

Graph Grammars and Operations on Graphs

Jan Joris Vereijken

May 19, 1993



Department of Computer Science
Leiden University
The Netherlands

Master's Thesis, Leiden University, The Netherlands.

Title : *Graph Grammars and Operations on Graphs*

Author : Jan Joris Vereijken

Supervisor : Dr. Joost Engelfriet

Completion date : May 19, 1993

Problems worthy
of attack
prove their worth
by hitting back.

Contents

1	Introduction	1
1.1	Graph grammars	1
1.2	The structure of this thesis	2
1.3	Closing remarks	4
2	Definitions	5
2.1	Terminology	5
2.2	Typing	8
2.3	Typed alphabets	9
2.4	Typed languages	10
2.5	Typed grammars	11
2.6	Typed languages versus typed grammars	12
3	I/O-hypergraphs	17
3.1	Definition	17
3.2	Terminology	18
3.3	Depiction of hypergraphs	19
3.4	Ordinary graphs	20
3.5	String graphs	20
3.6	External versus internal nodes	22
3.7	Hypergraph languages	22
3.8	Union of hypergraph languages	23
4	Composition	25
4.1	Sequential composition	25
4.2	Sequential composition versus degree and loops	27
4.3	Parallel composition	28
4.4	Sequential versus parallel composition	29

4.5	Expressions used as a function	31
5	Decomposition	33
5.1	Definition	33
5.2	$\mathbf{HGR} \xrightarrow{\sim} L_A$	34
5.3	$L_A \xrightarrow{\sim} L_B$	35
5.4	$L_B \xrightarrow{\sim} L_{C_2} \cup L_{C_3}$	36
5.5	$L_{C_3} \xrightarrow{\sim^+} L_{C_1} \cup L_{C_2}$	37
5.6	$L_{C_2} \xrightarrow{\sim^+} L_{C_4} \cup L_{C_5}$	37
5.7	$L_{C_4} \xrightarrow{\sim^+} L_{C_6}$	40
5.8	$L_{C_5} \xrightarrow{\sim^+} L_{C_6}$	41
5.9	$L_B \xrightarrow{\sim^+} L_C$	41
5.10	Conclusions	41
6	Folds and flips	43
6.1	Definition	43
6.2	Basic properties	44
6.3	Derived properties	45
7	Interpretation	47
7.1	Definition of an interpreter	47
7.2	Definition of Int	48
7.3	Examples of interpretation	49
7.4	Edge Normal Form	51
7.5	Existence of isomorphic copies	52
7.6	Bounded degree implies bounded cutwidth	53
8	Power of interpretation	57
8.1	$\text{Int}(\text{RLIN}) = \text{Int}(\text{LIN})$	57
8.2	$\text{Int}(\text{RLIN}) = \text{Int}(\text{DB})$	64
8.3	$\text{Int}(\text{STR}(\text{Int}(\mathbf{K}))) = \text{Int}(\mathbf{K})$	72
8.4	About $\text{STR}(\text{Int}(\text{RLIN}))$	78
8.5	The power of interpretation theorems	79
8.6	Conclusions	80
9	Closure properties of $\text{Int}(\mathbf{K})$	83
9.1	Closure under sequential composition	83
9.2	Closure under union	84

9.3	Closure under Kleene closure	84
9.4	Closure under $+\{U_n\}$ and $\{U_n\}+$	85
9.5	Closure under parallel composition	86
9.6	Closure under fold and backfold	86
9.7	Closure under flip	86
9.8	Closure under split	88
9.9	Closure under edge relabeling	88
9.10	Conclusions	88
10	Another characterization	91
10.1	Using HGR	91
10.2	Using the sequential pseudo base set	92
10.3	Using the full base set	93
10.4	Conclusions	93
11	Other literature	95
11.1	Introduction	95
11.2	Engelfriet and Heyker	96
11.3	Context-Free Hypergraph Grammars	97
11.4	$\text{split}(\mathbf{\Gamma}(\text{LIN-CFHG})) = \text{Int}(\text{RLIN})$	98
11.5	$\text{Int}(\text{RLIN}) \subsetneq \text{Int}(\text{CF})$	101
11.6	$\text{Int}(\text{CF}) \subsetneq \text{split}(\mathbf{\Gamma}(\text{CFHG}))$	102
11.7	Bauderon and Courcelle	104
11.8	Habel and Kreowski	106
11.9	Further reading	107
12	Summary	109
13	Acknowledgments	111
A	Naming conventions	113
B	Proofs	117
B.1	Proofs concerning Section 6.3	117
B.2	Proofs concerning Section 8.2	120
	Bibliography	127
	Index	129

For those who like this sort of thing,
this is the sort of thing they like.

— *Abraham Lincoln*

1

Introduction

1.1 Graph grammars

Just like sets of strings (string languages) can be characterized by string grammars, sets of graphs (*graph languages*) can be characterized by *graph grammars*. Over the past decade, a lot of research has been done into this subject. The approach mainly taken was either to rewrite edges, or to rewrite nodes.

However, a completely different approach to define graph languages is also imaginable: one defines a number of operations on graphs (including constants), and considers a string language of expressions over these operations. The graphs obtained by evaluating these expressions then form a graph language.

In this thesis we describe and investigate a simple formalism of the latter kind. We define only *one* nonconstant operation on graphs: *sequential composition*.

When we now take a string language over some alphabet, and a function from that alphabet to a set of graphs, we can do the following. For each string in the language, take the graphs one obtains by applying the function to all the symbols that form that string. Now take the sequential composition of all those graphs, in the order as indicated by the order of the symbols that form the string. This yields a graph. So, for every string in the language, we obtain a graph. Together they form a graph language. We have called this formalism *interpretation*: we take a string language and a function, and interpret the strings in that language as graphs, in the way indicated by that function (called the *interpreter*). Note that the symbols in the alphabet may be viewed as graph constants,

that can be arbitrarily interpreted as graphs, by the interpreter. Strings may be viewed as expressions over these constants and sequential composition. Thus, concatenation of strings is interpreted as sequential composition of graphs.

This formalism to associate graph languages with string languages has the following obvious advantage above a graph grammar formalism. From formal language theory, a lot of different classes (for example, the Chomsky hierarchy) of string languages are known. By means of interpretation, we can immediately derive classes of graph languages from these classes of string languages. For example, from the class of all regular languages, we instantly derive the class of all graph languages obtainable by interpretation of regular languages. In this way, for every known class of string languages, we now also have a corresponding class of graph languages.

In the chapters to follow, we will investigate the relations between several classes obtained by interpretation in this way. We will look at their properties, and how our formalism relates to other formalisms (mainly graph grammars) that were proposed to define graph languages. That brings us to the title of this thesis: “*Graph Grammars and Operations on Graphs*”. In our view, a string grammar/interpreter pair is just a “graph grammar in disguise”. This will be justified by the results we will find; they are very much alike the results obtained by using a “real” graph grammar.

1.2 The structure of this thesis

The structure of this thesis is the following. In Chapter 2, we lay out the mathematical framework needed to express ourselves. In particular, we will introduce the concept of *typing*: every symbol, string, string language, graph, or whatever, has two integers associated with it, its *input type*, and its *output type*.

As we will want to interpret concatenation as sequential composition, we will need a special kind of graphs, namely *i/o-hypergraphs* (Chapter 3). These have distinguished *input nodes* and *output nodes*, so we have an easy way to define how the sequential composition acts on them: roughly speaking, sequential composition connects two graphs to one another by “hooking” the first’s output nodes to the second’s input nodes, just like railroad cars are hooked together to form a train. By the typing, we indicate the number of nodes that need to be hooked. If we now make sure that the types of two neighboring symbols within a string match, and that the interpretation function preserves the type of the symbols, we can guarantee that in the process of interpreting we will only apply sequential composition to graphs that “fit”. This is, in short, the reason we need all our objects to be typed: to ensure that all our operations are applied in a way that makes sense, i.e., the objects operated on must fit.

Then, in Chapter 4, we formally define the sequential composition operation, and its less important counterpart, *parallel composition*. Using these operations, we can take “small” graphs, and use them to build larger ones. As noted, sequential composition is like hooking railroad cars together, with graphs. Continuing this metaphor, parallel composition is like stacking railroad cars on top of one another (for example, in order to build a double-deck automobile carrier).

Just like the composition operations build larger graphs from small ones, we can also do the opposite: take a large graph to pieces, namely small graphs. In Chapter 5, we investigate this process, which is called *decomposition*. We will try to find the “smallest” set of “small” graphs from which all other ones can be built. Or in other words, we search for the “basic building blocks” of graphs (the answer, more or less, is: *edges*).

In Chapter 6 we introduce four auxiliary operations on graphs. These operations do not operate on the internal structure of graphs, but only on the “superficial” structure of which distinguished nodes are input nodes, and which ones are output nodes. Some properties of these operations are investigated. They will “only” be needed to conveniently express the technical details of some proofs, but nonetheless they have a beauty of their own.

After that, in Chapter 7, we are finally ready to introduce the formalism of interpretation, and will devote three chapters to the investigation of its power and properties. First of all, we will prove some theorems about classes of graph languages obtained by interpretation, one of them concerning a *normal form* for interpretation. Most importantly, these theorems will gain us an insight in the structure of these kinds of classes.

Then, in Chapter 8, we look at the classes of regular, linear, and derivation-bounded languages under interpretation. As it turns out, all three give rise to the same class, which we propose as the “class of regular graph languages”. Furthermore, we indicate how large a class can get so that under interpretation it is still the same as the class of all regular graph languages. This culminates in the two “power of interpretation” theorems, which give strong indications on the power of interpretation.

In Chapter 9 we look at some closure properties of a general class of graph languages obtained by interpretation. The conditions for closure will be given in terms of closure properties of the underlying class of string languages on which the interpretation acted.

Following that, there are two chapters where we make comparisons with other formalisms. In Chapter 10 we give a “smallest class closed under . . .” characterization of the class of all regular graph languages, and in Chapter 11 we look at the relations between the formalism of interpretation and some other formalisms that have been proposed in the literature. Luckily, our idea of “the class of regular graph languages” corresponds very well with some classes proposed by other researchers.

There are two appendices. In Appendix A we account for the naming conventions (e.g.,

n is always an integer, w is always a string of symbols) we have used throughout this thesis. By strictly adhering to these conventions, we hope to have made our constructions easier to read. Appendix B contains some proofs that we did not want to give in full in the main text.

Finally, there is a Bibliography, which contains information about the literature we refer to, and an extensive Index.

1.3 Closing remarks

This Master's thesis was written in the final fulfillment of the requirements for a Master's degree in Theoretical Computer Science at the *Rijksuniversiteit te Leiden*, The Netherlands, under the supervision of Dr. Joost Engelfriet.

For those curious, the motto on page iii appeared as a fortune cookie on my computer terminal one winter night at 4:30 AM, when after a whole night of T_EX'ing and fiddling with the definition of interpretation, I found out that I had "fixed" the definition, but *broken all my proofs*, seemingly beyond repair. Disillusioned I logged out of our VAX/VMS system, only to end up with a screen that said, in large, friendly, VT100 letters:

Problems worthy of attack prove
their worth by hitting back.

It seemed very appropriate.

We will restrict ourselves to natural numbers only, as there are quite enough of these.

— *Edsger Wybe Dijkstra*

2

Definitions

In this chapter the definitions and notations are laid out of the mathematical framework we will need. Some of them are very common, and in wide use, and some are quite novel. In particular, we introduce typed variants of the concepts of a grammar and a language, as noted in the introduction.

2.1 Terminology

We assume the reader to be familiar with elementary set theory (see, e.g., [Kam50], [BS87, §5], or [Her75, §1.1]), and elementary formal language theory (see, e.g., [HU79] or [CL89]). Some basic knowledge about graphs is also useful (see, e.g., [BS87, §11] or [Joh84, §3]).

Numbers: \mathbb{N} denotes the set $\{0, 1, 2, \dots\}$ of all nonnegative natural numbers. The interval $\{1, \dots, n\}$ is denoted by $[n]$, and the interval $\{m, \dots, n\}$ by $[m, n]$. For a finite set $V \subseteq \mathbb{N}$, by $\mathbf{max}(V)$ we denote the largest element of V , and by $\mathbf{min}(V)$ the smallest element.

Sets: Set inclusion is denoted by \subseteq , proper set inclusion by \subsetneq , set union by \cup , set intersection by \cap . Set difference is denoted by \setminus or $-$. The symbols \supseteq and \supsetneq denote the inverses of \subseteq and \subsetneq . The empty set is denoted by \emptyset , set membership by \in , or inversely, \ni . For a set V , $\mathcal{P}(V)$ denotes the *power set* of V ; $\mathcal{P}(V) = \{W \mid W \subseteq V\}$. The cardinality

of a set V is denoted¹ by $|V|$. The cartesian product of two sets V and W is denoted by $V \times W$, and the n times repeated cartesian product of a set V with itself by V^n .

Sequences: A sequence over a set V is denoted (v_1, \dots, v_n) , the empty sequence $()$ by λ , and V^* denotes the set of all sequences. The length of a sequence $\alpha \in V^*$ is denoted by $|\alpha|$. A sequence of length n will also be called an n -sequence.

Logic: By TRUE and FALSE we denote the boolean constants for true and false. Logical or is denoted by \vee , and logical and by \wedge . The symbol \implies denotes logical implication, and \iff logical equivalence (if and only if). We follow the convention to write “iff” as a shorthand for “if and only if”. The symbol \forall denotes universal quantification (“for all”) and \exists denotes existential quantification (“there exists”). These quantors are always subscripted by the declarations of the variables that are local to the quantification, e.g., $\forall_{a,b,c,n \in \mathbb{N}^+} (n \geq 3 \implies a^n + b^n \neq c^n)$. Name clash ambiguities between local and global variables are not allowed.

Relations: The symbol \equiv always denotes an equivalence relation. For a set V and an equivalence relation \equiv on V , the equivalence class of $v \in V$ with respect to \equiv is denoted by $[v]_{\equiv}$, and the set of all equivalence classes by V/\equiv . An equivalence relation \equiv on some set V may be thought of as a set, namely the set $\{(v, v') \in V \times V \mid v \equiv v'\}$ of all pairs from V that are equivalent.

We extend the notation to sequences and functions in the following way. For a sequence (v_1, \dots, v_n) over a set V , and \equiv an equivalence relation on V , by $[(v_1, \dots, v_n)]_{\equiv}$ we denote the sequence $([v_1]_{\equiv}, \dots, [v_n]_{\equiv})$. For a function $f : V_1 \rightarrow V_2$ and an equivalence relation \equiv on V , the function f_{\equiv} is defined as $f_{\equiv}(v) = [f(v)]_{\equiv}$, for all $v \in V_1$. By the symbol \approx we denote the informal concept of “approximate” equality. Formally, \approx means *nothing at all!*

Functions: For two functions $f : V_1 \rightarrow V_2$ and $g : V_2 \rightarrow V_3$ the composition is written as $g \circ f$, and for all $v \in V_1$, $(g \circ f)(v) = g(f(v))$. The restriction of a function $f : V \rightarrow W$ to a subset V' of V is denoted $f \upharpoonright V'$. A function $f : V \rightarrow W$ whose domain V contains exactly one element may be denoted $a \mapsto f(a)$, where a is that one element. For a bijective function (also called bijection) $f : V_1 \rightarrow V_2$, its inverse is denoted $f^{-1} : V_2 \rightarrow V_1$.

Alphabets: An *alphabet* (also called *ordinary alphabet*) is a nonempty finite set of symbols. A *ranked alphabet* is an alphabet that has a rank from \mathbb{N} associated with every

¹Warning: the symbol $\#$ is only used in identifier names, *not* to denote set cardinality.

symbol. For an alphabet Σ and a symbol $a \in \Sigma$, this rank is denoted as $\mathbf{rank}_\Sigma(a)$. To express its rank, a symbol a with rank n may be denoted (a, n) .

Strings: A *string*² over an alphabet Σ is a sequence $w \in \Sigma^*$, a *substring* $v \in \Sigma^*$ of a string $w \in \Sigma^*$ is a string such that there exists strings $u, z \in \Sigma^*$ such that $uvz = w$. If $u = \lambda$ the string v is called a *prefix* of w , and if furthermore v and z are both nonempty it is called a *proper prefix*. Conversely, when $z = \lambda$, the string v is called a *postfix*, and a *proper postfix* if also u and v are nonempty. By the symbol \cdot we denote the concatenation of strings. For a string $w = a_1 \dots a_n$ its reverse $a_n \dots a_1$ is denoted by w^R . This operation is called reversal. A *language* over an alphabet Σ is a set $L \subseteq \Sigma^*$, in other words, a set of strings over Σ . If we say that L is *strictly over* Σ , we mean that all symbols are really used, i.e., for all $a \in \Sigma$ there exists a string $w \in L$ such that a is a substring of w .

Grammars: A context-free grammar is denoted $G = (N, T, P, S)$, where N is the non-terminal alphabet, T is the terminal alphabet (disjoint with N), P is the set of productions (of the form $A \rightarrow \alpha$ with $A \in N$, and $\alpha \in (N \cup T)^*$), and $S \in N$ is the initial symbol. A derivation of an $\alpha \in (N \cup T)^*$ by a nonterminal $A \in N$ will be denoted $A \Rightarrow^* \alpha$. When we want to explicitly mention the length k of the derivation, we write $A \Rightarrow^k \alpha$. We may prefix any of these syntactic constructs by the name G of the grammar in order to stress to which grammar it belongs. E.g.: $G : A \Rightarrow^* \alpha$, meaning A derives α in G . The set of all context-free grammars is denoted by $G(\text{CF})$. For a context-free grammar G the language it defines is denoted $L(G)$.

Classes of languages: The class of all context-free languages, $\{L(G) \mid G \in G(\text{CF})\}$ is denoted by $L(\text{CF})$. The following well-known classes of context-free languages are used in this thesis: $L(\text{RLIN})$, the class of all *right-linear* languages, $L(\text{LIN})$, the class of all *linear* languages and $L(\text{DB})$, the class of all *derivation-bounded* languages. Where there can be no confusion, we may omit the L's.

A language L is said to be right-linear iff it can be generated by a context-free grammar G that satisfies the restriction that for every $p \in P$, p is of the form $A \rightarrow wB$, or $A \rightarrow w$, where $A, B \in N$ and $w \in T^*$. Such a grammar G is also called right-linear. The class of all right-linear languages is also often called the class of all *regular* languages, and therefore, sometimes denoted as REG.

A language L is linear iff it can be generated by a context-free grammar G that satisfies the restriction that for every $p \in P$, p is of the form $A \rightarrow vBw$, or $A \rightarrow v$, where $A, B \in N$ and $v, w \in T^*$. Such a grammar G is also called linear.

²Traditionally, strings are also often called *words*.

A language L is derivation-bounded iff it can be generated by a context-free grammar G such that for some $m \in \mathbb{N}$, for every $w \in L$ there is a derivation $S \Rightarrow \alpha_1 \Rightarrow \dots \Rightarrow \alpha_n \Rightarrow w$, where $\alpha_i \in (N \cup T)^*$, in G such that there is no α_i that contains more than m occurrences of symbols from N . This bound m is called the *derivation bound*. Such a grammar G is also called derivation-bounded.

The relation between the above mentioned classes of languages is $\text{RLIN} \subsetneq \text{LIN} \subsetneq \text{DB} \subsetneq \text{CF}$ (note that all inclusions are proper).

Formally, if X is a *property of a grammar*, then $G(X)$ is the class of all grammars that have that property, and $L(X) = \{L(G) \mid G \in G(X)\}$ is the class of all languages generated by those grammars. So, strictly speaking, RLIN (for example) is a *property of grammars* (namely, right-linearity), $G(\text{RLIN})$ is a *class of grammars*, and $L(\text{RLIN})$ is a *class of languages*. As noted, we will informally often omit the L , and use \mathbf{X} to denote $L(\mathbf{X})$.

2.2 Typing

As noted in the introduction, we want our objects to be “typed”. That is, with almost every object, be it a symbol, an alphabet, a language, or whatever, we want to associate a type, denoted $(m \rightarrow n)$ (where $m, n \in \mathbb{N}$).

The general idea behind this is twofold. Firstly, for some our functions we will want to require *type preservingness*, i.e., the function must always return an object of the same type as the object it took as argument. Secondly, we will want to specify *type conditions*, i.e., some binary operations will only be defined under certain conditions on the *types* of the two arguments. In this way, we can always assure that the results of our computations are defined and meaningful.

Compare this to what happens in strongly *typed* programming languages. Take for example PASCAL. There all variables have to be declared, and must be used in accordance with their declaration. So when, e.g., we declare `var n:integer; x:real;` the assignment `x := sqrt(n);` makes sense, but `n := sqrt(x);` will result in a compile-time error. Or compare it to what happens in physics, where we have the concept of *unit*. We are only allowed to operate on quantities in a way that makes sense with respect to their respective units. So, one kilogram plus two kilograms makes three, but two meters plus four seconds is always nonsense. Just as obeying the declarations in PASCAL program, and the units in a physical computation, is a sine qua non for the results to make sense, we will have to obey certain rules of typing too.

The reason that we want to give each object an input type *and* an output type, is that our graphs (to be defined later) will have two distinguished types of nodes: *input nodes*,

and *output nodes*. Intuitively, an object of type $(m \rightarrow n)$ stands as a placeholder for a graph with m input nodes and n output nodes.

In the three sections to follow, we will define typed variants of the concepts of an alphabet, a language and a grammar.

2.3 Typed alphabets

A *typed alphabet* is an alphabet that has two ranks from \mathbb{N} associated with every symbol. For an alphabet Σ and a symbol $a \in \Sigma$, these two ranks are denoted as $\#\mathbf{in}_\Sigma(a)$ (the *input type*) and $\#\mathbf{out}_\Sigma(a)$ (the *output type*). We may drop the subscribed Σ where there can be no confusion. A symbol $a \in \Sigma$ with input type m and output type n may also be denoted $(a, m \rightarrow n)$. It is said to be of *type* $(m \rightarrow n)$. If we want to stress the fact that a symbol belongs to a typed alphabet, we may call it a *typed symbol*. Consequently, a symbol from an ordinary alphabet may be called an *ordinary symbol*. Mutatis mutandis, we define a *typed set* to be a set that has two ranks from \mathbb{N} associated with every element, and refer to a nontyped set as an *ordinary set*.

A typed alphabet Σ_1 and an ordinary alphabet Σ_2 such that both contain exactly the same symbols are considered equal (denoted $\Sigma_1 = \Sigma_2$), albeit there are two functions that are defined on the first that are undefined on the second (also see the “philosophical sidenote” at the end of Section 2.4).

Let Σ be a typed alphabet. For two symbols $a_1, a_2 \in \Sigma$, the concatenation $w = a_1 \cdot a_2$ is only defined when $\#\mathbf{out}(a_1) = \#\mathbf{in}(a_2)$. The type of the resulting string w is $(\#\mathbf{in}(a_1) \rightarrow \#\mathbf{out}(a_2))$. Kleene closure on Σ is defined as follows:

$$\Sigma^+ = \{ a_1 \dots a_n \mid n \geq 1, \#\mathbf{out}_\Sigma(a_i) = \#\mathbf{in}_\Sigma(a_{i+1}) \text{ for } 1 \leq i < n, a_1, \dots, a_n \in \Sigma \},$$

$$\Sigma^* = \{ (\lambda, n \rightarrow n) \mid n \in \mathbb{N} \} \cup \Sigma^+.$$

Here $(\lambda, n \rightarrow n)$ denotes the empty string of type $(n \rightarrow n)$ (note that there is no such thing as a $(\lambda, m \rightarrow n)$ where $m \neq n$). For a nonempty string $w \in \Sigma^*$, $\alpha = a_1 \dots a_n$ the input type of w , denoted $\#\mathbf{in}_\Sigma(w)$, is $\#\mathbf{in}_\Sigma(a_1)$. The output type, denoted $\#\mathbf{out}_\Sigma(w)$, is $\#\mathbf{out}_\Sigma(a_n)$. A string $w \in \Sigma^*$ of type $(m \rightarrow n)$ can be denoted $(w, m \rightarrow n)$. Note that Σ^* is a typed set.

A string $w = a_1 \dots a_n$ over a typed alphabet Σ is called *correctly internally typed* if $\#\mathbf{out}_\Sigma(a_i) = \#\mathbf{in}_\Sigma(a_{i+1})$ for all $1 \leq i < n$. Note that, by the above definition of Kleene closure on a typed set, Σ^* consists of exactly all correctly internally typed strings over Σ .

Concatenation of two strings $v, w \in \Sigma^*$ is only defined when the output type of the first

matches the input type of the second. So for $(v, m \rightarrow n)$ and $(w, n \rightarrow k)$ we define:

$$(v, m \rightarrow n) \cdot (w, n \rightarrow k) = (vw, m \rightarrow k).$$

Note that for a string $v \in \Sigma^*$ all substrings of w are also in Σ^* . By definition of a substring, the empty string $(\lambda, n \rightarrow n)$ is a substring of v iff there is a symbol $a \in \Sigma$ in v such that $\#\mathbf{in}_\Sigma(a) = n$ or $\#\mathbf{out}_\Sigma(a) = n$.

As with symbols, we will use the terms *ordinary string* and *typed string* to discriminate between the two.

2.4 Typed languages

A *typed language* L is a set of correctly internally typed strings over some typed alphabet Σ , such that all strings have the same type:

$$\forall_{w_1, w_2 \in L} (\#\mathbf{in}(w_1) = \#\mathbf{in}(w_2) \text{ and } \#\mathbf{out}(w_1) = \#\mathbf{out}(w_2)).$$

A typed language L in which all strings are of type $(m \rightarrow n)$ can be denoted $(L, m \rightarrow n)$ when clarity demands it. We say that L is of *type* $(m \rightarrow n)$. If necessary, the input and output type can be denoted as follows: $\#\mathbf{in}(L) = m$ and $\#\mathbf{out}(L) = n$. In the case of the empty language of type $(m \rightarrow n)$, we need to write $(\emptyset, m \rightarrow n)$ to make its type explicit. If $m = n$, L is called of *uniform type*. Note that a typed language that contains the empty string λ must necessarily be of uniform type: a typed λ always has the same input and output type, say $(\lambda, k \rightarrow k)$, for some $k \in \mathbb{N}$, and all strings in L have the same type, so L has type $(k \rightarrow k)$.

Concatenation, union, and Kleene closure are defined on typed languages over the same alphabet Σ in the following way (let $(L, m \rightarrow n)$, $(L_1, m_1 \rightarrow n_1)$ and $(L_2, m_2 \rightarrow n_2)$ be typed languages).

- $L_1 \cdot L_2 = \{w_1 \cdot w_2 \mid w_1 \in L_1 \text{ and } w_2 \in L_2\}$, in the case that $n_1 = m_2$, and undefined otherwise. Note that $L_1 \cdot L_2$ is of type $(m_1 \rightarrow n_2)$,
- $L_1 \cup L_2 = \{w \mid w \in L_1 \text{ or } w \in L_2\}$, in the case that $m_1 = m_2$ and $n_1 = n_2$, and undefined otherwise. Note that $L_1 \cup L_2$ is of type $(m_1 \rightarrow n_1)$,
- $L^* = \bigcup_{k=0}^{\infty} L^k$, in the case that $m = n$, and undefined otherwise. Here L^k denotes L , k times concatenated to itself. L^0 denotes the appropriate unity element, $\{(\lambda, m \rightarrow m)\}$ in this case. Note that L^* is of type $(m \rightarrow m)$.

Finally, when we want to stress the fact that a string language is *not* typed, we will refer to it as an *ordinary* string language. For a typed string language L , the ordinary string language L' such that L and L' contain exactly the same strings (albeit those in L are typed, and those in L' are not) is called the *underlying language* of L' .

For a class K of ordinary string languages, we denote by $L_\tau(K)$ the class of all typed languages whose underlying languages are in $L(K)$. Note that since we use \mathbf{X} to denote $L(\mathbf{X})$, $L_\tau(\mathbf{X})$ denotes $L_\tau(L(\mathbf{X}))$.

As a “philosophical” sidenote: normally we would consider two languages equal iff *they contain exactly the same strings*. In order to remain faithful to this intuitive concept, we will allow a typed language L_1 and a nontyped language L_2 to be equal to each other, albeit that nonetheless there is a difference: the strings in L_1 have a type associated with them, those in L_2 do not. This (non)typing, however, is not considered to be all that important for the essence of the language, it is merely something that is added on for convenience. We will write: $L_1 = L_2$.

If all this sounds counter-intuitive, notice that something similar is being done in ordinary formal language theory, where two languages over different alphabets can be the same, provided only a common subset of symbols from both alphabets is actually used in the two languages.

In other words: not *where* the symbol came from (the one alphabet or the other) matters, but *what* the symbol is.

Note that from this point of view $L(\text{CF}) = L_\tau(\text{CF})!$ (Proof: assign type $(0 \rightarrow 0)$ to all symbols.) So when we write $L \in L_\tau(\text{CF})$ formally speaking we could just as well have dropped the τ . However, in what follows we will implicitly assume that the τ in $L \in L_\tau(\text{CF})$ means that we have a *fixed typing in mind* for that L .

Unfortunately, this also prohibits us from writing $L_1 = L_2$, if, for two typed languages L_1, L_2 , we want to express that they are equal and also have the same typing defined on them (or $=$ would not be an equivalence relation anymore, something we certainly do not want to happen). Therefore, we will write $L_1 =_\tau L_2$ instead, if we want to express that $L_1 = L_2$ and $L_1, L_2 \subseteq \Sigma^*$ for some typed alphabet Σ , i.e., L_1 and L_2 are equal *even with the typing*. Note that $=_\tau$ indeed is an equivalence relation.

2.5 Typed grammars

A context-free grammar $G = (N, T, P, S)$ where N and T are typed alphabets is called *typed*, if for every production $p : A \rightarrow \alpha$, $A \in N$, $\alpha \in (N \cup T)^*$, A and α are of the same type. Be aware that the Kleene closure is over a typed alphabet, so α is a correctly internally typed string with respect to $N \cup T$. Let $G' = (N', T', P', S')$ be the *underlying*

grammar of G , i.e., $N' = N$, $T' = T$ (albeit there is no typing defined on N' and T'), $P' = P$, and $S' = S$. We now define $L(G)$, the typed language generated by the typed grammar G :

$$L(G) = \begin{cases} L(G'), \text{ typed according to } T & \text{if } L(G') \neq \emptyset, \\ (\emptyset, \#\mathbf{in}_N(S) \rightarrow \#\mathbf{out}_N(S)) & \text{if } L(G') = \emptyset \end{cases}$$

Warning: note that this definition *only* makes sense if $L(G')$ is a typed language with respect to T . That this is indeed the case, will be proved in the next section, where we will also prove that $L(G)$ is always of the same type as S .

When we want to stress the fact that a grammar is *not* typed, we will refer to it as an *ordinary* grammar.

We extend the notation $G(X)$, all grammars with a certain property, (for example, $X = \text{CF}$), to $G_\tau(X)$, all typed grammars whose underlying grammar is in $G(X)$.

2.6 Typed languages versus typed grammars

Context-free typed languages and typed grammars are equivalent in the sense that every typed grammar generates a context-free typed language, and that every context-free typed language is generated by some context-free typed grammar. Formally:

$$\begin{aligned} \forall_{G \in G_\tau(\text{CF})} L(G) \in L_\tau(\text{CF}), \\ \text{and} \\ \forall_{L \in L_\tau(\text{CF})} \exists_{G \in G_\tau(\text{CF})} L =_\tau L(G). \end{aligned} \tag{2.1}$$

Part 1:

Firstly, we will prove that for every typed context-free grammar G , we have $L(G) \in L_\tau(\text{CF})$ ³. Let $G = (N, T, P, S) \in G_\tau(\text{CF})$, and $G' = (N', T', P', S')$ the underlying grammar of G . Now for every production $p : A \rightarrow \alpha$, $A \in N$, $\alpha \in (N \cup T)^*$, we have, by definition, $\#\mathbf{in}_N(A) = \#\mathbf{in}_{(N \cup T)}(\alpha)$, and $\#\mathbf{out}_N(A) = \#\mathbf{out}_{(N \cup T)}(\alpha)$. To start with, we will prove that for every derivation:

$$G' : S' \Rightarrow^* \alpha,$$

where $\alpha \in (N' \cup T')^*$, we have that:

- α is correctly internally typed with respect to $N \cup T$, and,

³Formally speaking, this is trivially true (by definition)! Recall however the above warning that we still need to verify that the definition is correct. This verification is what the now following proof is about.

- with respect to $N \cup T$, α is of the same type as S' .

We proceed by induction on the length of the derivation. Induction basis: length is 0. There is only one derivation of length 0: $S' \Rightarrow^0 S'$, for which both conditions trivially hold. Induction step: length is $k + 1$. We assume that our claim holds for length k . Consider a derivation of length $k + 1$, which has the form:

$$S' \Rightarrow^k \alpha \xrightarrow{R} \beta.$$

To prove: β is correctly internally typed, and has the same type as S' . Let the production applied in the last step be $p : A' \rightarrow \gamma$. Then α must have the form $\alpha' A' \alpha''$, and consequently β the form $\alpha' \gamma \alpha''$. As α is correctly typed internally, by the induction hypothesis, so are α' and α'' . And, as A' has the same type as γ , and γ is correctly typed internally (both by the definition of p), necessarily β is correctly typed internally also. Furthermore, as obviously β has the same type as α , and, by the induction hypothesis, α has the same type as S' , it is clear that β has the same type as S' . End of induction proof.

Now for any $w \in T^*$ in $L(G')$, i.e., $S' \Rightarrow^* w$, w has the same type as S' with respect to $N \cup T$, and is correctly internally typed with respect to $N \cup T$. As an important consequence, we now have verified that $L(G')$ is indeed a correctly typed language with respect to T , which ensures that the definition of the language generated by a typed languages (see the previous section) is indeed meaningful.

Consequently, all the above statements also hold for $L(G)$, and, hence, $L(G)$ is a typed language of type $(\#in_N(S) \rightarrow \#out_N(S))$. Therefore, $L(G) \in L_\tau(\text{CF})$, which completes the first part of the proof.

Part 2:

Secondly, for a given context-free typed language $L \in L_\tau(\text{CF})$ over some typed alphabet Σ , we will construct a context-free typed grammar $G \in G_\tau(\text{CF})$ such that $L =_\tau L(G)$. We distinguish three cases:

- $L = (\emptyset, m \rightarrow n)$, for some $m, n \in \mathbb{N}$, or,
- $L = \{(\lambda, n \rightarrow n)\}$, for some $n \in \mathbb{N}$, or,
- none of the above.

The first two cases are almost trivial. If $L = (\emptyset, m \rightarrow n)$, it is generated by the context-free typed grammar $(\{(S, m \rightarrow n)\}, \emptyset, \emptyset, S)$. If $L = \{(\lambda, n \rightarrow n)\}$, it is generated by the context-free typed grammar $(\{(S, n \rightarrow n)\}, \emptyset, \{S \rightarrow (\lambda, n \rightarrow n)\}, S)$. The last case

(“none of the above”), is the difficult one. There, choose⁴ an ordinary, reduced, λ -free, context-free grammar $G = (N, T, P, S)$ (where $T = \Sigma$) such that $L - \{\lambda\} = L(G)$ (albeit that there is a typing associated with L , and not with $L(G)$). Now extend G to a typed grammar by defining functions $\#in$ and $\#out$ on N and T . Take $\#in_T(a) = \#in_\Sigma(a)$, and $\#out_T(a) = \#out_\Sigma(a)$, for every $a \in T$. For a nonterminal $A \in N$, define $\#in_N$ and $\#out_N$ in the following way: if $A \Rightarrow^* w$ in G (with $w \in T^+$) then $\#in_N(A) = \#in_\Sigma(w)$, and $\#out_N(A) = \#out_\Sigma(w)$. Such a w always exists, as G is reduced and λ -free.

That this definition is consistent, i.e., if $A \Rightarrow^* v$, and $A \Rightarrow^* w$, then v and w are of the same type, is quite straightforward. Choose an arbitrary derivation $S \Rightarrow^* uAz$ (where $u, z \in T^*$), the existence of which is guaranteed by the reachability of A . Because G is context-free we now have $S \Rightarrow^* uvz$ and $S \Rightarrow^* uwz$. As $uvz, uwz \in L \subseteq \Sigma^*$ we know that $\#in_\Sigma(v) = \#out_\Sigma(u) = \#in_\Sigma(w)$, and $\#out_\Sigma(v) = \#in_\Sigma(z) = \#out_\Sigma(w)$ ⁵. Together with the usefulness of A this consistency guarantees that $\#in_N(A)$, and $\#out_N(A)$, are always properly defined.

Left to show that the thus defined typing functions on $N \cup T$ indeed make G a typed grammar, i.e., that G satisfies the restriction that for all productions $p : A \rightarrow \alpha$, α must be a correctly internally typed string over $N \cup T$, of the same type as A . Suppose α has the following form:

$$\alpha = w_0 A_1 w_1 \dots w_{n-1} A_{n-1} w_n,$$

where $w_0, \dots, w_n \in T^*$, and $A_1, \dots, A_{n-1} \in N$. By the reducedness of G , there now exist derivations $A_i \Rightarrow^* v_i$, where $v_i \in T^*$, for all $1 \leq i < n$. So:

$$A \Rightarrow^* w_0 v_1 w_1 \dots w_{n-1} v_{n-1} w_n = z.$$

By the reachability of A , there exists a $u \in L$, such that z is a substring of u . As u is correctly internally typed, so is z . Furthermore, as for all $A_i \Rightarrow^* v_i$, $1 \leq i < n$, A_i and v_i have the same type, α is also correctly internally typed, and α has the same type as z , which is the type of A .

Because we did not modify G in any way (we only extended N and T to be typed) the thus typed grammar obviously generates $L - \{\lambda\}$. We now distinguish two cases. First case: $\lambda \notin L$. This means $L(G) =_\tau L$, so we have arrived at the typed grammar G we are looking for. Second case: $\lambda \in L$. By adding the production $S \rightarrow (\lambda, \#in_N(S) \rightarrow \#out_N(S))$ to P

⁴Such a grammar always exists. See for example [HU79, §4.4].

⁵These expressions are trickier than one might perceive at first sight. Note that we *require* a u and z to be there. This in effect means that when u or z should happen to be “empty” we will use the empty typed string *of the appropriate type* to represent them. In this case that would amount to $(\lambda, \#in(L) \rightarrow \#in(L))$ for u , and likewise, $(\lambda, \#out(L) \rightarrow \#out(L))$ for z . The reader should be alert for this “trick” which occurs several times in the chapters to follow (see also the index under “lambda trick”).

we can extend the typed grammar G in such a way that $L(G) =_{\tau} L$. Note that necessarily $\#\mathbf{in}_N(S) = \#\mathbf{out}_N(S)$: as L is a typed language that contains a λ , it must be of uniform type. This completes the proof that for every $L \in L_{\tau}(\text{CF})$ there exists a $G \in G_{\tau}(\text{CF})$ such that $L =_{\tau} L(G)$.

Finally, note that in proving (2.1) the only use we made of CF was that all languages in $L(\text{CF})$ are context-free and all grammars in $G(\text{CF})$ are λ -free reducible. Consequently, we can easily extend it to other classes than CF. Let X be a property of grammars such that $L(X) \subsetneq L(\text{CF})$ and $G(X)$ λ -free reducible, then:

$$\begin{aligned} \forall_{G \in G_{\tau}(X)} L(G) \in L_{\tau}(X), \\ \text{and} \\ \forall_{L \in L_{\tau}(X)} \exists_{G \in G_{\tau}(X)} L =_{\tau} L(G). \end{aligned} \tag{2.2}$$

In particular, the equivalence of typed languages with typed grammars holds for the properties RLIN, LIN, and DB (this is not trivial, but checking the conditions is beyond the scope of this thesis).

Leibniz spoke of it first, calling it *geometria situs*. This branch of geometry deals with relations dependent on position alone, and investigates the properties of position.

— Leonhard Euler

3

I/O-hypergraphs

In this chapter, we will define the kind of graphs we will be working with. Instead of taking the standard definition of a directed graph, where edges stand for 2-sequences of nodes, we will generalize to *hypergraphs*, where the *hyperedges* stand for n -sequences of nodes. The reason for this is, that we want to compare our method to a well-known type of context-free graph grammar that works with hypergraphs.

