. The second is an example of a condition composed of two simpler conditions, e.g., a disjunction, and the last production describes a condition including a loop, e.g., a condition consisting of a searching procedure.

So far, we could apply a production \( p \) to a given graph \( G_l \) if there is a morphism mapping the left-hand object of \( p \) into \( G_l \) (although we have not yet discussed how to find the context graph). Our next example shows that this unrestricted use of productions may sometimes lead to undesired effects.

**Example 3.4.4 (Tree grammar):**

We consider the following production, in which all the nodes and edges are labeled identically:

```
\[ stmt \]
```

We apply this production to a simple tree. Then, we can distinguish two different applications. In Figure (a), we map node \( 1^l \) to a leaf: \( g^l(1^l) = 6 \). Another possibility is mapping node \( 1^l \) to the root: \( g^l(1^l) = 4 \) as shown in Figure (b).

The details of these derivation steps are left to the reader. It is easy to see that repeated application of this production allows us to construct all trees with an even number of edges, if we start from a single node. If, however, we restrict application in such a way that \( g^l(1^l) \) is a leaf, then this production constructs all binary trees.
Definition 3.4.5 (Graph grammar with application conditions):
A graph grammar with application conditions \( G_A \) is defined by a graph grammar \( G = (L, T, P, S) \) together with a mapping \( A \) associating a decidable subset \( A(p) \) of morphisms with each production \( p \in P \): \( A(p) \subseteq \bigcup_{G} \text{Mor}_K(B^l, G) \). The derivability relation \( \xRightarrow{G_A} \) is defined as in Definition 3.1.1 with the restriction: \( g^l \in A(p) \).

In Example 3.4.4, we may restrict \( A(p) \) to the morphisms that map \( 1^l \) to a leaf. Then \( L_{G_A} \) is the set of all binary trees. Another application condition that is often used in various applications is restricting \( g^l \) to monomorphisms.

Application conditions can also be used to describe some “programmed” use of productions, e.g., a production \( p_1 \) must not be used if production \( p_2 \) is applicable. This technique is similar to the programmed grammars in formal language theory [41, 78]. We can apply this idea to describing the scope rules of block structured programming languages.

Example 3.4.6 (Block structure):
We abstract from the whole program and consider only the block structure, declaring identifiers, and the use of identifiers. We omit the types and the context in which identifiers are used in order to simplify the graphs. The figure on the next page shows a small graph representing a program segment with three nested blocks. Nesting is indicated by in-edges. Furthermore, there are some nodes representing declarations and other nodes representing places where the declared identifiers are used. The positions of these nodes within the hierarchy of blocks are also given by in-edges. Finally, a def-edge is associated with each use-node indicating the declaration its identifier refers to. According to the usual scope rules, this is the innermost declaration of this identifier.

We need three productions to describe inserting a new use-node and constructing the correct def-edge. Production \( p_1 \) inserts a new occurrence of identifier \( X \). An in-edge defines the block into which this occurrence is inserted. Furthermore, we insert an edge labeled with the nonterminal label \( \delta \). This nonterminal label ensures that the graph resulting from applying this production is not an element.
Graph representing a small program segment as described in Example 3.4.6

\[
\begin{align*}
1^l : block & ::= & 1^r : block & \quad (p_1) \\
& & & \quad \downarrow \delta \\
& & & \quad \downarrow in \\
& & & \quad 2^r : use X \\
1^l : block & \quad in & 2^l : dcl X & ::= & 1^r : block & \quad in & 2^r : dcl X & \quad (p_2) \\
& \quad \delta & & & \quad \downarrow \delta \\
& & & \quad 3^l : use X & & & \quad 3^r : use X \\
1^l : block & \quad in & 2^l : block & ::= & 1^r : block & \quad in & 2^r : block & \quad (p_3) \\
& & & \quad \delta & & & \quad \delta \\
& & & \quad 3^l : use X & & & \quad 3^r : use X \\
\end{align*}
\]

Productions used in Example 3.4.6
of the language. We have to remove this label. Production $p_2$ replaces the $\delta$-edge pointing to a block node by a def-edge referring to the declaration if a suitable declaration is in this block. If there is no such declaration, we can not apply $p_2$, but production $p_3$ that shifts the $\delta$-edge to the next block in the hierarchy. Then, we can again try to apply production $p_2$. To get a correct def-edge, production $p_3$ must not be applied if production $p_2$ is applicable, since we have to look for the innermost declaration. Therefore, the application condition of $p_3$ is

$$A(p_3) = \{ B^1_3 \to G \mid B^1_3 \to G \text{ injective } \land \neg (\exists B^1_2 \to G) \}.$$ 

whereas $A(p_1)$ and $A(p_2)$ do not impose any restrictions.

The productions of this example describe inserting a use-occurrence of a special identifier $X$. We may, however, interpret these productions as a schema representing an infinite set of productions the elements of which we can get by replacing $X$ with a concrete identifier. Another interpretation is to modify the definition of morphisms between labeled graphs. This will be done next.

### 3.5 Graph Grammars vs. Chomsky Grammars

In Section 3.4, we have introduced the notion of a graph grammar. A graph grammar is a formal system to define a set of graphs. This definition is motivated by the study of the Chomsky grammar that is a formal system to define a set of strings:

**Definition 3.5.1** (Chomsky Grammar [9]):

A Chomsky grammar (phrase structure grammar) is a quadruple $G = (T, N, P, S)$ where $T$ and $N$ are disjoint finite sets (alphabets), $S$ is a distinguished element of $N$, and $P$ is a finite subset of $L^* \times L^*$ with $L = T \cup N$.

The elements of $T$ and $N$ are called terminal symbols and nonterminal symbols, respectively. $S$ is the start symbol or axiom, and $P$ is the set of productions. Usually, the productions are written in the Backus-Naur-style [2] as $u ::= v$ instead of $(u, v)$.

**Example 3.5.2:**

We consider a small example with the terminal symbols $T = \{a, b, c\}$ and the nonterminal symbols $N = \{S, A, B, C\}$. The productions are as follows:

- $S ::= aSBC \mid aBC$
- $CB ::= BC$
- $aB ::= ab$
- $bB ::= bb$
- $bC ::= bc$
- $cC ::= cc$

Of course, $S$ is the start symbol.