Furthermore, we will define a sequence of input nodes, and a sequence of output nodes. Hence the name: i/o-hypergraphs. This we do because in that way we can easily define operations (read: sequential composition) on our graphs; our main operation will take two graphs, and then “hook” the output nodes of the first to the input nodes of the second, thereby forming a larger graph, just like railroad cars are hooked together to form a train.

3.1 Definition

Let Δ be a ranked alphabet. An *i/o-hypergraph* H over Δ is a 6-tuple $(V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out})$ where V is the finite set of *nodes*¹, E is the finite set of (*hyper*)*edges*, $\mathbf{nod} : E \rightarrow V^*$ is the *incidence function*, $\mathbf{lab} : E \rightarrow \Delta$ is the *labeling function*, $\mathbf{in} \in V^*$ is the sequence of *input nodes* and $\mathbf{out} \in V^*$ is the sequence of *output nodes*.

If necessary, we can denote these components by V_H , E_H , \mathbf{nod}_H , \mathbf{lab}_H , \mathbf{in}_H and \mathbf{out}_H respectively, in order to avoid possible confusion with other hypergraphs. It is required

¹Traditionally, nodes are sometimes also called *vertices*.

that for every $e \in E$, $\text{rank}_\Delta(\text{lab}(e)) = |\text{nod}(e)|$. If $\text{nod}(e) = (v_1, \dots, v_n)$, $n \in \mathbb{N}$, then v_i is also denoted by $\text{nod}(e, i)$, and we say that e and v_i are *incident*. In the same fashion, if $\mathbf{in} = (v_1, \dots, v_n)$, then v_i is denoted $\mathbf{in}(i)$, and similarly for \mathbf{out} . Furthermore, we define $\#\mathbf{in}(H) = |\mathbf{in}|$ and $\#\mathbf{out}(H) = |\mathbf{out}|$, the length of the input and output sequences. Where convenient, we may choose to view \mathbf{in} and \mathbf{out} as being sets. In such a case, they denote the sets $\{\mathbf{in}(i) \mid 1 \leq i \leq |\mathbf{in}|\}$ and $\{\mathbf{out}(i) \mid 1 \leq i \leq |\mathbf{out}|\}$ respectively.

Finally, we assume the reader to be experienced in the problem of *concrete* versus *abstract* graphs (where an abstract graph is a class of isomorphic concrete graphs). As usual in the theory of graph grammars we consider graph languages (to be defined for hypergraphs in Section 3.7) to consist of abstract graphs; however, in all our constructions we deal with concrete graphs (taking an isomorphic copy when necessary). The notion of isomorphism is the obvious one, preserving the incidence structure, the edge labels, and the sequences of input and output nodes.

3.2 Terminology

Let H be the hypergraph $(V, E, \text{nod}, \text{lab}, \mathbf{in}, \mathbf{out})$. For $v \in V$, by $\text{deg}(v)$ we denote the number of edges incident with v , its *degree*. We extend this notation to H by defining $\text{deg}(H) = \max(\{\text{deg}(v) \mid v \in V_H\})$.

An i/o-hypergraph with m input nodes and n output nodes is said to be of *type* $(m \rightarrow n)$. Where convenient, it is called of *input type* m and of *output type* n . An i/o-hypergraph of type $(n \rightarrow n)$ is called of *uniform type* n . An i/o-hypergraph of type $(0 \rightarrow 0)$ is called *simple*. An i/o-hypergraph H such that no node $v \in V_H$ appears more than twice in \mathbf{in}_H , nor in \mathbf{out}_H , is called *identification free*.

For a ranked alphabet Δ the set of all i/o-hypergraphs over Δ is denoted by $\mathbf{HGR}(\Delta)$. Note that $\mathbf{HGR}(\Delta)$ is a typed set, under the following² typing functions: for all $H \in \mathbf{HGR}(\Delta)$, $\#\mathbf{in}_{\mathbf{HGR}(\Delta)}(H) = \#\mathbf{in}(H)$ and $\#\mathbf{out}_{\mathbf{HGR}(\Delta)}(H) = \#\mathbf{out}(H)$. The set of all i/o-hypergraphs of type $(m \rightarrow n)$ over Δ is denoted by $\mathbf{HGR}_{m,n}(\Delta)$. Finally, by \mathbf{HGR} , we denote the set of *all* hypergraphs (not restricted to some fixed alphabet).

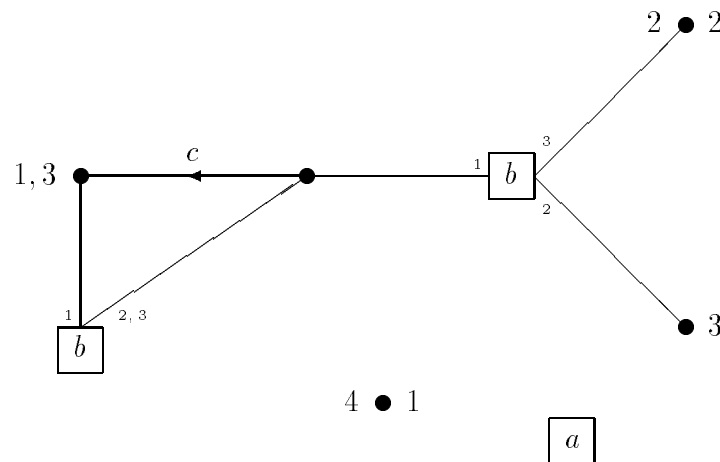
From now on, unless the context proves otherwise, the word “hypergraph” is used as a synonym to “i/o-hypergraph”.

²Note that the left-hand side refers to $\#\mathbf{in}$, the *input type of a typed set*, and the right-hand side to $\#\mathbf{in}$, the *length of the input sequence of H*

3.3 Depiction of hypergraphs

Sometimes, instead of just formally defining a hypergraph, we will also give a graphical (hypergraphical?) representation of it. In depicting a hypergraph H we will follow the following conventions (almost literally adopted from [EH91, page 331]). A node of H is indicated by a fat dot, as usual, and an edge e of H is indicated by a box containing $\mathbf{lab}(e)$, with a thin line between e and $\mathbf{nod}(e, i)$, labeled by a tiny i . These lines (or the corresponding integers) are also called the *tentacles* of the hyperedge e . An edge e with two tentacles (i.e., with $|\mathbf{nod}(e)| = 2$) may also be drawn as a thick directed line from $\mathbf{nod}(e, 1)$ to $\mathbf{nod}(e, 2)$, with label $\mathbf{lab}(e)$, as usual in ordinary graphs. The input nodes $\mathbf{in}_H(i)$ are indicated by a label i to the *left* of the nodes, the output nodes $\mathbf{out}_H(i)$ by a label i to the *right* of the nodes.

In the following example, of a hypergraph $H = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out})$ of type $(4 \rightarrow 3)$, all conventions can be observed.



Here $V = \{v_1, v_2, v_3, v_4, v_5\}$, $E = \{e_1, e_2, e_3, e_4\}$, $\mathbf{nod}(e_1) = (v_1, v_2, v_2)$, $\mathbf{nod}(e_2) = (v_2, v_1)$, $\mathbf{nod}(e_3) = (v_2, v_4, v_3)$, $\mathbf{nod}(e_4) = \lambda$, $\mathbf{lab}(e_1) = b$, $\mathbf{lab}(e_2) = c$, $\mathbf{lab}(e_3) = b$, $\mathbf{lab}(e_4) = a$, $\mathbf{in} = (v_1, v_3, v_1, v_5)$, and $\mathbf{out} = (v_5, v_3, v_4)$.

There is only *one* hypergraph that cannot be satisfactorily depicted by these conventions: the “empty” hypergraph $(\emptyset, \emptyset, \emptyset, \emptyset, \lambda, \lambda)$, which has no nodes, and no edges. We will depict it by a big \emptyset symbol.

3.4 Ordinary graphs

Ordinary (directed) graphs, that is hypergraphs where all edges are incident with exactly two nodes, are special cases of hypergraphs. In other words: a hypergraph over a ranked alphabet Δ is an ordinary graph iff for all symbols $a \in \Delta$, $\mathbf{rank}_\Delta(a) = 2$. For an ordinary graph H , an edge $e \in E_H$ such that $\mathbf{nod}_H(e, 1) = \mathbf{nod}_H(e, 2)$ (i.e., both tentacles of e are incident with the same node) is called a *loop*.

On ordinary graphs we will define the notion of *cutwidth*. Let H be an ordinary graph, and suppose that $|V_H| = n$. Now a bijection $f : V_H \rightarrow \{1, \dots, n\}$ is called a *linear layout of H* . A *cut* is one of the numbers i with $1 \leq i < n$. Intuitively, H is cut between node $f^{-1}(i)$ and $f^{-1}(i+1)$, and by the width of cut i is meant the number of edges that cross this cut, i.e., those edges e for which either $f(\mathbf{nod}(e, 1)) \leq i$ and $f(\mathbf{nod}(e, 2)) > i$, or $f(\mathbf{nod}(e, 2)) \leq i$ and $f(\mathbf{nod}(e, 1)) > i$. The *cutwidth of H under f* , denoted $\mathbf{cw}(H, f)$, is the maximum number of edges e , for all $1 \leq i < n$, for which either $f(\mathbf{nod}(e, 1)) \leq i$ and $f(\mathbf{nod}(e, 2)) > i$, or $f(\mathbf{nod}(e, 2)) \leq i$ and $f(\mathbf{nod}(e, 1)) > i$. Thus, $\mathbf{cw}(H, f)$ is the maximal width of all cuts. The *cutwidth of H* , denoted $\mathbf{cw}(H)$, is defined as follows:

$$\mathbf{cw}(H) = \min(\{ \mathbf{cw}(H, f) \mid f \text{ is a linear layout of } H \}). \quad (3.1)$$

The following lemma on cutwidth gives an absolute upper bound on the cutwidth of *any* given ordinary graph, as a function of the number of nodes it has. For a given ordinary graph H , over some alphabet Δ , and a linear layout f of H the cutwidth of H is bounded by $\frac{1}{2} \cdot \mathbf{deg}(H) \cdot |V_H|$:

$$\forall_{\substack{H \in \mathbf{HGR}(\Delta) \\ f: V_H \rightarrow [|V_H|]}} \mathbf{cw}(H, f) \leq \frac{1}{2} \cdot \mathbf{deg}(H) \cdot |V_H|. \quad (3.2)$$

The proof is trivial, as obviously $\mathbf{cw}(H, f) \leq |E_H|$, and clearly, by elementary graph theory, $|E_H| \leq \frac{1}{2} \cdot \mathbf{deg}(H) \cdot |V_H|$. Therefore, no more than $\frac{1}{2} \cdot \mathbf{deg}(H) \cdot |V_H|$ edges can possibly be cut in any linear layout. This observation immediately leads to the following, somewhat weaker, lemma:

$$\forall_{H \in \mathbf{HGR}(\Delta)} \mathbf{cw}(H) \leq \frac{1}{2} \cdot \mathbf{deg}(H) \cdot |V_H|. \quad (3.3)$$

Finally, a linear layout $f : V_H \rightarrow \{1, \dots, n\}$ may intuitively be thought of as the sequence $(f^{-1}(1), \dots, f^{-1}(n))$ of all nodes in H .

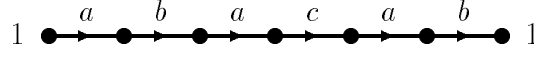
3.5 String graphs

An ordinary graph where all edges “lie in line” is called a *string graph*. This is due to the fact that every string can be uniquely represented as a string graph. For example, the

string:

$abacab$

corresponds to the following string graph (i.e. ordinary graph, i.e. hypergraph):



Formally, for an ordinary string $w = a_1 \dots a_n$ over an alphabet Σ , the string graph corresponding with it, denoted $\mathbf{gr}(w)$, is defined as follows:

$$\begin{aligned} \mathbf{gr}(w) &= (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out}), \\ \text{where } V &= \{v_0, \dots, v_n\}, E = \{e_1, \dots, e_n\}, \\ \mathbf{nod}(e_i) &= (v_{i-1}, v_i) \text{ and } \mathbf{lab}(e_i) = a_i, \text{ for } 1 \leq i \leq n, \\ \mathbf{in} &= (v_0), \mathbf{out} = (v_n). \end{aligned}$$

Note that by this definition, $\mathbf{gr}(\lambda)$ consists of just one node, and no edges (as one would expect), and:

$$\forall_{w_1, w_2 \in \Sigma^*} \mathbf{gr}(w_1 \cdot w_2) = \mathbf{gr}(w_1) \cdot \mathbf{gr}(w_2). \quad (3.4)$$

The function $\mathbf{gr} : \Sigma^* \rightarrow \mathbf{HGR}(\Sigma)$ is, as can be easily checked, injective. In other words: no two different strings are ever mapped to the same string graph, so, as noted, a string graph uniquely represents a certain string. The degree of a string graph is a direct function of the length of the underlying string. For a string w :

$$\mathbf{deg}(\mathbf{gr}(w)) = \begin{cases} 0 & \text{if } |w| = 0, \\ 1 & \text{if } |w| = 1, \\ 2 & \text{if } |w| \geq 2. \end{cases} \quad (3.5)$$

Therefore, a string graph H always has $\mathbf{deg}(H) \leq 2$. Finally, note that a string graph never contains a loop, as can be directly seen in the definition.

Now for a class \mathbf{K} of hypergraph languages, by $\mathbf{STR}(\mathbf{K})$ we denote the class of all ordinary string languages L such that $\mathbf{gr}(L) \in \mathbf{K}$:

$$\mathbf{STR}(\mathbf{K}) = \{ L \mid \mathbf{gr}(L) \in \mathbf{K} \}.$$

Note that $\mathbf{STR}(\mathbf{K})$ is a class of *string* languages, not *hypergraph* languages, and furthermore that for every class \mathbf{K} of hypergraph languages, and every class K of string

languages:

$$\mathbf{STR}(\mathbf{gr}(K)) = K \quad (3.6)$$

$$\mathbf{gr}(\mathbf{STR}(K)) \subseteq K. \quad (3.7)$$

3.6 External versus internal nodes

Nodes that are either input nodes, output nodes, or both, are called *external nodes*. Two hypergraphs $H_1 = (V_1, E_1, \mathbf{nod}_1, \mathbf{lab}_1, \mathbf{in}_1, \mathbf{out}_1)$, $H_2 = (V_2, E_2, \mathbf{nod}_2, \mathbf{lab}_2, \mathbf{in}_2, \mathbf{out}_2)$ are said to be *equal modulo i/o*, denoted $H_1 \equiv_{i/o} H_2$, when $V_1 = V_2$, $E_1 = E_2$, $\mathbf{nod}_1 = \mathbf{nod}_2$, and $\mathbf{lab}_1 = \mathbf{lab}_2$. In other words, $H_1 \equiv_{i/o} H_2$ holds when H_1 is equal to H_2 , except possibly for their input or output sequences. Note that $\equiv_{i/o}$ is an equivalence relation, and that in terms of external nodes a simple hypergraph is just a hypergraph with no external nodes.

This notation is extended to sets of hypergraphs in the following way. For two sets of hypergraphs L_1 and L_2 we have $L_1 \equiv_{i/o} L_2$ if for every $H \in L_1$ there exists a $H' \in L_2$ such that $H \equiv_{i/o} H'$, and, vice versa, for every $H \in L_2$ there exists a $H' \in L_1$ such that $H \equiv_{i/o} H'$.

Nodes that are not external are called *internal nodes*.

3.7 Hypergraph languages

A *hypergraph language* over a ranked alphabet Δ is a set of hypergraphs $\mathbf{L} \subseteq \mathbf{HGR}(\Delta)$. We require that all hypergraphs in \mathbf{L} be of type $(m \rightarrow n)$. \mathbf{L} is called of type $(m \rightarrow n)$. Like hypergraphs, when \mathbf{L} is of type $(n \rightarrow n)$, it is called of uniform type n . Where convenient, we will write $(\mathbf{L}, m \rightarrow n)$ instead of just \mathbf{L} , to explicitly express the type. The empty language must have a type too. The empty hypergraph language of type $(m \rightarrow n)$ is almost always denoted $(\emptyset, m \rightarrow n)$ because its type cannot simply be derived from the hypergraphs within it, as is the case with a nonempty hypergraph language.

The notion of a type extends to classes of hypergraph languages. When a class \mathbf{K} of hypergraph languages only contains hypergraph languages of type $(m \rightarrow n)$ it is also called of type $(m \rightarrow n)$. Otherwise, it is called of *mixed type*. Note that there is no such thing as a hypergraph language of mixed type³.

When \mathbf{L} contains exactly one hypergraph, it is called a *singleton hypergraph language*. For a set of hypergraphs (not necessarily a hypergraph language) \mathbf{L} , $\mathbf{Sing}(\mathbf{L})$

³It is very well possible to put hypergraphs of different types in one set, but that is not a hypergraph language as far as our definition is concerned.

denotes the class of all singleton hypergraph languages in $\mathcal{P}(\mathbf{L})$: $\mathbf{Sing}(\mathbf{L}) = \{\mathbf{L}' \mid \mathbf{L}' \subseteq \mathbf{L} \text{ and } \mathbf{L}' \text{ is a singleton}\}$. This class $\mathbf{Sing}(\mathbf{L})$ is called the *singleton class derived from \mathbf{L}* .

A hypergraph language \mathbf{L} is said to be of *bounded degree k* if for all graphs $H \in \mathbf{L}$, the degree of H is $\leq k$. A hypergraph language \mathbf{L} that contains only ordinary graphs, is said to be of *bounded cutwidth k* (where $k \in \mathbb{N}$) if for all graphs $H \in \mathbf{L}$ the cutwidth of H is $\leq k$.

3.8 Union of hypergraph languages

For two hypergraph languages $\mathbf{L}_1 = (V_1, m \rightarrow n)$ and $\mathbf{L}_2 = (V_2, m \rightarrow n)$ of the same type, the union of both is defined as the hypergraph language $\mathbf{L}_3 = \mathbf{L}_1 \cup \mathbf{L}_2 = (V_1 \cup V_2, m \rightarrow n)$. Note that the resulting hypergraph language is also of type $(m \rightarrow n)$. The empty language of type $(m \rightarrow n)$, $(\emptyset, m \rightarrow n)$, is the unity element of this union.

Actually we are defining countably infinite union operators: one for every type $(m \rightarrow n)$. When convenient, we will write $\cup_{m,n}$ instead of just \cup to make clear we are applying the union operator to hypergraph languages of type $(m \rightarrow n)$.

Text processing has made it possible to right-justify any idea, even one which cannot be justified on any other grounds.

— *J. Finnigan*

4

Composition

In this chapter we will define two binary operations on hypergraphs: sequential composition (the main one), and parallel composition. Using these operations it is possible to take several hypergraphs and use them to build a larger one.

4.1 Sequential composition

The *sequential composition* of two hypergraphs H_1 and H_2 , denoted $H_1 \cdot H_2$, is only defined when $\#\mathbf{out}(H_1) = \#\mathbf{in}(H_2)$. One finds $H_1 \cdot H_2$ by first taking the disjoint union of H_1 and H_2 , and then identifying the i th node of \mathbf{out}_{H_1} with the i th node of \mathbf{in}_{H_2} , for every $1 \leq i \leq \#\mathbf{out}(H_1)$. The resulting graph $H_1 \cdot H_2$ has \mathbf{in}_{H_1} as input node sequence, and \mathbf{out}_{H_2} as output node sequence. Formally, $H_1 \cdot H_2$ is defined as:

$$((V_{H_1} \cup V_{H_2})/\equiv, E_{H_1} \cup E_{H_2}, (\mathbf{nod}_{H_1} \cup \mathbf{nod}_{H_2})_{\equiv}, \mathbf{lab}_{H_1} \cup \mathbf{lab}_{H_2}, [\mathbf{in}_{H_1}]_{\equiv}, [\mathbf{out}_{H_2}]_{\equiv}),$$

where \equiv is the smallest equivalence relation on $V_{H_1} \cup V_{H_2}$ that contains the following pairs:

$$\{(\mathbf{out}_{H_1}(i), \mathbf{in}_{H_2}(i)) \mid 1 \leq i \leq \#\mathbf{out}(H_1)\}.$$

All this is supposing H_1 and H_2 are disjoint. If not, we take an isomorphic copy. The result of the sequential composition of two hypergraphs is also called their *product*. Note that the product of two hypergraphs is an ordinary graph iff both hypergraphs are also ordinary.

Actually we are defining countably infinite sequential composition operators: one for every $(m, n, k) \in \mathbb{N}^3$. When convenient, we will write $\cdot_{m,n,k}$ instead of just \cdot to make clear we are applying the \cdot operator to a H_1 of type $(m \rightarrow n)$ and a H_2 of type $(n \rightarrow k)$, for all Σ a typed alphabet:

$$\cdot_{m,n,k} : \mathbf{HGR}_{m,n}(\Sigma) \times \mathbf{HGR}_{n,k}(\Sigma) \rightarrow \mathbf{HGR}_{m,k}(\Sigma).$$

For every $n \in \mathbb{N}$ we define a hypergraph U_n , called the *unity hypergraph of type $(n \rightarrow n)$* :

$$U_n = (\{v_1, \dots, v_n\}, \emptyset, \emptyset, \emptyset, (v_1, \dots, v_n), (v_1, \dots, v_n)).$$

So, for example, U_4 is:

$$\begin{array}{c} 1 \bullet 1 \\ 2 \bullet 2 \\ 3 \bullet 3 \\ 4 \bullet 4 \end{array}$$

Note that U_n indeed is of type $(n \rightarrow n)$, that for every hypergraph H of input type n we have

$$U_n \cdot H = H, \tag{4.1}$$

and for every hypergraph G of output type n similarly

$$G \cdot U_n = G. \tag{4.2}$$

The unity hypergraph of type $(0 \rightarrow 0)$, U_0 , is also called the *empty hypergraph*, for obvious reasons.

The sequential composition can also be applied to two hypergraph languages, provided the output type of the first matches the input type of the second. Let $\mathbf{L}_1 = (V_1, m \rightarrow n)$ and $\mathbf{L}_2 = (V_2, n \rightarrow k)$ be hypergraph languages. Now the hypergraph language $\mathbf{L}_3 = \mathbf{L}_1 \cdot \mathbf{L}_2$ is defined as follows:

$$\mathbf{L}_3 = (\{H_1 \cdot H_2 \mid H_1 \in V_1, H_2 \in V_2\}, m \rightarrow k). \tag{4.3}$$

The sequential composition operation is associative, although proving this is surprisingly hard. Sketch of proof is as follows. Take disjoint hypergraphs H_1 , H_2 , and H_3 . Now by

completely writing out the expressions $H_1 \cdot (H_2 \cdot H_3)$ and $(H_1 \cdot H_2) \cdot H_3$ by the definition, in both cases we arrive at hypergraphs that are isomorphic to:

$$\begin{aligned} & ((V_{H_1} \cup V_{H_2} \cup V_{H_3})/\equiv, E_{H_1} \cup E_{H_2} \cup E_{H_3}, (\mathbf{nod}_{H_1} \cup \mathbf{nod}_{H_2} \cup \mathbf{nod}_{H_3})_{\equiv}, \\ & \quad \mathbf{lab}_{H_1} \cup \mathbf{lab}_{H_2} \cup \mathbf{lab}_{H_3}, [\mathbf{in}_{H_1}]_{\equiv}, [\mathbf{out}_{H_3}]_{\equiv}). \end{aligned}$$

Where \equiv denotes the smallest equivalence relation on $V_{H_1} \cup V_{H_2} \cup V_{H_3}$, that contains the following pairs:

$$\begin{aligned} & \{ (\mathbf{out}_{H_1}(i), \mathbf{in}_{H_2}(i)) \mid 1 \leq i \leq \#\mathbf{out}(H_1) \}, \\ & \{ (\mathbf{out}_{H_2}(i), \mathbf{in}_{H_3}(i)) \mid 1 \leq i \leq \#\mathbf{out}(H_2) \}. \end{aligned}$$

It can be easily seen that in general $H_1 \cdot H_2 \neq H_2 \cdot H_1$, so the sequential composition operation is noncommutative. However, when both H_1 and H_2 are simple hypergraphs the sequential composition operation *is* commutative, as in that case it reduces to just taking the disjoint union of both hypergraphs.

The symbol \prod stands for a repeatedly applied sequential composition:

$$\prod_{i=1}^n H_i = H_1 \dots H_n, \quad (4.4)$$

for all $n \in \mathbb{N}, n \neq 0$.

Sequential composition is also often called *concatenation*, because of the analogy with string concatenation¹. We may write the shorthand version $H_1 H_2$ for $H_1 \cdot H_2$.

Because of the associativity of the concatenation we can define Kleene closure on uniform hypergraph languages. For a hypergraph language \mathbf{L} of uniform type n :

$$\mathbf{L}^* = \bigcup_{k=0}^{\infty} \mathbf{L}^k,$$

where \mathbf{L}^k denotes \mathbf{L} , k times concatenated to itself. \mathbf{L}^0 denotes the appropriate unity element, $\{U_n\}$ in this case. Note that \mathbf{L}^* is again of uniform type n .

4.2 Sequential composition versus degree and loops

Sequential composition can only increase the degree of the hypergraphs involved. Formally expressed:

$$\forall_{H_1, H_2 \in \mathbf{HGR}(\Delta)} \begin{cases} \mathbf{deg}(H_1 \cdot H_2) \geq \mathbf{deg}(H_1), \\ \mathbf{deg}(H_1 \cdot H_2) \geq \mathbf{deg}(H_2). \end{cases} \quad (4.5)$$

¹To summarize: “sequential composition” and “concatenation” are synonyms, and can stand for both the operation and for the result. The word “product” always denotes the result, and never the operation.

The proof is trivial, as in the process of sequential composition, a node always remains incident with all the edges it was incident with before. Therefore, the degree of the product must be at least the degree of the original two hypergraphs. The degree of the product can be larger, however. This is the case when during identification two or more nodes that have edges attached to them get merged in such a way that the total degree of the resulting node is larger than the degree of the original hypergraphs. We can generalize (4.5) to the product of n hypergraphs in the following way:

$$\forall_{H_1, \dots, H_n \in \mathbf{HGR}(\Delta)} \mathbf{deg} \left(\prod_{i=1}^n H_i \right) \geq \mathbf{max} (\{ \mathbf{deg}(H_i) \mid 1 \leq i \leq n \}). \quad (4.6)$$

Regarding loops, note that by the definition of sequential composition, for hypergraphs H, H_1, H_2 , such that $H = H_1 \cdot H_2$, if H_1 , or H_2 , or both, contain a loop, then H itself also contains a loop.

As a special cases of these properties, for a string graph H , and hypergraphs H_1, \dots, H_n , such that $H = H_1 \dots H_n$, for all H_i , $1 \leq i \leq n$, $\mathbf{deg}(H_i) \leq 2$. Proof: as H is a string graph, $\mathbf{deg}(H) \leq 2$ (see Section 3.5), and the result directly follows from (4.6). Furthermore, as H (being a string graph) does not contain a loop, neither does H_i , for $1 \leq i \leq n$.

4.3 Parallel composition

The *parallel composition* of two hypergraphs H_1 and H_2 , denoted $H_1 + H_2$, is always defined. One finds $H_1 + H_2$ by taking the disjoint union of H_1 and H_2 , and putting $\mathbf{in}_{H_1+H_2} = \mathbf{in}_{H_1} \cdot \mathbf{in}_{H_2}$ and $\mathbf{out}_{H_1+H_2} = \mathbf{out}_{H_1} \cdot \mathbf{out}_{H_2}$. Formally, $H_1 + H_2$ is defined as:

$$(V_{H_1} \cup V_{H_2}, E_{H_1} \cup E_{H_2}, \mathbf{nod}_{H_1} \cup \mathbf{nod}_{H_2}, \mathbf{lab}_{H_1} \cup \mathbf{lab}_{H_2}, \mathbf{in}_{H_1} \cdot \mathbf{in}_{H_2}, \mathbf{out}_{H_1} \cdot \mathbf{out}_{H_2}),$$

All this is supposing H_1 and H_2 are disjoint. If not, we take an isomorphic copy. The result of the parallel composition of two hypergraphs is called their *sum*.

The unity element of the parallel composition is the empty hypergraph, U_0 . It can be easily verified that for every hypergraph H we have $U_0 + H = H$ and likewise $H + U_0 = H$. The parallel composition acts on unity hypergraphs in the following way:

$$U_n + U_m = U_{n+m}. \quad (4.7)$$

The parallel composition operation is associative. The proof, which directly follows from the associativity of \cup and \cdot (on sequences) is very easily accomplished by writing out in full $(H_1 + H_2) + H_3$ and $H_1 + (H_2 + H_3)$ where $H_1, H_2, H_3 \in \mathbf{HGR}(\Sigma)$ for some Σ , and H_1, H_2, H_3 mutually disjoint. Both expressions yield the same hypergraph:

$$(V_1 \cup V_2 \cup V_3, E_1 \cup E_2 \cup E_3, \mathbf{nod}_1 \cup \mathbf{nod}_2 \cup \mathbf{nod}_3, \mathbf{lab}_1 \cup \mathbf{lab}_2 \cup \mathbf{lab}_3, \mathbf{in}_1 \cdot \mathbf{in}_2 \cdot \mathbf{in}_3, \mathbf{out}_1 \cdot \mathbf{out}_2 \cdot \mathbf{out}_3),$$

which proves the associativity of $+$.

The parallel composition $H_1 + H_2$ is commutative when one or both of H_1 and H_2 are simple hypergraphs, but *not* in general.

The symbol \sum stands for a repeatedly applied parallel composition:

$$\sum_{i=1}^n H_i = H_1 + \cdots + H_n, \quad (4.8)$$

for all $n \in \mathbb{N}, n \neq 0$. For the case $n = 0$ we define $\sum_{i=1}^0 H_i = U_0$.

By the associativity of $+$ on both natural numbers and hypergraphs we can now generalize (4.7) to:

$$\sum_{k=1}^m U_{n_k} = U_{(\sum_{k=1}^m n_k)}. \quad (4.9)$$

The parallel composition can also be applied to two hypergraph languages. Let $\mathbf{L}_1 = (V_1, m_1 \rightarrow n_1)$ and $\mathbf{L}_2 = (V_2, m_2 \rightarrow n_2)$ be hypergraph languages. Now the hypergraph language $\mathbf{L}_3 = \mathbf{L}_1 + \mathbf{L}_2$ is defined as follows:

$$\mathbf{L}_3 = (\{H_1 + H_2 \mid H_1 \in V_1, H_2 \in V_2\}, m_1 + m_2 \rightarrow n_1 + n_2).$$

Note that \mathbf{L}_3 is by definition of type $(m_1 + m_2 \rightarrow n_1 + n_2)$.

4.4 Sequential versus parallel composition

The basic relationship between the sequential and parallel composition is as follows:

$$(H_1 + H_2) \cdot (H'_1 + H'_2) = H_1 H'_1 + H_2 H'_2, \quad (4.10)$$

where we require that $\#\mathbf{out}_{H_1} = \#\mathbf{in}_{H_2}$, and $\#\mathbf{out}_{H'_1} = \#\mathbf{in}_{H'_2}$. Sketch of proof is as follows. By completely writing out the left-hand and right-hand expression by the definitions, we arrive in both cases at:

$$\begin{aligned} & ((V_{H_1} \cup V_{H_2} \cup V_{H'_1} \cup V_{H'_2}) / \equiv, E_{H_1} \cup E_{H_2} \cup E_{H'_1} \cup E_{H'_2}), \\ & (\mathbf{nod}_{H_1} \cup \mathbf{nod}_{H_2} \cup \mathbf{nod}_{H'_1} \cup \mathbf{nod}_{H'_2}) / \equiv, \mathbf{lab}_{H_1} \cup \mathbf{lab}_{H_2} \cup \mathbf{lab}_{H'_1} \cup \mathbf{lab}_{H'_2}, \\ & (\mathbf{in}_{H_1} \cdot \mathbf{in}_{H_2}) / \equiv, (\mathbf{out}_{H'_1} \cdot \mathbf{out}_{H'_2}) / \equiv. \end{aligned}$$

Where \equiv denotes the smallest equivalence relation on $V_{H_1} \cup V_{H_2} \cup V_{H'_1} \cup V_{H'_2}$, that contains the following pairs:

$$\left\{ (\mathbf{out}_{H_1}(i), \mathbf{in}_{H'_1}(i)) \mid 1 \leq i \leq \#\mathbf{out}(H_1) \right\},$$

$$\left\{ (\mathbf{out}_{H_2}(i), \mathbf{in}_{H'_2}(i)) \mid 1 \leq i \leq \#\mathbf{out}(H_2) \right\}.$$

All this is, or course, supposing that $H_1, H_2, H'_1,$ and H'_2 are mutually disjoint. If not, we take isomorphic copies.

By repeatedly applying (4.10) to itself we get, for all $n \in \mathbb{N}, n \geq 1$:

$$(H_1 + \cdots + H_n) \cdot (H'_1 + \cdots + H'_n) = H_1 H'_1 + \cdots + H_n H'_n, \quad (4.11)$$

under the condition that $H_1 H'_1, \dots, H_n H'_n$ are all defined. Proof by induction on n . The induction basis ($n = 1$) is trivially fulfilled: $(H_1) \cdot (H'_1) = H_1 H'_1$. Induction step, assuming the induction hypothesis holds for $n = k$:

$$\begin{aligned} (H_1 + \cdots + H_{k+1}) \cdot (H'_1 + \cdots + H'_{k+1}) &= \text{(adding parentheses)} \\ ((H_1 + \cdots + H_k) + H_{k+1}) \cdot ((H'_1 + \cdots + H'_k) + H'_{k+1}) &\stackrel{(4.10)}{=} \\ (H_1 + \cdots + H_k) \cdot (H'_1 + \cdots + H'_k) + H_{k+1} H'_{k+1} &= \text{(induction hypothesis)} \\ (H_1 H'_1 + \cdots + H_k H'_k) + H_{k+1} H'_{k+1} &= \text{(removing parentheses)} \\ H_1 H'_1 + \cdots + H_{k+1} H'_{k+1}, & \end{aligned}$$

which proves (4.11). Note that we can rephrase this equation as:

$$\left(\sum_{i=1}^n H_i \right) \cdot \left(\sum_{j=1}^n H'_j \right) = \sum_{i=1}^n H_i H'_i. \quad (4.12)$$

This relation too can be made more general by repeatedly applying it to itself, which ultimately gives us the most general case:

$$\prod_{i=1}^m \sum_{j=1}^n H_{ij} = \sum_{j=1}^n \prod_{i=1}^m H_{ij}. \quad (4.13)$$

for every $m, n \in \mathbb{N}, m, n \geq 1$. Proof by induction on m (only). Induction basis ($m = 1$):

$$\prod_{i=1}^1 \sum_{j=1}^n H_{ij} \stackrel{(4.4)}{=} \sum_{j=1}^n H_{1j} \stackrel{(4.4)}{=} \sum_{j=1}^n \prod_{i=1}^1 H_{ij}.$$

Induction step, assuming the induction hypothesis holds for $m = k$:

$$\begin{aligned} \prod_{i=1}^{k+1} \sum_{j=1}^n H_{ij} &\stackrel{(4.4)}{=} \\ \left(\prod_{i=1}^k \sum_{j=1}^n H_{ij} \right) \cdot \sum_{j=1}^n H_{(k+1)j} &= \text{(induction hypothesis)} \end{aligned}$$

$$\begin{aligned}
& \left(\sum_{j=1}^n \prod_{i=1}^k H_{ij} \right) \cdot \sum_{j=1}^n H_{(k+1)j} \stackrel{(4.4)}{=} \\
& \left(\sum_{j=1}^n \prod_{i=1}^k H_{ij} \right) \cdot \left(\sum_{j=1}^n \prod_{i=k+1}^{k+1} H_{ij} \right) \stackrel{(4.12)}{=} \\
& \sum_{j=1}^n \left(\left(\prod_{i=1}^k H_{ij} \right) \cdot \left(\prod_{i=k+1}^{k+1} H_{ij} \right) \right) \stackrel{(4.4)}{=} \\
& \sum_{j=1}^n \prod_{i=1}^{k+1} H_{ij},
\end{aligned}$$

which proves (4.13). Note that (4.10) is just the case $m = 2, n = 2$ of (4.13), and (4.11) is just the case $m = 2$.

Furthermore, for simple hypergraphs H_1 and H_2 we have:

$$H_1 \cdot H_2 = H_1 + H_2.$$

Finally, the \cdot operator has precedence over the $+$ operator.

4.5 Expressions used as a function

Let $\mathbf{H}, \mathbf{G} : V \rightarrow \mathbf{HGR}(\Delta)$ be functions, with a common input domain V , that yield hypergraphs over some ranked alphabet Δ . We now define two new functions $\mathbf{F}_1, \mathbf{F}_2 : V \rightarrow \mathbf{HGR}(\Delta)$. For all $v \in V$:

$$\begin{aligned}
\mathbf{F}_1(v) &= \mathbf{H}(v) \cdot \mathbf{G}(v), \\
\mathbf{F}_2(v) &= \mathbf{H}(v) + \mathbf{G}(v).
\end{aligned}$$

In the case that $V = \mathbf{HGR}(\Delta)$ we also define $\mathbf{F}_3, \mathbf{F}_4, \mathbf{F}_5, \mathbf{F}_6 : V \rightarrow \mathbf{HGR}(\Delta)$. For all $v \in V$:

$$\begin{aligned}
\mathbf{F}_3(v) &= v \cdot \mathbf{H}(v), \\
\mathbf{F}_4(v) &= \mathbf{H}(v) \cdot v, \\
\mathbf{F}_5(v) &= v + \mathbf{H}(v), \\
\mathbf{F}_6(v) &= \mathbf{H}(v) + v.
\end{aligned}$$

(These six new functions need not be complete.) These constructions deserve their own notation: we may notate these ad hoc functions $\mathbf{F}_1, \dots, \mathbf{F}_6$ as $\mathbf{H} \cdot \mathbf{G}, \mathbf{H} + \mathbf{G}, \cdot \mathbf{H}, \mathbf{H} \cdot, + \mathbf{H}$, and $\mathbf{H} +$ respectively. When we consider a hypergraph $H \in \mathbf{HGR}(\Delta)$ as a function that

returns H for every $v \in V$, we may now write expressions like “ $+U_n$ ”, the function that adds the unity hypergraph of uniform order n to its input, or, by recursively applying this new notation, “ $(+\mathbf{flip}) \cdot \mathbf{backfold}$ ”. (\mathbf{flip} and $\mathbf{backfold}$ are total functions on hypergraphs, that will be defined in Section 6.1). Finally, we may do the same thing for functions that yield hypergraph *languages*.

"What's one and one and one and one and one
and one and one and one and one and one?"
"I don't know," said Alice. "I lost count."

— *Lewis Carroll*

5

Decomposition

Sequential and parallel composition are essentially about taking small hypergraphs and using them to build larger ones. But we can also do the opposite: take a large hypergraph, and try to break it down in smaller hypergraphs. This process is called *decomposition*.

In this chapter we will try to find a “small” set of “small” hypergraphs that can be used to build all other hypergraphs. The result will be useful in defining the class of all “regular” hypergraph languages (see Chapter 10) and in finding a normal form for interpreters (see Chapter 7).

5.1 Definition

Let $\mathbf{L}, \mathbf{L}' \subseteq \mathbf{HGR}$ be sets of hypergraphs. We now write $\mathbf{L} \dot{\rightarrow} \mathbf{L}'$, pronounced \mathbf{L} decomposes sequentially into \mathbf{L}' , if for every $H \in \mathbf{L}$ there exist $H_1, \dots, H_n \in \mathbf{L}'$, $n \geq 1$, such that $H = \prod_{i=1}^n H_i$. In words: every graph H in \mathbf{L} can be built from graphs in \mathbf{L}' by just using sequential composition.

If we also allow parallel composition to be used, we write $\mathbf{L} \dot{\rightarrow}^+ \mathbf{L}'$, pronounced \mathbf{L} fully decomposes into \mathbf{L}' . In words: for every $H \in \mathbf{L}$ there exist $H_1, \dots, H_n \in \mathbf{L}'$ such that H can be built using just H_1, \dots, H_n , \cdot , and $+$. Formally: \mathbf{L} is a subset of the smallest set of hypergraphs that contains \mathbf{L}' and is closed under \cdot and $+$.

It can easily be seen that both $\dot{\rightarrow}$ and $\dot{\rightarrow}^+$ are reflexive, transitive relations, and that sequential decomposition implies full decomposition. In the sections to follow we will construct sets $\mathbf{L}_A, \mathbf{L}_B$, and \mathbf{L}_C such that $\mathbf{HGR} \dot{\rightarrow} \mathbf{L}_A \dot{\rightarrow} \mathbf{L}_B \dot{\rightarrow}^+ \mathbf{L}_C$. The last step will

involve several substeps.

5.2 HGR $\dot{\rightarrow}$ L_A

We define L_A as follows:

$$L_A = \{H \in \mathbf{HGR} \mid \text{all nodes } v \in V_H \text{ are external}\}.$$

Now to prove $\mathbf{HGR} \dot{\rightarrow} L_A$ it suffices to prove that:

$$\forall_{H \in \mathbf{HGR}} \exists_{H_1, H_2 \in L_A} H = H_1 \cdot H_2.$$

For a given H we can always construct such a H_1 and H_2 by making all nodes in H external, resulting in H_1 , and constructing H_2 (without edges) in such a way that only the nodes that are supposed to be external are passed through. Formally, let

$$H = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out}) : p \rightarrow q.$$

Furthermore, let $\{v_1, \dots, v_n\}$ be the set of internal nodes of H . We now define:

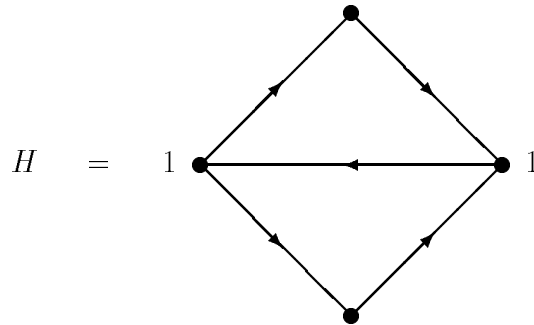
$$H_1 = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out} \cdot (v_1, \dots, v_n)) : p \rightarrow q + n,$$

$$H_2 = (\{w_1, \dots, w_{q+n}\}, \emptyset, \emptyset, \emptyset, (w_1, \dots, w_{q+n}), (w_1, \dots, w_q)) : q + n \rightarrow q.$$

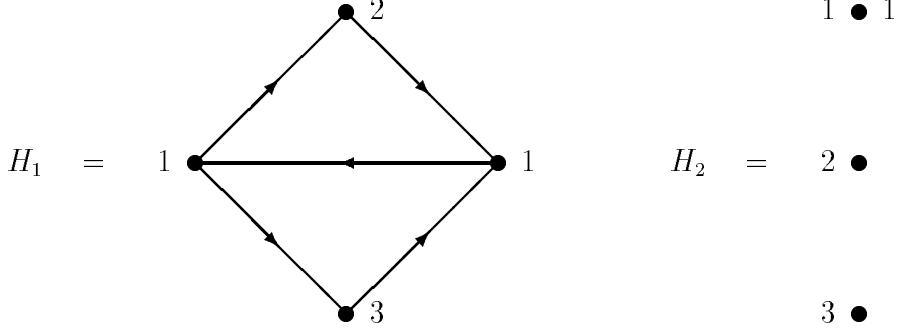
We now have $H = H_1 \cdot H_2$.

Example:

Let H be the hypergraph of type $(1 \rightarrow 1)$, with 2 internal nodes (so $p = q = 1$ and $n = 2$), depicted by:



Then H_1 and H_2 are of type $(1 \rightarrow 3)$ and $(3 \rightarrow 1)$ respectively, and look as follows:



Note that the edge labels have been left out, as they are irrelevant for the construction.

5.3 $L_A \xrightarrow{\cdot} L_B$

We define L_B as follows:

$$L_B = \{H \in L_A \mid |E_H| \leq 1\}.$$

Now to prove $L_A \xrightarrow{\cdot} L_B$ it suffices to prove that:

$$\forall_{H \in L_A, |E_H| > 1} \exists_{H_1, H_2 \in L_A} (H = H_1 \cdot H_2 \text{ and } |E_{H_1}| < |E_H| \text{ and } |E_{H_2}| < |E_H|).$$

For a given H we can always construct such a H_1 and H_2 by picking an edge e and reroute it outside H : $H_1 \approx H - \{e\}$, $H_2 \approx \{e\}$. Formally, let:

$$H = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out}) : p \rightarrow q,$$

and choose an edge $e \in E$. Suppose $|\mathbf{nod}(e)| = n$. We now define:

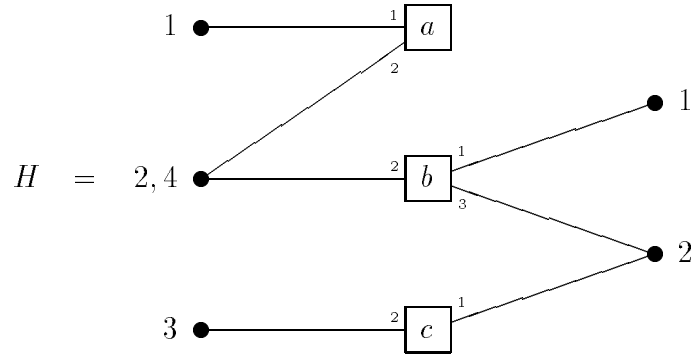
$$H_1 = (V, E - \{e\}, \mathbf{nod} \upharpoonright (E - \{e\}), \mathbf{lab} \upharpoonright (E - \{e\}), \mathbf{in}, \mathbf{out} \cdot \mathbf{nod}(e)) : p \rightarrow q + n,$$

$$H_2 = (\{w_1, \dots, w_{q+n}\}, \{e\}, e \mapsto (w_{q+1}, \dots, w_{q+n}), \\ e \mapsto \mathbf{lab}(e), (w_1, \dots, w_{q+n}), (w_1, \dots, w_q)) : q + n \rightarrow q.$$

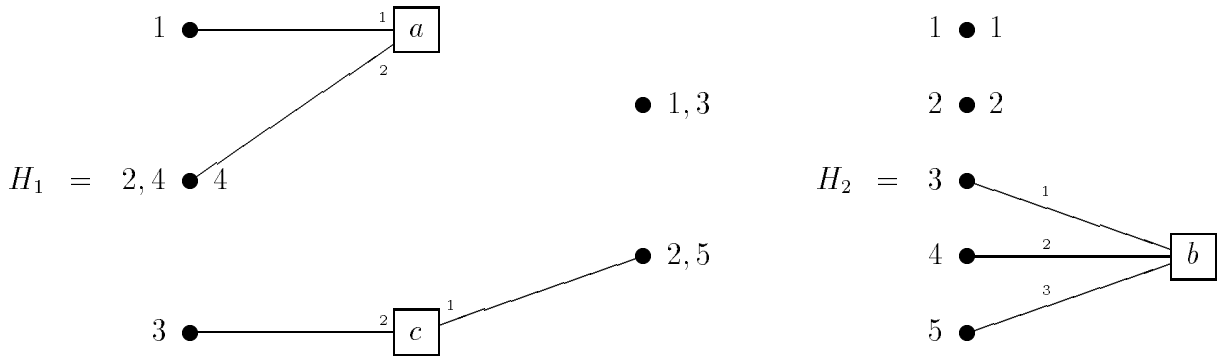
We now have $H = H_1 \cdot H_2$, $|E_{H_1}| = |E_H| - 1 < |E_H|$, and $|E_{H_2}| = 1 < |E_H|$. This method to remove an edge by sequential decomposition is called *edge removal*.

Example:

Let H be the hypergraph, of type $(4 \rightarrow 2)$ (so $p = 4$ and $q = 2$), depicted by:



Then applying edge removal at the edge labeled b (so $n = 3$), yields the hypergraphs H_1 and H_2 , of type $(4 \rightarrow 5)$ and $(5 \rightarrow 2)$ respectively, that look as follows:



5.4 $\mathbf{L}_B \xrightarrow{\cdot} \mathbf{L}_{C_2} \cup \mathbf{L}_{C_3}$

First we define sets \mathbf{L}_{C_1} , \mathbf{L}_{C_2} , and \mathbf{L}_{C_3} as follows:

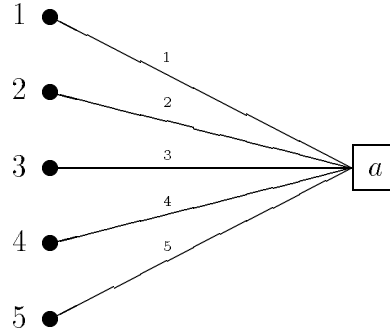
$$\begin{aligned} \mathbf{L}_{C_1} &= \{ \Omega_a \mid a \text{ a ranked symbol } \}, \\ \mathbf{L}_{C_2} &= \{ H \in \mathbf{L}_B \mid |E_H| = 0 \}, \\ \mathbf{L}_{C_3} &= \{ H \in \mathbf{L}_B \mid \exists_{n \in \mathbb{N}, a \text{ a ranked symbol}} H = U_n + \Omega_a \}, \end{aligned}$$

where Ω_a , for $\mathbf{rank}(a) = n$, is defined as follows:

$$\Omega_a = (\{v_1, \dots, v_n\}, \{e\}, e \mapsto (v_1, \dots, v_n), e \mapsto a, (v_1, \dots, v_n), \lambda).$$

For example, when a has rank 5,

$$\Omega_a = (\{v_1, v_2, v_3, v_4, v_5\}, \{e\}, e \mapsto (v_1, v_2, v_3, v_4, v_5), e \mapsto a, (v_1, v_2, v_3, v_4, v_5), \lambda) =$$



Now to prove $L_B \xrightarrow{;\,+} L_{C_2} \cup L_{C_3}$, take an $H \in L_B$. If $|E_H| = 1$, sequentially decompose it into H_1 and H_2 using edge removal. We now have: $H = H_1H_2$, $H_1 \in L_{C_2}$ and $H_2 \in L_{C_3}$ (note that $H_2 = U_q + \Omega_{\text{lab}(e)}$, for the q and e as meant in the definition of edge removal). If $|E_H| \neq 1$, it must be 0, so $H \in L_{C_2}$. This completes the proof.

5.5 $L_{C_3} \xrightarrow{;\,+} L_{C_1} \cup L_{C_2}$

Proof. For all $H \in L_{C_3}$, $H = U_n + \Omega_a$ for some $n \in \mathbb{N}$ and a a ranked symbol. But $U_n \in L_{C_2}$ and $\Omega_a \in L_{C_1}$, so $L_{C_3} \xrightarrow{;\,+} L_{C_1} \cup L_{C_2}$. This completes the proof.

5.6 $L_{C_2} \xrightarrow{;\,+} L_{C_4} \cup L_{C_5}$

Define sets L_{C_4} and L_{C_5} as follows:

$$L_{C_4} = \{ \mathbf{1}_{m,n} \mid m, n \in \mathbb{N}, (m, n) \neq (0, 0) \},$$

$$L_{C_5} = \{ \Pi_{\pi, \pi', k} \mid \pi, \pi' \text{ permutations of } (1, \dots, k), k \in \mathbb{N} \}.$$

Here $\mathbf{1}_{m,n}$ is defined as follows:

$$\mathbf{1}_{m,n} = (\{v\}, \emptyset, \emptyset, \emptyset, \underbrace{(v, \dots, v)}_{m \text{ times}}, \underbrace{(v, \dots, v)}_{n \text{ times}}).$$

For example,

$$\mathbf{1}_{3,2} = (\{v\}, \emptyset, \emptyset, \emptyset, (v, v, v), (v, v)) =$$

$$1, 2, 3 \bullet 1, 2$$

Furthermore, $\Pi_{\pi, \pi', k}$ is defined as follows. Suppose $\pi = (i_1, \dots, i_k)$ and $\pi' = (j_1, \dots, j_k)$.

$$\Pi_{\pi, \pi', k} = (\{v_1, \dots, v_k\}, \emptyset, \emptyset, \emptyset, (v_{i_1}, \dots, v_{i_k}), (v_{j_1}, \dots, v_{j_k}))$$

For example,

$$\Pi_{(2,1,3), (3,1,2), 3} = (\{v_1, v_2, v_3\}, \emptyset, \emptyset, \emptyset, (v_2, v_1, v_3), (v_3, v_1, v_2)) =$$

$$2 \bullet 2$$

$$1 \bullet 3$$

$$3 \bullet 1$$

This kind of hypergraph is called a *permutation hypergraph*. Note that it is overkill to use two permutations, as both can always be combined into one. For technical reasons however (the symmetry allows us to specify the inverse permutation by swapping π and π') we choose to use two.

Choose an $H \in \mathbf{L}_{C_2}$. Suppose $|V_H| = k$, $V = \{v_1, \dots, v_k\}$, and $H : m \rightarrow n$. Now H can always be sequentially decomposed as follows¹:

$$H = H_\pi \cdot H_\sigma \cdot H_{\pi'}.$$

Here $H_\sigma : m \rightarrow n$ is defined as follows. Let p_i indicate the number of occurrences of v_i in \mathbf{in}_H , and p'_i the number of occurrences of v_i in \mathbf{out}_H . Note that for all $1 \leq i \leq k$, $(p_i, p'_i) \neq (0, 0)$ as $H \in \mathbf{L}_B$.

$$H_\sigma = \sum_{i=1}^k \mathbf{1}_{p_i, p'_i}$$

This H_σ we now have is almost equal to H , albeit that \mathbf{in} and \mathbf{out} have been “sorted”: $\mathbf{in}_{H_\sigma} = \mathbf{sort}(\mathbf{in}_H)$ and $\mathbf{out}_{H_\sigma} = \mathbf{sort}(\mathbf{out}_H)$. This is illustrated by the following example:

$$H = (\{v_1, v_2, v_3, v_4\}, \emptyset, \emptyset, \emptyset, (v_1, v_4, v_3), (v_1, v_3, v_2, v_2)) =$$

¹In order to avoid confusion: the π , π' and σ in H_π , $H_{\pi'}$ and H_σ are just used as a subscript, and have no meaning of their own. So, the π in H_π is *not* a permutation. It is only used to give a hint that H_π will be defined as a permutation graph.

$$\begin{aligned}
 & 1 \bullet 1 \\
 & \bullet 3, 4 \\
 & 3 \bullet 2 \\
 & 2 \bullet
 \end{aligned}$$

Now:

$$\begin{aligned}
 H_\sigma &= \mathbf{1}_{1,1} + \mathbf{1}_{0,2} + \mathbf{1}_{1,1} + \mathbf{1}_{1,0} \\
 &= (\{v_1, v_2, v_3, v_4\}, \emptyset, \emptyset, \emptyset, (v_1, v_3, v_4), (v_1, v_2, v_2, v_3)) =
 \end{aligned}$$

$$\begin{aligned}
 & 1 \bullet 1 \\
 & \bullet 2, 3 \\
 & 2 \bullet 4 \\
 & 3 \bullet
 \end{aligned}$$

Note that $(v_1, v_3, v_4) = \mathbf{sort}((v_1, v_4, v_3))$ and $(v_1, v_2, v_2, v_3) = \mathbf{sort}((v_1, v_3, v_2, v_2))$ (where $v_i \leq v_j \iff i \leq j$). Now consider a permutation (a_1, \dots, a_m) of $(1, \dots, m)$ such that:

$$\mathbf{in}_H(a_1) \leq \dots \leq \mathbf{in}_H(a_m),$$

so we have $(\mathbf{in}_H(a_1), \dots, \mathbf{in}_H(a_m)) = \mathbf{sort}(\mathbf{in}_H)$. Similarly, consider a permutation (b_1, \dots, b_n) of $(1, \dots, n)$ such that:

$$\mathbf{out}_H(b_1) \leq \dots \leq \mathbf{out}_H(b_n),$$

so we have $(\mathbf{out}_H(b_1), \dots, \mathbf{out}_H(b_n)) = \mathbf{sort}(\mathbf{out}_H)$. Now define $H_\pi = \Pi_{(1, \dots, m), (a_1, \dots, a_m), m}$ and $H_{\pi'} = \Pi_{(b_1, \dots, b_n), (1, \dots, n), n}$. So, continuing the previous example we have: $(a_1, a_2, a_3) = (1, 3, 2)$ and $(b_1, b_2, b_3, b_4) = (1, 3, 4, 2)^2$, and so:

$$\begin{aligned}
 H_\pi &= (\{v_1, v_2, v_3\}, \emptyset, \emptyset, \emptyset, (v_1, v_2, v_3), (v_1, v_3, v_2)), \\
 H_{\pi'} &= (\{v_1, v_2, v_3, v_4\}, \emptyset, \emptyset, \emptyset, (v_1, v_3, v_4, v_2), (v_1, v_2, v_3, v_4)).
 \end{aligned}$$

²Note that these permutations need not be uniquely determined. For example, we could have chosen $(1, 4, 3, 2)$ instead of $(1, 3, 4, 2)$ for (b_1, b_2, b_3, b_4)

$$\begin{array}{ccc}
& 1 \bullet 1 & 1 \bullet 1 \\
H_\pi = & 2 \bullet 3 & H_{\pi'} = 4 \bullet 2 \\
& 3 \bullet 2 & 2 \bullet 3 \\
& & 3 \bullet 4
\end{array}$$

Now, applying the permutation (a_1, \dots, a_m) to \mathbf{in}_H results in $\mathbf{sort}(\mathbf{in}_H)$, and likewise, applying (b_1, \dots, b_n) to \mathbf{out}_H in $\mathbf{sort}(\mathbf{out}_H)$. As $\mathbf{in}_{H_\sigma} = \mathbf{sort}(\mathbf{in}_H)$, and $\mathbf{out}_{H_{\sigma'}} = \mathbf{sort}(\mathbf{out}_H)$, we now have $H = H_\pi \cdot H_\sigma \cdot H_{\pi'}$. This completes the proof that $\mathbf{L}_2 \xrightarrow{\cdot, +} \mathbf{L}_4 \cup \mathbf{L}_5$.

5.7 $\mathbf{L}_{C_4} \xrightarrow{\cdot, +} \mathbf{L}_{C_6}$

Define \mathbf{L}_{C_6} as follows:

$$\mathbf{L}_{C_6} = \{ U_0, \mathbf{1}_{0,1}, \mathbf{1}_{1,0}, \mathbf{1}_{1,2}, \mathbf{1}_{2,1}, \mathbf{X} \},$$

where $\mathbf{X} = (\{v_1, v_2\}, \emptyset, \emptyset, \emptyset, (v_1, v_2), (v_2, v_1))$. Graphically, \mathbf{L}_{C_6} looks as follows:

$$\left\{ \begin{array}{ccccccc}
& & & & & & 2 \bullet 1 \\
& & & & & & \\
\emptyset, & \bullet 1, & 1 \bullet, & 1 \bullet 1, 2, & 1, 2 \bullet 1, & & \\
& & & & & & 1 \bullet 2
\end{array} \right\}$$

The following induction steps now suffice to fully decompose \mathbf{L}_{C_4} into \mathbf{L}_{C_6} :

$$\begin{aligned}
\mathbf{1}_{1,1} &= \mathbf{1}_{1,2} \cdot \mathbf{1}_{2,1}, \\
\mathbf{1}_{m+1,1} &= (\mathbf{1}_{m,1} + \mathbf{1}_{1,1}) \cdot \mathbf{1}_{2,1} \quad \text{for } m \geq 1, \\
\mathbf{1}_{1,n+1} &= \mathbf{1}_{1,2} \cdot (\mathbf{1}_{1,n} + \mathbf{1}_{1,1}) \quad \text{for } n \geq 1, \\
\mathbf{1}_{m,n} &= \mathbf{1}_{m,1} \cdot \mathbf{1}_{1,n} \quad \text{for } m, n \in \mathbb{N}.
\end{aligned}$$

Therefore, $\mathbf{L}_{C_4} \xrightarrow{\cdot, +} \mathbf{L}_{C_6}$.

5.8 $L_{C_5} \xrightarrow{\cdot^+} L_{C_6}$

Choose an $H \in L_5$. Suppose $|V_H| = n$. By definition, we now have that \mathbf{out}_H is a permutation π of \mathbf{in}_H . From combinatorics/algebra it is well known³ that every permutation π can be decomposed into permutations π_1, \dots, π_k where each of the permutations π_i only swaps two neighboring elements. Using these permutations π_i we can sequentially decompose H :

$$H = \Pi_{(1, \dots, n), \pi, n} = \prod_{i=1}^k \Pi_{(1, \dots, n), \pi_i, n}.$$

These hypergraphs $\Pi_{(1, \dots, n), \pi_i, n}$ are obviously of the form:

$$\mathbf{1}_{1,1} + \dots + \mathbf{1}_{1,1} + \mathbf{X} + \mathbf{1}_{1,1} + \dots + \mathbf{1}_{1,1},$$

so we have fully decomposed L_{C_5} into L_{C_6} . Therefore, $L_{C_5} \xrightarrow{\cdot^+} L_{C_6}$. As a consequence of this, and by the previous sections, $L_{C_2} \xrightarrow{\cdot^+} L_{C_6}$.

5.9 $L_B \xrightarrow{\cdot^+} L_C$

We define L_C as follows: $L_C = L_{C_1} \cup L_{C_6}$. By the transitivity of $\xrightarrow{\cdot^+}$, and because $\xrightarrow{\cdot}$ implies $\xrightarrow{\cdot^+}$, the results of the previous sections now justify the conclusion $L_B \xrightarrow{\cdot^+} L_C$.

5.10 Conclusions

In trying to decompose \mathbf{HGR} a result (not the strongest) we got with sequential decomposition was

$$\mathbf{HGR} \xrightarrow{\cdot} L_B.$$

This set L_B is called the *sequential pseudo base set*, because all graphs in \mathbf{HGR} can be built from it, using just sequential composition. Note that it is *not* finite, and by no means minimal, as L_B can be further sequentially decomposed. (for example to $L_{C_2} \cup L_{C_3}$, and further). For these reason, it is only a *pseudo* base. There does not exist a “real” sequential base set (finite, minimal), as \mathbf{HGR} contains hypergraphs of type $(m \rightarrow n)$ for arbitrarily large $m, n \in \mathbb{N}$, and by sequential composition it is not possible to “pump up” the type of hypergraph.

Using full decomposition we ultimately arrived at

$$\mathbf{HGR} \xrightarrow{\cdot^+} L_C.$$

³See for example Knuth, [Knu73, §5.1], or Herstein, [Her75, §2.10].

This set \mathbf{L}_C is called the *full base set* because all graphs in \mathbf{HGR} can be built from it, using both sequential and parallel composition. Contrary to the sequential pseudo base set, at least for a fixed edge label alphabet, the full base set *is* minimal: it cannot be further decomposed (into a proper subset). Furthermore, for a fixed edge label alphabet, the full base set is finite. Formally, by \mathbf{L}_C “for a fixed edge label alphabet Δ ”, we mean $\mathbf{L}_C \cap \mathbf{HGR}(\Delta)$. That this set is indeed minimal under this condition can be easily understood from the following considerations:

- For a fixed label alphabet Δ , all hypergraphs Ω_a , for $a \in \Delta$, are really needed, because without all of them, we would be unable to generate hypergraphs that contain hyperedges labeled a for all $a \in \Delta$,
- U_0 is really needed, as it obviously cannot be built from any other hypergraph other than itself,
- $\mathbf{1}_{0,1}$ and $\mathbf{1}_{1,0}$ are really needed, as without them we could *not* generate edge-less hypergraphs with internal nodes. Sketch of proof: in all other edge-less hypergraphs in \mathbf{L}_C , i.e. $\{U_0, \mathbf{1}_{1,2}, \mathbf{1}_{2,1}, \mathbf{X}\}$, all nodes are both input and output nodes. This property clearly remains invariant under both sequential and parallel composition. End of sketch. That $\mathbf{1}_{0,1}$ and $\mathbf{1}_{1,0}$ are *both* needed can be shown by similar arguments.

Intuitively, $\mathbf{1}_{0,1}$ and $\mathbf{1}_{1,0}$ are needed to be able to “get rid of external nodes”.

- $\mathbf{1}_{1,2}$ and $\mathbf{1}_{2,1}$ are really needed, as without them we could *only* generate identification-free hypergraphs. Sketch of proof: all other hypergraphs in \mathbf{L}_C are identification-free, and this property remains invariant under both parallel composition (trivial) and sequential composition (almost trivial). End of sketch. Again, the fact that $\mathbf{1}_{1,2}$ and $\mathbf{1}_{2,1}$ are both needed can be show by similar arguments.

Intuitively, $\mathbf{1}_{1,2}$ and $\mathbf{1}_{2,1}$ are needed to be able to “split up external nodes”.

- \mathbf{X} is really needed, as without it we could *only* generate hypergraphs H such that for all $1 \leq i < j \leq \#\mathbf{in}(H)$, $1 \leq i' < j' \leq \#\mathbf{in}(H)$, such that $\mathbf{in}_H(i) \neq \mathbf{in}_H(j)$ and $\mathbf{out}_H(i') \neq \mathbf{out}_H(j')$, not *both* $\mathbf{in}_H(i) = \mathbf{out}_H(j')$ and $\mathbf{in}_H(j) = \mathbf{out}_H(i')$. Sketch of proof: this property holds for all other hypergraphs in \mathbf{L}_C , and remains invariant under both parallel composition (trivial), and sequential composition (almost trivial). End of sketch.

Intuitively, \mathbf{X} is needed to be able to “permute external nodes”.

We do not know whether $\mathbf{HGR}(\Delta)$ can be fully decomposed into another minimal set that has less elements, but that eventuality seems highly unlikely.

6

Fold: to bend over or double up so that one part lies on another part.

— *The American Heritage Dictionary*

Folds and flips

In this chapter we will define four natural functions on hypergraphs: *fold*, *backfold*, *flip*, and *split*. They exhibit several nice properties that make them very suitable for building the constructions needed to prove two classes of hypergraph languages equal.

6.1 Definition

Put $H = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out}) : m \rightarrow n$, and define the following unary operations *flip*, *fold*, *backfold*, and *split* on H :

$$\mathbf{flip}(H) = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{out}, \mathbf{in}) : n \rightarrow m,$$

$$\mathbf{fold}(H) = (V, E, \mathbf{nod}, \mathbf{lab}, \lambda, \mathbf{in} \cdot \mathbf{out}) : 0 \rightarrow m + n,$$

$$\mathbf{backfold}(H) = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in} \cdot \mathbf{out}, \lambda) : m + n \rightarrow 0.$$

Put $H = (V, E, \mathbf{nod}, \mathbf{lab}, (v_1, \dots, v_m), (v_{m+1}, \dots, v_{m+n})) : m \rightarrow n$, and define the unary operation $\mathbf{split}_{p,q}$ on H for all $p, q \in \mathbb{N}$, $m + n = p + q$:

$$\mathbf{split}_{p,q}(H) = (V, E, \mathbf{nod}, \mathbf{lab}, (v_1, \dots, v_p), (v_{p+1}, \dots, v_{p+q})) : p \rightarrow q.$$

Recall that λ denotes the empty sequence, and \cdot denotes the concatenation of sequences. Note that these operations only manipulate the input and output sequences, the V , E , \mathbf{nod} , and \mathbf{lab} components are not changed.

We will use the **flip**, **fold**, **backfold**, and **split** as verbs where convenient, for example as in “(6.33) is proved by flipping both sides of (6.32)”, i.e., by applying the operation **flip** to both sides of the equation.

6.2 Basic properties

These definitions lead to, amongst others, the following properties. Let H be of type $(m \rightarrow n)$, and $p, q, p', q' \in \mathbb{N}$ such that $p + q = p' + q' = m + n$.

$$\mathbf{split}_{m,n}(H) = H, \quad (6.1)$$

$$\mathbf{flip}(\mathbf{flip}(H)) = H, \quad (6.2)$$

$$\mathbf{flip}(\mathbf{fold}(H)) = \mathbf{backfold}(H), \quad (6.3)$$

$$\mathbf{flip}(\mathbf{backfold}(H)) = \mathbf{fold}(H), \quad (6.4)$$

$$\mathbf{fold}(\mathbf{fold}(H)) = \mathbf{fold}(H), \quad (6.5)$$

$$\mathbf{fold}(\mathbf{backfold}(H)) = \mathbf{fold}(H), \quad (6.6)$$

$$\mathbf{fold}(\mathbf{split}_{p,q}(H)) = \mathbf{fold}(H), \quad (6.7)$$

$$\mathbf{backfold}(\mathbf{fold}(H)) = \mathbf{backfold}(H), \quad (6.8)$$

$$\mathbf{backfold}(\mathbf{backfold}(H)) = \mathbf{backfold}(H), \quad (6.9)$$

$$\mathbf{backfold}(\mathbf{split}_{p,q}(H)) = \mathbf{backfold}(H), \quad (6.10)$$

$$\mathbf{split}_{p,q}(\mathbf{fold}(H)) = \mathbf{split}_{p,q}(H), \quad (6.11)$$

$$\mathbf{split}_{p,q}(\mathbf{backfold}(H)) = \mathbf{split}_{p,q}(H), \quad (6.12)$$

$$\mathbf{split}_{p,q}(\mathbf{split}_{p',q'}(H)) = \mathbf{split}_{p,q}(H). \quad (6.13)$$

There are no simpler expressions in terms of **flip**, **fold**, **backfold**, and **split** for the expressions $\mathbf{fold}(\mathbf{flip}(H))$, $\mathbf{backfold}(\mathbf{flip}(H))$, $\mathbf{split}_{p,q}(\mathbf{flip}(H))$, and $\mathbf{flip}(\mathbf{split}_{p,q}(H))$. Furthermore we have, directly from the definitions, the following properties:

$$\#\mathbf{in}(\mathbf{flip}(H)) = \#\mathbf{out}(H), \quad (6.14)$$

$$\#\mathbf{out}(\mathbf{flip}(H)) = \#\mathbf{in}(H), \quad (6.15)$$

$$\#\mathbf{in}(\mathbf{fold}(H)) = 0, \quad (6.16)$$

$$\#\mathbf{out}(\mathbf{fold}(H)) = \#\mathbf{in}(H) + \#\mathbf{out}(H), \quad (6.17)$$

$$\#\mathbf{in}(\mathbf{backfold}(H)) = \#\mathbf{in}(H) + \#\mathbf{out}(H), \quad (6.18)$$

$$\#\mathbf{out}(\mathbf{backfold}(H)) = 0, \quad (6.19)$$

$$\#\mathbf{in}(\mathbf{split}_{p,q}(H)) = p, \quad (6.20)$$

$$\#\text{out}(\text{split}_{p,q}(H)) = q. \quad (6.21)$$

As previously noted, the **flip**, **fold**, **backfold**, and **split** operations only manipulate the input and output sequences (recall from Section 3.6 that \equiv_{io} is an equivalence relation):

$$H \equiv_{\text{io}} \text{flip}(H) \equiv_{\text{io}} \text{fold}(H) \equiv_{\text{io}} \text{backfold}(H) \equiv_{\text{io}} \text{split}_{p,q}(H). \quad (6.22)$$

The **flip** operation is linked to sequential and parallel composition in the following way:

$$\text{flip}(G \cdot H) = \text{flip}(H) \cdot \text{flip}(G), \quad (6.23)$$

$$\text{flip}(G + H) = \text{flip}(G) + \text{flip}(H), \quad (6.24)$$

$$\text{flip}(U_n) = U_n. \quad (6.25)$$

As it turns out, $\text{split}_{p,q}$ can be expressed in terms of **fold** and **backfold**, and vice versa. So, in a sense, either **split**, or **fold** and **backfold** are superfluous. In practice we need both of them, as some concepts are more elegantly expressed in folds and backfolds, others in splits. The relations between them are as follows:

$$\text{fold}(H) = \text{split}_{0, \#\text{in}(H) + \#\text{out}(H)}(H), \quad (6.26)$$

$$\text{backfold}(H) = \text{split}_{\#\text{in}(H) + \#\text{out}(H), 0}(H), \quad (6.27)$$

$$\text{split}_{p,q}(H) = (U_p + \text{fold}(U_q)) \cdot (\text{backfold}(H) + U_q). \quad (6.28)$$

We extend **split** to apply to classes of hypergraph languages. Let \mathbf{K} be any class of hypergraph languages.

$$\text{split}(\mathbf{K}) = \{ \text{split}_{p,q}(L) \mid L \in \mathbf{K} \text{ and } p, q \in \mathbb{N}, p + q = \#\text{in}(L) + \#\text{out}(L) \} \quad (6.29)$$

Note that now, by (6.26) and (6.27), we have:

$$\text{fold}(\mathbf{K}) \subseteq \text{split}(\mathbf{K}) \quad (6.30)$$

$$\text{backfold}(\mathbf{K}) \subseteq \text{split}(\mathbf{K}) \quad (6.31)$$

The above properties are easily verified by expanding the respective definitions.

6.3 Derived properties

A few properties that can be derived from the basic properties are listed below. They are illustrated in Appendix B, and will be used in Section 8.1 (page 57) and Section 8.2 (page 64) where they are needed in an induction proof.

$$\text{fold}(H_2) \cdot (\text{flip}(H_1) + H_3) = \text{fold}(H_1 H_2 H_3), \quad (6.32)$$

$$(H_1 + \mathbf{flip}(H_3)) \cdot \mathbf{backfold}(H_2) = \mathbf{backfold}(H_1H_2H_3), \quad (6.33)$$

both under the condition that $H_1H_2H_3$ is defined.

$$(H_1 + \mathbf{fold}(H_3)) \cdot (\mathbf{backfold}(H_2) + H_4) = H_1H_2H_3H_4, \quad (6.34)$$

$$(\mathbf{fold}(H_2) + \mathbf{flip}(H_4)) \cdot (\mathbf{flip}(H_1) + \mathbf{backfold}(H_3)) = \mathbf{flip}(H_1H_2H_3H_4), \quad (6.35)$$

under the condition that $H_1H_2H_3H_4$ is defined.

$$(\mathbf{fold}(H_2) + \mathbf{fold}(H_4)) \cdot (\mathbf{flip}(H_1) + \mathbf{backfold}(H_3) + H_5) = \mathbf{fold}(H_1H_2H_3H_4H_5), \quad (6.36)$$

$$(H_1 + \mathbf{fold}(H_3) + \mathbf{flip}(H_5)) \cdot (\mathbf{backfold}(H_2) + \mathbf{backfold}(H_4)) = \mathbf{backfold}(H_1H_2H_3H_4H_5), \quad (6.37)$$

under the condition that $H_1H_2H_3H_4H_5$ is defined.

Note that by flipping both sides of (6.32), (6.34), or (6.36), we get (6.33), (6.35), or (6.37) respectively, and vice versa. The above six properties can be proved from the basic properties, albeit in a highly nontrivial way. See also Section B.1.

Be aware of bugs in the above code; I have only proved it correct, not tried it.

— *Donald Ervin Knuth*

7

Interpretation

In this chapter we devise a method to interpret a string as a hypergraph. As this method can be extended to interpret string languages as hypergraph languages, and classes of string languages as classes of hypergraph languages, we can instantly define a lot of classes of hypergraph languages. For example, from LIN, all linear languages, we instantly derive **Int**(LIN), all linear languages under interpretation as hypergraph languages.

7.1 Definition of an interpreter

Let Σ be a typed alphabet, and Δ a ranked alphabet. Let h be a function, $h : \Sigma \rightarrow \mathbf{HGR}(\Delta)$, that is type preserving, that is, it must satisfy the condition that for every $a \in \Sigma$ the hypergraph $h(a)$ is of the same type as the symbol a . Let w be a string from Σ^* , $w = a_1 \dots a_n$. Now extend h to Σ^* in the following way: $h(w) = h(a_1) \dots h(a_n)$. Furthermore, if $w = (\lambda, n \rightarrow n)$, we define $h((\lambda, n \rightarrow n))$ as U_n . By the definition of h and \cdot we now have, for all $v, w \in \Sigma^*$:

$$h(vw) = h(v)h(w), \tag{7.1}$$

provided vw is defined. By the associativity of \cdot on both strings and hypergraphs this generalizes to:

$$h\left(\prod_{k=1}^n w_k\right) = \prod_{k=1}^n h(w_k), \tag{7.2}$$

provided, again, $\prod_{k=1}^n w_k$ is defined.

The 3-tuple (Σ, Δ, h) is called an *interpreter*, and h is called the *interpretation function*. For an interpreter I , we will denote its three components by Σ_I , Δ_I , and h_I respectively.

Let L be a typed string language over a typed alphabet Σ , and $I = (\Sigma, \Delta, h)$ an interpreter. Now, I is called an *interpreter for L* .

7.2 Definition of Int

Let L be a typed string language and I an interpreter for L . We now define $\mathbf{Int}_I(L)$, the *interpretation of L under I* to be the hypergraph language:

$$\mathbf{Int}_I(L) = \{ h_I(w) \mid w \in L \}. \quad (7.3)$$

Note that $\mathbf{Int}_I(L) \subseteq \mathbf{HGR}(\Delta_I)$. We can extend \mathbf{Int} to interpret one specific language as a class of hypergraph languages, without specifying an interpreter. The *generic interpretation* of L , denoted $\mathbf{Int}(L)$ is defined as follows:

$$\mathbf{Int}(L) = \{ \mathbf{Int}_I(L) \mid I \text{ an interpreter for } L \}. \quad (7.4)$$

We extend \mathbf{Int} to *classes* of languages. Let K be a class of ordinary string languages. We now define $\mathbf{Int}(K)$, the *generic class interpretation* of K , as follows:

$$\mathbf{Int}(K) = \{ \mathbf{Int}_I(L) \mid L \in L_\tau(K), I \text{ an interpreter for } L \}. \quad (7.5)$$

Recall that $L \in L_\tau(K)$ means that the underlying language of L is in K . By the definition of an interpreter $\mathbf{Int}_I(L)$ is always of the type $(\#\mathbf{in}(L) \rightarrow \#\mathbf{out}(L))$. A generic interpretation $\mathbf{Int}(L)$ or a class interpretation $\mathbf{Int}(K)$ is always of mixed type. Finally, we define three different kinds of *typed class interpretation* (let $m, n \in \mathbb{N}$):

$$\begin{aligned} \mathbf{Int}_{m \rightarrow}(K) &= \{ \mathbf{L} \in \mathbf{Int}(K) \mid \#\mathbf{in}(\mathbf{L}) = m \}, \\ \mathbf{Int}_{\rightarrow n}(K) &= \{ \mathbf{L} \in \mathbf{Int}(K) \mid \#\mathbf{out}(\mathbf{L}) = n \}, \\ \mathbf{Int}_{m \rightarrow n}(K) &= \{ \mathbf{L} \in \mathbf{Int}(K) \mid \#\mathbf{in}(\mathbf{L}) = m \text{ and } \#\mathbf{out}(\mathbf{L}) = n \}. \end{aligned}$$

These last three definitions lead immediately to the following two sequences of inclusions:

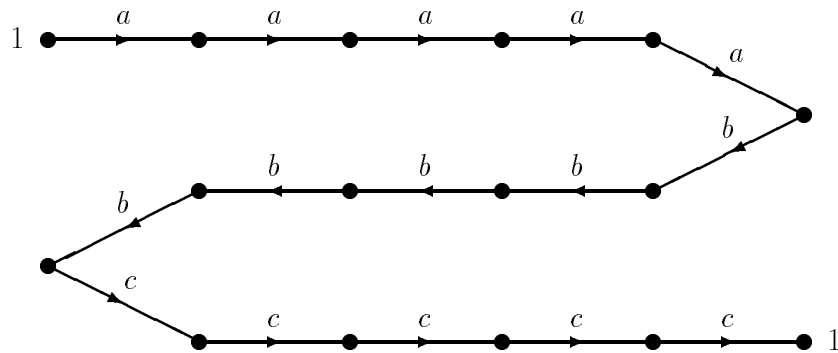
$$\mathbf{Int}_{m \rightarrow n}(K) \subseteq \mathbf{Int}_{m \rightarrow}(K) \subseteq \mathbf{Int}(K), \quad (7.6)$$

$$\mathbf{Int}_{m \rightarrow n}(K) \subseteq \mathbf{Int}_{\rightarrow n}(K) \subseteq \mathbf{Int}(K). \quad (7.7)$$

Finally, note that for an ordinary alphabet Σ , and an ordinary language L over Σ , by (3.4) and (7.1), we can express the hypergraph language $\mathbf{gr}(L)$ of string graphs in terms of \mathbf{Int} :

$$\mathbf{gr}(L) = \mathbf{Int}_I(L), \quad (7.8)$$

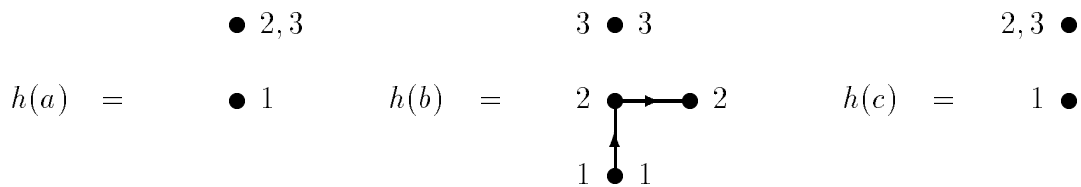
Now, $h(ab^n c) = \mathbf{gr}(a^n b^n c^n)$. For example, for $n = 5$, $h(abbbbbc)$ is:



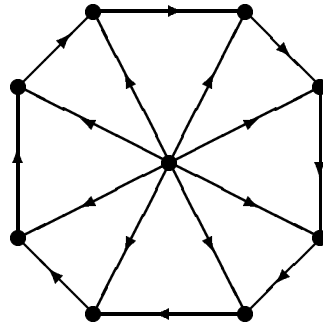
Therefore, $\mathbf{Int}_I(L) = \{ \mathbf{gr}(a^n b^n c^n) \mid n \in \mathbb{N} \} = \mathbf{gr}(\{ a^n b^n c^n \mid n \in \mathbb{N} \})$ (all of which are of type $(1 \rightarrow 1)$). This is somewhat surprising, as the string language $\{ a^n b^n c^n \mid n \in \mathbb{N} \}$ is the classic example of a language that is not context-free!