**Definition 3.5.3** (Derivability):

A string $x$ is directly derivable to $y$ in $G$, written as $x \xrightarrow{G} y$, if and only if there is a production $(u, v) \in P$ such that $(\exists w, u, v, z \in L^*)(x = wuz \land y = wzv \land p = (u, v))$

A string $x$ is derivable to $y$ in $G$, written as $x \xrightarrow{G} y$, if and only if there is an integer $n \geq 0$ and a sequence of strings $w_0, w_1, \ldots, w_n \in L^*$ such that

$$x = w_0 \land (\forall 1 \leq i \leq n)(w_{i-1} \xrightarrow{G} w_i) \land y = w_n$$
\( n = 0 \) implies that each string is derivable to itself in any grammar. If there is no ambiguity, we write \( x \Rightarrow y \) instead of \( x \rightarrow_G y \).

**Definition 3.5.4** (Chomsky Language):

Each Chomsky grammar defines a set of strings that can be derived from the start symbol and that does not contain any nonterminal symbol:

\[
\mathcal{L}(G) := \{ w \mid w \in T^* \land S \Rightarrow_G w \}
\]

A set of strings that can be defined by a Chomsky grammar is called a **Chomsky language**.

**Example 3.5.2** (Cont’d):

The above Chomsky grammar defines the set \( \mathcal{L}(G) = \{ a^n b^n c^n \mid n > 0 \} \). The proof is left to the reader.

Unfortunately, the relation \( \ast \Rightarrow_G \) is undecidable, i.e., there is no algorithm that can decide whether \( x \Rightarrow_G y \) holds or does not hold if we consider an arbitrary Chomsky grammar \( G \) and two arbitrary strings \( x \) and \( y \). Therefore, it is necessary to restrict the form of the productions to make derivability decidable:

**Definition 3.5.5** (Chomsky Hierarchy):

We consider only two cases of the hierarchy introduced by Chomsky:

(a) A Chomsky grammar is called **context-sensitive** if the productions are of the form \( wAz := wuz \) with \( A \in N, w, z \in L^*, \) and \( u \in L^+ \).

(b) A Chomsky grammar is called **context-free** if the productions are of the form \( A := u \) with \( A \in N \) and \( u \in L^*. \)

(c) A set of strings is called context-sensitive (context-free) if there exists a context-sensitive (context-free) Chomsky grammar that generates it.

For context-sensitive Chomsky grammars, the relation \( \ast \Rightarrow_G \) becomes decidable. A derivation step \( x \Rightarrow y \) can not make the string shorter. Checking \( x \Rightarrow_G z \), we can start with \( x \) and generate all the strings that are derivable from it and that are not longer than \( z \). We can decide whether \( z \) is contained in this set, since it is finite. This algorithm, however, needs exponential time and is therefore not well-suited for syntax analysis. In the case of context-free grammars, however, there exist efficient algorithms.

The proof of decidability does not use the special form of context-sensitive productions, but only the fact that the derivation steps do not shorten the strings. We can use this property as the definition of context-sensitive grammars by saying that the right-hand side of a production must not be shorter than its left-hand side. The equivalence of these definitions can be shown by replacing a production that does not satisfy the first definition of context-sensitivity with a sequence of productions satisfying it by introducing additional nonterminal symbols.

Thus, Example 3.5.2 is a context-sensitive grammar and \( \{ a^n b^n c^n \mid n > 0 \} \) is a context-sensitive language. In the theory of formal languages, it is a well-known fact that this

\( L^+ \) does not contain the empty string, i.e., \( L^+ = L^* \setminus \{ \varepsilon \} \).
set cannot be generated by any context-free grammar. The proof is based on the so-called **pumping lemma**.

We now turn our attention to the relationship between Chomsky grammars and graph grammars. We can simulate the derivation steps of a Chomsky grammar by derivation steps of a graph grammar. As in Example 3.3.3, we represent a string by an ordered sequence of edges:

**Definition 3.5.6:**

If \( x = x_1x_2 \ldots x_n \) is a string where the \( x_i \) are symbols of an alphabet, we define the graph

\[
\gamma(x) := (\{e_1, e_2, \ldots, e_n\}, \{0, 1, \ldots, n\}, s, t)
\]

with \( s(e_i) := i - 1 \) and \( t(e_i) := i \) for \( i = 1, 2, \ldots, n \).

Then, we can translate Chomsky grammars into graph grammars as follows:

**Definition 3.5.7:**

Given a Chomsky grammar \( G' = (T', N', P', S') \), we associate a graph grammar \( G = \gamma(G') = (L, T, P, S) \) with \( G' \) by defining:

(a) \( L = (T' \cup N', \mathbb{N}_0) \)

(b) \( T = (T', \mathbb{N}_0) \)

(c) \( S = 0 \)

(d) If and only if there is a production \( u_1u_2 \ldots u_k := v_1v_2 \ldots v_l \) in \( P' \), then \( P \) contains a graph production of the form

\[
B^l | 0 \overset{u_1}{\rightarrow} 1 \overset{u_2}{\rightarrow} 2 \cdots k' \overset{u_k}{\rightarrow} k |
\]

\[
\overset{p^l(I)}{0} \overset{l}{\rightarrow} 1 \overset{p^r(B^r)}{0} \overset{v_1}{\rightarrow} 1 \overset{v_2}{\rightarrow} 2 \cdots l' \overset{v_l}{\rightarrow} l
\]

with

\[
p^l(0) = 0 \quad p^r(0) = 0 \quad p^l(1) = k \quad p^r(1) = l
\]

Please note that \( l = 0 \) induces \( p^r(0) = p^r(1) = 0 \) and \( B^r_E = \emptyset \).

This construction is straightforward such that the following lemma becomes obvious:

**Lemma 3.5.8:**

If we have a Chomsky grammar \( G' \) and the graph grammar \( \gamma(G') \) associated with it, the following assertions hold:

(a) \( x \xrightarrow{G'} y \) implies \( \gamma(x) \xrightarrow{\gamma(G')} \gamma(y) \).

(b) If \( G \xrightarrow{\gamma(G')} H \) holds and if there exists a string \( x \) with \( \gamma(x) = G \), then there exists a string \( y \) such that \( \gamma(y) = H \) and \( x \xrightarrow{G'} y \).

---

\( ^5 \mathbb{N}_0 \) is the set of natural numbers including 0.

\( ^6 \) For typographical reasons, we write \( k' \) and \( l' \) instead of \( k - 1 \) and \( l - 1 \), respectively.
The assumption that there exists a suitable $x$ ensures that the derivation step is based on an injective embedding, i.e., the two nodes of the interface graph are not put together in the context graph. Otherwise, the image of the left-hand side in the host graph would be a cycle, since the first and the last node coincide. On the right-hand side of the production, however, $p_r(0)$ and $p_r(1)$ may coincide, namely in the case of an empty right-hand side. But in this case, a cycle does not occur because of $B_r^E = \emptyset$.