Example 2

As second example, an interpretation that is slightly more difficult. We take the same right-linear language $L = \{ ab^n c \mid n \in \mathbb{N} \}$, over the same alphabet Σ , and the interpreter $I = (\Sigma, \Delta, h)$ for L , where this time $\Delta = \{a\}$, and h defined as follows:



We omit the edge labels, as they are all a . Now, $h(ab^n c)$ is the “clockwise spinning wheel with n spokes”. To understand what we mean by this, look at the case $n = 8$. Then $h(abbbbbb c)$ is:



Therefore, $\mathbf{Int}_I(L)$ consists of all such clockwise spinning wheels with n spokes, for all $n \in \mathbb{N}$ (all these hypergraphs are simple).

Example 3

Let L be the ordinary string language $\{a\}$, and $K = \{L\}$ the class that only contains L . Now, $\mathbf{Int}(K)$ is the class of all singleton hypergraph languages. This can be easily proved. First, for every interpreter I for L , $\mathbf{Int}_I(L)$ obviously only contains *one* hypergraph (namely $h_I(a)$), so it is a singleton hypergraph language. Secondly, for every singleton hypergraph language $\{H\}$, we can construct an interpreter I for L such that $\mathbf{Int}_I(L) = \{H\}$, namely that interpreter that has $a \mapsto H$ as interpretation function.

7.4 Edge Normal Form

Using the results on decomposition from Chapter 5 we can derive a normal form for interpreters. This normal form is called *Edge Normal Form*, or ENF for short. An interpreter $I = (\Sigma, \Delta, h)$ is in Edge Normal Form if for all $a \in \Sigma$, the hypergraph $h(a)$ contains at most one edge, and no internal nodes:

$$\forall_{a \in \Sigma} \left(|E_{h(a)}| \leq 1 \text{ and } V_{h(a)} = \mathbf{in}_{h(a)} \cup \mathbf{out}_{h(a)} \right).$$

The following theorem intuitively says that, for almost all classes K , all hypergraph languages in $\mathbf{Int}(K)$ can be obtained using an interpreter in Edge Normal Form.

Edge Normal Form Theorem:

Let K be a class of string languages that is closed under λ -free homomorphisms. Now for all $L \in L_\tau(K)$ and $I = (\Sigma, \Delta, h)$ an interpreter for L there exists an $L' \in L_\tau(K)$ and an

interpreter $I' = (\Sigma', \Delta', h')$ for L' such that I' is in ENF and $\mathbf{Int}_{I'}(L') = \mathbf{Int}_I(L)$.

Proof. As proved in sections 5.2 and 5.3, every hypergraph H can be sequentially decomposed in hypergraphs H_1, \dots, H_n , where H_i contains at most one edge, and no internal nodes, for $1 \leq i \leq n$. For every hypergraph $h(a)$, for $a \in \Sigma$, choose such a decomposition:

$$h(a) = \prod_{i=1}^{n_a} H_{a,i}.$$

Here n_a denotes the length of the decomposition for $h(a)$. Let $\langle a, i \rangle$ denote an element from the typed alphabet Σ' , which is defined as follows:

$$\Sigma' = \{ \langle a, i \rangle \mid a \in \Sigma \text{ and } 1 \leq i \leq n_a \}.$$

Here the type of a symbol $\langle a, i \rangle$ is defined as $(\# \mathbf{in}(H_{a,i}) \rightarrow \# \mathbf{out}(H_{a,i}))$. Now define the typed language $L' = \sigma(L)$ over Σ' , where $\sigma : \Sigma \rightarrow \Sigma'$ denotes the following λ -free homomorphism: for all $a \in \Sigma$, $\sigma(a) = \prod_{i=1}^{n_a} \langle a, i \rangle$. Define $I' = (\Sigma', \Delta', h')$, where $\Delta' = \Delta$, and for all $\langle a, i \rangle \in \Sigma'$, $h'(\langle a, i \rangle) = H_{a,i}$. Note that by this definition (keep in mind the fixed decompositions) for all $a \in \Sigma$, $h'(\sigma(a)) = h(a)$. We now have: $\mathbf{Int}_{I'}(L') = \mathbf{Int}_I(L)$. Proof:

$$\begin{aligned} \mathbf{Int}_{I'}(L') &\stackrel{(7.3)}{=} \\ &\{ h'(w) \mid w \in L' \} \stackrel{(\text{definition of } L')}{=} \\ &\{ h'(w) \mid w \in \sigma(L) \} \stackrel{(\text{set theory})}{=} \\ &\{ h'(\sigma(v)) \mid v \in L \} \stackrel{(\text{definition of } h', \sigma, \text{ and } h)}{=} \\ &\{ h(v) \mid v \in L \} \stackrel{(7.3)}{=} \\ &\mathbf{Int}_I(L). \end{aligned}$$

This completes the proof of the Edge Normal Form theorem.

7.5 Existence of isomorphic copies

In a sense, every hypergraph language that can be obtained by interpretation, can be so in an infinite number of ways. We will formally express this by the following theorem.

Isomorphic Copies Theorem:

For a class K that is closed under isomorphism, a typed language $L \in K$, and an interpreter I for L , we can always find an $L' \in K$, and an interpreter I' for L' , such that $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$, and L disjoint with L' .

Proof. Given a hypergraph language \mathbf{L} over a ranked alphabet Δ , a typed string language L over a typed alphabet Σ , and an interpreter $I = (\Sigma, \Delta, h)$ for L , such that $\mathbf{Int}_I(L) = \mathbf{L}$, there exists a typed string language L' over a typed alphabet Σ' such that $L' = f(L)$ for some isomorphism f , and an interpreter $I' = (\Sigma', \Delta', h')$ for L' , such that $\Sigma \cap \Sigma' = \emptyset$ and $\mathbf{Int}_{I'}(L') = \mathbf{L}$:

$$\forall_{\substack{L \in \mathbf{HGR}(\Delta) \\ L \subseteq \Sigma^* \\ I = (\Sigma, \Delta, h)}} \left((\mathbf{Int}_I(L) = \mathbf{L}) \implies \exists_{\substack{f: \Sigma \rightarrow \Sigma' \text{ (bijective)} \\ L' \subseteq \Sigma'^* \\ I' = (\Sigma', \Delta', h')}} \left(\begin{array}{l} \Sigma \cap \Sigma' = \emptyset \wedge \\ f(L) = L' \wedge \\ \mathbf{Int}_{I'}(L') = \mathbf{L} \end{array} \right) \right). \quad (7.10)$$

This can be proved by taking the following Σ' , f , L' , Δ' , and h' :

$$\begin{aligned} \Sigma' &= \{ (a', m \rightarrow n) \mid (a, m \rightarrow n) \in \Sigma \}, \\ f((a, m \rightarrow n)) &= (a', m \rightarrow n), \text{ for all } (a, m \rightarrow n) \in \Sigma, \\ L' &= \{ a'_1 \dots a'_n \mid a_1 \dots a_n \in L \}, \\ \Delta' &= \Delta, \\ h'(a') &= h(a). \end{aligned}$$

Obviously $L' = f(L)$. Now for given L , I , and \mathbf{L} such that $\mathbf{Int}_I(L) = \mathbf{L}$ we have:

$$\begin{aligned} \mathbf{Int}_{I'}(L') &\stackrel{(7.3)}{=} \\ &\{ h'(w') \mid w' \in L' \} \stackrel{(\text{rewriting } w' \text{ as symbols})}{=} \\ &\{ h'(a'_1 \dots a'_n) \mid a'_1 \dots a'_n \in L' \} \stackrel{(7.2)}{=} \\ &\{ h'(a'_1) \dots h'(a'_n) \mid a'_1 \dots a'_n \in L' \} \stackrel{(\text{definition of } h, h', \text{ and } L')}{=} \\ &\{ h(a_1) \dots h(a_n) \mid a_1 \dots a_n \in L \} \stackrel{(7.2)}{=} \\ &\{ h(a_1 \dots a_n) \mid a_1 \dots a_n \in L \} \stackrel{(\text{rewriting symbols as } w)}{=} \\ &\{ h(w) \mid w \in L \} \stackrel{(7.3)}{=} \\ &\mathbf{Int}_I(L) \stackrel{(\text{by definition})}{=} \\ &\mathbf{L}. \end{aligned}$$

This completes the proof of (7.10), from which the Isomorphic Copies Theorem follows as a corollary.

7.6 Bounded degree implies bounded cutwidth

By the nature of interpretation, some kinds of hypergraph languages are inherently impossible to form. As an example of such a limitation, for all $\mathbf{Int}_I(L)$ that only contain

ordinary graphs, we have that if $\mathbf{Int}_I(L)$ is of bounded degree, necessarily $\mathbf{Int}_I(L)$ is also of bounded cutwidth (see Section 3.4):

Degree versus Cutwidth Theorem:

$$\forall \mathbf{Int}_I(L) \text{ that only contain ordinary graphs } \mathbf{Int}_I(L) \text{ of bounded degree} \implies \mathbf{Int}_I(L) \text{ of bounded cutwidth. (7.11)}$$

Proof. Denote $\mathbf{Int}_I(L)$ by \mathbf{L} , with L strictly over some typed alphabet Σ , and $I = (\Sigma, \Delta, h)$ an interpreter for L . Let d be the bound on $\mathbf{deg}(\mathbf{L})$, and note that by the definition of interpretation, the definition of $\mathbf{deg}(\mathbf{L})$, and (4.6), now also $\mathbf{deg}(h(a)) \leq d$ for all $a \in \Sigma$. Define μ as the maximum number of nodes in any interpreted symbol:

$$\mu = \mathbf{max} \left(\left\{ |V_{h(a)}| \mid a \in \Sigma \right\} \right).$$

Now define the following property P of $H \in \mathbf{HGR}(\Delta)$:

$$\begin{aligned} &P(H) \\ &\iff \end{aligned}$$

There exists a linear layout $w_1, \dots, w_p, w_{p+1}, \dots, w_{p+q}$ of H (formally: $|V_H| = p+q$ for some $p, q \in \mathbb{N}$, and there exists a linear layout f for H such that $f(w_i) = i$ for all $1 \leq i \leq p+q$) such that the following three conditions hold:

1. $\mathbf{cw}(H, f) \leq d\mu$,
2. $\{w_1, \dots, w_p\} = V_H - \mathbf{out}_H$,
3. $\{w_{p+1}, \dots, w_{p+q}\} = \mathbf{out}_H$.

In words: $P(H)$ means that H has a linear layout with cutwidth $\leq d\mu$, such that all \mathbf{out} nodes come at the end.

We are now going to prove that:

$$\forall H \in \mathbf{L} \mathbf{cw}(H) \leq d\mu. \quad (7.12)$$

Let $H \in \mathbf{L}$. If $H = U_n$ for some $n \in \mathbb{N}$, $\mathbf{cw}(H) = 0$, and we are done (this case arises if $(\lambda, n \rightarrow n) \in L$, because then $h((\lambda, n \rightarrow n)) = U_n$). Otherwise, by (7.3) and (7.2), there exists a string $a_1 \dots a_n \in L$, for some $n \geq 1$ such that:

$$H = h(a_1 \dots a_n) = h(a_1) \dots h(a_n).$$

So in order to prove $\mathbf{cw}(H) \leq d\mu$, it suffices to prove $P(H)$, or, $P(h(a_1) \dots h(a_n))$. We will prove this by proving the following proposition $Q(k)$, which is defined for $1 \leq k \leq n$, for $k = n$:

$$Q(k) \iff P(h(a_1) \dots h(a_k)).$$

We will proceed by induction on k .

Induction basis, $k = 1$:

$Q(1)$ ($\iff P(h(a_1))$), is trivially fulfilled by (3.2), because $\mathbf{deg}(h(a_1)) \leq d$, and $|V_{h(a_1)}| \leq \mu$, and therefore $\mathbf{cw}(h(a_1), f) \leq \frac{1}{2}d\mu \leq d\mu$ for any layout f for $h(a_1)$. Hence, a layout where the **out** nodes of $h(a_1)$ come last will satisfy all three conditions in $P(h(a_1))$. Such a layout obviously exists, which completes the induction basis.

Induction step, $k = m + 1$:

Assuming $Q(m)$ ($\iff P(h(a_1) \dots h(a_m))$), where $1 \leq m < n$, we are now going to prove $Q(m + 1)$ ($\iff P(h(a_1) \dots h(a_{m+1}))$). This is done as follows. Note that $h(a_1) \dots h(a_{m+1}) = (h(a_1) \dots h(a_m)) \cdot h(a_{m+1})$. Look at the conditions in the definition of $P(h(a_1) \dots h(a_{m+1}))$. Take as linear layout for $h(a_1) \dots h(a_{m+1})$ a layout that starts with w_1, \dots, w_p as meant in $P(h(a_1) \dots h(a_m))$. As none of these nodes were **out** nodes of $h(a_1) \dots h(a_m)$, no identification has taken place on them during the concatenation with $h(a_{m+1})$, and consequently no new edges have become incident with them. Therefore, this first part of the layout still has cutwidth $\leq d\mu$.

The remainder of the layout can be chosen at wish, under the restriction that the **out** nodes of $h(a_{m+1})$ come last. As this second part of the layout contains at most μ nodes (by the definition of concatenation), and because $\mathbf{deg}(h(a_1) \dots h(a_{m+1})) \leq d$ (by (4.6)), there can be at most $d\mu$ edges running either within this second part, or between this second part and the first part (formally: there can be at most $d\mu$ edges that are incident with a node in the second part). Therefore, the cut between the first and the second part of the layout, and the cuts within the second part, all have width $\leq d\mu$.

The layout thus constructed, as can be easily seen, satisfies the conditions in the definition of $P(h(a_1) \dots h(a_{m+1}))$. Therefore, $P(h(a_1) \dots h(a_{m+1}))$ holds, and consequently $Q(m + 1)$ holds also. By induction we have now proven that $Q(n)$ holds, and as an immediate consequence $\mathbf{cw}(H) \leq d\mu$. This completes the proof of (7.12), so $\mathbf{cw}(\mathbf{L}) \leq d\mu$.

This theorem puts a clear restriction on the kind of hypergraph languages \mathbf{L} that can be generated using interpretation: an \mathbf{L} with bounded degree, but no bounded cutwidth is fundamentally impossible. An example of such a language \mathbf{L} is the language of all ordinary

graphs that form binary trees. As Lengauer proved in 1982 [Len82], a complete binary tree of depth $2k$ has cutwidth $k + 1$ (for $k \geq 1$). As \mathbf{L} contains all complete binary trees, this means that \mathbf{L} is not of bounded cutwidth. Since \mathbf{L} is of bounded degree, it cannot be generated by means of interpretation.

As a final remark, let it be noted that something quite similar has been done for eNCE graph grammars (a node rewriting based approach), by Engelfriet and Leih in 1989 (see [EL89, §5]).

We will not give formal correctness proofs of our constructions, because we feel that these would only obscure the underlying intuitions.

— *Joost Engelfriet and George Leih*

8

Power of interpretation

In this chapter we will examine RLIN, LIN, and DB under interpretation. As it turns out: $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{LIN}) = \mathbf{Int}(\mathbf{DB})$. However, $\mathbf{Int}(\mathbf{RLIN}) \subsetneq \mathbf{Int}(\mathbf{CF})$. We will also examine $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K)))$, and will find that under a few weak conditions on K , it is equal to $\mathbf{Int}(K)$. From these results, we will be able to prove two theorems that nicely indicate the power of interpretation.

8.1 $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{LIN})$

Firstly, to prove $\mathbf{Int}(\mathbf{RLIN}) \subseteq \mathbf{Int}(\mathbf{LIN})$ we only have to observe that $\mathbf{RLIN} \subseteq \mathbf{LIN}$. So for any $L \in L_\tau(\mathbf{RLIN})$ and I an interpreter for L , there exists an $L' \in L_\tau(\mathbf{LIN})$ and an I' an interpreter for L' such that $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$ (namely: $L' = L$ and $I' = I$). This proves that $\mathbf{Int}(\mathbf{RLIN}) \subseteq \mathbf{Int}(\mathbf{LIN})$. As a matter of fact this proof extends to any $K_1 \subseteq K_2$:

$$(K_1 \subseteq K_2) \implies (\mathbf{Int}(K_1) \subseteq \mathbf{Int}(K_2)). \quad (8.1)$$

Secondly, to prove $\mathbf{Int}(\mathbf{RLIN}) \supseteq \mathbf{Int}(\mathbf{LIN})$ we construct for every $L \in L_\tau(\mathbf{LIN})$ and $I = (\Sigma, \Delta, h)$ an interpreter for L an $L' \in L_\tau(\mathbf{RLIN})$ and an interpreter I' for L' such that $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$.

Choose a typed linear grammar $G = (N, T, P, S)$ (where $T = \Sigma$) such that $L(G) = L$. Now construct the following typed right-linear grammar $G' = (N', T', P', S')$, and I' an interpreter for $L(G')$. Note that from now on we will mostly write h and h' instead of h_I and h'_I respectively, as there can be no confusion.

- $I' = (\Sigma', \Delta', h')$, where $\Sigma' = T'$, $\Delta' = \Delta$, and h' as defined below.

- $N' = N \cup \{D\}, D \notin N$,

For $A \in N', A \neq D$, $\#in_{N'}(A) = \#in_N(A) + \#out_N(A) + \#out(L)$, $\#out_{N'}(A) = \#out(L)$. Furthermore, $\#in_{N'}(D) = \#in(L)$, and $\#out_{N'}(D) = \#out(L)$,

- $T' = \{a_p \mid p \in P\} \cup \{b\}$,

- $P' = \{p' \mid p \in P\} \cup \{q\}$.

If $p : A \rightarrow vBw$, where $A, B \in N$ and $v, w \in T^*$, then¹:

$$p' : A \rightarrow a_p B,$$

where $h'(a_p) = h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)}$, $\#in_{T'}(a_p) = \#in(h'(a_p)) = \#in(v) + \#out(w) + \#out(L)$, and $\#out_{T'}(a_p) = \#out(h'(a_p)) = \#out(v) + \#in(w) + \#out(L)$.

If $p : A \rightarrow v$ then:

$$p' : A \rightarrow a_p,$$

where $h'(a_p) = \mathbf{backfold}(h(v)) + U_{\#out(L)}$, $\#in_{T'}(a_p) = \#in(h'(a_p)) = \#in(v) + \#out(v) + \#out(L)$, and $\#out_{T'}(a_p) = \#out(h'(a_p)) = \#out(L)$.

And finally:

$$q : D \rightarrow bS,$$

where $h'(b) = U_{\#in(L)} + \mathbf{fold}(U_{\#out(L)})$, $\#in_{T'}(b) = \#in(h'(b)) = \#in(L)$, and $\#out_{T'}(b) = \#out(h'(b)) = \#in(L) + 2\#out(L)$,

- $S' = D$.

We now claim $\mathbf{Int}_{I'}(L(G')) = \mathbf{Int}_I(L)$, so $L(G')$ is the L' we are looking for. Proof of the claim by head recursion as follows.

Invariant:

For all $H \in \mathbf{HGR}(\Delta)$, $A \in N$, and $i \in \mathbb{N}$:

$$\begin{aligned} \exists_{v,w \in T^*} \left(G : S \Rightarrow^i vAw \text{ and } H = h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)} \right) \\ \iff \\ \exists_{v' \in T^*} \left(G' : S \Rightarrow^i v'A \text{ and } H = h'(v') \right). \end{aligned}$$

¹Note the λ -case! (See also the footnote on page 14.)

Proof:

By induction on the length of the derivations (= on i):

Induction basis, $i = 0$:

$$\begin{aligned}
G &: S \Rightarrow^0 S, \\
v &= (\lambda, \#\mathbf{in}(L) \rightarrow \#\mathbf{in}(L)), \\
w &= (\lambda, \#\mathbf{out}(L) \rightarrow \#\mathbf{out}(L)), \\
G' &: S \Rightarrow^0 S, \\
v' &= (\lambda, \#\mathbf{in}(L) + 2\#\mathbf{out}(L) \rightarrow \#\mathbf{in}(L) + 2\#\mathbf{out}(L)).
\end{aligned}$$

To prove:

$$h(v) + \mathbf{flip}(h(w)) + U_{\#\mathbf{out}(L)} = h'(v').$$

Proof:

$$\begin{aligned}
h(v) + \mathbf{flip}(h(w)) + U_{\#\mathbf{out}(L)} &= \text{(definition of } h) \\
&U_{\#\mathbf{in}(L)} + \mathbf{flip}(U_{\#\mathbf{out}(L)}) + U_{\#\mathbf{out}(L)} \stackrel{(6.25)}{=} \\
&U_{\#\mathbf{in}(L)} + U_{\#\mathbf{out}(L)} + U_{\#\mathbf{out}(L)} \stackrel{(4.9)}{=} \\
&U_{\#\mathbf{in}(L)+2\#\mathbf{out}(L)} = \text{(definition of } h') \\
&h'(v').
\end{aligned}$$

Induction step (assuming the hypothesis holds for $i = n$), from left to right, $i = n + 1$:

Consider a derivation of length $n + 1$ in G :

$$G : S \Rightarrow^n vAw \Rightarrow^1 vuBzw,$$

where $A, B \in N$, and $u, v, w, z \in T^*$ with the production $p : A \rightarrow uBz$ applied in the last step. By the induction hypothesis, there exists a derivation $G' : S \Rightarrow^n v'A$, where $v' \in T^*$, such that:

$$h'(v') = h(v) + \mathbf{flip}(h(w)) + U_{\#\mathbf{out}(L)}. \quad (8.2)$$

Let $p' : A \rightarrow a_p B$ be the production of G' corresponding to $p : A \rightarrow uBz$, by the definition of P' . We now have:

$$G' : S \Rightarrow^n v'A \Rightarrow^1 v'a_p B.$$

To prove:

$$h'(v'a_p) = h(vu) + \mathbf{flip}(h(zw)) + U_{\#\mathbf{out}(L)}.$$

Proof:

$$\begin{aligned}
& h'(v'a_p) \stackrel{(7.1)}{=} \\
& h'(v') \cdot h'(a_p) \stackrel{(8.2)}{=} \\
& (h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)}) \cdot h'(a_p) \stackrel{(\text{definition of } h'(a_p))}{=} \\
& (h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)}) \cdot (h(u) + \mathbf{flip}(h(z)) + U_{\#out(L)}) \stackrel{(4.11)}{=} \\
& h(v) \cdot h(u) + \mathbf{flip}(h(w)) \cdot \mathbf{flip}(h(z)) + U_{\#out(L)} \stackrel{(7.1)}{=} \\
& h(vu) + \mathbf{flip}(h(w)) \cdot \mathbf{flip}(h(z)) + U_{\#out(L)} \stackrel{(6.23)}{=} \\
& h(vu) + \mathbf{flip}(h(z)h(w)) + U_{\#out(L)} \stackrel{(7.1)}{=} \\
& h(vu) + \mathbf{flip}(h(zw)) + U_{\#out(L)}.
\end{aligned}$$

A similar proof applies for the direction from right to left. This completes the proof of the invariant for $i = n + 1$, and overall proof of the invariant.

Now for any $H \in \mathbf{Int}_I(L)$:

$$\exists_{v,w,u \in T^*} G : S \Rightarrow^* vAw \Rightarrow^1 vuw, \text{ such that } h(vuw) = H,$$

where $A \in N$, and production $p : A \rightarrow u$ applied in the last step. Choose such a $v, w, u \in T^*$. As implied by the invariant:

$$\exists_{v' \in T'^*} G' : S \Rightarrow^* v'A \text{ and } h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)} = h'(v').$$

Choose such a $v' \in T'^*$. Let $p' : A \rightarrow a_p$ be the production in P' corresponding to p . Then, $G' : S \Rightarrow^* v'A \Rightarrow^1 v'a_p$, and:

$$\begin{aligned}
& h'(v'a_p) \stackrel{(7.1)}{=} \\
& h'(v') \cdot h'(a_p) \stackrel{(\text{invariant})}{=} \\
& (h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)}) \cdot h'(a_p) \stackrel{(\text{definition of } h'(a_p))}{=} \\
& (h(v) + \mathbf{flip}(h(w)) + U_{\#out(L)}) \cdot (\mathbf{backfold}(h(u)) + U_{\#out(L)}) \stackrel{(6.33)}{=} \\
& \mathbf{backfold}(h(v)h(u)h(w)) + U_{\#out(L)} \stackrel{(7.2)}{=} \\
& \mathbf{backfold}(h(vuw)) + U_{\#out(L)} \stackrel{(\text{as } h(vuw)=H)}{=} \\
& \mathbf{backfold}(H) + U_{\#out(L)}.
\end{aligned}$$

And therefore also:

$$G' : D \Rightarrow^1 bS \Rightarrow^* bv'A \Rightarrow^1 bv'a_p,$$

and:

$$\begin{aligned}
 h'(bv'a_p) &\stackrel{(7.1)}{=} \\
 &h'(b) \cdot h'(v'a_p) \stackrel{(\text{as by definition } h'(b)=U_{\#in(L)}+\mathbf{fold}(U_{\#out(L)}))}{=} \\
 &(U_{\#in(L)} + \mathbf{fold}(U_{\#out(L)})) \cdot h'(v'a_p) \stackrel{(\text{proven above})}{=} \\
 &(U_{\#in(L)} + \mathbf{fold}(U_{\#out(L)})) \cdot (\mathbf{backfold}(H) + U_{\#out(L)}) \stackrel{(6.34)}{=} \\
 &U_{\#in(L)} \cdot H \cdot U_{\#out(L)} \cdot U_{\#out(L)} \stackrel{(\text{unity})}{=} \\
 &H.
 \end{aligned}$$

This proves that $H \in \mathbf{Int}_{I'}(L')$, so $\mathbf{Int}_I(L) \subseteq \mathbf{Int}_{I'}(L')$. The proof in the other direction, $\mathbf{Int}_I(L) \supseteq \mathbf{Int}_{I'}(L')$, works in the same way. This completes the proof of our claim that $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$, and thus also completes the overall proof $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{LIN})$.

In order to make the construction we used in this proof more clear, we will give an example.

Example:

Let L be the typed language $\{a^nbc^n \mid n \in \mathbb{N}\}$, over the typed alphabet $\Sigma = \{(a, 1 \rightarrow 1), (b, 1 \rightarrow 1), (c, 1 \rightarrow 1)\}$. By elementary formal language theory, this is a linear language that is *not* right-linear². Let $I = (\Sigma, \Delta, h)$ be the interpreter for L defined by Σ as above, $\Delta = ((a, 2), (b, 2), (c, 2))$, and $h = \mathbf{gr}$. So:



As can be easily seen, for all $w \in L$, $h(w) = \mathbf{gr}(w)$, so:

$$\mathbf{Int}_I(L) = \mathbf{gr}(L) = \{\mathbf{gr}(a^nbc^n) \mid n \in \mathbb{N}\}.$$

We will now apply the construction given at the beginning of this section to find a typed right-linear grammar G' and an interpreter I' for $L(G')$, such that $\mathbf{Int}_{I'}(L(G')) = \mathbf{Int}_I(L)$. First, we need to choose a typed linear grammar $G = (N, T, P, S)$ (where $T = \Sigma$) such that $L(G) = L$. We take:

- $N = \{(S, 1 \rightarrow 1)\}$,

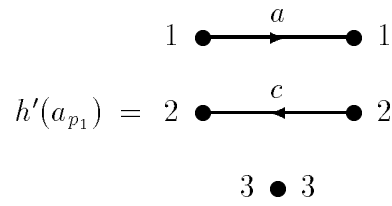
²Standard procedure to prove this: give a linear grammar that generates L , which proves that L is indeed linear, and then use the pumping lemma to show that L is not regular, i.e., not right-linear.

- $T = \Sigma = \{(a, 1 \rightarrow 1), (b, 1 \rightarrow 1), (c, 1 \rightarrow 1)\}$,
- $P = \{p_1 : S \rightarrow aSc, p_2 : S \rightarrow b\}$,
- $S = S$.

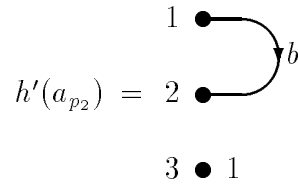
The typed right-linear grammar $G' = (N', T', P', S')$ and the interpreter $I' = (\Sigma', \Delta', h')$ for $L(G')$ are now, following the construction, defined as:

- $\Sigma' = T'$ (defined below), $\Delta' = \Delta$, and h' also as defined below,
- $N' = \{(S, 3 \rightarrow 1), (D, 1 \rightarrow 1)\}$,
- $T' = \{a_{p_1}, a_{p_2}, b\}$ (types will be defined below),
- $P' = \{p'_1, p'_2, q\}$,

$p'_1 : S \rightarrow a_{p_1}S$, where a_{p_1} has type $(3 \rightarrow 3)$, and $h'(a_{p_1}) = h(a) + \mathbf{flip}(h(c)) + U_1$:



$p'_2 : S \rightarrow a_{p_2}$, where a_{p_2} has type $(3 \rightarrow 1)$, and $h'(a_{p_2}) = \mathbf{backfold}(h(b)) + U_1$:



$q : D \rightarrow bS$, where b has type $(1 \rightarrow 3)$, and $h'(b) = U_1 + \mathbf{fold}(U_1)$:

$$h'(b) = \begin{matrix} 1 \bullet 1 \\ \bullet 2, 3 \end{matrix}$$

- $S' = D$.

As was previously proved, now $\mathbf{Int}_{I'}(L(G')) = \mathbf{Int}_I(L)$. Is this plausible? Let us take a look. It can be easily seen that the very simple typed right-linear grammar G' generates the typed language:

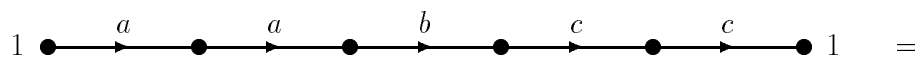
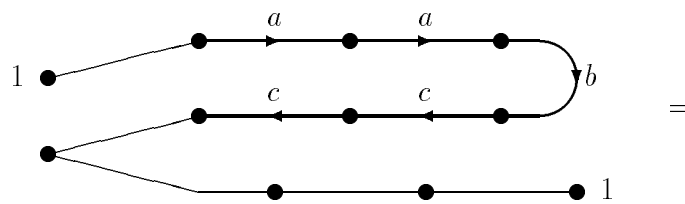
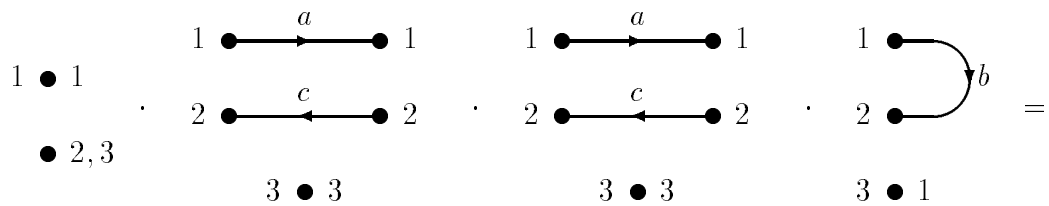
$$L(G') = \{ ba_{p_1}^n a_{p_2} \mid n \in \mathbb{N} \}.$$

Note that (modulo isomorphism) this is just the language $\{ ab^n c \mid n \in \mathbb{N} \}$ in disguise. The question now is:

$$\mathbf{Int}_{I'}(\{ ba_{p_1}^n a_{p_2} \mid n \in \mathbb{N} \}) \stackrel{?}{=} \{ \mathbf{gr}(a^n bc^n) \mid n \in \mathbb{N} \}.$$

Consider the case $n = 2$ on the left-hand side:

$$h'(ba_{p_1} a_{p_1} a_{p_2}) =$$



$$\mathbf{gr}(aabcc).$$

N.B.: the thin lines in the above denote that the nodes they connect have been identified. As we have just shown:

$$h'(ba_{p_1}^2 a_{p_2}) = h(a^2 bc^2).$$

Because instead of 2 we could have taken any $n \in \mathbb{N}$, this more or less “proves” that $\mathbf{Int}_{I'}(L') = \mathbf{Int}_I(L)$. So, yes, the claim *is* plausible. This concludes the example of the construction used to prove $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{LIN})$.

8.2 $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{DB})$

Firstly, it is trivial that $\mathbf{Int}(\mathbf{RLIN}) \subseteq \mathbf{Int}(\mathbf{DB})$ because $\mathbf{RLIN} \subseteq \mathbf{DB}$ (8.1). Secondly, to prove $\mathbf{Int}(\mathbf{RLIN}) \supseteq \mathbf{Int}(\mathbf{DB})$ we construct for every $L \in L_\tau(\mathbf{DB})$ and $I = (\Sigma, \Delta, h)$ an interpreter for L an $L' \in L_\tau(\mathbf{RLIN})$ and an interpreter I' for L' such that $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$.

Choose a typed derivation-bounded grammar $G = (N, T, P, S)$ (where $T = \Sigma$) such that $L(G) = L$. Now construct the following right-linear grammar $G' = (N', T', P', S')$, where k denotes the derivation bound of G , and I' an interpreter for $L(G')$.

- $I' = (\Sigma', \Delta', h')$, where $\Sigma' = T'$, $\Delta' = \Delta$, and h' as defined above,
- $N' = \left(\bigcup_{i=1}^k N'_i \right) \cup \{D\}$, where $N'_i = \{[A_1 \dots A_i] \mid A_1, \dots, A_i \in N\}$.

Intuitively: there exists a nonterminal in N' for every sequence of up to k nonterminals in N . Furthermore, there is a new nonterminal D .

The type of $[A_1 \dots A_n] \in N'$ is:

$$\begin{aligned} \#\mathbf{in}_{N'}([A_1 \dots A_m]) &= \left(\sum_{k=1}^m (\#\mathbf{in}_N(A_k) + \#\mathbf{out}_N(A_k)) \right) + \#\mathbf{out}(L), \\ \#\mathbf{out}_{N'}([A_1 \dots A_m]) &= \#\mathbf{out}(L). \end{aligned}$$

Furthermore, $\#\mathbf{in}_{N'}(D) = \#\mathbf{in}(L)$ and $\#\mathbf{out}_{N'}(D) = \#\mathbf{out}(L)$.

- $T' = \{a_p \mid p \in P'\}$.

Intuitively: there exists a unique terminal for every production in P' ,

- $P' = \left(\bigcup_{p \in P} Q(p) \right) \cup \{p_0\}$.

Intuitively: $Q(p)$ comprises a set of productions that “accomplishes” the same in G' as p “accomplishes” in G . This will become clearer after $Q(p)$ has been defined. Furthermore, there is a production p_0 . For

$$p : A_0 \rightarrow v_1 A_1 v_2 A_2 \dots v_{n-1} A_{n-1} v_n, \quad v_1, \dots, v_n \in T^*, \quad A_0, \dots, A_{n-1} \in N, \quad n \geq 1,$$

$Q(p)$ contains the production $q(p, [B_1 \dots B_m], i)$ for every $[B_1, \dots, B_m] \in N'$, and every $i, 1 \leq i \leq m$, such that $A_0 = B_i$, under the condition that $m + n - 2 \leq k$.

The production $p' = q(p, [B_1 \dots B_m], i)$ looks as follows:

$$[B_1 \dots B_m] \rightarrow a_{p'} [B_1 \dots B_{i-1} A_1 \dots A_{n-1} B_{i+1} \dots B_m]$$

N.B.:

1. Should $[B_1 \dots B_{i-1} A_1 \dots A_{n-1} B_{i+1} \dots B_m]$ reduce to “[]” because $m = 1, n = 1$, and $i = 1$, we will interpret this as the syntactically empty string (in other words, we will pretend the [] is not there at all). When that happens p' reduces to the production $[B_1] \rightarrow a_{p'}$.
2. Because $m + n - 2 \leq k$ it is guaranteed that $[B_1 \dots B_{i-1} A_1 \dots A_{n-1} B_{i+1} \dots B_m] \in N'$ (unless, of course, it reduces to []).
3. This condition $m + n - 2 \leq k$ is not a restrictive one, as the derivation boundedness of G guarantees that there is “no need” for sentential forms that contain more than k nonterminals.
4. Intuitively the above production rewrites the i th nonterminal according to p .

For this production $h'(a_{p'})$ is defined as follows³:

$$h'(a_{p'}) = \mathbf{H}(B_1) + \dots + \mathbf{H}(B_{i-1}) + \mathbf{H}(v_1) + \dots + \mathbf{H}(v_n) + \mathbf{H}(B_{i+1}) + \dots + \mathbf{H}(B_m) + U_{\#\mathbf{out}(L)},$$

where:

$$\begin{aligned} \mathbf{H}(B_j) &= U_{\#\mathbf{in}(B_j) + \#\mathbf{out}(B_j)} && \text{for } 1 \leq j \leq i-1 \text{ or } i+1 \leq j \leq m, \\ \mathbf{H}(v_1) &= \mathbf{backfold}(h(v_1)) && \text{for } n = 1, \\ \mathbf{H}(v_1) &= h(v_1) && \text{for } n \geq 2, \\ \mathbf{H}(v_j) &= \mathbf{fold}(h(v_j)) && \text{for } n \geq 2 \text{ and } 2 \leq j \leq n-1, \\ \mathbf{H}(v_n) &= \mathbf{flip}(h(v_n)) && \text{for } n \geq 2. \end{aligned}$$

Furthermore, we need to define:

$$\begin{aligned} \#\mathbf{in}_{T'}(a_{p'}) &= \#\mathbf{in}(h'(a_{p'})) = \left(\sum_{l=1}^m (\#\mathbf{in}_N(B_l) + \#\mathbf{out}_N(B_l)) \right) + \#\mathbf{out}(L), \\ \#\mathbf{out}_{T'}(a_{p'}) &= \#\mathbf{out}(h'(a_{p'})) = \left(\sum_{l=1}^{i-1} (\#\mathbf{in}_N(B_l) + \#\mathbf{out}_N(B_l)) \right) + \\ &\quad \left(\sum_{l=1}^{n-1} (\#\mathbf{in}_N(A_l) + \#\mathbf{out}_N(A_l)) \right) + \\ &\quad \left(\sum_{l=i+1}^m (\#\mathbf{in}_N(B_l) + \#\mathbf{out}_N(B_l)) \right) + \#\mathbf{out}(L). \end{aligned}$$

³Note the λ -case! (See also the footnote on page 14.)

And finally:

$$p_0 : D \rightarrow a_{p_0}[S],$$

where $h'(a_{p_0}) = U_{\#\mathbf{in}(L)} + \mathbf{fold}(U_{\#\mathbf{out}(L)})$, $\#\mathbf{in}_{T'}(a_{p_0}) = \#\mathbf{in}(L)$, and $\#\mathbf{out}_{T'}(a_{p_0}) = \#\mathbf{in}(L) + 2\#\mathbf{out}(L)$,

- $S' = D$.

We now have $\mathbf{Int}_{T'}(L(G')) = \mathbf{Int}_I(L)$, so $L(G')$ is the L' we are looking for. Proof by head recursion as follows:

Invariant:

For all $H \in \mathbf{HGR}(\Delta)$, $A_i \in N$, $n \geq 2$, and $j \in \mathbb{N}$:

$$\begin{aligned} \exists_{v_1, \dots, v_n \in T^*} \left(\begin{array}{c} G : S \Rightarrow^j v_1 A_1 v_2 \dots v_{n-1} A_{n-1} v_n \\ \text{and} \\ H = h(v_1) + \mathbf{fold}(h(v_2)) + \dots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)} \end{array} \right) \\ \iff \\ \exists_{v' \in T'^*} (G' : [S] \Rightarrow^j v' [A_1 \dots A_{n-1}] \text{ and } H = h'(v')). \end{aligned}$$

Proof:

In proving this invariant, we reason along the same lines as we did in the proof in Section 8.1. Consequently, we will notate somewhat more tersely now, as there is enough verbatim correspondence between the two proofs as it is. Induction on the length of the derivations (= on j):

Induction basis, $j=0$:

$$\begin{aligned} G : S &\Rightarrow^0 S, \\ n &= 2, \\ v_1 &= (\lambda, \#\mathbf{in}(L) \rightarrow \#\mathbf{in}(L)), \\ v_2 &= (\lambda, \#\mathbf{out}(L) \rightarrow \#\mathbf{out}(L)), \\ G' : [S] &\Rightarrow^0 [S], \\ v' &= (\lambda, \#\mathbf{in}(L) + 2\#\mathbf{out}(L) \rightarrow \#\mathbf{in}(L) + 2\#\mathbf{out}(L)). \end{aligned}$$

To prove:

$$h(v_1) + \mathbf{flip}(h(v_2)) + U_{\#\mathbf{out}(L)} = h'(v').$$

Proof:

$$\begin{aligned}
h(v_1) + \mathbf{flip}(h(v_2)) + U_{\#\mathbf{out}(L)} & \stackrel{(\text{definition of } h)}{=} \\
U_{\#\mathbf{in}(L)} + \mathbf{flip}(U_{\#\mathbf{out}(L)}) + U_{\#\mathbf{out}(L)} & \stackrel{(6.25)}{=} \\
U_{\#\mathbf{in}(L)} + U_{\#\mathbf{out}(L)} + U_{\#\mathbf{out}(L)} & \stackrel{(4.9)}{=} \\
U_{\#\mathbf{in}(L)+2\#\mathbf{out}(L)} & \stackrel{(\text{definition of } h')}{=} \\
h'(v'). &
\end{aligned}$$

Induction step, assuming the hypothesis holds for \Rightarrow^l , $j = l + 1$:

Consider a derivation of length $l + 1$ in G . Such a derivation necessarily consists of a derivation of length l :

$$G : S \Rightarrow^l v_1 A_1 v_2 \dots v_{n-1} A_{n-1} v_n,$$

followed by a last step in which we apply the production:

$$p : A_i \rightarrow w_1 B_1 w_2 \dots w_{m-1} B_{m-1} w_m.$$

Firstly, we examine the case where in the last derivation step we are applying a production that does *not* have a terminal-only right-hand side (so $m \geq 2$), and of that case the subcase $1 < i < n - 1$, and $n > 2$. Then, by the induction hypothesis, there exists a derivation in G' :

$$G' : [S] \Rightarrow^l v'[A_1 \dots A_{n-1}],$$

such that:

$$h'(v') = h(v_1) + \mathbf{fold}(h(v_2)) + \dots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}. \quad (8.3)$$

Let p' be the production $q(p, [A_1 \dots A_{n-1}], i)$ of G' (which corresponds to applying p in the current context):

$$p' : [A_1 \dots A_{n-1}] \rightarrow a_{p'}[A_1 \dots A_{i-1} B_1 \dots B_{m-1} A_{i+1} \dots A_{n-1}].$$

Combining these derivations of length l with these last steps, we now have:

$$\begin{aligned}
G : S \Rightarrow^l v_1 A_1 v_2 \dots v_{n-1} A_{n-1} v_n & \Rightarrow^1 v_1 A_1 v_2 \dots v_i w_1 B_1 w_2 \dots w_{m-1} B_{m-1} w_m v_{i+1} \dots v_{n-1} A_{n-1} v_n, \\
G' : [S] \Rightarrow^l v'[A_1 \dots A_{n-1}] & \Rightarrow^1 v' a_{p'}[A_1 \dots A_{i-1} B_1 \dots B_{m-1} A_{i+1} \dots A_{n-1}].
\end{aligned}$$

Note that from the existence of these derivations we can derive, for $1 \leq r < n$:

$$\begin{aligned}
\#\mathbf{in}(A_r) & = \#\mathbf{out}(h(v_r)), \\
\#\mathbf{out}(A_r) & = \#\mathbf{in}(h(v_{r+1})),
\end{aligned} \quad (8.4)$$

which will come in very handy in our proof.

To prove:

$$\begin{aligned} h'(v'a_{p'}) &= h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{i-1})) + \\ &\quad \mathbf{fold}(h(v_i w_1)) + \mathbf{fold}(w_2) + \cdots + \mathbf{fold}(w_{m-1}) + \mathbf{fold}(h(w_m v_{i+1})) + \\ &\quad \mathbf{fold}(h(v_{i+2})) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}. \end{aligned}$$

Although the expressions in what now follows look quite abhorrent, the steps in between them are actually very simple. All what happens is some reshuffling of U 's using (4.9), and the rewriting of $\#\mathbf{in}(A_i)$ and $\#\mathbf{out}(A_i)$, as $\#\mathbf{out}(h(v_i))$ and $\#\mathbf{in}(h(v_{i+1}))$ respectively. Only in the last step, where we apply (B.2)⁴, some “real” work is being done.

Proof:

$$h'(v'a_{p'}) \stackrel{(7.1)}{=}$$

$$h'(v') \cdot h'(a_{p'}) \stackrel{(8.3)}{=}$$

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot h'(a_{p'}) \stackrel{(\text{definition of } h'(a_{p'}))}{=}$$

$$\begin{aligned} &(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \\ &\left(\begin{array}{l} U_{\#\mathbf{in}(A_1)+\#\mathbf{out}(A_1)} + \cdots + U_{\#\mathbf{in}(A_{i-1})+\#\mathbf{out}(A_{i-1})} + \\ h(w_1) + \mathbf{fold}(h(w_2)) + \cdots + \mathbf{fold}(h(w_{m-1})) + \mathbf{flip}(h(w_m)) + \\ U_{\#\mathbf{in}(A_{i+1})+\#\mathbf{out}(A_{i+1})} + \cdots + U_{\#\mathbf{in}(A_{n-1})+\#\mathbf{out}(A_{n-1})} + U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(4.9)}{=} \end{aligned}$$

$$\begin{aligned} &(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \\ &\left(\begin{array}{l} U_{\#\mathbf{in}(A_1)+\#\mathbf{out}(A_1)+\cdots+\#\mathbf{in}(A_{i-1})+\#\mathbf{out}(A_{i-1})} + \\ h(w_1) + \mathbf{fold}(h(w_2)) + \cdots + \mathbf{fold}(h(w_{m-1})) + \mathbf{flip}(h(w_m)) + \\ U_{\#\mathbf{in}(A_{i+1})+\#\mathbf{out}(A_{i+1})+\cdots+\#\mathbf{in}(A_{n-1})+\#\mathbf{out}(A_{n-1})} + U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(8.4)}{=} \end{aligned}$$

⁴The “B” in “(B.2)” refers to *Appendix B*.

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \left(\begin{array}{c} U_{\#\mathbf{out}(h(v_1))+\#\mathbf{in}(h(v_2))+\#\mathbf{out}(h(v_2))+\cdots+\#\mathbf{in}(h(v_{i-1}))+\#\mathbf{out}(h(v_{i-1}))+\#\mathbf{in}(h(v_i)) +} \\ h(w_1) + \mathbf{fold}(h(w_2)) + \cdots + \mathbf{fold}(h(w_{m-1})) + \mathbf{flip}(h(w_m)) + \\ U_{\#\mathbf{out}(h(a_{i+1}))+\#\mathbf{in}(h(v_{i+2}))+\#\mathbf{out}(h(v_{i+2}))+\cdots+\#\mathbf{in}(h(v_{n-1}))+\#\mathbf{out}(h(v_{n-1}))+\#\mathbf{in}(h(v_n)) +} \\ U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(4.9)}{=} \left(\begin{array}{c} U_{\#\mathbf{out}(h(v_1)) + U_{\#\mathbf{in}(h(v_2))+\#\mathbf{out}(h(v_2))} + \cdots + U_{\#\mathbf{in}(h(v_{i-1}))+\#\mathbf{out}(h(v_{i-1}))} + \\ U_{\#\mathbf{in}(h(v_i))} + h(w_1) + \mathbf{fold}(h(w_2)) + \cdots + \mathbf{fold}(h(w_{m-1})) + \mathbf{flip}(h(w_m)) + U_{\#\mathbf{out}(h(a_{i+1}))} + \\ U_{\#\mathbf{in}(h(v_{i+2}))+\#\mathbf{out}(h(v_{i+2}))} + \cdots + U_{\#\mathbf{in}(h(v_{n-1}))+\#\mathbf{out}(h(v_{n-1}))} + U_{\#\mathbf{in}(h(v_n))} + \\ U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(B.2)}{=} \left(\begin{array}{c} h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{i-1})) + \\ \mathbf{fold}(h(v_i w_1)) + \mathbf{fold}(w_2) + \cdots + \mathbf{fold}(w_{m-1}) + \mathbf{fold}(h(w_m v_{i+1})) + \\ \mathbf{fold}(h(v_{i+2})) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)} \end{array} \right).$$

$$\left(\begin{array}{c} h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{i-1})) + \\ \mathbf{fold}(h(v_i w_1)) + \mathbf{fold}(w_2) + \cdots + \mathbf{fold}(w_{m-1}) + \mathbf{fold}(h(w_m v_{i+1})) + \\ \mathbf{fold}(h(v_{i+2})) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)} \end{array} \right).$$

The equivalents for the subcases ($i = 1$ and $n > 2$), ($i = n - 1$ and $n > 2$), and ($i = 1$ and $n = 2$) are not written out in full here, as they are very similar. Most importantly, they differ in the fact that instead of applying (B.2) in the last step, one needs to apply (B.3), (B.4), and (B.5) respectively. Note that it is more or less imaginable to combine all these four cases using a kind meta notation just as in (B.1), but only at the cost of an even more crippled clarity of the above derivation.

Secondly, we examine the case where in the last derivation step we are applying a production that *does* have a terminal-only right-hand side (so $m = 1$). We start with the subcase $1 < i < n - 1$ and $n > 2$. This time, we arrive at the following derivations of length $l + 1$:

$$\begin{aligned} G : S &\Rightarrow^l v_1 A_1 v_2 \dots v_{n-1} A_{n-1} v_n \Rightarrow^1 v_1 A_1 v_2 \dots v_i w_1 v_{i+1} \dots v_{n-1} A_{n-1} v_n, \\ G' : [S] &\Rightarrow^l v' [A_1 \dots A_{n-1}] \Rightarrow^1 v' a_{p'} [A_1 \dots A_{i-1} A_{i+1} \dots A_{n-1}]. \end{aligned}$$

Note that (8.3) and (8.4) also hold for these derivations.

To prove:

$$\begin{aligned} h'(v' a_{p'}) &= h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{i-1})) + \mathbf{fold}(h(v_i w_1 v_{i+1})) + \\ &\quad \mathbf{fold}(h(v_{i+2})) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}. \end{aligned}$$

Proof (derivation is almost the same as the previous one):

$$h'(v'_{a_{p'}}) \stackrel{(7.1)}{=}$$

$$h'(v') \cdot h'(a_{p'}) \stackrel{(8.3)}{=}$$

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot h'(a_{p'}) \stackrel{(\text{definition of } h'(a_{p'}))}{=}$$

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \left(\begin{array}{c} U_{\#\mathbf{in}(A_1)+\#\mathbf{out}(A_1)} + \cdots + U_{\#\mathbf{in}(A_{i-1})+\#\mathbf{out}(A_{i-1})} + \\ \mathbf{backfold}(h(w_1)) + \\ U_{\#\mathbf{in}(A_{i+1})+\#\mathbf{out}(A_{i+1})} + \cdots + U_{\#\mathbf{in}(A_{n-1})+\#\mathbf{out}(A_{n-1})} + U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(4.9)}{=}$$

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \left(\begin{array}{c} U_{\#\mathbf{in}(A_1)+\#\mathbf{out}(A_1)+\cdots+\#\mathbf{in}(A_{i-1})+\#\mathbf{out}(A_{i-1})} + \\ \mathbf{backfold}(h(w_1)) + \\ U_{\#\mathbf{in}(A_{i+1})+\#\mathbf{out}(A_{i+1})+\cdots+\#\mathbf{in}(A_{n-1})+\#\mathbf{out}(A_{n-1})} + U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(8.4)}{=}$$

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \left(\begin{array}{c} U_{\#\mathbf{out}(h(v_1))+\#\mathbf{in}(h(v_2))+\#\mathbf{out}(h(v_2))+\cdots+\#\mathbf{in}(h(v_{i-1}))+\#\mathbf{out}(h(v_{i-1}))+\#\mathbf{in}(h(v_i))} + \\ \mathbf{backfold}(h(w_1)) + \\ U_{\#\mathbf{out}(h(v_{i+1}))+\#\mathbf{in}(h(v_{i+2}))+\#\mathbf{out}(h(v_{i+2}))+\cdots+\#\mathbf{in}(h(v_{n-1}))+\#\mathbf{out}(h(v_{n-1}))+\#\mathbf{in}(h(v_n))} + \\ U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(4.9)}{=}$$

$$(h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)}) \cdot \left(\begin{array}{c} U_{\#\mathbf{out}(h(v_1))} + U_{\#\mathbf{in}(h(v_2))+\#\mathbf{out}(h(v_2))} + \cdots + U_{\#\mathbf{in}(h(v_{i-1}))+\#\mathbf{out}(h(v_{i-1}))} + \\ U_{\#\mathbf{in}(h(v_i))} + \mathbf{backfold}(h(w_1)) + U_{\#\mathbf{out}(h(v_{i+1}))} + \\ U_{\#\mathbf{in}(h(v_{i+2}))+\#\mathbf{out}(h(v_{i+2}))} + \cdots + U_{\#\mathbf{in}(h(v_{n-1}))+\#\mathbf{out}(h(v_{n-1}))} + U_{\#\mathbf{in}(h(v_n))} + \\ U_{\#\mathbf{out}(L)} \end{array} \right) \stackrel{(B.7)}{=}$$

$$\left(\begin{array}{c} h(v_1) + \mathbf{fold}(h(v_2)) + \cdots + \mathbf{fold}(h(v_{i-1})) + \\ \mathbf{fold}(h(v_i w_1 v_{i+1})) + \\ \mathbf{fold}(h(v_{i+2})) + \cdots + \mathbf{fold}(h(v_{n-1})) + \mathbf{flip}(h(v_n)) + U_{\#\mathbf{out}(L)} \end{array} \right)$$

The equivalents for the subcases ($i = 1$ and $n > 2$), ($i = n - 1$ and $n > 2$), and ($i = 1$ and $n = 2$) are not written out in full here, as they are very similar. Most importantly, they differ in the fact that instead of applying (B.7) in the last step, one needs to apply (B.8), (B.9), and (B.10) respectively.

The similar proof applies for the direction from right to left, where we use the same pairs of derivations for each value of j . This completes the overall proof of the invariant.

Now for any $H \in \mathbf{Int}_I(L)$:⁵

$$\exists_{v,w,u \in T^*} G : S \Rightarrow^* vAw \Rightarrow^1 vuw, \text{ such that } h(vuw) = H,$$

where $A \in N$, and production $p : A \rightarrow u$ applied in the last step. Choose such a $v, w, u \in T^*$. As implied by the invariant:

$$\exists_{v' \in T'^*} G' : [S] \Rightarrow^* v'[A] \text{ and } h(v) + \mathbf{flip}(h(w)) + U_{\#\mathbf{out}(L)} = h'(v').$$

Choose such a $v' \in T'^*$. Define p' to be $q(p, [A], 1)$, the production in P' corresponding to the application of p in the context of $[A]$. Thus, $p' : [A] \rightarrow a_{p'}$. Then, $G' : [S] \Rightarrow^* v'[A] \Rightarrow^1 v'a_{p'}$, and:

$$\begin{aligned} & h'(v'a_{p'}) \stackrel{(7.1)}{=} \\ & h'(v') \cdot h'(a_{p'}) \stackrel{(\text{invariant})}{=} \\ & (h(v) + \mathbf{flip}(h(w)) + U_{\#\mathbf{out}(L)}) \cdot h'(a_{p'}) \stackrel{(\text{definition of } h'(a_{p'}))}{=} \\ & (h(v) + \mathbf{flip}(h(w)) + U_{\#\mathbf{out}(L)}) \cdot (\mathbf{backfold}(h(u)) + U_{\#\mathbf{out}(L)}) \stackrel{(6.33)}{=} \\ & \mathbf{backfold}(h(v)h(u)h(w)) + U_{\#\mathbf{out}(L)} \stackrel{(7.2)}{=} \\ & \mathbf{backfold}(h(vuw)) + U_{\#\mathbf{out}(L)} \stackrel{(\text{as by definition } h(vuw)=H)}{=} \\ & \mathbf{backfold}(H) + U_{\#\mathbf{out}(L)}. \end{aligned}$$

And therefore also:

$$G' : D \Rightarrow^1 a_{p_0}[S] \Rightarrow^* a_{p_0}v'[A] \Rightarrow^1 a_{p_0}v'a_{p'},$$

and:

$$\begin{aligned} & h'(a_{p_0}v'a_{p'}) \stackrel{(7.1)}{=} \\ & h'(a_{p_0}) \cdot h'(v'a_{p'}) \stackrel{(\text{as by definition } h'(a_{p_0})=U_{\#\mathbf{in}(L)}+\mathbf{fold}(U_{\#\mathbf{out}(L)}))}{=} \\ & (U_{\#\mathbf{in}(L)} + \mathbf{fold}(U_{\#\mathbf{out}(L)})) \cdot h'(v'a_{p'}) \stackrel{(\text{as proven above})}{=} \end{aligned}$$

⁵N.B.: this part is almost identical to the last part of Section 8.1.

$$\begin{aligned}
& (U_{\#in(L)} + \mathbf{fold}(U_{\#out(L)})) \cdot (\mathbf{backfold}(H) + U_{\#out(L)}) \stackrel{(6.34)}{=} \\
& U_{\#in(L)} \cdot H \cdot U_{\#out(L)} \cdot U_{\#out(L)} \stackrel{(\text{unity})}{=} \\
& H.
\end{aligned}$$

Which proves that $H \in \mathbf{Int}_{I'}(L')$, so $\mathbf{Int}_I(L) \subseteq \mathbf{Int}_{I'}(L')$. The proof in the other direction, $\mathbf{Int}_I(L) \supseteq \mathbf{Int}_{I'}(L')$, works in the same way. This completes the proof of our claim that $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$, and thus also completes the overall proof of $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{DB})$.

8.3 $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(\mathbf{K}))) = \mathbf{Int}(\mathbf{K})$

Given a class K of string languages, it turns out that under the remarkably weak conditions that:

- K is closed under λ -free finite substitution, and,
- K is closed under intersection with a regular language,

we have that:

$$\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K))) = \mathbf{Int}(K). \quad (8.5)$$

Both RLIN and CF, and most other classes not deliberately constructed to violate these conditions, can be substituted for K . We will not give a complete formal proof, but only the construction for a graph language L in the one side of (8.5), given L is contained in the other side.

Proof. The direction $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K))) \supseteq \mathbf{Int}(K)$ is easy, as by (7.9) $\mathbf{gr}(K) \subseteq \mathbf{Int}(K)$, so therefore also $\mathbf{STR}(\mathbf{gr}(K)) \subseteq \mathbf{STR}(\mathbf{Int}(K))$, and by (3.6) $K \subseteq \mathbf{STR}(\mathbf{Int}(K))$, from which the result $\mathbf{Int}(K) \subseteq \mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K)))$ immediately follows.

The other direction, $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K))) \subseteq \mathbf{Int}(K)$, involves a lot of hard work. Given a hypergraph language $L \in \mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K)))$, we construct⁶ an $L' \in L_\tau(K)$, and an interpreter $I' = (\Sigma', \Delta', h')$ for L' , such that $\mathbf{Int}_{I'}(L') = L$.

Construction:

There exist alphabets Σ_1, Σ_2 , languages $L_0 \subseteq \Sigma_1^*, L_1 \subseteq \Sigma_2^*$, with $L_0 \in K$, and interpreters $I_1 = (\Sigma_1, \Delta_1, h_1)$, and $I_2 = (\Sigma_2, \Delta_2, h_2)$ for L_0 and L_1 respectively, such that $\mathbf{Int}_{I_1}(L_0) =$

⁶Be warned: the construction we give is flooded with technical detail to such an extent, that it borders on the totally incomprehensible. However, it should be possible to grasp the essence of its inner workings by studying the explanation that follows the construction.

$\mathbf{gr}(L_1)$, and $\mathbf{Int}_{I_2}(L_1) = \mathbf{L}$ (note that necessarily $\Sigma_2 = \Delta_1$, save the types and ranks). Assume (without loss of generality, as K is closed under λ -free finite substitution) that I_1 is in ENF. Define μ to be the maximum input or output type of a $h_2(a)$ for all $a \in \Sigma_2$:

$$\mu = \mathbf{max} \{ \#\mathbf{in}(h_2(a)), \#\mathbf{out}(h_2(a)) \mid a \in \Sigma_2 \}.$$

L' is formed by applying a λ -free finite substitution to L_0 , and then intersecting the result with a regular language:

$$L' = \sigma(L_0) \cap L_T.$$

Here σ is a mapping from Σ_1 to finite subsets of Σ' :

$$\sigma : \Sigma_1 \rightarrow \mathcal{P}(\Sigma'),$$

and $L_T \subseteq \Sigma'^*$ is a regular language. Σ' is derived from Σ_1 :

$$\Sigma' = \bigcup_{a \in \Sigma_1} \Sigma_a.$$

Here, for every symbol $(a, m \rightarrow n) \in \Sigma_1$, Σ_a is defined as follows:

$$\Sigma_a = \left\{ \langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle \mid \begin{array}{l} i_1, \dots, i_m, o_1, \dots, o_n \in [\mu] \wedge \\ \mathbf{P}(\langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle) \end{array} \right\},$$

where for $\bar{a} = \langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle$, and $h_1(a) = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out})$, $\mathbf{P}(\bar{a})$ denotes the following condition (note that because $\mathbf{Int}_{I_1}(L_0)$ only contains ordinary graphs,

$h_1(a)$ is necessarily also an ordinary graph):

$$\left(\bigvee_{\substack{e \in E \\ \mathbf{nod}(e) = (v_1, v_2) \\ \mathbf{lab}(e) = d}} \left[\begin{array}{c} (\forall_{1 \leq j \leq m} (v_1 = \mathbf{in}(j) \implies \#\mathbf{in}_{\Sigma_2}(d) = i_j)) \\ \wedge \\ (\forall_{1 \leq j \leq n} (v_1 = \mathbf{out}(j) \implies \#\mathbf{in}_{\Sigma_2}(d) = o_j)) \\ \wedge \\ (\forall_{1 \leq j \leq m} (v_2 = \mathbf{in}(j) \implies \#\mathbf{out}_{\Sigma_2}(d) = i_j)) \\ \wedge \\ (\forall_{1 \leq j \leq n} (v_2 = \mathbf{out}(j) \implies \#\mathbf{out}_{\Sigma_2}(d) = o_j)) \end{array} \right] \right)$$

\wedge

$$\left[\begin{array}{c} (\forall_{1 \leq j, j' \leq m} (\mathbf{in}(j) = \mathbf{in}(j') \implies i_j = i_{j'})) \\ \wedge \\ (\forall_{1 \leq j, j' \leq n} (\mathbf{out}(j) = \mathbf{out}(j') \implies o_j = o_{j'})) \\ \wedge \\ (\forall_{1 \leq j \leq m, 1 \leq j' \leq n} (\mathbf{in}(j) = \mathbf{out}(j') \implies i_j = o_{j'})) \end{array} \right]$$

The typing on Σ' is defined as follows. For all $(\bar{a}, m \rightarrow n) \in \Sigma'$, where $\bar{a} = \langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle$:

$$\begin{aligned} \#\mathbf{in}_{\Sigma'}(\bar{a}) &= \sum_{j=1}^m i_j, \\ \#\mathbf{out}_{\Sigma'}(\bar{a}) &= \sum_{j=1}^n o_j. \end{aligned}$$

Now for all $a \in \Sigma_1$ the substitution σ is defined as follows:

$$\sigma(a) = \Sigma_a,$$

and the language L_T as follows:

$$L_T = \left\{ \bar{a}_1 \dots \bar{a}_k \left| \begin{array}{l} k \geq 0, \bar{a}_1 \dots \bar{a}_k \in \Sigma'^*, \forall_{1 \leq i < k} \mathbf{match}(\bar{a}_i, \bar{a}_{i+1}) \wedge \\ \#\mathbf{in}_{\Sigma'}(\bar{a}_1) = \#\mathbf{in}(\mathbf{L}) \wedge \#\mathbf{out}_{\Sigma'}(\bar{a}_k) = \#\mathbf{out}(\mathbf{L}) \end{array} \right. \right\} \cup \{(\lambda, \#\mathbf{in}(\mathbf{L}) \rightarrow \#\mathbf{in}(\mathbf{L}))\}$$

where **match** is defined as follows. For all $\bar{a}, \bar{b} \in \Sigma'$,

$$\begin{aligned}\bar{a} &= \langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle, \\ \bar{b} &= \langle b, \langle i'_1, \dots, i'_{m'} \rangle, \langle o'_1, \dots, o'_{n'} \rangle \rangle,\end{aligned}$$

$$\mathbf{match}(\bar{a}, \bar{b}) \iff \langle o_1, \dots, o_n \rangle = \langle i'_1, \dots, i'_{m'} \rangle.$$

Note that $\mathbf{match}(\bar{a}, \bar{b})$ implies that $\#\mathbf{out}(\bar{a}) = \#\mathbf{in}(\bar{b})$, but *not* the other way around! This condition, together with the conditions $\#\mathbf{in}_{\Sigma'}(\bar{a}_1) = \#\mathbf{in}(\mathbf{L})$ and $\#\mathbf{out}_{\Sigma'}(\bar{a}_k) = \#\mathbf{out}(\mathbf{L})$, guarantees that L_T , and therefore L' also, is a typed language with respect to Σ' . Finally, we now have fully and uniquely defined L' :

$$L' = \sigma(L_0) \cap L_T.$$

In order to completely define I' , we still have to specify h' . This interpretation function h' we define as follows⁷. For all $(\bar{a}, m \rightarrow n) \in \Sigma'$:

$$\bar{a} = \langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle,$$

and

$$h_1(a) = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out}),$$

we define:

$$h'(\bar{a}) = (V', E', \mathbf{nod}', \mathbf{lab}', \mathbf{in}', \mathbf{out}').$$

Here we distinguish two cases:

- The hypergraph $h_1(a)$ contains no edges; $E = \emptyset$,
- The hypergraph $h_1(a)$ contains exactly one edge; $E = \{e\}$.

Other cases do not occur, as I_1 is in ENF.

Case 1, $E = \emptyset$.

$$\begin{aligned}V' &= \{ \langle \mathbf{in}(j), k \rangle \mid 1 \leq j \leq m \text{ and } 1 \leq k \leq i_j \} \cup \\ &\quad \{ \langle \mathbf{out}(j), k \rangle \mid 1 \leq j \leq n \text{ and } 1 \leq k \leq o_j \}, \\ E' &= \mathbf{nod}' = \mathbf{lab}' = \emptyset, \\ \mathbf{in}' &= \begin{pmatrix} \langle \mathbf{in}[1], 1 \rangle, & \dots, & \langle \mathbf{in}[1], i_1 \rangle, \\ \vdots & \ddots & \vdots \\ \langle \mathbf{in}[m], 1 \rangle, & \dots, & \langle \mathbf{in}[m], i_m \rangle, \end{pmatrix}\end{aligned}$$

⁷Keep in mind that $\bar{a} \in \Sigma'$ implies that the condition $P(\bar{a})$ holds.

$$\mathbf{out}' = \begin{pmatrix} \langle \mathbf{out}[1], 1 \rangle, & \dots, & \langle \mathbf{out}[1], o_1 \rangle, \\ \vdots & \ddots & \vdots \\ \langle \mathbf{out}[n], 1 \rangle, & \dots, & \langle \mathbf{out}[m], o_n \rangle. \end{pmatrix}.$$

Case 2, $E = \{e\}$, $\mathbf{nod}(e) = (v_1, v_2)$, $\mathbf{lab}(e) = d$. Note that necessarily $v_1 \neq v_2$, because otherwise $h_1(a)$ would contain a loop, and therefore some $H \in \mathbf{Int}_{I_1}(L_0)$ also. As this is a language of string graphs, this cannot happen (see also Section 4.2).

$$\begin{aligned} V' &= \{ \langle \mathbf{in}(j), k \rangle \mid 1 \leq j \leq m \text{ and } 1 \leq k \leq i_j \text{ and } \mathbf{in}(j) \notin \{v_1, v_2\} \} \cup \\ &\quad \{ \langle \mathbf{out}(j), k \rangle \mid 1 \leq j \leq n \text{ and } 1 \leq k \leq o_j \text{ and } \mathbf{out}(j) \notin \{v_1, v_2\} \} \cup \\ &\quad V_{h_2(d)}, \\ E' &= E_{h_2(d)}, \\ \mathbf{nod}' &= \mathbf{nod}_{h_2(d)}, \\ \mathbf{lab}' &= \mathbf{lab}_{h_2(d)}, \\ \mathbf{in}' &= \mathbf{in}'_1 \dots \mathbf{in}'_m, \text{ where:} \end{aligned}$$

$$\mathbf{in}'_k = \begin{cases} (\langle \mathbf{in}(k), 1 \rangle, \dots, \langle \mathbf{in}(k), i_k \rangle) & \text{if } \mathbf{in}(k) \notin \{v_1, v_2\}, \\ \mathbf{in}_{h_2(d)} & \text{if } \mathbf{in}(k) = v_1, \\ \mathbf{out}_{h_2(d)} & \text{if } \mathbf{in}(k) = v_2. \end{cases}$$

$\mathbf{out}' = \mathbf{out}'_1 \dots \mathbf{out}'_n$, where:

$$\mathbf{out}'_k = \begin{cases} (\langle \mathbf{out}(k), 1 \rangle, \dots, \langle \mathbf{out}(k), o_k \rangle) & \text{if } \mathbf{out}(k) \notin \{v_1, v_2\}, \\ \mathbf{in}_{h_2(d)} & \text{if } \mathbf{out}(k) = v_1, \\ \mathbf{out}_{h_2(d)} & \text{if } \mathbf{out}(k) = v_2. \end{cases}$$

This fully completes the construction of I' .

We have now constructed an L' and an interpreter I' for L' , for which we claim that $\mathbf{Int}_{I'}(L') = \mathbf{L}$. However, we will not give a formal proof of this claim, but instead make it plausible by explaining how the construction works.

The construction used in this proof is based on the intuition that (8.5):

$$\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K))) = \mathbf{Int}(K)$$

holds because we can always “move” an outermost interpretation I_2 on the left-hand side into its corresponding innermost one I_1 , so that we get an interpretation I' that performs

“ $I_2 \circ I_1$ ”. This is done by applying the interpretation function h_2 of I_2 to the hypergraphs that occur in the definition of h_1 . In that process, every edge e that occurs in h_1 is replaced by the hypergraph $h_2(\mathbf{lab}(e))$. In order to do that, we need to split up the node $\mathbf{nod}(e, 1)$ into $\#\mathbf{in}(h_2(\mathbf{lab}(e)))$ new ones, and the node $\mathbf{nod}(e, 2)$ into $\#\mathbf{out}(h_2(\mathbf{lab}(e)))$ new ones, so that the replacing hypergraph $h_2(\mathbf{lab}(e))$ “fits”.

In that way, all nodes in h_1 get split up properly, except for those that are not incident with an edge. These nodes also need to split up, namely in p new ones, where $0 \leq p \leq \mu$. However, we do not directly know p . Therefore, using σ , we generate all possible “split-up’s”. Then to remove the ones that are incorrect, we intersect $\sigma(L_0)$ with the “type checking” language L_T . This L_T contains all sequences of split-up’s that match up. To be more precise, for an $(a, m \rightarrow n) \in \Sigma_1$, the typed set Σ_a of all its split-up’s consists of all symbols $\bar{a} = \langle a, \langle i_1, \dots, i_m \rangle, \langle o_1, \dots, o_n \rangle \rangle$, where the i ’s are used to indicate that the p th input node of $h_1(a)$ should be split up into i_p new nodes, and the q th output node into j_q new nodes.

So, $L' = \sigma(L_0) \cap L_T$ consists of all words $w \in L_0$, together with the intended split-up numbers of their symbols, which numbers are meaningful. Then $h'(\bar{a})$ is defined as $h_1(a)$ properly split up, with the eventual edge e replaced by $h_2(\mathbf{lab}(e))$. Note that as h_1 is in Edge Normal Form, there can be at most one edge. The h' thus obtained, intuitively, is “ h_2 applied to h_1 ”. Finally, the closure conditions on K for (8.5) arise from the operations used to define L' , i.e., to guarantee that $L' \in K$.

We would very much have liked to give a tangible example of all this. However, the tendency of the construction to lead to a combinatorial explosion seems to defy any attempt at such an example. The only examples that are manageable are the highly trivial ones, where \mathbf{L} consists of string graphs only. To give an indication of the exponential growth of the construction, if we would use the L and I from Example 1 in Section 7.3 (which are quite simple) for L_0 and I_1 , and about the simplest nontrivial interpretation I_2 (where, say, μ is only 2), Σ' already contains 747 symbols, and $\sigma(L_0)$ and L_T are apparently impossible to write down in a way that is essentially simpler than just giving their definition. But things are worse than that: I_1 is required to be in Edge Normal Form. This gives rise to even more combinatorial headaches!

Conclusion: there seems to be no way to give a proper example, one that is neither trivial, nor monstrous. We do consider this, together with absence of a formal proof, *major* shortcomings of our construction. We would very much welcome a better one.

For what it is worth, we are convinced that the conjecture (8.5) is true, because the construction and the intuition leading to it are quite convincing (to us). Nonetheless, the conjecture should be considered a conjecture, and nothing more. At best, its status may be described as an “interesting and promising direction for further research”.

8.4 About $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$

By the results of the previous section we can now make some strong statements on the class $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. To start with:

$$\mathbf{DB} \subsetneq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})). \quad (8.6)$$

Proof. Firstly, we prove that $\mathbf{DB} \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. By (3.6), $\mathbf{DB} = \mathbf{STR}(\mathbf{gr}(\mathbf{DB}))$, by (7.9), $\mathbf{gr}(\mathbf{DB}) \subseteq \mathbf{Int}(\mathbf{DB})$, and therefore also $\mathbf{STR}(\mathbf{gr}(\mathbf{DB})) \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{DB}))$, and finally, by Section 8.2, $\mathbf{Int}(\mathbf{DB}) = \mathbf{Int}(\mathbf{RLIN})$, and therefore also $\mathbf{STR}(\mathbf{Int}(\mathbf{DB})) = \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. Merging these equations, we get:

$$\mathbf{DB} = \mathbf{STR}(\mathbf{gr}(\mathbf{DB})) \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{DB})) = \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})).$$

So, $\mathbf{DB} \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. Now, secondly, we have to find a string language L such that $L \notin \mathbf{DB}$, but still $L \in \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. We take: $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. This L , as is well known from formal language theory, is not in \mathbf{DB} (not even in \mathbf{CF}). And as shown in Section 7.3, $L \in \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. This completes the proof of (8.6).

The above result enables us to extend our sequence of properly included classes of string-languages:

$$\mathbf{RLIN} \subsetneq \mathbf{LIN} \subsetneq \mathbf{DB} \subsetneq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})).$$

Furthermore, in the previous three sections, we proved that these classes are all the same under interpretation.

Now we have proven that \mathbf{RLIN} and $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$ are the same under interpretation, from this, and the definition of $\mathbf{Int}(K)$, it follows that for any class of languages K such that $\mathbf{RLIN} \subseteq K \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$ it must be the case that $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(K)$:

$$\mathbf{RLIN} \subseteq K \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})) \implies \mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(K). \quad (8.7)$$

Proof: because $\mathbf{RLIN} \subseteq K$, by (8.1), we have $\mathbf{Int}(\mathbf{RLIN}) \subseteq \mathbf{Int}(K)$. And because $K \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$, we also have $\mathbf{Int}(K) \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. Now from $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$ it directly follows that $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(K)$.

At this point, one might be curious whether there exists a class K still larger than $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$, such that $\mathbf{Int}(K) = \mathbf{Int}(\mathbf{RLIN})$. The answer is: *no*. Proof by reductio ad absurdum: given a class $K \supsetneq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$, suppose $\mathbf{Int}(K) = \mathbf{Int}(\mathbf{RLIN})$. Then of course also $\mathbf{STR}(\mathbf{Int}(K)) = \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. Now because $\mathbf{gr}(K) \subseteq \mathbf{Int}(K)$ (by (7.9)), $\mathbf{STR}(\mathbf{gr}(K)) \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. But then, by (3.6), $K \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$, and we have a contradiction.

Combining the above results, for any class K of string languages, we have:

$$\mathbf{Int}(K) \subseteq \mathbf{Int}(\mathbf{RLIN}) \iff K \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})). \quad (8.8)$$

The proof is just a recapitulation of earlier proofs in this section. Firstly, the direction \implies holds because, as proved in Section 8.3, $\mathbf{Int}(\mathbf{RLIN}) = \mathbf{Int}(\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})))$, and hence if $\mathbf{Int}(K) \subseteq \mathbf{Int}(\mathbf{RLIN})$, then also $K = \mathbf{STR}(\mathbf{gr}(K)) \subseteq \mathbf{STR}(\mathbf{Int}(K)) \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. Secondly, the direction \impliedby holds because, if $K \subseteq \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$, then also $\mathbf{Int}(K) \subseteq \mathbf{Int}(\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))) = \mathbf{Int}(\mathbf{RLIN})$.

Finally, the class $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$ was found to be equal to the class $\mathbf{OUT}(2\mathbf{DGSM})$ by⁸ Engelfriet and Heyker in 1991.

About this class $\mathbf{OUT}(2\mathbf{DGSM})$, the class of all output languages of two-way deterministic generalized sequential machines, quite a lot is known; for example, it is a substitution closed full AFL. For further properties, see [ERS80] and its references.

8.5 The power of interpretation theorems

From the results we have now obtained, we will derive two theorems on the power of interpretation. Given two classes K and K' , they give necessary and sufficient conditions for:

- $\mathbf{Int}(K) \subseteq \mathbf{Int}(K')$, and,
- $\mathbf{Int}(K) = \mathbf{Int}(K')$.

These conditions will be in terms of ordinary *string* languages only.

Power of Interpretation Theorem I:

For all classes K' that are closed under λ -free finite substitution, and under intersection with a regular language, we have that for *any* class K :

$$\mathbf{Int}(K) \subseteq \mathbf{Int}(K') \iff K \subseteq \mathbf{STR}(\mathbf{Int}(K')). \quad (8.9)$$

Proof. As in the proof of (8.8) we did not specifically use properties of \mathbf{RLIN} , other than that it is closed under λ -free finite substitution, and under intersection with a regular language, we can extend it from \mathbf{RLIN} to any class K' that satisfies these closure properties.