Obviously, the graph grammars associated with Chomsky grammars have the same properties as the Chomsky grammars. Especially, a context-free graph grammar has exactly one edge on the left-hand side of each production. Then, we get the same theorems as in the theory of formal languages. If we replace graphs with hypergraphs and edges with hyperedges, however, we get interesting new results.

In the following, we use the notations introduced in the definitions 2.2.4 and 2.2.9.

**Definition 3.5.9** (Context-free hypergraph production):

A context-free hypergraph production of order $(k_1, k_2)$ satisfies:

(a) $B^I_E = \{e^I\}$ with $|s'(e^I)| = k_1$ and $|t'(e^I)| = k_2$
(b) All the nodes of $B^V_V$ occur either in $s'(e^I)$ or in $t'(e^I)$.
(c) $I_E = \emptyset$, $I_V = \{1, 2, \ldots, k_1 + k_2\}$
(d) $p^I_V$ is bijective.

This means that the left-hand side of a context-free hypergraph production consists of one hyperedge and the nodes it visits. The interface graph is a discrete graph with $k_1 + k_2$ nodes, and the bijectivity of $p^I_V$ ensures that the left-hand side also has $k_1 + k_2$ nodes, i.e., there are no nodes visited twice by the hyperedge.

If we additionally require the right-hand morphism $p^I_V$ to be injective, we get the hyperedge replacement approach investigated by A. Habel in great detail [46]. She could prove that this concept generalizes a lot of properties well-known from the theory of context-free Chomsky grammars, e.g., the closure properties or the pumping lemma. But there is an interesting difference: The set $L(G) = \{a^n b^n c^n | n > 0\}$ can be generated by a context-free hypergraph grammar.

**Definition 3.5.10** (Context-free hypergraph grammar):

A context-free hypergraph grammar of order $(K_1, K_2)$ is a graph grammar all the productions of which are context-free hypergraph productions of order $(k_1, k_2)$ with $k_1 \leq K_1$ and $k_2 \leq K_2$.

**Lemma 3.5.11**:

The set $\{\gamma(a^n b^n c^n) | n > 0\}$ is a context-free hypergraph language of order $(2, 2)$.

Proof:

The hypergraph grammar depicted in Figure 3.5.11 defines $\{\gamma(a^n b^n c^n) | n > 0\}$. We use a BNF-like notation with a nonterminal symbol on the left-hand side representing a hyperedge labeled with this symbol, and we omit the interface graph since it is trivial. By definition, it is discrete and mapping its nodes onto the left-hand side is bijective. Therefore, it simply consists of the nodes $1, 2, \ldots, k_1 + k_2$, where $k_1$ and $k_2$ are determined by the symbol on the left-hand side. We use two nonterminal
symbols: the start symbol $S$ with $k_1 = k_2 = 0$ and the recursive symbol $A$ with $k_1 = k_2 = 2$. The terminal symbols $a$, $b$, and $c$ have one entry node and one exit node, i.e., $k_1 = k_2 = 1$.

![Diagram of hypergraph grammar](image)

Figure 3.5.11: Hypergraph grammar generating $\{a^n b^n c^n\}$

The first line of the figure contains the productions with $S$ on the left-hand side. The second production immediately constructs the graph representing the string $abc$. The first production also constructs one occurrence of each terminal symbol, but in this case, they are not connected one to another. Instead, we get something like

$$
\cdot \rightarrow a \rightarrow 1 \quad 2 \rightarrow b \rightarrow 3 \quad 4 \rightarrow c \rightarrow \cdot
$$

where the nodes 1, 2, 3, and 4 are connected by a hyperedge labeled with $A$. This hyperedge is used to generate additional $a$’s, $b$’s, and $c$’s simultaneously. Consider the last production: We insert $\rightarrow a \rightarrow b \rightarrow$ between 1 and 2 and we insert $\rightarrow c \rightarrow$ between 3 and 4. Whereas this production removes the nonterminal hyperedge and its nodes, the third production preserves it. Here, we insert

$$
\rightarrow a \rightarrow 1' \quad 2' \rightarrow b \rightarrow \quad \text{and} \quad \rightarrow c \rightarrow 3' \quad 4' \rightarrow
$$

between 1 and 2 and between 3 and 4, respectively. The new nodes are connected by $A$ such that the recursion can be continued.

On the one hand, context-free hypergraph grammars have properties very similar to the context-free Chomsky grammars, but on the other hand, they are able to describe facts that can be treated only by context-sensitive means on the level of string grammars. This makes hypergraph grammars an interesting concept in modeling applications.
3.6 Graphs With Structured Alphabet

Both $\mathcal{L}_{\text{graph}}$ and $\mathcal{L}_{\text{Hgraph}}$ are defined on the basis of label-preserving morphisms, and therefore, they do not allow graph transformations changing the labels on the nodes or on the edges. Of course, it is possible to define productions that remove a node and replace it by a new node that takes a different label. Alternately, we may interpret a production as a schema representing a (possibly infinite) set of productions, as we have done in Example 3.4.6. This works well if some (or all) labels in the production can be seen to be variables which are to be replaced by concrete values. But this is not sufficient, i.e., playing the token game on a Petri net:

Example 3.6.1:

A Petri net consists of two disjoint sets of nodes: the places and the transitions. Edges connect input places to transitions and transitions to output places. Whereas this underlying bipartite graph defines the static structure of the system, its dynamic behavior is controlled by labeling the places. A typical example is the well-known producer-consumer net:

A transition is enabled if all the places preceding the transition (input places) are marked and all the places following the transition (output places) are not marked. Then, the transition can fire; this step removes the marks from the input places and sets a mark onto each output place. In the state given by the picture above, only transition $t_3$ can fire.\(^7\) The result is a new state:

We want to model the token game by a graph transformation system. We associate a production with each transition in such a way that (a) the production is applicable if and only if its transition is enabled and (b) firing corresponds to a derivation step and vice versa. The production for transition $t_3$ must look like this:

\(^7\) $t_2$ can not fire because one of its output places is marked. $t_3$ having fired, however, either $t_2$ or $t_4$ can fire in the next step.
We can not use the category $\mathcal{L}_{\text{graph}}$ since it is impossible to label the places of the interface graph in such a way that both $p'$ and $p''$ are label preserving.$^8$