⁸See [EH91, page 356], where $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$ is called $\mathbf{STR}(\mathbf{LIN}\text{-CFHG})$. However, beware: their definition of \mathbf{STR} is not identical to ours. See also Section 11.4, where the relation between $\mathbf{Int}(\mathbf{RLIN})$ and $\mathbf{LIN}\text{-CFHG}$ is discussed (“essentially equal”).

Power of Interpretation Theorem II:

For all classes K and K' , such that both are closed under λ -free finite substitution, and under intersection with a regular language, we have:

$$\mathbf{Int}(K) = \mathbf{Int}(K') \iff \mathbf{STR}(\mathbf{Int}(K)) = \mathbf{STR}(\mathbf{Int}(K')). \quad (8.10)$$

Proof. The direction \implies is trivial. The direction \impliedby is as follows: if $\mathbf{STR}(\mathbf{Int}(K)) = \mathbf{STR}(\mathbf{Int}(K'))$, then obviously also $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K))) = \mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K')))$. By Section 8.3, $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K))) = \mathbf{Int}(K)$, and $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(K')) = \mathbf{Int}(K')$, from which the result then immediately follows.

These two theorems have clear implications. The second one, intuitively, says that for two classes of string languages K and K' , if we want to know whether $\mathbf{Int}(K)$ and $\mathbf{Int}(K')$ are equal, we just have to check some simple closure properties, and prove the equality for the string-graph languages in $\mathbf{Int}(K)$ and $\mathbf{Int}(K')$ only.

The first one gives, given a class K' that satisfies some closure properties, the largest class K such that $\mathbf{Int}(K)$ is still contained in $\mathbf{Int}(K')$.

8.6 Conclusions

The results from this chapter give strong indications on the “power” of interpretation. The fact that $\mathbf{Int}(\mathbf{RLIN})$ is equal to $\mathbf{Int}(\mathbf{DB})$, although there is quite a gap between \mathbf{RLIN} and \mathbf{DB} , tells us that a lot of complexity can be moved out of the interpreted language, into the interpreter. For example: we can choose a complex language $L \in \mathbf{DB}$ and any interpreter I for L , and then construct a simple language $L' \in \mathbf{RLIN}$ and an interpreter I' for L' that generates the same hypergraph language: $\mathbf{Int}_I(L) = \mathbf{Int}_{I'}(L')$. Somehow the complexity has “fled” out of L , into I' !

However, there are limits to the amount of complexity that can be moved in this way. As we will prove in Section 11.5, $\mathbf{Int}(\mathbf{CF})$ is a proper superset of $\mathbf{Int}(\mathbf{DB})$. Confusingly enough, there are languages that are not even context-free, and still under interpretation do not yield stronger results than \mathbf{RLIN} under interpretation (see section 8.4). As an example of this, take $L = \{a^n b^n c^n \mid n \in \mathbb{N}\}$. As shown in Section 7.3, $L \in \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. And by Section 8.3: $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))) = \mathbf{Int}(\mathbf{RLIN})$.

This is strange! On the one hand, the class $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$ contains a wealth of complex languages, amongst which all of \mathbf{DB} , and even non context-free languages. On the other hand, all the complexity of $\mathbf{Int}(\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN})))$ can be moved into an interpreter

acting on RLIN. Apparently, the complexity of $\mathbf{STR}(\mathbf{Int}(\text{RLIN}))$ is of a “specific kind” which can be “handled” by interpretation on RLIN. However, as noted, the complexity of CF is not of that specific kind, and it can not be handled by interpretation on RLIN. As a side result of these considerations, by the way, one can easily see that $\mathbf{STR}(\mathbf{Int}(\text{RLIN}))$ and CF are incomparable, i.e. neither is a subset of the other.

For the case of interpretation on RLIN, we have proven that the kind of complexity interpretation can “cope with”, is exactly the kind of complexity of the class $\mathbf{STR}(\mathbf{Int}(\text{RLIN}))$. And in general, for a class K that is closed under λ -free finite substitution, and under intersection with a regular language, interpretation on K can exactly cope with the complexity generated by $\mathbf{STR}(\mathbf{Int}(K))$, and nothing more (Power of Interpretation Theorem I).

Furthermore, the Power of Interpretation Theorem II is of great help to prove two classes equal under interpretation. If for some classes K and K' we want to prove that $\mathbf{Int}(K) = \mathbf{Int}(K')$, essentially all we have to do is to prove it for the string graphs only.

Still a question remains: given a class K , what is the nature of the class $\mathbf{STR}(\mathbf{Int}(K))$? What does it look like? What is its relation with other known classes? As for RLIN, we know that it leads to $\text{OUT}(2\text{DGSM})$. For a similar problem (but beyond the reach of the formalism of interpretation), Engelfriet and Heyker [EH91, page 355], arrived at the class $\text{OUT}(\text{DTWT})$, all output languages of deterministic tree-walking transducers. Hence, it seems reasonable to assume $\mathbf{STR}(\mathbf{Int}(K))$, for different K , will be equal to other known classes, and possibly even interesting unknown classes.

Maybe future research will give a more profound insight into the exact nature of $\mathbf{STR}(\mathbf{Int}(K))$, or in other words, into the nature of the complexity interpretation can cope with. In that way, interpretation might serve as a vehicle to obtain further knowledge in the field of traditional formal language theory; by taking a detour through the realm of graph languages, we have found complexity measures on string languages.

A mathematician is a machine for
converting coffee into theorems.

— A. N. Oonymous

9

Closure properties of $\mathbf{Int}(K)$

In this chapter we will examine the closure properties of $\mathbf{Int}(K)$ under several operations. We will give sufficient conditions for closure in terms of closure properties of K . Sometimes we can give several, mutually independent, sufficient conditions.

9.1 Closure under sequential composition

Sufficient condition: K closed under concatenation and isomorphism.

Proof. Given two hypergraph languages $\mathbf{L}_1, \mathbf{L}_2 \in \mathbf{Int}(K)$ choose two typed languages $L_1, L_2 \in L_\tau(K)$ over disjoint alphabets (the existence of which is guaranteed by the definition of \mathbf{Int} , the isomorphic copies theorem, and the closure of K under isomorphism), and two interpreters I_1, I_2 for L_1 and L_2 such that $\mathbf{Int}_{I_1}(L_1) = \mathbf{L}_1$ and $\mathbf{Int}_{I_2}(L_2) = \mathbf{L}_2$.

As K is closed under concatenation we know that $L_3 = L_1 \cdot L_2$ is in K . Taking $I_3 = I_1 \cup I_2$ (performing the union pairwise on all elements of the 3-tuple) as interpreter for L_3 we get $\mathbf{Int}_{I_3}(L_3) = \mathbf{L}_1 \cdot \mathbf{L}_2$. This is proved as follows:

$$\begin{aligned} \mathbf{Int}_{I_3}(L_3) &\stackrel{(7.3)}{=} \\ &\{ h_3(w) \mid w \in L_3 \} \stackrel{(\text{definition of } L_3)}{=} \\ &\{ h_3(w_1 \cdot w_2) \mid w_1 \in L_1, w_2 \in L_2 \} \stackrel{(7.1)}{=} \\ &\{ h_3(w_1) \cdot h_3(w_2) \mid w_1 \in L_1, w_2 \in L_2 \} \stackrel{(\text{definition of } h_3)}{=} \end{aligned}$$

$$\begin{aligned}
& \{ h_1(w_1) \cdot h_2(w_2) \mid w_1 \in L_1, w_2 \in L_2 \} \stackrel{(4.3)}{=} \\
& \{ h_1(w) \mid w \in L_1 \} \cdot \{ h_2(w) \mid w \in L_2 \} \stackrel{(7.3)}{=} \\
& \mathbf{Int}_{I_1}(L_1) \cdot \mathbf{Int}_{I_2}(L_2) \stackrel{(\text{definition of } L_1, L_2)}{=} \\
& L_1 \cdot L_2.
\end{aligned}$$

Therefore, $\mathbf{Int}(K)$ is closed under sequential composition. Proving $\mathbf{Int}(K)$ closed under union and Kleene closure works in a similar way. Be warned however: the following two sections are boringly alike this one!

9.2 Closure under union

Sufficient condition: K closed under union and isomorphism.

Proof. Given two hypergraph languages $L_1, L_2 \in \mathbf{Int}(K)$ choose two typed languages $L_1, L_2 \in L_\tau(K)$ over disjoint alphabets (the existence of which is guaranteed by the definition of \mathbf{Int} , the isomorphic copies theorem, and the closure under isomorphism), and two interpreters I_1, I_2 for L_1 and L_2 such that $\mathbf{Int}_{I_1}(L_1) = L_1$ and $\mathbf{Int}_{I_2}(L_2) = L_2$.

As K is closed under union we know that $L_3 = L_1 \cup L_2$ is in K . Taking $I_3 = I_1 \cup I_2$ (performing the union pairwise on all elements of the 3-tuple) as interpreter for L_3 we get $\mathbf{Int}_{I_3}(L_3) = L_1 \cup L_2$. This is proved as follows:

$$\begin{aligned}
& \mathbf{Int}_{I_3}(L_3) \stackrel{(7.3)}{=} \\
& \{ h_3(w) \mid w \in L_3 \} \stackrel{(\text{definition of } L_3)}{=} \\
& \{ h_3(w) \mid w \in L_1 \cup L_2 \} \stackrel{(\text{set theory})}{=} \\
& \{ h_3(w) \mid w \in L_1 \} \cup \{ h_3(w) \mid w \in L_2 \} \stackrel{(\text{definition of } h_3)}{=} \\
& \{ h_1(w) \mid w \in L_1 \} \cup \{ h_2(w) \mid w \in L_2 \} \stackrel{(7.3)}{=} \\
& \mathbf{Int}_{I_1}(L_1) \cup \mathbf{Int}_{I_2}(L_2) \stackrel{(\text{definition of } L_1, L_2)}{=} \\
& L_1 \cup L_2.
\end{aligned}$$

Therefore, $\mathbf{Int}(K)$ is closed under union.

9.3 Closure under Kleene closure

Sufficient condition: K closed under Kleene closure.

Proof. Given a hypergraph language $L \in \mathbf{Int}(K)$ choose a typed language $L \in L_\tau(K)$ and an interpreter I for L , such that $\mathbf{Int}_I(L) = L$.

As K is closed under Kleene closure we know that $L' = L^*$ is in K . We now have $\mathbf{Int}_I(L') = L^*$. This is proved as follows:

$$\begin{aligned}
 \mathbf{Int}_I(L') &\stackrel{(7.3)}{=} \\
 &\{ h(w) \mid w \in L' \} \stackrel{(\text{definition of } L')}{=} \\
 &\{ h(w) \mid w \in L^* \} \stackrel{(7.2)}{=} \\
 &\{ h(w) \mid w \in L \}^* \stackrel{(\text{definition of } L)}{=} \\
 &L^*.
 \end{aligned}$$

Therefore, $\mathbf{Int}(K)$ is closed under Kleene closure.

9.4 Closure under $+\{U_n\}$ and $\{U_n\}+$

Sufficient condition: TRUE.

Proof. Given a hypergraph language $L \in \mathbf{Int}(K)$ and an $n \in \mathbb{N}$, choose an $L \in L_\tau(K)$ and an $I = (\Sigma, \Delta, h)$, an interpreter for L , such that $\mathbf{Int}_I(L) = L$. Define $I' = (\Sigma', \Delta', h') = (\Sigma, \Delta, h + U_n)$. We now have: $\mathbf{Int}_{I'}(L) = L + \{U_n\}$. This can be proved as follows:

$$\begin{aligned}
 \mathbf{Int}_{I'}(L) &\stackrel{(7.3)}{=} \\
 &\{ h'(w) \mid w \in L \} \stackrel{(\text{rewriting } w \text{ as symbols})}{=} \\
 &\{ h'(a_1 \dots a_m) \mid a_1 \dots a_m \in L \} \stackrel{(7.2)}{=} \\
 &\{ h'(a_1) \dots h'(a_m) \mid a_1 \dots a_m \in L \} \stackrel{(\text{definition of } h')}{=} \\
 &\{ (h(a_1) + U_n) \dots (h(a_m) + U_n) \mid a_1 \dots a_m \in L \} \stackrel{(4.13)}{=} \\
 &\{ (h(a_1) \dots h(a_m)) + (U_n \dots U_n) \mid a_1 \dots a_m \in L \} \stackrel{(\text{unity})}{=} \\
 &\{ (h(a_1) \dots h(a_m)) + U_n \mid a_1 \dots a_m \in L \} \stackrel{(7.2)}{=} \\
 &\{ h(a_1 \dots a_m) + U_n \mid a_1 \dots a_m \in L \} \stackrel{(\text{rewriting symbols as } w)}{=} \\
 &\{ h(w) + U_n \mid w \in L \} \stackrel{(\text{definition of } +\{U_n\})}{=} \\
 &\{ h(w) \mid w \in L \} + \{U_n\} \stackrel{(7.3)}{=} \\
 &\mathbf{Int}_I(L) + \{U_n\} \stackrel{(\text{definition of } L)}{=} \\
 &L + \{U_n\}.
 \end{aligned}$$

Therefore, $\mathbf{Int}(K)$ is closed under $+\{U_n\}$. Similarly, we can prove $\mathbf{Int}(K)$ to be closed under $\{U_n\}+$.

9.5 Closure under parallel composition

Sufficient condition: $\mathbf{Int}(K)$ closed under sequential composition. So in terms of a sufficient condition on K : K closed under concatenation and isomorphism.

Proof. Let $L_1, L_2 \in \mathbf{Int}(K)$ be hypergraph languages. We can now derive:

$$\begin{aligned} (L_1 + \{U_{\#in(L_2)}\}) \cdot (\{U_{\#out(L_1)}\} + L_2) &\stackrel{(4.10)}{=} \\ L_1 \cdot \{U_{\#out(L_1)}\} + \{U_{\#in(L_2)}\} \cdot L_2 &\stackrel{(\text{unity})}{=} \\ L_1 + L_2. \end{aligned}$$

Because $\mathbf{Int}(K)$ is closed under all operations involved $(\cdot, +\{U_n\}, \{U_n\}+)$, see sections 9.1 and 9.4) this proves that $L_1 + L_2 \in \mathbf{Int}(K)$, so $\mathbf{Int}(K)$ is closed under parallel composition.

9.6 Closure under fold and backfold

Sufficient condition: $\mathbf{fold}(\{U_n\}) \in \mathbf{Int}(K)$ ($\mathbf{backfold}(\{U_n\}) \in \mathbf{Int}(K)$ for closure under **backfold**) and $\mathbf{Int}(K)$ closed under \cdot and $+$. So in terms of a sufficient condition on K : $\{a\} \in K$ for some symbol a , and K is closed under concatenation and isomorphism.

Proof. Let $L \in \mathbf{Int}(K)$ be a hypergraph language. We can now derive:

$$\begin{aligned} \mathbf{fold}(\{U_{\#in(L)}\}) \cdot (\{U_{\#in(L)}\} + L) &\stackrel{(6.32)}{=} \\ \mathbf{fold}(\mathbf{flip}(\{U_{\#in(L)}\}) \cdot \{U_{\#in(L)}\} \cdot L) &\stackrel{(6.25)}{=} \\ \mathbf{fold}(\{U_{\#in(L)}\} \cdot \{U_{\#in(L)}\} \cdot L) &\stackrel{(\text{unity})}{=} \\ \mathbf{fold}(L). \end{aligned}$$

Because $\mathbf{Int}(K)$ is closed under all operations involved $(\cdot, +)$, and because $\{U_{\#in(L)}\}$, $\mathbf{fold}(\{U_{\#in(L)}\}) \in \mathbf{Int}(K)$ as we can obtain it by interpreting $\{a\} \in K$ in the appropriate way, this proves that $\mathbf{fold}(L) \in \mathbf{Int}(K)$, so $\mathbf{Int}(K)$ is closed under folding. Proving $\mathbf{Int}(K)$ closed under backfolding works in the same manner, using (6.33) instead of (6.32).

9.7 Closure under flip

Sufficient condition: $\{U_n\} \in \mathbf{Int}(K)$ and $\mathbf{Int}(K)$ closed under \cdot , $+$, **fold**, and **backfold**. So in terms of a sufficient condition on K : $\{a\} \in K$ for some symbol a , and K is closed under concatenation and isomorphism.

Proof. Let $L \in \mathbf{Int}(K)$ be a hypergraph language. We can now derive:

$$\begin{aligned}
 & (\mathbf{fold}(L) + \{U_{\#out(L)}\}) \cdot (\{U_{\#in(L)}\} + \mathbf{backfold}(\{U_{\#out(L)}\})) \stackrel{(6.25)}{=} \\
 & (\mathbf{fold}(L) + \mathbf{flip}(\{U_{\#out(L)}\})) \cdot (\mathbf{flip}(\{U_{\#in(L)}\}) + \mathbf{backfold}(\{U_{\#out(L)}\})) \stackrel{(6.35)}{=} \\
 & \mathbf{flip}(\{U_{\#in(L)}\}) \cdot L \cdot \{U_{\#out(L)}\} \cdot \{U_{\#out(L)}\} \stackrel{(\text{unity})}{=} \\
 & \mathbf{flip}(L).
 \end{aligned}$$

Because $\mathbf{Int}(K)$ is closed under all operations involved (\cdot , $+$, **fold**, **backfold**), and because $\{U_{\#out(L)}\}, \{U_{\#in(L)}\} \in \mathbf{Int}(K)$ as we can obtain them by interpreting $\{a\} \in K$ in the appropriate way, this proves that $\mathbf{flip}(L) \in \mathbf{Int}(K)$, so $\mathbf{Int}(K)$ is closed under flipping. However, there is also another, very natural, sufficient condition for closure under flipping:

Sufficient condition: K is closed under reversal.

Proof. Given an $L \in \mathbf{Int}(K)$, choose an $L \in L_\tau(K)$ and an $I = (\Sigma, \Delta, h)$, an interpreter for L , such that $\mathbf{Int}_I(L) = L$. As we know that K is closed under reversal, $L^R \in K$. Define $I' = (\Sigma', \Delta', h') = (\Sigma, \Delta, \mathbf{flip} \circ h)$. We now have: $\mathbf{Int}_{I'}(L^R) = \mathbf{flip}(L)$. This can be proved as follows:

$$\begin{aligned}
 & \mathbf{Int}_{I'}(L^R) \stackrel{(7.3)}{=} \\
 & \{ h'(w) \mid w \in L^R \} \stackrel{(\text{definition of reversal})}{=} \\
 & \{ h'(w^R) \mid w \in L \} \stackrel{(\text{rewriting } w \text{ as symbols})}{=} \\
 & \{ h'(a_n \dots a_1) \mid a_1 \dots a_n \in L \} \stackrel{(7.2)}{=} \\
 & \{ h'(a_n) \dots h'(a_1) \mid a_1 \dots a_n \in L \} \stackrel{(\text{definition of } h')}{=} \\
 & \{ \mathbf{flip}(h(a_n)) \dots \mathbf{flip}(h(a_1)) \mid a_1 \dots a_n \in L \} \stackrel{(6.23)}{=} \\
 & \{ \mathbf{flip}(h(a_1) \dots h(a_n)) \mid a_1 \dots a_n \in L \} \stackrel{(7.2)}{=} \\
 & \{ \mathbf{flip}(h(a_1 \dots a_n)) \mid a_1 \dots a_n \in L \} \stackrel{(\text{rewriting symbols as } w)}{=} \\
 & \{ \mathbf{flip}(h(w)) \mid w \in L \} \stackrel{(\text{set theory})}{=} \\
 & \mathbf{flip}(\{ h(w) \mid w \in L \}) \stackrel{(7.3)}{=} \\
 & \mathbf{flip}(\mathbf{Int}_I(L)) \stackrel{(\text{definition of } L)}{=} \\
 & \mathbf{flip}(L).
 \end{aligned}$$

Therefore, $\mathbf{Int}(K)$ is closed under flipping.

9.8 Closure under split

Sufficient condition: $\{U_n\} \in \mathbf{Int}(K)$ and $\mathbf{Int}(K)$ closed under **fold**, **backfold**, \cdot and $+$. So in terms of a sufficient condition on K : $\{a\} \in K$ for some symbol a , and K is closed under concatenation and isomorphism.

Proof. Let $\mathbf{L} \in \mathbf{Int}(K)$ be a hypergraph language, and $p, q \in \mathbb{N}$ arbitrary integers such that $p + q = \#\mathbf{in}(\mathbf{L}) + \#\mathbf{out}(\mathbf{L})$. We can now derive:

$$\mathbf{split}_{p,q}(\mathbf{L}) \stackrel{(6.28)}{=} (\{U_p\} + \mathbf{fold}(\{U_q\}) \cdot (\mathbf{backfold}(\mathbf{L}) + \{U_q\})).$$

Because $\mathbf{Int}(K)$ is closed under all operations involved (**fold**, **backfold**, \cdot , $+$), and because $\{U_p\}, \{U_q\} \in K$ as we can obtain them by interpreting $\{a\} \in K$ in the appropriate way, this proves that $\mathbf{split}_{p,q}(\mathbf{L}) \in \mathbf{Int}(K)$, so $\mathbf{Int}(K)$ is closed under $\mathbf{split}_{p,q}$. As p and q were arbitrarily chosen, and by (6.29), this means that $\mathbf{Int}(K)$ is also closed under **split** (in general, without specified p and q).

9.9 Closure under edge relabeling

Sufficient condition: TRUE.

Proof. Let $\mathbf{L} \in \mathbf{Int}(K)$ be a hypergraph language over a ranked alphabet Δ , Δ' a ranked alphabet, and $f : \Delta \rightarrow \Delta'$ a rank preserving function ($\forall a \in \Delta \mathbf{rank}(f(a)) = \mathbf{rank}(a)$). Extend f in the obvious way to operate on hypergraphs, such that it returns the same hypergraph, only with the edges relabeled. Now choose a language $L \in L_\tau(K)$ and $I = (\Sigma, \Delta, h)$ an interpreter for L such that $\mathbf{Int}_I(L) = \mathbf{L}$. If we now choose $I' = (\Sigma, \Delta', f \circ h)$, then clearly $\mathbf{Int}_{I'}(L)$ is the hypergraph language that results from applying the edge relabeling f to (the edge labels in) \mathbf{L} . Therefore, $\mathbf{Int}(K)$ is closed under the relabeling of edges.

9.10 Conclusions

Summarizing the results from this chapter, we can draw the following table to express the sufficient conditions for closure of $\mathbf{Int}(K)$. Each row indicates for a certain operation which closure conditions on K are sufficient to guarantee closure on $\mathbf{Int}(K)$ under that operation. Needed conditions are marked by a bullet (\bullet), “edge relab.” stands for edge relabeling, and “isomorph” for isomorphism.

	\cdot	\cup	$*$	isomorph	$\{a\} \in K$	reversal
\cdot	•			•		
$+$	•			•		
\cup		•		•		
$*$			•			
$+\{U_n\}$						
$\{U_n\}+$						
fold	•			•	•	
backfold	•			•	•	
split _{p,q}	•			•	•	
flip	•			•	•	
flip						•
edge relab.						

Finally, note that RLIN and CF satisfy *all* mentioned closure conditions. Therefore, both **Int**(RLIN) and **Int**(CF) are closed under *all* mentioned operations.

10

If you cannot convince them, confuse them.

— Harry S. Truman

Another characterization

It turns out that there is another beautiful way to characterize the class of hypergraph languages $\mathbf{Int}(\mathbf{RLIN})$, *without* the concept of “interpreting” at all! This characterization is as follows:

$\mathbf{Int}(\mathbf{RLIN})$ is the smallest class of hypergraph languages that is closed under concatenation, parallel composition, union, and Kleene closure, and that contains the singleton class derived from the full base set.

Before we will be able to prove this characterization, we will first prove that some other characterizations that look like it also define the class $\mathbf{Int}(\mathbf{RLIN})$.

10.1 Using HGR

To begin with, we characterize $\mathbf{Int}(\mathbf{RLIN})$ as the smallest class of hypergraph languages that is closed under concatenation, union, and Kleene closure, and that contains all singleton hypergraph languages. To prove that this characterization indeed defines the class $\mathbf{Int}(\mathbf{RLIN})$, we need to prove three things. Firstly, by Section 9.10, it is clear that $\mathbf{Int}(\mathbf{RLIN})$ is indeed closed under concatenation, union, and Kleene closure. Secondly, it is trivial that $\mathbf{Int}(\mathbf{RLIN})$ contains all singleton hypergraph languages, as $\mathbf{Int}(\{\{a\}\})$ consists of exactly all singleton hypergraph languages (see Example 3 of Section 7.3), and clearly $\{a\} \in \mathbf{RLIN}$. Therefore by (8.1), $\mathbf{Int}(\{\{a\}\}) \subseteq \mathbf{Int}(\mathbf{RLIN})$. The third (and last) part is the hardest: $\mathbf{Int}(\mathbf{RLIN})$ is the *smallest* class that is closed under concatenation,

Kleene closure, and union, and that contains all singleton hypergraph languages. In order to prove this, we have to show that every hypergraph language $\mathbf{L} \in \mathbf{Int}(\mathbf{RLIN})$ over some alphabet Δ can be denoted by a finite expression over $\mathbf{Sing}(\mathbf{HGR}(\Delta))$, \cdot , $*$, and \cup (also called a *regular expression*).

Proof. Choose such an $\mathbf{L} \in \mathbf{Int}(\mathbf{RLIN})$. Let L be a typed right-linear language over some alphabet Σ , and $I = (\Sigma, \Delta, h)$ an interpreter for L , such that $\mathbf{Int}_I(L) = \mathbf{L}$. From formal language theory, it is a well known fact¹ that L can be expressed by a regular expression over \cdot , $*$, \cup , and Σ . So, let r be a regular (string) expression that denotes L . Now let r' be the regular (graph) expression derived from r by replacing every occurrence of $\{a\}$ by $\{h(a)\}$, for all $a \in \Sigma$. As we know that for all string languages L' and L'' :

- if $L' \cdot L''$ is typed, then so are L' and L'' ,
- if L'^* is typed, then so is L' ,
- if $L' \cup L''$ is typed, then so are L' and L'' ,
- $h(L' \cdot L'') = h(L') \cdot h(L'')$,
- $h(L'^*) = h(L')^*$,
- $h(L' \cup L'') = h(L') \cup h(L'')$,

this proves that $r' = h(r)$ (by induction on the length of r). Because $h(r) = \mathbf{Int}_I(L) = \mathbf{L}$, the regular expression r' thus obtained denotes the language \mathbf{L} . This completes the proof that every language $\mathbf{L} \in \mathbf{Int}(\mathbf{RLIN})$ can be denoted by a regular expression. Together with the first two parts, this third part concludes the overall proof that the new characterization indeed exactly characterizes the class $\mathbf{Int}(\mathbf{RLIN})$ we already know.

10.2 Using the sequential pseudo base set

Having obtained the result of the previous section, we can easily make it stronger by using the results from Chapter 5. In this way, we can characterize $\mathbf{Int}(\mathbf{RLIN})$ as the smallest class of hypergraph languages that is closed under concatenation, union, and Kleene closure, and that contains the singleton class derived from the sequential pseudo base set. In addition to what we already proved in the previous section, it now suffices to only prove that every singleton language in \mathbf{HGR} can be represented by an expression over the sequential pseudo base set and sequential composition. Or, formally expressed, $\mathbf{HGR} \xrightarrow{\rightarrow} \mathbf{L}_B$. But this we have already proved in Section 5.10, so we are done.

¹See for example Hopcroft and Ullman [HU79, §2.5], or Carroll and Long [CL89, §6.2].

Recall that the sequential pseudo base set contains exactly all hypergraphs that have at most one edge. So, intuitively, this characterization says that $\mathbf{Int}(\mathbf{RLIN})$ is the smallest class of hypergraph languages that is closed under concatenation, union, and Kleene closure, and that contains all singleton hypergraph languages $\{H\}$ where H has zero or one edges.

10.3 Using the full base set

The same as in the previous section can be done for the full base set. Now, we characterize $\mathbf{Int}(\mathbf{RLIN})$ as the smallest class of hypergraph languages that is closed under concatenation, parallel composition, union, and Kleene closure, and that contains the singleton class derived from the full base set (this is the same characterization as given at the beginning of this chapter). The additionally needed proof follows directly from the fact that $\mathbf{Int}(\mathbf{RLIN})$ is closed under parallel composition (Section 9.5), and $\mathbf{HGR} \xrightarrow{+} L_C$.

Recall that the full set consists of all “edges” and six “auxiliary hypergraphs”. So, intuitively, this characterization says that $\mathbf{Int}(\mathbf{RLIN})$ is the smallest class of hypergraph languages that is closed under concatenation, parallel composition, union, and Kleene closure, and that contains all “edges” (and a few auxiliary hypergraphs).

10.4 Conclusions

There is strong resemblance between the characterization given at the beginning of this chapter, and the characterization (from formal language theory) of \mathbf{RLIN} as the smallest class that is closed under concatenation, union, and Kleene closure, and that contains all “symbols” (i.e., languages of the form $\{a\}$, where a is a symbol). Just as that characterization intuitively says “the class of regular string languages can be build by regular expressions over the basic building blocks, namely symbols”, ours says “the class $\mathbf{Int}(\mathbf{RLIN})$ can be build by regular expressions over the basic building blocks, namely edges”. We think that because of this resemblance, $\mathbf{Int}(\mathbf{RLIN})$ deserves to be called “the class of regular graph languages”.

There's too many men, too many people,
making too many problems. Can't you
see this is a land of confusion?

— *Phil Collins*

11

Other literature

More than 20 years ago, the concept of a graph grammar was introduced by A. Rosenfeld as a formulation of some problems in pattern recognition and image processing, as well as by H. J. Schneider in Germany as a method for data type specification. In this chapter we will make a comparison between our $\mathbf{Int}(K)$ classes of hypergraph languages, and some classes of hypergraph languages generated by graph grammars on which results have been published in the scientific literature. In particular we will find that $\mathbf{Int}(\text{RLIN}) \approx \text{LIN-CFHG}^1$, where LIN-CFHG is the class of (languages generated by) linear context-free hypergraph grammars, as defined by Engelfriet and Heyker.

11.1 Introduction

Be aware that there is no universally, or even widely, agreed on concept of the “right” way to define a graph grammar. Instead, there are many fundamentally different approaches. For example: what should nonterminals stand for, nodes, edges, or perhaps even something else? As a consequence of this controversy, seemingly simple questions like “what is context freeness with regard to a graph grammar?” have no conclusive answer (yet).

Although our means of defining hypergraph languages by interpretation has strictly spoken little or nothing to do with a graph *grammar*, a string grammar/interpreter pair

¹We write \approx instead of $=$ because their hypergraphs have only one sequence of external nodes, while ours have two. Except for that small matter, both classes are completely identical. We will turn this into a precise formal statement, and prove it, in Section 11.4.

is, in our view, a “graph grammar in disguise”. Not surprisingly, some of our results (for example, our claim that the class $\mathbf{Int}(\mathbf{RLIN})$ deserves to be called “the class of regular graph languages”, and the decomposition of $\mathbf{HGR}(\Delta)$ in a base set) are strikingly similar to some results derived using a hyper-edge replacement graph grammar approach. Hence the title of this thesis: “Graphs Grammars and Operations on Graphs”.

In what follows, we will make comparisons to work of Engelfriet and Heyker ([EH91], [EH92]), Bauderon and Courcelle ([BC87]), and Habel and Kreowski ([Hab92]), all of which are hyper-edge replacement oriented approaches.

No comparison whatsoever is made with node-replacement oriented approaches. It may be interesting to observe that node replacement is in general more powerful than edge replacement. For example, the set of all complete graphs can easily be generated by a node-replacement graph grammar, but not by means of the above mentioned edge-replacement graph grammars.

For an overview of developments in the field, recent proceedings of the “International Workshop on Graph Grammars and Their Application to Computer Science” ([ENRR87], [EKR91]) are a good place to start.

11.2 Engelfriet and Heyker

The class $\mathbf{Int}(\mathbf{RLIN})$ we defined is very similar to the class $\mathbf{LIN-CFHG}$ defined by Engelfriet and Heyker in 1991 [EH91]. Before we can make precise statements, however, we need a way to “translate” their notion of “hypergraphs” into our idea of hypergraphs as defined in Chapter 2.

For clarity’s sake, we will call the “hypergraphs” as defined by Engelfriet and Heyker *BC-hypergraphs*², and correspondingly their equivalent of our set of all hypergraphs over a ranked alphabet Δ , $\mathbf{HGR}(\Delta)$, we will call $\mathbf{BC-HGR}(\Delta)$. The reader is referred to [EH91, pages 330–331] for the exact definitions of BC-hypergraphs, and the notation involved. Let it suffice here to summarize that BC-hypergraphs are very much like our hypergraphs albeit that they do not distinguish between input nodes and output nodes. Instead, they have only one kind of external node, called *external node*(!)³. The sequence of all external nodes is called \mathbf{ext} . A BC-hypergraph \mathcal{H} is denoted as: $\mathcal{H} = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{ext})$, where V , E , \mathbf{nod} , and \mathbf{lab} are defined in the same manner as we do, and $\mathbf{ext} \in V^*$. The “type” of a BC-hypergraph is called *rank*, and is defined as the length of \mathbf{ext} : $\mathbf{rank}(\mathcal{H}) = |\mathbf{ext}|$. This

²After Bauderon and Courcelle, who first proposed this kind of hypergraph.

³Note the difference between an external node of a BC-hypergraph, as defined here, and an external node of a hypergraph in our definition, as defined on page 22. Luckily, although defined on different kinds of hypergraphs, both concepts of external node stand for exactly the same thing.

brings us to our translation functions Γ and Γ^\leftarrow . Let Δ be a ranked alphabet.

$\Gamma : \mathbf{BC-HGR}(\Delta) \rightarrow \mathbf{HGR}(\Delta)$, for a BC-hypergraph $\mathcal{H} = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{ext})$:

$$\Gamma(\mathcal{H}) \stackrel{\text{def}}{=} (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{ext}, \lambda),$$

$\Gamma^\leftarrow : \mathbf{HGR}(\Delta) \rightarrow \mathbf{BC-HGR}(\Delta)$, for a hypergraph $H = (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in}, \mathbf{out})$:

$$\Gamma^\leftarrow(H) \stackrel{\text{def}}{=} (V, E, \mathbf{nod}, \mathbf{lab}, \mathbf{in} \cdot \mathbf{out}).$$

Note that for $\Gamma(\mathcal{H})$ we simply use the sequence of external nodes of \mathcal{H} as input nodes, and leave the sequence of output nodes empty. For $\Gamma^\leftarrow(H)$ we concatenate **in** and **out** and use this product as sequence of external nodes. Be aware that although Γ and Γ^\leftarrow are each other's inverse in some sense, they are *not* so mathematically speaking. The connection between the two of them is as follows (let $H \in \mathbf{HGR}(\Delta)$ of type $(m \rightarrow n)$, and $\mathcal{H} \in \mathbf{BC-HGR}(\Delta)$ for some ranked alphabet Δ):

$$\Gamma^\leftarrow(\Gamma(\mathcal{H})) = \mathcal{H}, \tag{11.1}$$

$$\Gamma(\Gamma^\leftarrow(H)) = \mathbf{backfold}(H), \tag{11.2}$$

$$\mathbf{split}_{m,n}(\Gamma(\Gamma^\leftarrow(H))) = H. \tag{11.3}$$

The proofs follow immediately from the definitions involved. Note that from (11.2), (11.3), and (6.12) it follows that for any class \mathbf{K} of hypergraph languages that is closed under **split**, we have:

$$\mathbf{split}(\Gamma(\Gamma^\leftarrow(\mathbf{K}))) = \mathbf{K}. \tag{11.4}$$

11.3 Context-Free Hypergraph Grammars

Using BC-hypergraphs, Engelfriet and Heyker follow a hyperedge replacement graph grammar approach to define classes of graph languages. Their grammars have the form (Σ, Δ, P, S) , where Σ is the (ranked) alphabet of edge labels, $\Delta \subseteq \Sigma$ is the alphabet of terminal edge labels, P is the (finite) set of productions, and $S \in \Sigma - \Delta$ is the initial nonterminal. Every production π in P is of the form $\pi = (X, \mathcal{H})$, where $X \in \Sigma - \Delta$ is a nonterminal symbol, and $\mathcal{H} \in \mathbf{BC-HGR}(\Sigma)$ is a hypergraph, such that $\mathbf{rank}_\Sigma(X) = \mathbf{rank}(\mathcal{H})$.

In short, a derivation in such a grammar proceeds as follows. Let $n = \mathbf{rank}_\Sigma(S)$. To begin, one takes a hypergraph which consists only of one edge, labeled S , and n nodes, all of which external:

$$(\{v_1, \dots, v_n\}, \{e\}, \mathbf{nod}(e) \mapsto (v_1, \dots, v_n), \mathbf{lab}(e) \mapsto S, (v_1, \dots, v_n)).$$

Then in each step, one chooses an edge labeled by a nonterminal symbol $X \in \Sigma - \Delta$, and a production $\pi = (X, \mathcal{H})$. The hypergraph under consideration then gets his chosen edge replaced with the hypergraph \mathcal{H} , where the former attachment nodes of the tentacles of the edge get identified with the corresponding external nodes of \mathcal{H} . This edge replacement process continues until we reach the moment where all edge labels are terminal. At that point, our derivation has completed. Note that all “sentential forms”, including the finally derived hypergraph, are of rank n .

For a given grammar $G = (\Sigma, \Delta, P, S)$, the hypergraph language $L(G)$ consists of all hypergraphs over Δ that have a derivation in G . The class of all hypergraph languages obtainable in this way, is denoted CFHG, standing for “*Context-Free Hypergraph Grammar*”. If we put the restriction on the grammars that the right hand side of a production may at most contain *one* nonterminal edge, we get the class LIN-CFHG (for Linear Context-Free Hypergraph Grammar). Loosely speaking, we will also use LIN-CFHG and CFHG to denote the sets of their *grammars*.

11.4 $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN-CFHG})) = \mathbf{Int}(\mathbf{RLIN})$

Firstly, we prove $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN-CFHG})) \subseteq \mathbf{Int}(\mathbf{RLIN})$. Given a linear grammar $G = (\Sigma, \Delta, P, S)$ conform [EH91], and two integers $m, n \in \mathbb{N}$ such that $\mathbf{rank}(S) = m + n$, we construct⁴ a right-linear typed grammar $G' = (N', T', P', S')$, that generates the typed language $L(G')$ of type $(m \rightarrow n)$, and an interpreter $I' = (\Sigma', \Delta', h')$ (where $\Sigma' = T'$ and $\Delta' = \Delta$) for $L(G)$, such that $\mathbf{Int}_{I'}(L(G')) = \mathbf{split}_{m,n}(\mathbf{\Gamma}(L(G)))$:

- $N' = (\Sigma - \Delta) \cup \{D\}$, $D \notin (\Sigma - \Delta)$,

where for $A \in N'$, $A \neq D$, its type is defined by $\#\mathbf{in}_{N'}(A) = \mathbf{rank}_{\Sigma}(A) + n$, and $\#\mathbf{out}_{N'}(A) = 0$. For D , $\#\mathbf{in}_{N'}(D) = m$, and $\#\mathbf{out}_{N'}(D) = n$,

- $T' = \{a_{\pi} \mid \pi \in P\} \cup \{b\}$,

where for $\pi = (X, \mathcal{H})$, the type of a_{π} is defined by $\#\mathbf{in}_{T'}(a_{\pi}) = \mathbf{rank}(\mathcal{H}) + n = \mathbf{rank}_{\Sigma}(X) + n$, and $\#\mathbf{out}_{T'}(a_{\pi}) = \mathbf{rank}_{\Sigma}(\mathbf{lab}_{\mathcal{H}}(\mathbf{nont}(\mathcal{H}))) + n$ if \mathcal{H} contains one nonterminal, and 0 otherwise. For b , $\#\mathbf{in}_{T'}(b) = m$, and $\#\mathbf{out}_{T'}(b) = m + 2n$,

- $P' = \{p_{\pi} \mid \pi \in P\} \cup \{q\}$.

⁴The construction is based on $\mathbf{split}_{m,n}(\mathbf{\Gamma}(\mathcal{H})) = (U_m + \mathbf{fold}(U_n)) \cdot (\mathbf{\Gamma}(\mathcal{H}) + U_n)$, by (6.28).

If $\pi = (X, \mathcal{H})$ and \mathcal{H} contains exactly one nonterminal edge (namely $\mathbf{nont}(\mathcal{H})$), then the production p_π looks as follows⁵:

$$\begin{aligned} p_\pi : X &\rightarrow a_\pi \mathbf{lab}_{\mathcal{H}}(\mathbf{nont}(\mathcal{H})), \\ h'(a_\pi) &= (V_{\mathcal{H}}, E_{\mathcal{H}} - \mathbf{nont}(\mathcal{H}), \mathbf{nod}_{\mathcal{H}}, \mathbf{lab}_{\mathcal{H}}, \mathbf{ext}_{\mathcal{H}}, \mathbf{nod}_{\mathcal{H}}(\mathbf{nont}(\mathcal{H}))) + U_n. \end{aligned}$$

If $\pi = (X, \mathcal{H})$ and \mathcal{H} does not contain a nonterminal edge (the only other alternative; as G is linear \mathcal{H} cannot contain more than one nonterminal edge) then the production p_π looks as follows:

$$\begin{aligned} p_\pi : X &\rightarrow a_\pi, \\ h'(a_\pi) &= \Gamma(\mathcal{H}) + U_n. \end{aligned}$$

The production q is defined as follows:

$$q : D \rightarrow bS,$$

and $h'(b) = U_m + \mathbf{fold}(U_n)$,

- $S' = D$,
- $\Sigma' = \Sigma$, $\Delta' = \Delta$, and h' as defined above.

We now claim that $\mathbf{split}_{m,n}(\Gamma(\mathbf{L}(G))) = \mathbf{Int}_I(\mathbf{L}(G'))$. Instead of giving a full formal proof, we will give the invariant that describes the relation between the derivations in G and G' .

Invariant:

For all $\mathcal{H} \in \mathbf{BC}\text{-HGR}(\Sigma)$, and $j \in \mathbb{N}$:

$$\begin{aligned} G : S &\Rightarrow^j \mathcal{H} \\ &\iff \\ \exists_{w \in T'^*, A \in N'} &\left(G' : S \Rightarrow^j wA \text{ and } h'(w) \cdot (\mathbf{H}(A) + U_n) = \Gamma(\mathcal{H}) + U_n \right), \end{aligned}$$

where, for $k = \mathbf{rank}_\Sigma(A)$:

$$\mathbf{H}(A) = (\{v_1, \dots, v_k\}, \{e\}, \mathbf{nod}(e) \mapsto (v_1, \dots, v_k), \mathbf{lab}(e) \mapsto A, (v_1, \dots, v_k), \lambda).$$

The proof of the correctness of this invariant, and of the fact that our claim follows from it, then proceeds along similar lines as in the comparable proof in Section 8.1. The validity

⁵Note the λ -case! (See also the footnote on page 14.)

of this claim completes the proof of $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN}\text{-CFHG})) \subseteq \mathbf{Int}(\mathbf{RLIN})$.

Conversely, we prove $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN}\text{-CFHG})) \supseteq \mathbf{Int}(\mathbf{RLIN})$. Given a right-linear typed grammar $G = (N, T, P, S)$ that generates the typed language $L(G)$ of type $(m \rightarrow n)$, and an interpreter $I = (\Sigma, \Delta, h)$ (where $\Sigma = T$) for $L(G)$, we construct a LIN-CFHG grammar $G' = (\Sigma', \Delta', P', S')$ such that $L(G') = \mathbf{\Gamma}^{-1}(\mathbf{Int}_I(L(G)))$, and hence, by (11.3), $\mathbf{split}_{m,n}(\mathbf{\Gamma}(L(G'))) = \mathbf{Int}_I(L(G))$:

- $\Sigma' = N \cup \Delta$,

$$\text{where for } a \in \Sigma', \mathbf{rank}_{\Sigma'}(a) = \begin{cases} \#\mathbf{in}_N(a) + \#\mathbf{out}_N(a) & \text{if } a \in N, \\ \mathbf{rank}_{\Delta}(a) & \text{if } a \in \Delta, \end{cases}$$

- $\Delta' = \Delta$ (the ranks of its symbols have already been determined above),
- $P' = \{ \pi_p \mid p \in P \}$.

If p has the form $p : A \rightarrow wB$, and $m' = \#\mathbf{out}_T(w) = \#\mathbf{in}_N(B)$, then the production π_p looks as follows⁶:

$$\pi_p = (A, \mathbf{\Gamma}^{-1}(h(w) \cdot \mathbf{H}(B))),$$

where:

$$\mathbf{H}(B) = (\{v_1, \dots, v_{m'+n}\}, \{e\}, \mathbf{nod}(e) \mapsto (v_1, \dots, v_{m'+n}), \\ \mathbf{lab}(e) \mapsto B, (v_1, \dots, v_{m'}), (v_{m'+1}, \dots, v_{m'+n})).$$

Note that this \mathbf{H} is type preserving.

If $p : A \rightarrow w$, then the production π_p looks as follows:

$$\pi_p = (A, \mathbf{\Gamma}^{-1}(h(w))),$$

- $S' = S$.

We now claim that $\mathbf{split}_{m,n}(\mathbf{\Gamma}(L(G'))) = \mathbf{Int}_I(L(G))$. Again, instead of giving a full formal proof, we will give the invariant that describes the relation between the derivations in G and G' .

⁶Note the λ -case! (See also the footnote on page 14.)

Invariant:

For all $\mathcal{H} \in \mathbf{BC}\text{-HGR}(\Sigma')$, and $j \in \mathbb{N}$:

$$\begin{aligned} \exists_{w \in T^*, A \in N} \left(G : S \Rightarrow^j wA \text{ and } \mathcal{H} = \mathbf{\Gamma}^-(h(w) \cdot \mathbf{H}(A)) \right) \\ \iff \\ G' : S' \Rightarrow^j \mathcal{H}. \end{aligned}$$

The validity of this claim completes the proof of $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN}\text{-CFHG})) \supseteq \mathbf{Int}(\mathbf{RLIN})$. Together with the previous result $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN}\text{-CFHG})) \subseteq \mathbf{Int}(\mathbf{RLIN})$, this completes the overall proof of $\mathbf{split}(\mathbf{\Gamma}(\mathbf{LIN}\text{-CFHG})) = \mathbf{Int}(\mathbf{RLIN})$.

What does this mean, i.e., what does $\mathbf{split} \circ \mathbf{\Gamma}$ do? Well, $\mathbf{\Gamma}$ just translates the sequence \mathbf{ext} of a BC-hypergraph in the sequences $\mathbf{in} = \mathbf{ext}$, and $\mathbf{out} = \lambda$, of an i/o-hypergraph. Then, \mathbf{split} just redistributes all input nodes over \mathbf{in} and \mathbf{out} , in all possible ways such that the order stays the same (i.e., $\mathbf{in} \cdot \mathbf{out}$ stays invariant). So, $\mathbf{split} \circ \mathbf{\Gamma}$ does not change the structure of its argument at all! The only thing it does is translate and redistribute external nodes. Therefore, intuitively speaking, $\mathbf{LIN}\text{-CFHG}$ contains exactly the same hypergraph languages as $\mathbf{Int}(\mathbf{RLIN})$, albeit that the former contains BC-hypergraphs, and the latter i/o-hypergraphs.

11.5 $\mathbf{Int}(\mathbf{RLIN}) \subsetneq \mathbf{Int}(\mathbf{CF})$

The proof of $\mathbf{Int}(\mathbf{RLIN}) \subsetneq \mathbf{Int}(\mathbf{CF})$ is quite involved. We will derive it from a result by Engelfriet and Heyker [EH91], who in turn rely on a result by Greibach [Gre78].

Firstly, by (8.1), it is obvious that $\mathbf{Int}(\mathbf{RLIN}) \subseteq \mathbf{Int}(\mathbf{CF})$. Secondly, on page 357 of [EH91] it is shown that there exists a string language $L \in \mathbf{L}(\mathbf{CF})$, such that $L \notin \mathbf{STR}(\mathbf{LIN}\text{-CFHG})$, so⁷ we also have $L \notin \mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$. From this, by the Power of Interpretation theorem I (or more specifically, by (8.8) for $K = \{L\}$), it follows that:

$$\exists_{L \in \mathbf{Int}(L)} L \notin \mathbf{Int}(\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))),$$

so by the results from Section 8.3:

$$\exists_{L \in \mathbf{Int}(L)} L \notin \mathbf{Int}(\mathbf{RLIN}).$$

Choose such a hypergraph language L . Now, as $L \in \mathbf{L}(\mathbf{CF})$, by (7.5), we have $\mathbf{Int}(L) \subseteq \mathbf{Int}(\mathbf{CF})$. Therefore, $L \notin \mathbf{Int}(\mathbf{RLIN})$, and at the same time, $L \in \mathbf{Int}(\mathbf{CF})$. This completes the proof of $\mathbf{Int}(\mathbf{RLIN}) \subsetneq \mathbf{Int}(\mathbf{CF})$.

⁷Note that, in [EH91], \mathbf{STR} is defined slightly different than here. In this section, however, we will only use $\mathbf{STR}(\mathbf{LIN}\text{-CFHG})$, which is equal to $\mathbf{STR}(\mathbf{Int}(\mathbf{RLIN}))$, as can be seen from the respective definitions of \mathbf{STR} , and the results from Section 11.4.

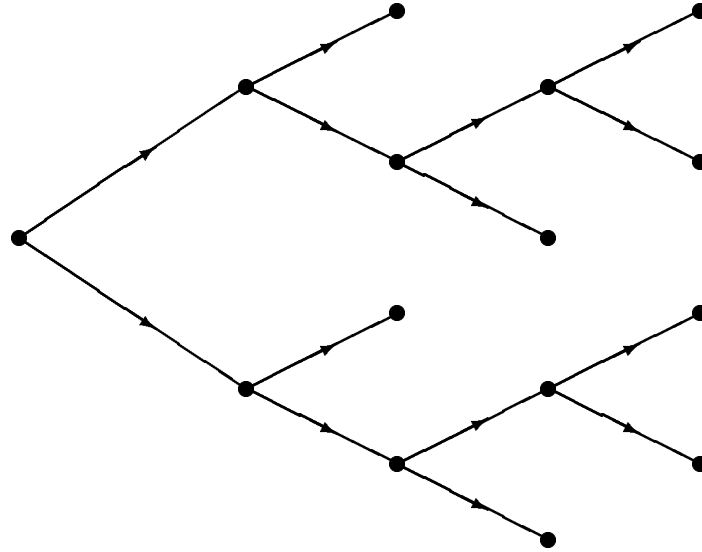
11.6 $\mathbf{Int}(\mathbf{CF}) \subsetneq \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$

In a sense, $\mathbf{Int}(\mathbf{CF})$ is completely contained in \mathbf{CFHG} , but the opposite does not hold: there are hypergraph languages in \mathbf{CFHG} that have no equivalent in $\mathbf{Int}(\mathbf{CF})$. Formally expressed:

$$\mathbf{Int}(\mathbf{CF}) \subsetneq \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG})). \quad (11.5)$$

In order to prove this, we need to do two things. Firstly, we will give a hypergraph language \mathbf{L} , the language of all binary trees, such that $\mathbf{L} \notin \mathbf{Int}(\mathbf{CF})$ and $\mathbf{L} \in \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$. Then, secondly, for every hypergraph language $\mathbf{L} \in \mathbf{Int}(\mathbf{CF})$ of type $(m \rightarrow n)$, we will give a graph grammar $G' \in \mathbf{CFHG}$ such that $\mathbf{split}_{m,n}(\mathbf{\Gamma}(\mathbf{L}(G'))) = \mathbf{L}$.

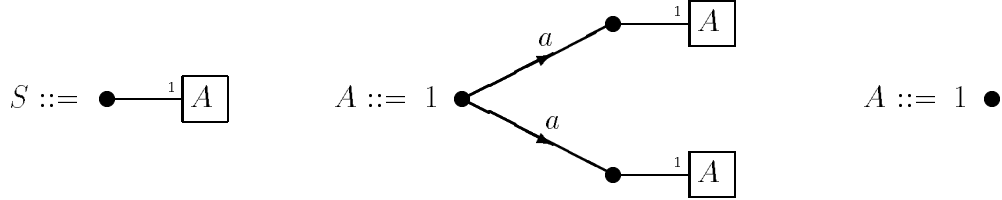
Firstly, the language \mathbf{L} of all binary trees. By a binary tree we here mean an ordinary graph of type $(0 \rightarrow 0)$ that forms a binary tree, and where the direction of all edges is from root to leaves. Instead of giving a complete formal definition, we give an example (the edge labels have been left out, as they are all a):



This language \mathbf{L} is not in $\mathbf{Int}(\mathbf{CF})$, as it is of bounded degree 3, but by [Len82] not of bounded cutwidth. Such a language, as we have proven in Section 7.6, can impossibly be obtained by interpretation.

That \mathbf{L} is indeed in $\mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$, is shown by the following grammar $G \in \mathbf{CFHG}$. $G = (\Sigma, \Delta, P, S)$, where $\Sigma = \{a, A, S\}$ (a of rank 2, A of rank 1, and S of rank 0), $\Delta = \{a\}$,

$P = \{\pi_1, \pi_2, \pi_3\}$. The three productions in P look as follows (we mimic the notation of [EH91]):



Clearly, this G generates all binary trees, or formally, $\mathbf{split}_{0,0}(\mathbf{\Gamma}(\mathbf{L}(G))) = \mathbf{L}$. Therefore, $\mathbf{L} \in \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$.

Secondly, for a given hypergraph language $\mathbf{L} \in \mathbf{Int}(\mathbf{CF})$ of type $(m \rightarrow n)$, we will construct a graph grammar $G' \in \mathbf{CFHG}$ such that $\mathbf{L}(G') = \mathbf{\Gamma}^{\leftarrow}(\mathbf{L})$, so $\mathbf{split}_{m,n}(\mathbf{\Gamma}(\mathbf{L}(G'))) = \mathbf{L}$. This grammar G' can be constructed as follows⁸. Let $G = (N, T, P, S)$ be a typed grammar, and $I = (\Sigma, \Delta, h)$ (where $\Sigma = T$) an interpreter for $\mathbf{L}(G)$, such that $\mathbf{Int}_I(\mathbf{L}(G)) = \mathbf{L}$. Now choose $G' = (\Sigma', \Delta', P', S')$ as follows:

- $\Sigma' = N \cup \Delta$,

$$\text{where for } a \in \Sigma', \mathbf{rank}_{\Sigma'}(a) = \begin{cases} \#\mathbf{in}_N(a) + \#\mathbf{out}_N(a) & \text{if } a \in N, \\ \mathbf{rank}_{\Delta}(a) & \text{if } a \in \Delta, \end{cases}$$

- $\Delta' = \Delta$ (the ranks of its symbols have already been determined above),
- $P' = \{\pi_p \mid p \in P\}$.

If p has the form⁹:

$$A_0 \rightarrow w_1 A_1 w_2 \dots w_{k-1} A_{k-1} w_k,$$

for some $k \geq 1$, and for all $1 \leq i < k$, $m_i = \#\mathbf{in}_N(A_i)$ and $n_i = \#\mathbf{out}_N(A_i)$, then the production π_p looks as follows:

$$\pi_p = (A_0, \mathbf{\Gamma}^{\leftarrow}(h(w_1) \cdot \mathbf{H}(A_1) \cdot h(w_2) \cdot \dots \cdot h(w_{k-1}) \cdot \mathbf{H}(A_{k-1}) \cdot h(w_k))),$$

where:

$$\mathbf{H}(A_i) = (\{v_1, \dots, v_{m_i+n_i}\}, \{e\}, \mathbf{nod}(e) \mapsto (v_1, \dots, v_{m_i+n_i}), \mathbf{lab}(e) = A_i, (v_1, \dots, v_{m_i}), (v_{m_i+1}, \dots, v_{m_i+n_i})) : m_i \rightarrow n_i.$$

Note that this \mathbf{H} is type preserving.

⁸The construction is a more-or-less straightforward extension of the second construction in Section 11.4.

⁹Note the λ -case! (See also the footnote on page 14.)

- $S' = S$.

We now claim that $\mathbf{split}_{m,n}(\mathbf{\Gamma}(L(G'))) = \mathbf{L}$. Instead of giving a full formal proof, we will give the invariant that describes the relation between the derivations in G and G' .

Invariant:

For all $\mathcal{H} \in \mathbf{BC}\text{-HGR}(\Sigma')$, $k = |\mathbf{nont}(\mathcal{H})| + 1$, and $j \in \mathbb{N}$:

$$\left[\exists_{\substack{w_1, \dots, w_k \in T^* \\ A_1, \dots, A_{k-1} \in N}} \left(\begin{array}{c} G : S \Rightarrow^j w_1 A_1 w_2 \dots w_{k-1} A_{k-1} w_k \\ \text{and} \\ \mathcal{H} = \mathbf{\Gamma}^-(h(w_1) \cdot \mathbf{H}(A_1) \cdot h(w_2) \cdot \dots \cdot h(w_{k-1}) \cdot \mathbf{H}(A_{k-1}) \cdot h(w_k)) \end{array} \right) \right] \\ \iff \\ G' : S' \Rightarrow^j \mathcal{H}.$$

By the validity of our claim, and because $L(G') \in \mathbf{CFHG}$ (by definition), we now have $\mathbf{L} \in \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$. This proves that $\mathbf{Int}(\mathbf{CF}) \subseteq \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$. Together with the first part, this completes the overall proof of $\mathbf{Int}(\mathbf{CF}) \subsetneq \mathbf{split}(\mathbf{\Gamma}(\mathbf{CFHG}))$.

11.7 Bauderon and Courcelle

The hypergraphs defined by Bauderon and Courcelle ([BC87]), are the same BC-hypergraphs we mentioned earlier in Section 11.2, except for minor notational differences. The (single) sequence of external nodes is called the sequence of *sources*, denoted \mathbf{src} , and the incidence function is called \mathbf{vert} (for *vertices*, i.e., nodes). To translate back and forth between our hypergraphs and these hypergraphs we will use the same functions $\mathbf{\Gamma}$ and $\mathbf{\Gamma}^-$ we used in Section 11.2.

Bauderon and Courcelle define three kinds of operations on BC-hypergraphs:

- Sum, a binary operation, denoted by $\mathcal{H} \oplus \mathcal{H}'$,
- Redefinition of Sources, a unary operation, denoted by $\sigma_\alpha(\mathcal{H})$,
- Source Fusion, a unary operation, denoted by $\theta_\delta(\mathcal{H})$.

Loosely speaking, these operations perform the following actions. Let \mathcal{H} and \mathcal{H}' be arbitrary BC-hypergraphs of rank n and rank n' respectively. The operation $\mathcal{H} \oplus \mathcal{H}'$ does the same thing as our parallel composition, but note the fact that it operates on hypergraphs that have only one sequence of external nodes. The sum $\mathcal{H} \oplus \mathcal{H}'$ has rank $n + n'$. The

operation $\sigma_\alpha(\mathcal{H})$, where α is a mapping from $[m]$ to $[n]$ redefines the sequence of n source nodes of \mathcal{H} in a new sequence of m source nodes, in such a way that, for $1 \leq i \leq m$, the new source node i is the old source node $\alpha(i)$. So, $\sigma_\alpha(\mathcal{H})$ is of rank m . The operation $\theta_\delta(\mathcal{H})$, where δ is an equivalence relation on $[n]$ (intuitively: on the external nodes), identifies those external nodes $\mathbf{src}_{\mathcal{H}}(i)$, $\mathbf{src}_{\mathcal{H}}(j)$, $1 \leq i, j \leq n$, such that $(i, j) \in \delta$. So intuitively, $\theta_\delta(\mathcal{H})$ identifies those external nodes of \mathcal{H} that are in the relation (as implied by) δ . The result is again a BC-hypergraph of rank n .

These operations can be expressed in terms of our sequential and parallel composition, and sequential and parallel composition can be expressed in terms of \oplus , σ_α , and θ_δ . in the following way. Let \mathcal{H} and \mathcal{H}' be arbitrary BC-hypergraphs of rank n and rank n' respectively.

Sum:

The sum $\mathcal{H} \oplus \mathcal{H}'$ can be expressed in terms of $+$ as follows:

$$\mathbf{\Gamma}(\mathcal{H} \oplus \mathcal{H}') = \mathbf{\Gamma}(\mathcal{H}) + \mathbf{\Gamma}(\mathcal{H}').$$

Redefinition of Sources:

For a mapping α from $[m]$ to $[n]$, $\sigma_\alpha(\mathcal{H})$ can be expressed in terms of \cdot as follows:

$$\mathbf{\Gamma}(\sigma_\alpha(\mathcal{H})) = H_\alpha \cdot \mathbf{\Gamma}(\mathcal{H}),$$

where $H_\alpha = (\{v_1, \dots, v_n\}, \emptyset, \emptyset, \emptyset, (v_{\alpha(1)}, \dots, v_{\alpha(m)}), (v_1, \dots, v_n)) : m \rightarrow n$.

Source Fusion:

For an equivalence relation δ on $[n]$, $\theta_\delta(\mathcal{H})$ can be expressed in terms of \cdot as follows:

$$\mathbf{\Gamma}(\theta_\delta(\mathcal{H})) = H_\delta \cdot \mathbf{\Gamma}(\mathcal{H}),$$

where $H_\delta = ([n]/\delta, \emptyset, \emptyset, \emptyset, ([1]_\delta, \dots, [n]_\delta), ([1]_\delta, \dots, [n]_\delta)) : n \rightarrow n$.

Parallel Composition:

The parallel composition of two hypergraphs $H : m \rightarrow n$ and $H' : m' \rightarrow n'$ can be expressed in terms of \oplus and σ_α as follows:

$$\mathbf{\Gamma}^-(H + H') = \sigma_\alpha(\mathbf{\Gamma}^-(H) \oplus \mathbf{\Gamma}^-(H')).$$

Here α denotes a mapping from $[m + m' + n + n']$ to $[m + n + m' + n']$, defined by:

$$\alpha(i) = \begin{cases} i & \text{for } 1 \leq i \leq m, \\ i + n & \text{for } m + 1 \leq i \leq m + m', \\ i - m' & \text{for } m + m' + 1 \leq i \leq m + m' + n, \\ i & \text{for } m + m' + n + 1 \leq i \leq m + m' + n + n'. \end{cases}$$

Sequential Composition:

The sequential composition of two hypergraphs $H : m \rightarrow n$ and $H' : n \rightarrow k$ can be expressed in terms of σ_α and θ_δ as follows:

$$\Gamma^-(H \cdot H') = \sigma_\alpha(\theta_\delta(\Gamma^-(H) \oplus \Gamma^-(H'))).$$

Here δ denotes the smallest equivalence relation on $[m + 2n + k]$ that contains the following pairs:

$$\{(i, j) \mid m + 1 \leq i \leq m + n \text{ and } j = i + n\},$$

and α is the mapping from $[m + k]$ to $[m + 2n + k]$ defined by:

$$\alpha(i) = \begin{cases} i & \text{for } 1 \leq i \leq m, \\ i + 2n & \text{for } m + 1 \leq i \leq m + k. \end{cases}$$

Concluding, we have now proved that the operations \oplus , σ_α , and θ_δ can be expressed in terms of \cdot and $+$, and vice versa. Intuitively, this means that the graph grammars (or rather expression grammars, where the expressions define graphs) that are defined by Bauderon and Courcelle in [BC87], could also have been defined using sequential and parallel composition, in such a way that their power stays exactly the same (it cannot increase, as the emulation of the operations goes both ways). Neither the formal expression of this intuition, nor its proof, are trivial. As both are beyond the scope of this thesis, we will make no attempt at formulating them.

11.8 Habel and Kreowski

In their paper “May we introduce to you: hyperedge replacement” [ENRR87, page 15–26] Habel and Kreowski make a strong case for taking a hypergraph edge rewriting approach to graph grammars. The hypergraphs they define look very much like our hypergraphs, having a sequence of “input” nodes (called *begin*) and a sequence of “output” nodes (called *end*). However, they differ in the fact that the hyperedge-label alphabet is *ranked* instead

of *typed*. Consequently, their hyperedges have two kind a tentacles: *source* tentacles, and *target* tentacles. This makes the hyperedge replacement scheme they propose conceptually very easy: one “lifts” a hyperedge out of a hypergraph, and then inserts an edge replacing hypergraph, thereby connecting the begin and end sequences to the former source and target nodes of the removed edge.

In this respect, their approach looks very much like the approach taken by Bauderon and Courcelle (and, later, by Engelfriet and Heyker), albeit that Habel and Kreowski choose to have two sequences of external nodes per hypergraph, and two sequences of tentacles per hyperedge. As Bauderon and Courcelle correctly point out [BC87, page 113], this does not increase the power of formalism: both kinds of graph grammars can be emulated by the other. Therefore, it may be considered a matter of taste whether one wants one or two sequences of external nodes. In the above mentioned paper Habel and Kreowski argue, by giving quite a few real-life examples, that having two sequences is preferable.

However, in order to avoid some complications that arise with identification, Habel and Kreowski do not allow nodes to appear more than once in the begin and end sequences of a graph. This puts a real restriction on the class of graph languages that can be generated, albeit not a severe one. Informally speaking, the same languages that could be generated without this restriction can still be generated, minus the non identification-free graphs they contained. This was proved by Engelfriet and Heyker in 1992, see [EH92, page 171].

11.9 Further reading

As noted, for an overview of recent developments in the field of graph grammars, [ENRR87] and [EKR91] are a good place to start, especially because they contain comprehensive tutorials. Furthermore, the Ph.D. thesis of Habel [Hab92] contains lots of interesting results. As a matter of fact, at the time of this writing it is the *only* existing book on the (modern) theory of context-free graph grammars, and one of the few books at all, and probably the most up-to-date one, on graph grammars.

Finally, the reader should be well aware of the fact that there is a large area of the field that has been barely touched upon in this master’s thesis: node replacement graph grammars. For an introduction to this kind of approach, read the tutorials by Engelfriet and Rozenberg, [ENRR87, page 55–66] and [EKR91, page 12–23].

There was things which he stretched
but mainly he told the truth.

— *Mark Twain*

12

Summary

This thesis is about the formalism of *interpretation*. This concept is closely linked to that of a *graph grammar*. In Chapter 2 we extended the well-known concepts of grammars and languages to *typed* variants of them. We had to do this to be able to exercise strong control on the types of our objects, in order to ensure they are properly defined. Then we defined a special kind of graph, the *i/o-hypergraph* (Chapter 3). On this type of graph, we defined two main operations: *sequential composition* and *parallel composition* (Chapter 4), which can be used to compose larger graphs from smaller ones. The relation between the two, and some of their properties were investigated. Then in Chapter 5 we looked at the opposite: *decomposing* large graphs into smaller ones. We found a small set of small graphs, called the *full base set*, that serve as building blocks to build all other graphs with sequential and parallel composition.

In Chapter 6, we defined several more operations (of minor importance than composition) on graphs, and gave some properties. These were mainly needed in proofs of theorems that do not contain these operations, but still intuitively are based on them. These operations seem quite natural to us, and their properties have a beauty of their own.

Having defined all the above concepts, we were finally able to introduce the pivot of this thesis: *interpretation* (Chapter 7). Interpretation is a mechanism by which we can obtain a graph language, on the basis of a given string language. As we argued, the formalism of interpretation may be considered a special kind of an edge-rewriting graph grammar formalism, albeit in disguise. We gave a few examples of how interpretation works, and developed a normal form for it. Furthermore, we proved that if a graph language obtained

by interpretation has *bounded degree*, it must also have *bounded cutwidth*.

Now having completely defined interpretation, in Chapter 8 we turned to the power of our formalism. We looked at right-linear, linear, and derivation-bounded languages, and found that all three yield the same class of hypergraph languages under interpretation. However, the class of all context-free languages under interpretation does give a larger class of hypergraph languages. We also proved (Section 8.3) that “repeated” interpretation (in some sense), usually does not increase its power. Finally, we gave two *theorems on the power of interpretation*.

The first one gives, given a class K' that satisfies some closure properties, the largest class K such that $\mathbf{Int}(K)$ is still contained in $\mathbf{Int}(K')$. The second one, intuitively, says that for two classes of string languages K and K' , if we want to know whether $\mathbf{Int}(K)$ and $\mathbf{Int}(K')$ are equal, we just have to check some simple closure properties, and prove the equality for the string graphs in $\mathbf{Int}(K)$ and $\mathbf{Int}(K')$ only.

In Chapter 9, we examined the *closure properties* of $\mathbf{Int}(K)$ in terms of sufficient closure properties on K , in particular for closure under sequential and parallel composition, union, Kleene closure, fold, backfold, split, flip, and edge relabeling.

Then, in Chapter 10 we showed that the class of all regular languages under interpretation is identical to the smallest class of hypergraph languages that is closed under both sequential and parallel composition, union, and Kleene closure, and that contains the full base set. As a similar characterization is used in traditional formal language theory to define the class of all regular languages, it can be argued that $\mathbf{Int}(\text{RLIN})$ deserves to be called “the class of all regular hypergraph languages”.

To conclude, in Chapter 11 we looked at a few other formalisms for graph grammars that have been described in the literature. We made a comparison between some of the classes of hypergraph languages we defined, and some of those that have been described by others. In particular we found that $\mathbf{Int}(\text{RLIN}) \approx \text{LIN-CFHG}$, which may also be used to argue that this class deserves to be called “the class of all regular hypergraph languages”.

So long, and thanks for all the fish.

— *Douglas Adams*

13

Acknowledgments

First and foremost, I would like to thank dr. Joost Engelfriet. For the honesty of rejecting preliminary versions of this thesis time after time, and for the extensive weekly guidance. For teaching me how to transform my vague intuitions into solid mathematical writing (did I make progress, or did you give up?), and for more than a year of pleasant Tuesday afternoon conversations. You have been as good a thesis supervisor as one could wish for, and I am in your debt for the numerous suggestions and ideas that have finally led to this thesis.

Then, I would like to thank drs. Gerard Borsboom for teaching me \TeX and putting up with the silly questions. Also thanks to Hans van Dongen for laboriously proofreading the complete final draft, for laughing himself to tears over a quite erroneous version of Section 5.6, and for all the beer. Thanks to Huibert Kreb for pointing out the difference between inherently true propositions, which need no proof, and true theorems, which are merely true by virtue of their proof, and will lose their validity real soon after that proof gets lost (see [Kre92]).

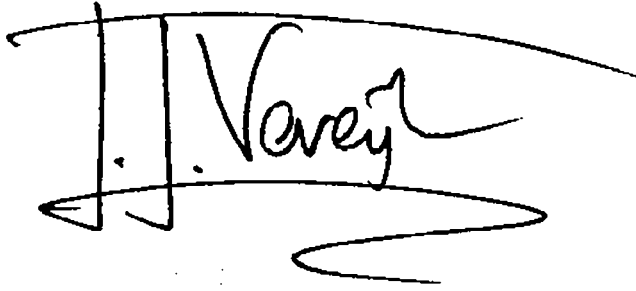
Furthermore thanks to ir. Erik Olofsen, ir. Tycho Lamerigts, Carin Tiggeloven, and Maarten van Dantzich, for their various useful comments, and to dr. Hendrik-Jan Hoogboom for showing me how real books number empty pages (they don't). Thanks to ir. Erik Kruyt too, for generously providing computer facilities, and to the various people who were involved in creating the \TeX , METAFONT, em\TeX , \LaTeX , and $\mathcal{A}\mathcal{M}\mathcal{S}\text{-}\text{\LaTeX}$ suites of typesetting software, all of which were extensively used in creating this document.

This thesis would not have been possible without the help of drs. Frits Vereijken and

Berth Vereijken-Baeten, whom I thank for their continuous moral and financial support throughout my study.

Last but not least, I am greatly indebted to Tieleke Stibbe, for being my main source of inspiration, and for trying to understand the difference between hypergraphs, hypergraph languages, and classes of the latter. It is to you, with love, I dedicate this thesis.

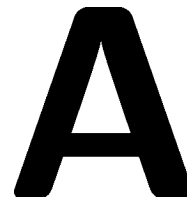
Jan Joris Vereijken,

A handwritten signature in black ink. The signature consists of the initials 'J.J.' followed by the name 'Vereijken'. The 'V' is large and stylized, with a long horizontal stroke extending to the right. The 'eijken' part is written in a cursive style. There are some additional scribbles and a long horizontal line below the signature.

Leiden, May 19, 1993.

The nice thing about standards is that there are so many to choose from.

— *Andrew S. Tanenbaum*



Naming conventions

The names of variables, functions, etc. are called *identifiers*. In assigning those names we have *strictly* adhered to the following conventions.

- **Uppercase roman letters:**

- For propositions we use a P , or a Q .

- **Uppercase italic letters:**

- For sets in general we use letters late in the alphabet: V , W .
- For a set that is a string language we use an L . For one that is a class of languages a K . For a set of terminal symbols: T . For a set of nonterminal symbols: N . For a set of productions: P or Q . For a set of vertices: V . For a set of edges: E .
- For the starting symbol in a nonterminal alphabet an S .
- For grammars: G . For a property of grammars: X .
- For hypergraphs we use letters near “H”: H , G , F .
- For interpreters we use an I .
- For nonterminal symbols we use letters early in the alphabet: A , B , C , D .

- **Lowercase italic letters:**

- For integers we use letters near the middle of the alphabet: $n, m, k, l, i, j, r, p, q$.
- For terminal symbols we use letters early in the alphabet: a, b, c, d . The a is also used for a symbol in general.
- For functions in general we use letters near “f”: f, g, h .
- For productions we use letters near “p”: p, q .
- For strings over terminal symbols we use letters near the end of the alphabet: w, v, u, z .
- For vertices: v, w, u, z . For edges: e .
- For (regular) expressions: r .

- **Greek upper case letters:**

- For alphabets: Σ, Δ .

- **Greek lower case letters:**

- For strings over terminal symbols, nonterminal symbols, or both, we use letters early in the alphabet: $\alpha, \beta, \gamma, \delta$.
- For morphisms on string languages: σ .
- For numbers that denote a maximum or minimum in some sense: μ .

- **Other cases:**

- For ad hoc functions that yield hypergraphs we use bold uppercase italic letters near “H”: $\mathbf{H}, \mathbf{G}, \mathbf{F}$.
- For hypergraph languages, and occasionally for sets of hypergraphs, we use a bold uppercase italic “L”: \mathbf{L} . For classes for hypergraph languages: \mathbf{K} .
- For EH-hypergraphs and BC-hypergraphs (see Chapter 11) we will use “calligraphic” uppercase letters near “H”: $\mathcal{H}, \mathcal{G}, \mathcal{F}$.
- General functions on hypergraphs are: $\#in, \#out, \mathbf{flip}, \mathbf{fold}, \mathbf{backfold}, \mathbf{split}$.
- In specific cases we are allowed to abandon the general naming rules, and introduce identifiers of our own choosing. These must however keep their fixed meaning throughout the whole thesis. The following is a complete list of them: $\mathbb{N}, \text{TRUE}, \text{FALSE}, L, L_\tau, G, G_\tau, \mathcal{P}, U_n, \lambda, \text{RLIN}, \text{LIN}, \text{DB}, \text{CF}, \mathbf{rank}, \mathbf{nod}$,

lab, in, out, deg, HGR, BC-HGR, Sing, Int, Γ , Γ^{\leftarrow} , Ω_a , $\mathbf{1}_{m,n}$, $\Pi_{\pi,\pi',k}$, gr, STR, match.

- When we refer to results from other sources, we may choose to adapt their notation. See for example Section 11.4, where we use notation from [EH91].

In general our symbols may be subscripted (e.g., A_1 , a_p , α_i) at wish if we need more of them than are available otherwise. The same holds for priming them (e.g., a' , H''), or barring them (e.g., \bar{a}). We also may build new symbols by grouping old symbols together in a list (e.g., $\langle a, i \rangle$, $[A_1A_2A_3]$). Although some symbols have more than one use (V , u , v , w , z , p , q and G), this is not a problem in practice.

Seek simplicity, and distrust it.

— *Alfred North Whitehead*

B

Proofs

As a rule, proofs are given right after the theorem they prove. However, some theorems are small, beautiful, and “obviously” true, but have nonetheless long, tedious, and bothersome proofs. It is this category of proofs that are not given in the main text, but in this appendix.

B.1 Proofs concerning Section 6.3

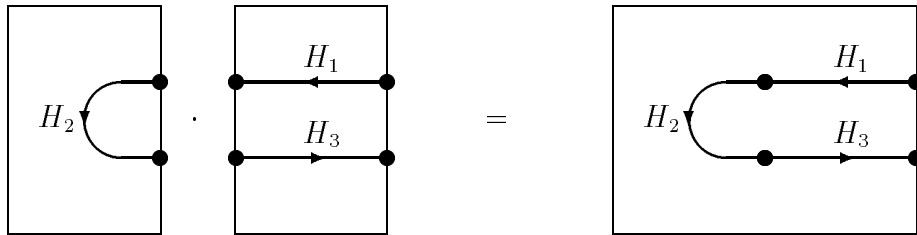
The six properties from Section 6.3 can be proven by introducing unity hypergraphs U to the left and the right of all H 's involved, and then “massaging” the H 's out by applying (4.10) and the basic properties from Section 6.2. As a result, we get the same properties back again, but now applied to the unity hypergraphs U only. The thus simplified properties can then be proven directly by expanding the definitions involved.

We will not write out these proof in full however, as they are very long, and worse, they do not attribute at all to a better understanding of the properties. Instead, we will give a graphical representation of the *structure* of the properties. In the following pictures, a thick line indicates a hypergraph, and a dot indicates external nodes. An arrow in the line points from **in** to **out** to avoid confusion. Folding is indicated folding the line, and flipping by flipping it (hence the names!). Finally, a thin-lined bounding box is drawn around the hypergraph that is formed by evaluating the parallel compositions in the properties. First we give the equation of the property, and then a picture of the structure.