Before turning our attention to looking for a suitable category, we consider a second example giving us an idea what type of extension we need:

**Example 3.6.2:**

We describe the problem of the dining philosophers in the following way: We have three places containing the thinking philosophers, the unused forks, and the eating philosophers. The transitions correspond to becoming hungry and satisfied, respectively:

As usual, there are five philosophers $p_1, \ldots, p_5$ and five forks $f_1, \ldots, f_5$. If philosopher $p_i$ wants to eat, he needs the forks $f_i$ and $f_{(i \mod 5)+1}$. In the situation shown in the picture on the previous page, philosophers $p_2$ and $p_5$ are eating using the forks $f_2, f_3, f_5, f_1$. Fork $f_4$ is not used. When $p_2$ has satisfied his hunger, he turns to thinking and puts back the forks $f_2$ and $f_3$. A production modeling this transition must be of the following form:

For each philosopher, we need such a production and another one modeling the transition when becoming hungry.

---

$^8$Why we do not make the transition an interface node, will become clear when we discuss the dangling condition (Theorem 4.5.8).
A first approach to overcome the limitations of label-preserving graph morphisms was given by Ehrig et al. [32]. We follow the approach by Parisi-Presicce et al. [67]; these authors impose a simple structure on the set of labels to allow relabeling:

**Definition 3.6.3 (Structured alphabet):**

A **structured alphabet** $SL = (L, \sqsubseteq)$ is an alphabet $L$ together with a reflexive and transitive relation on $L$.

Please note that the structured alphabet is more general than a partial order since we do not require antisymmetry: $x \sqsubseteq y \land y \sqsubseteq x \Rightarrow x = y$. Nevertheless, the subset relation is one of the most used examples of such a structured alphabet; it is used, e.g., to complete Example 3.6.2. If however, we allow a variable to be replaced by a constant or by another variable, the situation arises that two variables may be substituted one for another, but they are not considered equal.

**Definition 3.6.4 (SL-graph):**

An **SL-graph** is a labeled graph $G = (E, V, s, t, l_E, l_V)$ with $l_E : E \to SL_E$ and $l_V : V \to SL_V$ where $SL_E$ and $SL_V$ are structured alphabets. SL-hypergraphs are defined analogously.

Of course, there is no problem with considering graphs the nodes of which are labeled with a structured alphabet and the edges are labeled in the usual way or vice versa, since equality trivially satisfies the assumptions of Definition 3.6.3.

**Definition 3.6.5 (SL-graph morphism):**

An **SL-graph morphism** $f : G \to H$ is a graph morphism $f = (f_E : E_G \to E_H, f_V : V_G \to V_H)$ which additionally satisfies: $(\forall v \in V_G)(l_{V_G}(v) \sqsubseteq l_{V_H}(f_V(v)))$ and $(\forall e \in E_G)(l_{E_G}(e) \sqsubseteq l_{E_H}(f_E(e)))$. The analogous conditions hold for SL-hypergraph morphisms\(^9\).

**Example 3.6.2 (Cont’d):**

Since we have used sets to label the places of the Petri net describing the problem of the dining philosophers, it is quite natural to use the set inclusion as the relation $\sqsubseteq$, i.e., mapping a place $v$ of $G$ onto a place $v'$ of $H$ requires that the label of $v$ be a subset of the label of $v'$. Therefore, we can make $p^l$ and $p^r$ SL-graph morphisms replacing the question marks by empty sets.

\[^9\]Parasi-Presicce uses the converse relation $\sqsupset$. Therefore, he needs the greatest lower bound in constructing the pushout in $SL\text{-graph}$. 

Of course, the labels of the transitions can be interpreted as sets with one element. To ensure the correct mapping of places, we have introduced edge labels: $pt$ (philosopher wants to think), $ph$ (philosopher becomes hungry), $pe$ (philosopher
wants to eat), \( ps \) (philosopher becomes satisfied), \( t \) (philosopher takes forks), \( b \) (philosopher puts back forks). Then, the original net looks like this:

![Diagram of a net](image)

On \( L_E \), the \( \sqsubseteq \)-relation is given by the identity.

It should be mentioned that we get a more elegant solution to this example considering the bipartite graph as a hypergraph: Transitions are hyperedges the source nodes of which are the input places and the target nodes of which are the output places (Exercise 3.8.16).

**Theorem 3.6.6 (\( S\mathcal{L}_{\text{Graph}} \)):**

The set of \( SL \)-graphs and the set of \( SL \)-graph morphisms together with component-wise composition constitute the category \( S\mathcal{L}_{\text{Graph}} \). Analogously, we get the category \( S\mathcal{L}_{\text{Hypergraph}} \) of \( SL \)-hypergraphs and \( SL \)-hypergraph morphisms.

**Proof:**

It is sufficient to show that composition satisfies the labeling condition of Definition 3.6.5. Let be \( f : G \rightarrow G' \) and \( g : G' \rightarrow G'' \) as in the diagram of the proof of Theorem 2.1.13. For each node \( v \in G_V \), we get \( l_{V_{G'}(v)} \sqsubseteq l_{V_{G'}(f_V(v))} \sqsubseteq l_{V_{G''}(g_V(f_V(v)))} \), and therefore, \( l_{V_G(v)} \sqsubseteq l_{V_{G''}(g_V(f_V(v)))} \) by transitivity. The same argument holds true for edges. \( \square \)

Next, we want to construct pushout diagrams in the category \( S\mathcal{L}_{\text{Graph}} \). Unfortunately, we can not take over the proof of Theorem 3.3.1:
In the label preserving case, we had \( l_{VC} \cdot g_V = l_{VB} \cdot f_V \) because both paths are equal to \( l_{VI} \), and from this, we have got an unambiguous \( l_{VC} \) by taking advantage of the pushout property. Now, we have only \( l_{VC} \cdot g_V \subseteq l_{VI} \) and \( l_{VB} \cdot f_V \subseteq l_{VI} \). Therefore, we must look for another solution. We go back to the canonical construction composing a pushout of a coproduct and a coequalizer (Theorem 2.5.6). This means that we have to study how to get the coproduct and the coequalizer in \( S\mathcal{L}_{\text{graph}} \).

**Lemma 3.6.7 (Coproduct in \( S\mathcal{L}_{\text{graph}} \)):**

\( S\mathcal{L}_{\text{graph}} \) and \( S\mathcal{L}_{\text{Hgraph}} \) have coproducts.

**Proof:**

We construct the coproducts \((f_E, g_E, G_E)\) and \((f_V, g_V, G_V)\) of the edge sets \((G'_E, G''_E)\) and of the node sets \((G'_V, G''_V)\), separately; then, we connect both diagrams using the set morphisms \( s_{G'} \) and \( s_{G''} \) defining the source nodes in \( G' \) and \( G'' \):

\[
\begin{array}{ccc}
G'_V & \xrightarrow{f_V} & G_V \\
\downarrow{s_{G'}} & & \downarrow{s_{G'}} \\
G'_V & \xrightarrow{f_V} & G_V
\end{array}
\]

By the coproduct property of \( G_E \), we get a unique \( s_G \) with \( s_G \cdot f_E = f_V \cdot s_{G'} \) and \( s_G \cdot g_E = g_V \cdot s_{G''} \). Of course, the same argument holds for \( t_G, t_{G''}, t_G \); therefore, \((G_E, G_V, s_G, t_G)\) is a graph, and \( f = (f_E, f_V) \) and \( g = (g_E, G_V) \) are graph morphisms. In an analogous way, we construct the labels:

\[
\begin{array}{ccc}
G'_V & \xrightarrow{f_V} & G_V \\
\downarrow{l_{G'V}} & & \downarrow{l_{G'V}} \\
L_V & = & L_V
\end{array}
\]

Since the \( \subseteq \)-relation is reflexive, \( l_{G'V} = l_{GV} \) implies \( l_{G'V} \subseteq l_{GV} \), etc. This means that \( f \) and \( g \) are morphisms in \( S\mathcal{L}_{\text{graph}} \). To complete the proof, we have to show that the triple \((f, g, G)\) we have constructed satisfies the coproduct property in \( S\mathcal{L}_{\text{graph}} \). For this, we consider two \( SL \)-graph morphisms \( h' : G' \to H \) and \( h'' : G'' \to H \):

\[
\begin{array}{ccc}
G' & \xrightarrow{f} & G \\
\downarrow{h'} & \downarrow{h''} & \downarrow{h''} \\
& H &
\end{array}
\]

\[
\begin{array}{ccc}
G'' & \xrightarrow{f} & G \\
\downarrow{h'} & \downarrow{h''} & \downarrow{h''} \\
& H &
\end{array}
\]
We can uniquely construct the components \((h_E, h_V)\) in \(\text{Set}\), and we can show as above that \(h := (h_E, h_V)\) is a graph morphism factorizing both \(h'\) and \(h''\). Therefore, \(h\) is uniquely determined in \(\text{Graph}\), too. (If there were another way to construct such an \(h\), we would get the same result or a contradiction in \(\text{Set}\).) In the right-hand diagram, we consider labeling of nodes. We have to show that \(l_{GV}(v) \sqsubseteq l_{HV}(h_V(v))\) is satisfied for all nodes \(v \in G_V\). Since \(G_V\) is the disjoint union of \(G'_V\) and \(G''_V\), we have either a unique \(v' \in G'_V\) with \(v = f_V(v')\) or a unique \(v'' \in G''_V\) with \(v = g_V(v'')\). Without loss of generality, we assume the first case. Then, we get \(l_{GV}(v) = l_{GV}(f_V(v')) = l_{GV}(v') \sqsubseteq l_{HV}(h'_V(v')) = l_{HV}(h_V(f_V(v'))) = l_{HV}(h_V(v))\). Obviously, the same line of reasoning applies to edge labels. 

Unfortunately, coequalizers do not exist in each case. We need an additional assumption:

**Lemma 3.6.8** (Coequalizer in \(\mathcal{SL}\text{graph}\)):

If in the structured alphabet \(SL = (L, \sqsubseteq)\), the least upper bound exists, then \(\mathcal{SL}\text{graph}\) and \(\mathcal{SL}\text{Hgraph}\) have coequalizers.

The least upper bound of two labels \(\alpha\) and \(\beta\) is the label \(\gamma = \text{lub}\{\alpha, \beta\}\) such that for all \(\gamma'\) satisfying \(\alpha \sqsubseteq \gamma' \land \beta \sqsubseteq \gamma'\), we have \(\gamma \sqsubseteq \gamma'\).\(^{10}\) In the case of set inclusion, the least upper bound is the union.

Proof:

Again, we construct the diagram for nodes and edges, separately:

\[
\begin{array}{c}
G' \xrightarrow{f} G'' \xrightarrow{q} G \\
G'_E \xrightarrow{f_E} G''_E \xrightarrow{q_E} G_E \\
G'_V \xrightarrow{f_V} G''_V \xrightarrow{q_V} G_V
\end{array}
\]

To prove that \(G\) is a graph and that \(q : G'' \to G\) is a graph morphism is left to the reader as an exercise (Exercise 3.8.18). Labeling the graph \(G\) with \(l_{GV}(v) := \text{lub}\{l_{G''_V}(v'') \mid q_V(v'') = v\}\) and \(l_{GE}(e) := \text{lub}\{l_{G''_E}(e'') \mid q_E(e'') = e\}\) results in an \(\mathcal{SL}\text{-graph}\) morphism because we have \(l_{G''_V}(\bar{v}) \sqsubseteq \text{lub}\{l_{G''_V}(v'') \mid q_V(v'') = q_V(\bar{v})\} = l_{GV}(q_V(\bar{v}))\) for all \(\bar{v} \in G''_V\). The universal construction

\[
\begin{array}{c}
G' \xrightarrow{f} G'' \xrightarrow{q} G \\
G' \xrightarrow{h'} H
\end{array}
\]

is unambiguous in \(\text{Graph}\), and furthermore, we have

\(^{10}\)Please note that the least upper bound can be considered a colimit construction (Exercise 3.8.17).
\[
l_{GV}(v) = \text{lub}\{l_{G''V}(v'') \mid q_{V}(v'') = v\}
\subseteq \text{lub}\{l_{HV}(h_{V}(v'')) \mid q_{V}(v'') = v\}
= l_{HV}(h_{V}(q_{V}(v'')))
= \text{lub}\{l_{HV}(h_{V}(v))\}
= l_{HV}(h_{V}(v))
\]

These two lemmata allow us to apply the canonical construction of pushouts:

**Theorem 3.6.9 (Pushout in SL_{graph}):**

If in the structured alphabet \( SL = (L, \sqsubseteq) \), the least upper bound exists, then \( SL_{graph} \) and \( SL_{Hgraph} \) have pushouts.