Property (6.32):

$$\mathbf{fold}(H_2) \cdot (\mathbf{flip}(H_1) + H_3) = \mathbf{fold}(H_1 H_2 H_3)$$

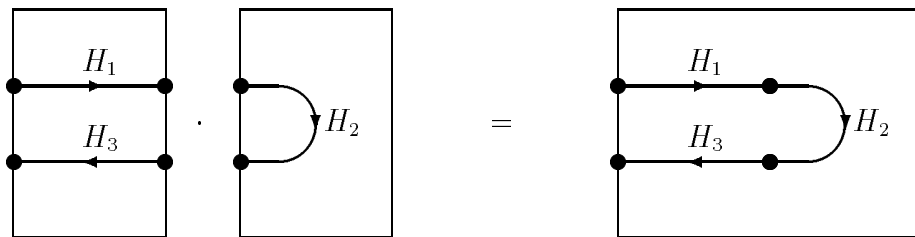
Structure:



Property (6.33):

$$(H_1 + \mathbf{flip}(H_3)) \cdot \mathbf{backfold}(H_2) = \mathbf{backfold}(H_1 H_2 H_3)$$

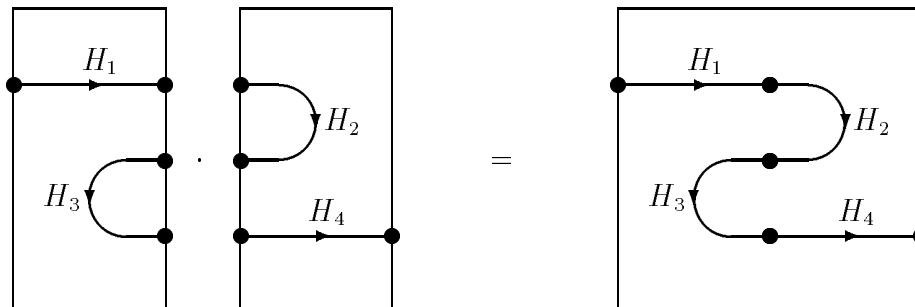
Structure:



Property (6.34):

$$(H_1 + \mathbf{fold}(H_3)) \cdot (\mathbf{backfold}(H_2) + H_4) = H_1 H_2 H_3 H_4$$

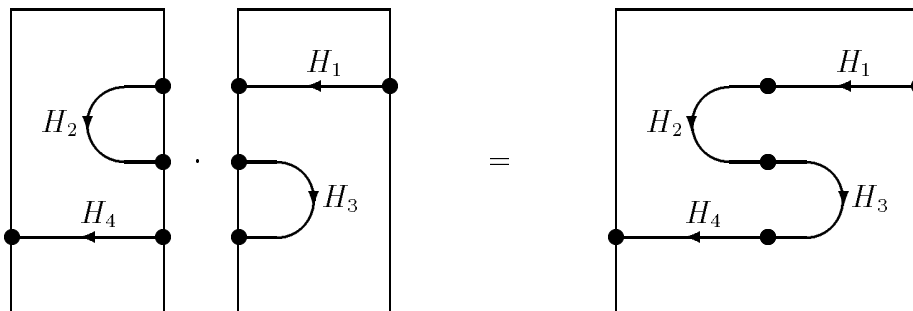
Structure:



Property (6.35):

$$(\text{fold}(H_2) + \text{flip}(H_4)) \cdot (\text{flip}(H_1) + \text{backfold}(H_3)) = \text{flip}(H_1 H_2 H_3 H_4)$$

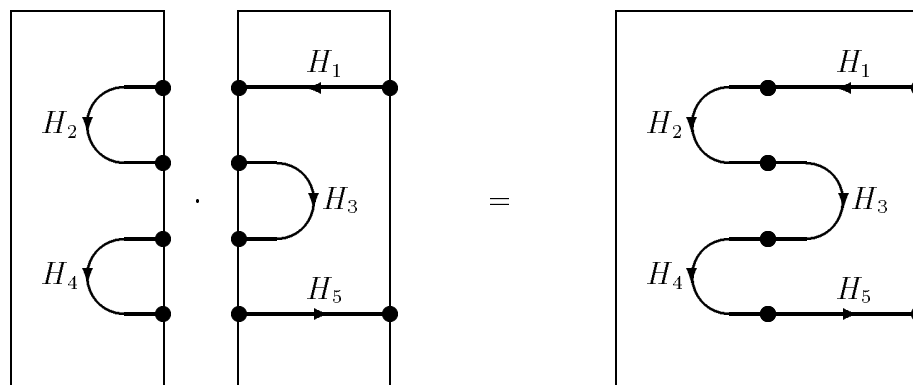
Structure:



Property (6.36):

$$(\text{fold}(H_2) + \text{fold}(H_4)) \cdot (\text{flip}(H_1) + \text{backfold}(H_3) + H_5) = \text{fold}(H_1 H_2 H_3 H_4 H_5)$$

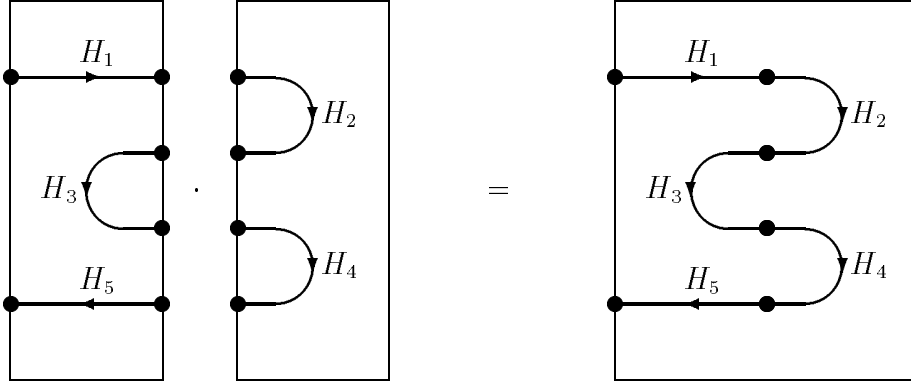
Structure:



Property (6.37):

$$(H_1 + \text{fold}(H_3) + \text{flip}(H_5)) \cdot (\text{backfold}(H_2) + \text{backfold}(H_4)) = \text{backfold}(H_1 H_2 H_3 H_4 H_5)$$

Structure:



B.2 Proofs concerning Section 8.2

The following properties are derived from the properties given in Sections 6.2 and 6.3.

First property:

$$\begin{aligned}
 & (H_1 + \text{fold}(H_2) + \cdots + \text{fold}(H_{n-1}) + \text{flip}(H_n)) \\
 & \cdot \\
 & \left(\begin{array}{l} U_{\#out(H_1)} + U_{\#in(H_2)+\#out(H_2)} + \cdots + U_{\#in(H_{i-1})+\#out(H_{i-1})} + U_{\#in(H_i)} + \\ G_1 + \text{fold}(G_2) + \cdots + \text{fold}(G_{m-1}) + \text{flip}(G_m) + \\ U_{\#out(H_{i+1})} + U_{\#in(H_{i+2})+\#out(H_{i+2})} + \cdots + U_{\#in(H_{n-1})+\#out(H_{n-1})} + U_{\#in(H_n)} \end{array} \right) \\
 & = \\
 & \left(\begin{array}{l} \mathbf{H}_1(H_1) + \mathbf{H}_2(H_2) + \cdots + \mathbf{H}_{i-1}(H_{i-1}) + \\ \mathbf{H}_i(H_i G_1) + \mathbf{H}_{i+1}(G_2) + \cdots + \mathbf{H}_{i+m-2}(G_{m-1}) + \mathbf{H}_{i+m-1}(G_m H_{i+1}) + \\ \mathbf{H}_{i+m}(H_{i+2}) + \cdots + \mathbf{H}_{m+n-3}(H_{n-1}) + \mathbf{H}_{m+n-2}(H_n) \end{array} \right).
 \end{aligned} \tag{B.1}$$

Where:

$$\mathbf{H}_j(H) = \begin{cases} H & \text{if } j = 1, \\ \text{fold}(H) & \text{if } 1 < j < m + n - 2, \\ \text{flip}(H) & \text{if } j = m + n - 2. \end{cases}$$

under the condition that $H_i G_1$ and $G_m H_{i+1}$ are both defined, $1 \leq i < n$, $n \geq 2$ and $m \geq 2$. Actually, this property could better be called a “meta property”. By this we mean that (B.1) is a property that comprises of four similar but distinct properties. We get this four properties by distinguishing the following four cases:

$$\begin{aligned} &1 < i < n - 1 \text{ and } n > 2, \\ &i = 1 \text{ and } n > 2, \\ &i = n - 1 \text{ and } n > 2, \\ &i = 1 \text{ and } n = 2. \end{aligned}$$

If we do this, all four properties can be written without the function \mathbf{H} , using **flip** and **fold** instead at the appropriate places.

To prove (B.1) (and to resolve the ambiguity that might arise when one tries to interpret it!), we write out the four cases in full, and prove them separately.

Firstly, the case where $1 < i < n - 1$ and $n > 2$:

$$\begin{aligned} &(H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ &\quad \cdot \\ &\left(\begin{array}{l} U_{\#\mathbf{out}(H_1)} + U_{\#\mathbf{in}(H_2)+\#\mathbf{out}(H_2)} + \cdots + U_{\#\mathbf{in}(H_{i-1})+\#\mathbf{out}(H_{i-1})} + U_{\#\mathbf{in}(H_i)} + \\ G_1 + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{flip}(G_m) + \\ U_{\#\mathbf{out}(H_{i+1})} + U_{\#\mathbf{in}(H_{i+2})+\#\mathbf{out}(H_{i+2})} + \cdots + U_{\#\mathbf{in}(H_{n-1})+\#\mathbf{out}(H_{n-1})} + U_{\#\mathbf{in}(H_n)} \end{array} \right) \quad (\text{B.2}) \\ &= \\ &\left(\begin{array}{l} H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{i-1}) + \mathbf{fold}(H_i G_1) + \mathbf{fold}(G_2) + \cdots + \\ \mathbf{fold}(G_{m-1}) + \mathbf{fold}(G_m H_{i+1}) + \mathbf{fold}(H_{i+2}) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n) \end{array} \right), \end{aligned}$$

under the condition that $H_i G_1$ and $G_m H_{i+1}$ are both defined. This property follows directly from (4.11), as the left-hand side can be written as:

$$\begin{aligned} &(H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_i) + U_0 + \mathbf{fold}(H_{i+1}) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ &\quad \cdot \\ &\left(\begin{array}{l} U_{\#\mathbf{out}(H_1)} + U_{\#\mathbf{in}(H_2)+\#\mathbf{out}(H_2)} + \cdots + U_{\#\mathbf{in}(H_{i-1})+\#\mathbf{out}(H_{i-1})} + \\ \underbrace{[U_{\#\mathbf{in}(H_i)} + G_1]}_{\text{connects to } \mathbf{fold}(H_i)} + \underbrace{[\mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1})]}_{\text{connects to } U_0} + \underbrace{[\mathbf{flip}(G_m) + U_{\#\mathbf{out}(H_{i+1})}]}_{\text{connects to } \mathbf{fold}(H_{i+1})} + \\ U_{\#\mathbf{in}(H_{i+2})+\#\mathbf{out}(H_{i+2})} + \cdots + U_{\#\mathbf{in}(H_{n-1})+\#\mathbf{out}(H_{n-1})} + U_{\#\mathbf{in}(H_n)} \end{array} \right). \end{aligned}$$

Applying (4.11) now yields the following expression:

$$\left(\begin{array}{l} H_1 U_{\# \text{out}(H_1)} + \mathbf{fold}(H_2) U_{\# \text{in}(H_2) + \# \text{out}(H_2)} + \cdots + \mathbf{fold}(H_{i-1}) U_{\# \text{in}(H_{i-1}) + \# \text{out}(H_{i-1})} + \\ \mathbf{fold}(H_i) [U_{\# \text{in}(H_i)} + G_1] + U_0 [\mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1})] + \mathbf{fold}(H_{i+1}) [\mathbf{flip}(G_m) + U_{\# \text{out}(H_{i+1})}] + \\ \mathbf{fold}(H_{i+1}) U_{\# \text{in}(H_{i+1}) + \# \text{out}(H_{i+1})} + \cdots + \mathbf{fold}(H_{n-1}) U_{\# \text{in}(H_{n-1}) + \# \text{out}(H_{n-1})} + \mathbf{flip}(H_n) U_{\# \text{in}(H_n)} \end{array} \right)$$

which easily reduces to

$$\left(\begin{array}{l} H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{i-1}) + \mathbf{fold}(H_i G_1) + \mathbf{fold}(G_2) + \cdots + \\ \mathbf{fold}(G_{m-1}) + \mathbf{fold}(G_m H_{i+1}) + \mathbf{fold}(H_{i+2}) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n) \end{array} \right)$$

by applying (4.2) and (6.32). This completes the proof of (B.2).

Secondly, the property (B.1) for the case $i = 1$ and $n > 2$ looks like this:

$$\begin{aligned} & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ & \quad \cdot \\ & \left(\begin{array}{l} G_1 + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{flip}(G_m) + \\ U_{\# \text{out}(H_2)} + U_{\# \text{in}(H_3) + \# \text{out}(H_3)} + \cdots + U_{\# \text{in}(H_{n-1}) + \# \text{out}(H_{n-1})} + U_{\# \text{in}(H_n)} \end{array} \right) \quad (\text{B.3}) \\ & \quad = \\ & \left(\begin{array}{l} H_1 G_1 + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{fold}(G_m H_2) + \\ \mathbf{fold}(H_3) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n) \end{array} \right), \end{aligned}$$

under the condition that $H_1 G_1$ and $G_m H_2$ are both defined. It is proved in a similar way as (B.2), using, amongst others, (6.32).

Thirdly, the case $i = n - 1$ and $n > 2$ is as follows:

$$\begin{aligned} & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ & \quad \cdot \\ & \left(\begin{array}{l} U_{\# \text{out}(H_1)} + U_{\# \text{in}(H_2) + \# \text{out}(H_2)} + \cdots + U_{\# \text{in}(H_{n-2}) + \# \text{out}(H_{n-2})} + U_{\# \text{in}(H_{n-1})} + \\ G_1 + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{flip}(G_m) \end{array} \right) \quad (\text{B.4}) \\ & \quad = \\ & \left(\begin{array}{l} H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-2}) + \\ \mathbf{fold}(H_{n-1} G_1) + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{flip}(G_m H_n) \end{array} \right), \end{aligned}$$

under the condition that $H_{n-1} G_1$ and $G_m H_n$ are both defined. This case too is proved in a way similar as (B.2), using, amongst others, (6.32) and (6.23).

Finally, the fourth case, $i = 1$ and $n = 2$:

$$\begin{aligned}
 & (H_1 + \mathbf{flip}(H_2)) \\
 & \quad \cdot \\
 & \left(G_1 + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{flip}(G_m) \right) \\
 & \quad = \\
 & \left(H_1 G_1 + \mathbf{fold}(G_2) + \cdots + \mathbf{fold}(G_{m-1}) + \mathbf{flip}(G_m H_2) \right),
 \end{aligned} \tag{B.5}$$

under the condition that $H_1 G_1$ and $G_m H_2$ are both defined. This case is proved using (4.11).

Second property:

$$\begin{aligned}
 & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\
 & \quad \cdot \\
 & \left(\begin{array}{c} U_{\#out(H_1)} + U_{\#in(H_2)+\#out(H_2)} + \cdots + U_{\#in(H_{i-1})+\#out(H_{i-1})} + U_{\#in(H_i)} + \\ \mathbf{backfold}(G) + \\ U_{\#out(H_{i+1})} + U_{\#in(H_{i+2})+\#out(H_{i+2})} + \cdots + U_{\#in(H_{n-1})+\#out(H_{n-1})} + U_{\#in(H_n)} \end{array} \right) \\
 & \quad = \\
 & \left(\begin{array}{c} \mathbf{H}_1(H_1) + \mathbf{H}_2(H_2) + \cdots + \mathbf{H}_{i-1}(H_{i-1}) + \\ \mathbf{H}_i(H_i G H_{i+1}) + \\ \mathbf{H}_{i+1}(H_{i+2}) + \cdots + \mathbf{H}_{n-2}(H_{n-1}) + \mathbf{H}_{n-1}(H_n) \end{array} \right).
 \end{aligned} \tag{B.6}$$

Where:

$$\mathbf{H}_j(H) = \begin{cases} H & \text{if } j = 1 \text{ and } n > 2, \\ \mathbf{fold}(H) & \text{if } 1 < j < n - 1 \text{ and } n > 2, \\ \mathbf{flip}(H) & \text{if } j = n - 1 \text{ and } n > 2, \\ \mathbf{backfold}(H) & \text{if } j = 1 \text{ and } n = 2. \end{cases}$$

under the condition that $H_i G H_{i+1}$ is defined, $1 \leq i < n$, and $n \geq 2$. As above, this meta property is distinguished in 4 cases:

$$\begin{aligned}
 & 1 < i < n - 1 \text{ and } n > 2, \\
 & i = 1 \text{ and } n > 2, \\
 & i = n - 1 \text{ and } n > 2,
 \end{aligned}$$

$$i = 1 \text{ and } n = 2.$$

Note that, in a way, (B.6) can intuitively be considered as the equivalent of (B.1) for the case $m = 1$. We write out all cases to prove (B.6).

Firstly, the case $1 < i < n - 1$ and $n > 2$:

$$\begin{aligned} & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ & \quad \cdot \\ & \left(\begin{array}{c} U_{\#out(H_1)} + U_{\#in(H_2)+\#out(H_2)} + \cdots + U_{\#in(H_{i-1})+\#out(H_{i-1})} + U_{\#in(H_i)} + \\ \mathbf{backfold}(G) + \\ U_{\#out(H_{i+1})} + U_{\#in(H_{i+2})+\#out(H_{i+2})} + \cdots + U_{\#in(H_{n-1})+\#out(H_{n-1})} + U_{\#in(H_n)} \end{array} \right) \\ & \quad = \\ & \left(\begin{array}{c} H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{i-1}) + \mathbf{fold}(H_iGH_{i+1}) + \\ \mathbf{fold}(H_{i+2}) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n) \end{array} \right), \end{aligned} \tag{B.7}$$

under the condition that H_iGH_{i+1} is defined. The proof of this property can be derived from (4.11) and (6.36) by rewriting the left hand side as:

$$\begin{aligned} & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ & \quad \cdot \\ & \left(\begin{array}{c} U_{\#out(H_1)} + U_{\#in(H_2)+\#out(H_2)} + \cdots + U_{\#in(H_{i-1})+\#out(H_{i-1})} + \\ \underbrace{[U_{\#in(H_i)} + \mathbf{backfold}(G) + U_{\#out(H_{i+1})}]}_{\text{connects to } \mathbf{fold}(H_i) + \mathbf{fold}(H_{i+1})} + \\ U_{\#in(H_{i+2})+\#out(H_{i+2})} + \cdots + U_{\#in(H_{n-1})+\#out(H_{n-1})} + U_{\#in(H_n)} \end{array} \right). \end{aligned}$$

Applying (4.11) now yields:

$$\left(\begin{array}{c} H_1U_{\#out(H_1)} + \mathbf{fold}(H_2)U_{\#in(H_2)+\#out(H_2)} + \cdots + \mathbf{fold}(H_{i-1})U_{\#in(H_{i-1})+\#out(H_{i-1})} + \\ [(\mathbf{fold}(H_i) + \mathbf{fold}(H_{i+1})) \cdot (U_{\#in(H_i)} + \mathbf{backfold}(G) + U_{\#out(H_{i+1})})] + \\ \mathbf{fold}(H_{i+2})U_{\#in(H_{i+2})+\#out(H_{i+2})} + \cdots + \mathbf{fold}(H_{n-1})U_{\#in(H_{n-1})+\#out(H_{n-1})} + \mathbf{flip}(H_n)U_{\#in(H_n)} \end{array} \right).$$

Rewriting the central part using (6.25) and (6.36) reduces this to:

$$\left(\begin{array}{c} H_1U_{\#out(H_1)} + \mathbf{fold}(H_2)U_{\#in(H_2)+\#out(H_2)} + \cdots + \mathbf{fold}(H_{i-1})U_{\#in(H_{i-1})+\#out(H_{i-1})} + \\ \mathbf{fold}(U_{\#in(H_i)}H_iGH_{i+1}U_{\#out(H_{i+1})}) + \\ \mathbf{fold}(H_{i+2})U_{\#in(H_{i+2})+\#out(H_{i+2})} + \cdots + \mathbf{fold}(H_{n-1})U_{\#in(H_{n-1})+\#out(H_{n-1})} + \mathbf{flip}(H_n)U_{\#in(H_n)} \end{array} \right),$$

which in turn easily reduces to

$$\left(\begin{array}{c} H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{i-1}) + \mathbf{fold}(H_iGH_{i+1}) + \\ \mathbf{fold}(H_{i+2}) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n) \end{array} \right)$$

by applying (4.2) and (4.1). This completes the proof of (B.7).

Secondly, the case $i = 1$ and $n > 2$ is as follows:

$$\begin{aligned} & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ & \quad \cdot \\ & \left(\begin{array}{c} \mathbf{backfold}(G) + U_{\#\mathbf{out}(H_2)} + U_{\#\mathbf{in}(H_3)+\#\mathbf{out}(H_3)} + \cdots + \\ U_{\#\mathbf{in}(H_{n-1})+\#\mathbf{out}(H_{n-1})} + U_{\#\mathbf{in}(H_n)} \end{array} \right) \quad (\text{B.8}) \\ & \quad = \\ & \left(H_1GH_2 + \mathbf{fold}(H_3) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n) \right), \end{aligned}$$

under the condition that H_1GH_2 is defined. It is proved in a similar way as (B.7) itself, using (6.34) instead of (6.36).

Thirdly, the case $i = n - 1$ and $n > 2$ is as follows:

$$\begin{aligned} & (H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-1}) + \mathbf{flip}(H_n)) \\ & \quad \cdot \\ & \left(\begin{array}{c} U_{\#\mathbf{out}(H_1)} + U_{\#\mathbf{in}(H_2)+\#\mathbf{out}(H_2)} + \cdots + \\ U_{\#\mathbf{in}(H_{n-2})+\#\mathbf{out}(H_{n-2})} + U_{\#\mathbf{in}(H_{n-1})} + \mathbf{backfold}(G) \end{array} \right) \quad (\text{B.9}) \\ & \quad = \\ & \left(H_1 + \mathbf{fold}(H_2) + \cdots + \mathbf{fold}(H_{n-2}) + \mathbf{flip}(H_{n-1}GH_n) \right), \end{aligned}$$

under the condition that $H_{n-1}GH_n$ is defined. It is proved in a similar way as (B.7) itself, using (6.35) instead of (6.36).

Finally, the fourth case, $i = 1$ and $n = 2$:

$$\begin{aligned} & (H_1 + \mathbf{flip}(H_2)) \\ & \quad \cdot \\ & \left(\mathbf{backfold}(G) \right) \quad (\text{B.10}) \\ & \quad = \\ & \left(\mathbf{backfold}(H_1GH_2) \right), \end{aligned}$$

under the condition that H_1GH_2 is defined. This case is identical to (6.33).

Bibliography

- [BC87] Michel Bauderon and Bruno Courcelle, *Graph expressions and graph rewritings*, *Mathematical Systems Theory* **20** (1987), no. 2 & 3, 83–127.
- [BS87] Mario Benedicty and Frank R. Sledge, *Discrete mathematical structures*, Harcourt Brace Jovanovich, Orlando, FL, 1987, ISBN 0-15-517683-8.
- [CL89] John Carroll and Darrell Long, *Theory of finite automata*, Prentice-Hall, Englewood Cliffs, NJ, 1989, ISBN 0-13-913708-4.
- [EH91] Joost Engelfriet and Linda Heyker, *The string generating power of context-free hypergraph grammars*, *Journal of Computer and System Sciences* **43** (1991), no. 2, 328–360.
- [EH92] Joost Engelfriet and Linda Heyker, *Context-free hypergraph grammars have the same term-generating power as attribute grammars*, *Acta Informatica* **29** (1992), 161–210.
- [EKR91] H. Ehrig, H.-J. Kreowski, and G. Rozenberg (eds.), *Graph grammars and their application to computer science*, *Lecture Notes in Computer Science*, vol. 532, Springer-Verlag, Berlin, 1991, 4th International Workshop, Bremen, Germany, March 1990, ISBN 3-540-54478-X.
- [EL89] Joost Engelfriet and George Leih, *Linear graph grammars: Power and complexity*, *Information and Computation* **81** (1989), no. 1, 88–121.
- [ENRR87] H. Ehrig, M. Nagl, G. Rozenberg, and A. Rosenfeld (eds.), *Graph grammars and their application to computer science*, *Lecture Notes in Computer Science*, vol. 291, Springer-Verlag, Berlin, 1987, 3rd International Workshop, Warrenton, Virginia, USA, December 1986, ISBN 3-540-18771-5.

-
- [ERS80] J. Engelfriet, G. Rozenberg, and G. Slutzki, *Tree transducers, L systems, and two-way machines*, Journal of Computer and System Sciences **20** (1980), 150–202.
- [Gre78] S. Greibach, *One-way visit automata*, Theoretical Computer Science **6** (1978), 175–221.
- [Hab92] Annegret Habel, *Hyperedge replacement: grammars and languages*, Lecture Notes in Computer Science, vol. 643, Springer-Verlag, Berlin, 1992, ISBN 3-540-56005-X (originally appeared as Ph.D. thesis, Bremen).
- [Her75] I. N. Herstein, *Topics in algebra*, second ed., John Wiley & Sons, New York, NY, 1975, ISBN 0-471-02371-X.
- [HU79] John E. Hopcroft and Jeffrey D. Ullman, *Introduction to automata theory, languages and computation*, Addison-Wesley, Reading, MA, 1979, ISBN 0-201-02988-X.
- [Joh84] Richard Johnsonbaugh, *Discrete mathematics*, Macmillan Publishing Company, New York, NY, 1984, ISBN 0-02-360900-1.
- [Kam50] E. Kamke, *Theory of sets*, Dover Publications, New York, NY, 1950, ISBN 0-486-60141-2 (this is a translation from the German second edition called *Mengenlehre*).
- [Knu73] Donald Ervin Knuth, *Sorting and searching*, The Art of Computer Programming, vol. 3, Addison-Wesley, Reading, MA, 1973, ISBN 0-201-03803-X.
- [Kre92] Huibert L. Kreb, *Proving the klets lemma*, Stand-up lecture at Leiden University, unpublished, 1992.
- [Len82] T. Lengauer, *Upper and lower bounds on the complexity of the min-cut linear arrangement problem on trees*, SIAM Journal of Algebraic Discrete Methods **3** (1982), 99–113.

Index

This index contains references to all definitions and notations used in this thesis. It is not uncommon for something to be defined more than once. For example, “type” is defined on page 9 (for a symbol) and on page 18 (for a hypergraph). Furthermore, important concepts and names of persons have been indexed at the relevant places. Definitions are indicated by a bold-face number, e.g. **42**, and references to naming conventions by a slanted number e.g. *42*.

$(\emptyset, m \rightarrow n)$, 22	L , <i>113</i>
(a, n) , 7	N , <i>113</i>
$(m \rightarrow n)$, 9, 10	P , <i>113</i>
(v_1, \dots, v_n) , 6	Q , <i>113</i>
$-$, 5	S , <i>113</i>
$=_r$, 11	T , <i>113</i>
A , <i>113</i>	U_n , 26 , <i>114</i>
B , <i>113</i>	V , <i>113, 115</i>
C , <i>113</i>	V/\equiv , 6
D , <i>113</i>	V^* , 6
E , <i>113</i>	V^n , 6
E_H , 17	V_H , 17
F , <i>113</i>	W , <i>113</i>
G , <i>113, 115</i>	X , <i>113</i>
$G :$, 7	$[(v_1, \dots, v_n)]_{\equiv}$, 6
H , <i>113</i>	$[\dots]$, <i>115</i>
I , <i>113</i>	$[m, n]$, 5
K , <i>113</i>	$[n]$, 5

- $[v]_{\equiv}$, 6
 $\#$, 6
BC-HGR, 115
BC-HGR(Δ), 96
 Δ , 114
G, 7, 114
 G_{τ} , 12, 114
HGR, 18, 115
HGR $_{m,n}$ (Δ), 18
Int, 115
Int(K), 48
Int(L), 48
Int $_I$ (L), 48
Int $_{\rightarrow n}$ (K), 48
Int $_{m \rightarrow n}$ (K), 48
Int $_{m \rightarrow}$ (K), 48
L, 7, 114
 L_{τ} , 114
 L_{τ} , 11
N, 5, 114
 Ω_a , 36, 115
P, 113
 $\Pi_{\pi, \pi', k}$, 38, 115
Q, 113
STR, 115
STR(K), 21
 Σ , 114
 Σ^* , 9
 Σ^+ , 9
Sing, 115
Sing(L), 22
 α , 114
 \approx , 6
 Γ , 97, 104, 115
 Γ^- , 97, 104, 115
 $\mathbf{1}_{m,n}$, 37, 115
X, 40
backfold, 43, 114
 β , 114
 \mathbf{F} , 114
 \mathbf{G} , 114
 \mathbf{H} , 114
 \mathbf{K} , 114
 \mathbf{L} , 114
 \bullet , 88
 \mathcal{F} , 114
 \mathcal{G} , 114
 \mathcal{H} , 114
 \cap , 5
 $|V|$, 6
 $|\alpha|$, 6
 \cdot , 7, 25
 $\cdot_{m,n,k}$, 26
 \circ , 6
 \cup , 5
 $\cup_{m,n}$, 23
 $\xrightarrow{+}$, 33
 $\xrightarrow{-}$, 33
deg, 18, 115
 δ , 114
 \Rightarrow^k , 7
 \Rightarrow^* , 7
 \emptyset , 5, 19
 \equiv , 6
 \equiv_{io} , 22
 \exists , 6
FALSE, 6, 114
flip, 43, 114
fold, 43, 114
 \forall , 6
 γ , 114
gr, 115
gr(w), 21
 $\#in$, 18, 114

- $\#in_\Sigma$, 9
 $\#out$, 18, 114
 $\#out_\Sigma$, 9
 \iff , 6
 \implies , 6
 \in , 5
 lab , 115
 lab_H , 17
 λ , 6, 114
 \wedge , 6
 $\langle \dots \rangle$, 115
 \vee , 6
 \mapsto , 6
 $match$, 115
 max , 5
 min , 5
 μ , 114
 \ni , 5
 nod , 114
 nod_H , 17
 in , 115
 in_H , 17
 out , 115
 out_H , 17
 \oplus , 104
 \mathcal{P} , 114
 $\mathcal{P}(V)$, 5
 \prod_i , 27
 $rank$, 114
 $rank(\mathcal{H})$, 96
 $rank_\Sigma$, 7
 L^* , 27
 L^k , 27
 \upharpoonright , 6
 \setminus , 5
 σ , 114
 σ_α , 104
 $split$, 114
 $split(K)$, 45
 $split_{p,q}$, 43
 src , 104
 \subsetneq , 5
 \subseteq , 5
 \sum_i , 29
 \supsetneq , 5
 \supseteq , 5
 θ_δ , 104
 \times , 6
 $TRUE$, 6, 114
 $vert$, 104
 a , 114
 b , 114
 c , 114
 d , 114
 e , 114
 f , 114
 f^{-1} , 6
 f_\equiv , 6
 g , 114
 h , 114
 i , 114
 j , 114
 k , 114
 l , 114
 m , 114
 n , 114
 p , 114, 115
 q , 114, 115
 r , 114
 u , 114, 115
 v , 114, 115
 w , 114, 115
 w^R , 7
 z , 114, 115

- abstract graph, 18
 algebra, 41
 alphabet, **6**
 ordinary, **6**
 ranked, **6**
 typed, **9**
 $\mathcal{A}\mathcal{M}\mathcal{S}$ - $\mathcal{L}\mathcal{A}\mathcal{T}\mathcal{E}\mathcal{X}$, ii, 111
 attack
 problems worthy of, iii, 4

 backfold
 in terms of split, 45
 backfolding, **44**
 closure under, 86
 Baeten, Berth, 112
 base set
 full, **42**
 sequential, **41**
 Bauderon, Michel, 96, 104, 106, 107
 BC-hypergraph, 96
 graph grammar using, 97
 binary tree, 56, 102
 Borsboom, Gerard, 111
 bounded cutwidth, **23**, 102
 theorem on, 54
 bounded degree, **23**, 102
 theorem on, 54

 cardinality, 5
 CF, **7**, 114
 CFHG, 97, 98, 102
 Chomsky hierarchy, 2
 closure
 summary table, 89
 under $+\{U_n\}$, $\{U_n\}+$, 85
 under backfolding, 86
 under concatenation, 83
 under edge relabeling, 88
 under flipping, 86
 under folding, 86
 under Kleene closure, 84
 under parallel composition, 86
 under sequential composition, 83
 under splitting, 88
 under union, 84
 combinatorics, 41
 composition
 parallel, 3, **28**
 relation between sequential and parallel, 29
 sequential, 1, **25**
 concatenation, **10**, **27**
 closure under, 83
 concrete graph, 18
 context-free grammar, **7**
 context-free hypergraph grammar, 97
 context-free language, **7**
 correctly internally typed, **9**
 Courcelle, Bruno, 96, 104, 106, 107
 cut, **20**
 cutwidth, **20**
 bounded, **23**, 102
 lemma on, 20

 van Dantzig, Maarten, 111
 DB, **7**, 114
 decomposition, 3, 33
 full, 33
 sequential, 33
 definitions
 regarding alphabets, 6
 regarding classes of languages, 7
 regarding functions, 6
 regarding grammars, 7
 regarding logic, 6
 regarding numbers, 5

- regarding relations, 6
- regarding sequences, 6
- regarding sets, 5
- regarding strings, 7
- degree, **18**
 - bounded, **23**, 102
 - vs sequential composition, 27
- degree of string graph, 21
- Degree versus Cutwidth Theorem, **54**
- derivation bound, **8**
- derivation-bounded grammar, **8**
- derivation-bounded language, **8**
- disguise
 - graph grammar in, 2
- van Dongen, Hans, 111
- edge, *see* hyperedge
 - “laying in line”, 20
- Edge Normal Form, **51**
 - theorem, **51**
- edge relabeling
 - closure under, 88
- edge removal, **35**
- empty hypergraph, **26**
- empty hypergraph language, 22
- emTeX, 111
- eNCE graph grammar, 56
- ENF, *see* Edge Normal Form
- Engelfriet, Joost, ii, 4, 56, 79, 81, 95, 96, 107, 111
- equal modulo i/o, **22**
- existence of isomorphic copies, 52
- external node, **22**, 96
- flipping, **44**
 - closure under, 86
- fold
 - in terms of split, 45
- folding, **44**
 - closure under, 86
- full base set, **42**
- full decomposition, 33
- generic class interpretation, **48**
- generic interpretation, **48**
- grammar, **7**
 - context-free, **7**
 - derivation-bounded, **8**
 - linear, **7**
 - ordinary, **12**
 - right-linear, **7**
 - typed, **11**
 - underlying, **12**
- graph
 - abstract, 18
 - concrete, 18
 - ordinary, **20**
 - string, **20**
- graph grammar, 1
 - context-free hypergraph ..., 97
 - eNCE, 56
 - in disguise, 2, 96
 - using BC-hypergraphs, 97
- graphical representation, 19
- Greibach, S. A., 101
- Habel, Annegret, 96, 106, 107
- Herstein, I. N., 41
- Heyker, Linda, 79, 81, 95, 96, 107
- homomorphism
 - λ -free, 51
- Hoogeboom, Hendrik-Jan, 111
- hyperedge, **17**
- hypergraph, 2
 - abstract, *see* graph
 - BC, 96

- concrete, *see* graph
- degree of, **18**
- empty, **26**
- graphical representation of, 19
- identification-free, **18**
- isomorphic, **18**
- permutation, 38
- product, 25
- simple, **18**
- sum, 28
- tentacle of, **19**
- unity, **26**
- hypergraph language, **22**
 - empty, 22
 - singleton, **22**
 - union on, **23**
- i/o
 - equal modulo, **22**
- i/o-hypergraph, **17**
- identification, **25**
- identification-free, **18**
- identifier, 6, **113**
- iff, **6**
- incidence function, **17**, 104
- incident, **18**
- input node, 2, **17**
- input type, 2, **9**, **18**
- internal node, **22**
- internally typed, *see* correctly ...
- interpretation, **48**
 - example of, 49
 - generic, **48**
 - generic class, **48**
 - limitations of, 53
 - power of, 80
 - string graph in terms of, 48
- interpretation function, **48**
- interpreter, 1, **48**
 - Edge Normal Form, 51
 - for L , **48**
- isomorphic copies
 - existence of, 52
- Isomorphic Copies Theorem, **52**
- isomorphic hypergraph, **18**
- isomorphism, 18, 52
- Kleene closure, **10**, **27**
 - closure under, 84
- Knuth, Donald Ervin, 41
- Kreb, Huibert, 111
- Kreowski, Hans-Jörg, 96, 106
- Kruyt, Erik, 111
- labeling function, **17**
- lambda trick, **14**, 58, 65, 99, 100, 103
- Lamerigts, Tycho, 111
- language
 - context-free, **7**
 - derivation-bounded, **8**
 - hypergraph, **22**
 - linear, **7**
 - ordinary, **11**
 - right-linear, **7**
 - typed, **10**
 - underlying, **11**
- language over, **7**
- late night fortune cookie, 4
- L^AT_EX, 111
- layout
 - linear, **20**
- Leih, George, 56
- lemma on cutwidth, 20
- Lengauer, T., 56
- LIN, **7**, 114
- LIN-CFHG, 98

- linear grammar, **7**
- linear language, **7**
- linear layout, **20**
- loop, **20**
 - in a string graph, 21
 - vs sequential composition, 27

- METAFONT, 111
- mixed type, **22**

- n*-sequence, *see* sequence
- naming, 113
- node, **17**
 - external, **22**, 96
 - input, **17**
 - internal, **22**
 - output, **17**
- normal form, *see* Edge Normal Form

- Olofsen, Erik, 111
- operator precedence, 31
- ordinary alphabet, **6**
- ordinary grammar, **12**
- ordinary graph, **20**
 - loop, 20
 - sequential composition on, 25
- ordinary language, **11**
- ordinary set, **9**
- ordinary string, **10**
- ordinary symbol, **9**
- OUT(2DGSM), 79
- OUT(DTWT), 81
- output node, 2, **17**
- output type, 2, **9**, **18**

- parallel composition, 3, **28**
 - associativity of, 28
 - closure under, 86
 - commutativity of, 29
 - relation with sequential composition, 29
 - stacking railroad cars metaphor, 3
 - unity element of, 28

- PASCAL, 8
- permutation hypergraph, 38
- philosophical sidenote, 11
- postfix, **7**
 - proper, **7**
- Power of Interpretation Theorem I, **79**
- Power of Interpretation Theorem II, **80**
- power set, **5**
- precedence of operators, 31
- prefix, **7**
 - proper, **7**
- problems
 - worthy of attack, iii, 4
- product, **25**
 - of ordinary graphs, 25
- pseudo base set, *see* sequential ...

- quotation from
 - A. N. Onymous, 83
 - Abraham Lincoln, 1
 - Alfred North Whitehead, 117
 - Andrew S. Tanenbaum, 113
 - Donald Ervin Knuth, 47
 - Douglas Adams, 111
 - Edsger Wybe Dijkstra, 5
 - Harry S. Truman, 91
 - J. Finnigan, 25
 - Joost Engelfriet and George Leih, 57
 - Leonhard Euler, 17
 - Lewis Carroll, 33
 - Mark Twain, 109
 - Phil Collins, 95
 - The American Heritage Dictionary, 43

-
- railroad cars metaphor, 2
 - rank, **6**
 - ranked alphabet, **6**
 - REG, 7
 - regular, **7**
 - regular expression, 92
 - reversal, **7**, 87
 - right-linear grammar, **7**
 - right-linear language, **7**
 - Rijksuniversiteit te Leiden, 4
 - RLIN, **7**, 114
 - Rosenfeld, A., 95
 - Rozenberg, Grzegorz, 107
 - Schneider, H. J., 95
 - n -sequence, *see* sequence
 - sequential composition, 1, **25**
 - associativity of, 26
 - closure under, 83
 - commutativity of, 27
 - hooking railroad cars metaphor, 3
 - of ordinary graphs, 25
 - relation with parallel composition, 29
 - unity element of, 26
 - vs degree, 27
 - vs loops, 27
 - sequential decomposition, 33
 - sequential pseudo base set, **41**
 - set
 - cardinality of, 5
 - ordinary, **9**
 - sequence over a, **6**
 - typed, **9**
 - silly index entry, 136
 - simple hypergraph, **18**
 - singleton class, **23**
 - singleton hypergraph language, **22**
 - source, 104
 - split
 - in terms of (back)fold, 45
 - split-up, 77
 - splitting, **44**
 - closure under, 88
 - Stibbe, Tieleke, 112
 - STR(Int(RLIN))**, 78
 - strictly over, **7**
 - string, **7**
 - ordinary, **10**
 - typed, **10**
 - string graph, **20**
 - contains no loops, 21
 - degree of, 21
 - in term of interpretation, 48
 - substring, **7**
 - sum, **28**
 - symbol
 - ordinary, **9**
 - typed, **9**
 - tentacle, **19**, 98, 107
 - T_EX, 111
 - theorem
 - on bounded cutwidth, 54
 - on bounded degree, 54
 - on degree versus cutwidth, 54
 - on Edge Normal Form, 51
 - on isomorphic copies, 52
 - on the power of interpretation, 79
 - Tiggeloven, Carin, 111
 - tree
 - binary, 56, 102
 - λ -trick, *see* lambda trick
 - type, **9**, **10**, **18**
 - input, 2, **9**, **18**
 - mixed, **22**
 - output, 2, **9**, **18**

- uniform, **10, 18**
- type conditions, **8**
- type preservingness, **8**
- typed
 - correctly internally, **9**
 - typed class interpretation, **48**
 - typed grammar, **11**
 - typed language, **10**
 - concatenation, **10**
 - Kleene closure, **10**
 - union, **10**
 - typed set, **9**
 - typed string, **10**
 - typed symbol, **9**
- typing, **2**
 - example from PASCAL, **8**
 - example from physics, **8**
- $+ \{U_n\}, \{U_n\} +$
 - closure under, **85**
- underlying grammar, **12**
- underlying language, **11**
- uniform type, **10, 18**
- union, **10**
 - closure under, **84**
 - on hypergraph languages, **23**
- unity element
 - of parallel composition, **28**
 - of sequential composition, **26**
- unity hypergraph, **26**
- variable name clash, **6**
- Vereijken, Frits, **111**
- Vereijken, Jan Joris, **ii, 112**
- vertex, *see* node
- wheel, **50**
- word, *see* string