We can construct the pushout in \( SL_{graph} \) stepwise. First, we take advantage of Theorem 3.1.4 to find the pushout in \( G_{graph} \). Then, we label the nodes (edges) by constructing the least upper bound of the labels of nodes (edges) that are thrown together into this node (edge). This second step uses Lemma 3.6.8.

It is very easy to see that both lemmata and the theorem do not only hold for the category of \( SL \)-graphs, but also for the category of \( SL \)-hypergraphs.

### 3.7 Examples With Structured Alphabet

We return to the example of the dining philosophers.

**Example 3.6.2 (Cont’d):**

As we have already seen, labeling the places in the interface graph \( I \) with empty sets makes \( p^l \) and \( p^r \) \( SL \)-graph morphisms. For reason of space, we do not show the full graphs \( G^l, C, \) and \( G^r \), but only the neighborhood of the transition \textit{satisfied}: stubs of arrows and some dots indicate the embedding into the remaining context. Since the underlying graph morphisms are injective, the pushout construction identifies one place of the left-hand object (or of the right-hand object) with one place of the context graph. The input place of transition \textit{satisfied} is labeled with \{\( p_2 \)\} in \( B^l \) and the corresponding place in \( C \) is labeled with \{\( p_5 \)\}.\(^{11}\) Therefore, the resulting node in \( G^l \) is labeled with the union \{\( p_2, p_5 \)\}. Since the input place in \( B^r \) is labeled with the empty set, the label of this place in \( G^r \) is still \{\( p_5 \)\}. In a similar way, the right-hand output place in \( G^r \) is labeled with \{\( p_1, p_2, p_3, p_4 \)\}, since we have \{\( p_1, p_3, p_4 \)\} in \( C \) and \{\( p_2 \)\} in \( B^r \).

Sets with set inclusion are a natural example of a structured alphabet. Now, we return to the producer-consumer example and look for a suitable labeling.

**Example 3.6.1 (Cont’d):**

There is an obvious way of labeling the places: We use \( m \) to indicate that a place is marked, and we use \( \overline{m} \) to indicate that it is not. Furthermore, we introduce edge labels. This facilitates distinguishing between different edges that point to the same transition or leave it. The figure on top of the next page shows this
version. Then, we can make $p^l$ and $p^r$ SL-graph morphisms by using a special symbol $\bot$ (called bottom) to label the places in the interface graph. This symbol represents something like an undefined label, and we can substitute $m$ as well as $\bar{m}$ for it:

\[
\begin{align*}
\bot & \sqsubseteq m & m & \sqsubseteq \top & \bot & \sqsubseteq \top \\
\bot & \sqsubseteq \bar{m} & \bar{m} & \sqsubseteq \top & \bot & \sqsubseteq \top
\end{align*}
\]

Additionally, we need the top symbol $\top$ to ensure existence of the least upper bound and therefore, the existence of the pushout construction. It may be interpreted as representing an overdefined label and must not occur in derivable graphs. With this labeling, we get the following derivation step describing tran-

\[\text{New version of the producer-consumer net}\]

\[\text{For convenience, we omit the braces in the diagram.}\]
If in this example, we label one of the input places in graph $C$ with $m$, we get the same graph $G'$, but on the right-hand side the corresponding place is labeled with $\text{lub}(m, \bar{m}) = \top$, and we get a forbidden graph.

The special symbols $\bot$ and $\top$ we have introduced in this example may be interpreted as nonterminal symbols that are not allowed to occur in valid graphs. But, they can be used in intermediate constructions.

In Example 3.4.4, we have defined a graph grammar generating trees. In order to restrict this grammar to generating only binary trees, we have used an application condition applying the production only to leaves. Since a structured alphabet allows us to relabel nodes, we can distinguish leaves from other nodes by labeling.

**Example 3.7.1 (Binary trees):**

We use two labels $A$ and $a$ that are allowed to occur in the derivable graphs:

$$
\begin{align*}
B'_{1} &\quad I_{1} &\quad B'_{1} \\
1^{r} &\quad 1 &\quad 1^{r} \\
1^{l} &\quad 1 &\quad 1^{r} \\
1^{l} &\quad A &\quad A
\end{align*}
$$

$$
\begin{align*}
B'_{2} &\quad I_{2} &\quad B'_{2} \\
1^{r} &\quad 1 &\quad 1^{r} \\
1^{l} &\quad 1 &\quad 1^{r} \\
1^{l} &\quad A &\quad A
\end{align*}
$$

$$
\begin{align*}
B'_{1} &\quad I_{1} &\quad B'_{1} \\
&\quad 1 &\quad 1 &\quad 1^{r} \\
&\quad a &\quad 1 &\quad a
\end{align*}
$$

$$
\begin{align*}
B'_{2} &\quad I_{2} &\quad B'_{2} \\
&\quad 1 &\quad 1 &\quad 1^{r} \\
&\quad a &\quad 1 &\quad a
\end{align*}
$$

$\bot \subseteq A \Leftrightarrow \bot \subseteq a$.\textsuperscript{12} Label $A$ denotes a leaf that can be expanded by production

\textsuperscript{12}Of course, we also need the top symbol.
This node then becomes a non-leaf and is labeled with $a$. Production $p_1$ can not be applied to this node once more. The second production removes the label $A$ from leaves that should not be expanded: \{a\} is the terminal alphabet of the grammar. Here is the beginning of a derivation:

$$
\text{production}
\begin{align*}
A & \Rightarrow a \\
A & \Rightarrow a \\
A & \Rightarrow a
\end{align*}
$$

Now, let us resume Example 2.1.11. In that example, we have distinguished female persons from male persons by adding edges labeled appropriately. The structured alphabet allows a solution without additional edges.

**Example 3.7.2:**

We start with the following graph depicting the mother-of relation:

```
mary1
mo1
/        /
\        \  
joan2    mo2
\        /  /
jane3
\       / /
dora4
\      /  /
david5
```

Now, we want to insert the sister-of edges by applying a suitable production.

The left-hand object of the production must ensure not only that both persons have the same mother, but also that the person at the source of the new edge is female. We ensure this by defining $w \subseteq \text{mary, joan, jane, dora}$ and $x \subseteq \text{mary, joan, jane, dora, david}$. Then, we have three ways of applying this production to the given graph. The first case is to map the nodes as follows.\text{Case 1:} $g^l(1) = 1 \quad g^l(2) = 2 \quad g^l(3) = 3$

This case results in the derivation step given above.\text{Case 1:} $g^l(1) = 1 \quad g^l(2) = 2 \quad g^l(3) = 3$

\text{Case 1:} $g^l(1) = 1 \quad g^l(2) = 2 \quad g^l(3) = 3$

For reason of space, we abbreviate the names by the first two letters.

---

\text{Case 1:} $g^l(1) = 1 \quad g^l(2) = 2 \quad g^l(3) = 3$

\text{Case 1:} $g^l(1) = 1 \quad g^l(2) = 2 \quad g^l(3) = 3$
also labeled with \( w \), but have the label \( joan \) in \( C \), and therefore, in \( G^l \), too. The definition of \( SL \)-graph morphisms does not require that all occurrences of a “variable” are replaced with the same label.

The other cases can be characterized by the following morphisms \( g^l \):

Case 2: \[
\begin{align*}
g^l(1) &= 1 \\
g^l(2) &= 3 \\
g^l(3) &= 2
\end{align*}
\]

Case 3: \[
\begin{align*}
g^l(1) &= 3 \\
g^l(2) &= 4 \\
g^l(3) &= 5
\end{align*}
\]

Whereas we can exchange the roles of node 2 and node 3 (case (1) and case (2), respectively), it is not possible to map node 2 to node 5 because we must not substitute the label \( david \) for \( w \).

In the introduction, we have presented another example that takes advantage of “variable” labels: removing common subexpressions (Example . . . ). Whereas the previous example shows us that our definition of \( SL \)-graph morphisms allows us to substitute different labels for a variable in the production, the common-subexpression example requires that two operations are identical. We look for a graph structure appropriate to model the removal of common subexpressions. Usually, expressions are depicted as trees. Since we prefer the edges to indicate the data flow, an expression leads to an “inverted” tree with edges pointing from the leaves to the root. The leaves represent the variables and constants, the root acts as the result. Taking advantage of common subexpressions, we have no longer (inverted) trees:

**Definition 3.7.3 (Term graph):**

A term graph is an acyclic graph with the following properties:

- If a node is labeled with an \( n \)-ary operation symbol, it is the target of \( n \) ingoing edges labeled with \( 1, 2, \ldots, n \).
- A node without ingoing edges is labeled with a variable or with a constant.
- There is at least one node without outgoing edges (result node, root).

In a normalized term graph, each variable or constant occurs only once.

We use term graph transformation to remove repeated calculations of subexpressions. It makes sense to take into consideration more than one expression. In this case, we get a term graph with more than one root.
Example 3.7.4 (Common subexpressions):
We consider arithmetic expressions consisting of constants, variables, and operators. If we have an expression of the form
\[ \ldots ((c_1 \times v_1) + (v_2 - c_2)) \ldots ((c_1 \times v_1) + (v_2 - c_2)) \ldots, \]
we can avoid calculating \((c_1 \times v_1) + (v_2 - c_2)\) twice. On the term-graph level, we can model the removal of repeated calculations of this type by the following production:

The relation on the structured alphabet is given by:
\[
\begin{align*}
op & \subseteq +, -, \times, / & \subseteq \top \\
x & \subseteq +, -, \times, / & v, c \subseteq \top \\
y & \subseteq +, -, \times, / & v, c \subseteq \top
\end{align*}
\]
where \(c\) is any constant and \(v\) any arithmetic variable. If we apply this production to a term graph representing an expression of the form mentioned above, we get a derivation step removing the unnecessary calculation as shown in Figure (a).

Figure 3.7.4(a): Removing common subexpressions

The fact that nodes 1 and 2 bear the same label is realized by choosing a suitable context graph. What happens if we label these nodes in the context graph \(C\)
differently, e.g., $l_C(1) = +$ and $l_C(2) = -$? The morphism $g$ allows this, but we get an illegal graph on the right-hand side because the label of node $[1,2]$ becomes $\top$. The reader should be aware that our definition of $\mathcal{SL}graph$ morphisms does not force us to replace a “symbolic label” by the same label at different nodes. On the other hand, we are allowed to replace different “symbolic” labels by the same label. Therefore, the same production is applicable if we have already found common subexpressions on the lower level. This situation is shown in Figure $(b)$.

Figure 3.7.4$(b)$: Removing common subexpressions after operands have been already identified

Discussing these examples, we have seen that choosing the context graph $C$ is not as trivial as in the string case. Applying a string production, we only remove the left-hand side of the production from the given string and we get the context string. Applying a graph production, we have some alternatives yielding the same $G^l$. In the next chapter, we shall discuss effectively constructing a derivation step, i.e., effectively constructing $C$ since the right-hand construction is straightforward. This discussion will also result in criteria that help us to choose between the alternatives.

### 3.8 Exercises

**Exercise 3.8.1:**

Consider a pushout diagram $\bar{g} \cdot p = \bar{p} \cdot g$ in $\mathcal{Graph}$. Show that the subdiagrams forgetting the graph structure are pushout diagrams in $\mathcal{Set}$:
Exercise 3.8.2 (Pushout in $\mathcal{H}graph$):
We have not proved formally that $\mathcal{H}graph$ has pushouts. But it is easy to draw an analogy between $\mathcal{G}raph$ and $\mathcal{H}graph$. Modify the proof of Theorem 3.1.4 to show existence of pushouts in $\mathcal{H}graph$.

Exercise 3.8.3:
Construct the pushout diagram of the graph morphisms given in the picture. The mappings of the nodes are defined by $p(i) = i'$ and $g(i) = \bar{i}$. As a consequence of this, the mappings of the edges are also defined. (Please note that $g(1) = g(2) = [1, 2]$.)

Exercise 3.8.4:
In the next few exercises, we consider a graph production and some variations of it. In the first exercise, the interface graph is discrete:

As usual, we assume $p^l(i) = i'$ and $p^r(i) = i'$. Construct the derivation step $G^l_1 \Rightarrow G^r_1$ using the context graph:
Exercise 3.8.5:
Now, we modify production $p_1$ by inserting an edge from node 2 to node 3 into the interface graph.

\[
\begin{array}{c}
B_2^l \\
1^l \\
\downarrow \\
2^l \\
\downarrow \\
3^l \\
\end{array}
\quad I_2
\quad
\begin{array}{c}
B_2^r \\
1^r \\
\downarrow \\
2^r \\
\downarrow \\
3^r \\
\end{array}
\]

Construct the derivation step $G_2^l \Rightarrow G_2^r$ applying this new production $p_2$ to the context graph $C_2 = C_1$ of Exercise 3.8.4. Compare the derivation steps!

Exercise 3.8.6:
Alternatively, we insert an edge from node 1 to node 2 into the interface graph of production $p_1$. This yields production $p_3$:

\[
\begin{array}{c}
B_3^l \\
1^l \\
\downarrow \\
2^l \\
\downarrow \\
3^l \\
\end{array}
\quad I_3
\quad
\begin{array}{c}
B_3^r \\
1^r \\
\downarrow \\
2^r \\
\downarrow \\
3^r \\
\end{array}
\]

Find a derivation step $G_3^l \Rightarrow G_3^r$ applying this production with the usual embedding $g_3^l(i^l) = i^l$, $g_3^r(i^r) = i^l$ into the following graph $G_3^l$:

\[
\begin{array}{c}
C_3^l \\
6^l \\
\downarrow \\
5^l \\
\downarrow \\
1^l \\
\end{array}
\quad
\begin{array}{c}
3^l \\
\downarrow \\
2^l \\
\end{array}
\quad
\begin{array}{c}
8^l \\
\downarrow \\
7^l \\
\end{array}
\]

(Hint: Construct a suitable context graph intuitively. How to construct it systematically, will be discussed in the next chapter.)

Exercise 3.8.7:
Now, apply the same production $p_3$ to the same graph $G_3^l$, but with a different embedding on the left-hand side: $g_3^l(1^l) = 2^l, g_3^l(2^l) = 7^l, g_3^l(3^l) = 3^l$. Show that this mapping can be made a graph morphism by suitably choosing the mapping of the edges, and find a derivation step $G_3^l \Rightarrow G_4^r$.

Exercise 3.8.8:
Consider the following production $p_5$: 

with $g(i) = i$. 

Find a context graph such that you can construct a derivation step $G^l_5 \Rightarrow G^r_5$ with the following graph $G^l_5$:

using the usual embedding $g^l_5(i) = \overline{i}$.

**Exercise 3.8.9:**
Consider the following situation:

We have chosen disjoint sets to identify the nodes occurring in the production and in the graph to be derived. Find as much embeddings $g^l_{6k}$ ($k = 1, 2, \ldots$) as possible and in each case, construct a derivation step $G^l_6 \Rightarrow G^r_{6k}$.

**Exercise 3.8.10:**
Now, we consider a production the left-hand side of which is not injective:
Apply this production to the following graph $G_l^i$ using the embedding $g_l^i(i') = l'$:

$G_l^i$

Construct as much derivation steps $G_l^i \Rightarrow G_r^i$ as possible, but (contrary to Exercise 3.8.9) do not change the embedding $g_l^i$.

Exercise 3.8.11:
In the proof of Theorem 3.3.1, we have shown the pushout property without presenting a suitable diagram. Add the morphism $D \rightarrow D'$ to the given diagram and explain the given proof with the aid of the completed diagram.

Exercise 3.8.12:
Consider the following production in $L_{graph}$ together with the embedding $g$:

As usual, the nodes are identified by numbers. For convenience, we use the same numbers in the left-hand graph, the interface graph, and the right-hand graph as well as in the context graph. (More precisely, we had to write $a_1^l$ instead of $a_1$ in the left-hand graph, etc.) Nodes are mapped in an intuitive way, i.e., $p_l(i) = i$, $p_r(i) = i$, and $g(i) = i$. We can avoid to identify the edges explicitly because in this example, edge mappings are unambiguously induced by node mappings. Complete the diagram to get a derivation step!

Exercise 3.8.13:
Now, turn your attention to an example with a non-injective left-hand side:
Nodes 1 and 2 of the interface graph are mapped onto one and the same node [1, 2] of the left-hand graph. This is possible because they bear the same label $a$. The notational conventions are the same as in Exercise 3.8.12.

(a) Complete the derivation step $G^l_1 \Rightarrow G^r_1$!

(b) Replace the embedding $g_1$ by another embedding $g_2(i) = i$ into the following graph: Compare the derivation step with the solution to (a).

(c) Is there another embedding resulting in the same graph to be derived?

Exercise 3.8.14:
In Example 3.3.5, we have translated some equations of a data-type specification into hypergraph productions. Construct the analogous translations of the remaining equations.

Exercise 3.8.15:
Example 3.4.6 considers some aspects of block structure in programming languages. The productions $(p_1) - (p_3)$ describe the effect of inserting an applied occurrence of an already declared identifier. Find productions describing how to insert a new declaration! Remember that such an insertion affects the $def$-edges coming from within the block and from the blocks it contains. (Hint: See [86].)

Exercise 3.8.16:
In Example 3.6.2, we have represented a Petri net by a (usual) graph the nodes of which are places or transitions. The edges are labeled to ensure the correct assignment. A more elegant solution is to represent transitions by hyperedges
the source nodes of which are the input places of the transition and the target
nodes of which are the output places. Give the details of this solution.

Exercise 3.8.17:
In Definition 3.6.3, we have introduced structured alphabets, and lateron, we
have restricted discussion to structured alphabets in which the least upper bound
exists. Prove that we can interpret the structured alphabet as a category and
that the least upper bound is the coproduct in this category.

Exercise 3.8.18:
In Lemma 3.6.8, we have constructed the coequalizer in $\mathcal{SL}_{\text{graph}}$ by constructing
coequalizers in $\mathcal{S}_{\text{et}}$ for nodes and edges separately. Prove that the result can be
unambiguously made a graph by suitably defining $s$ and $t$!

Exercise 3.8.19:
Complete the following pushout diagram in $\mathcal{S}_{\text{etinclgraph}}$, where $p(i) = i'$ and
g($i$) = $\bar{i}$ are assumed:

Exercise 3.8.20:
Apply the production of Example 3.7.4 to the expression

$$((10 \times a) \times (a - (b/2))) + ((10 \times a) \times (a - (b/2)))$$

as often as possible. Draw the normalized term graphs for each derivation step!
Construct a term-graph production implementing the simplification $x + x = 2 \times x$, and apply it to the term graph you have just derived.
3 Derivability in Categories

3.1 Definition of Derivation Steps .............................................. 60
3.2 Examples in Set and Graph with Noninjective Mappings .......... 66
3.3 Labeled Graphs ................................................................. 70
3.4 Graph Grammars ............................................................... 78
3.5 Graph Grammars vs. Chomsky Grammars ......................... 83
3.6 Graphs With Structured Alphabet ................................. 88
3.7 Examples With Structured Alphabet ........................... 94
3.8 Exercises ................................................................. 100