An Automata-Theoretic Perspective on Polyadic Quantification in Natural Language

MSc Thesis *(Afstudeerscriptie)*

written by

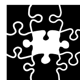
Sarah McWhirter
(born October 11, 1989 in San Antonio, Texas)

under the supervision of **Dr Jakub Szymanik**, and submitted to the Board of Examiners in partial fulfillment of the requirements for the degree of

MSc in Logic

at the *Universiteit van Amsterdam.*

Date of the public defense:	**Members of the Thesis Committee:**
July 2, 2014	Dr Maria Aloni
	Prof. dr Johan van Benthem
	Dr Paul Dekker
	Prof. dr Jan van Eijck
	Dr Jakub Szymanik



INSTITUTE FOR LOGIC, LANGUAGE AND COMPUTATION

Abstract

As part of the general project of procedural semantics, nearly thirty years ago van Benthem first proposed semantic automata as a computational model of natural language quantification. While automata-theoretic characterization results have been obtained for monadic quantifiers, very little has been done to investigate polyadic quantifiers from this perspective. Polyadic quantification in natural language includes but is not limited to iteration, cumulation, resumption, reciprocals, and branching. A natural extension of the semantic automata model is to study the properties of automata recognizing polyadic quantifiers, and the operations on simple automata corresponding to the lifts giving rise to them.

The thesis gives automata constructions for iteration and cumulation and answers (affirmatively) the open question of whether deterministic PDA are closed under these operations. These efforts pave the way toward a novel understanding of the closely related Frege boundary between reducible and "genuinely polyadic" quantification (studied by van Benthem, Keenan, Dekker, and van Eijck, among others) in automata-theoretic terms. An extension of semantic automata for the polyadic lifts which are largely non-Fregean in this sense is left for future work. Finally, the thesis concludes with discussion of the applications of our results and reflection on the importance of representations to the further advancement of this paradigm.

Contents

1 Introduction **3**
1.1 Motivations and Contributions 3
1.2 Themes 4
1.2.1 Meaning as Algorithm 5
1.2.2 Determinism 6
1.2.3 Production vs. Recognition 7
1.2.4 Compositionality 8
1.2.5 Representation 9
1.3 Overview 10

2 Prerequisites **11**
2.1 Generalized Quantifier Theory 11
2.1.1 Properties of Simple Monadic Quantifiers 11
2.1.2 Polyadic Lifts 15
2.2 Formal Languages and Automata Theory 19
2.2.1 Regular Languages and Finite Automata 19
2.2.2 Context-free Languages and Pushdown Automata 21
2.2.3 Deterministic CFL and Deterministic PDA 23

3 Survey of Semantic Automata for Monadic Quantifiers **27**
3.1 Models as Strings 27
3.2 Semantic Finite Automata 29
3.3 Semantic Pushdown Automata 31

4 Known Results for Iteration Automata **35**
4.1 Translating Models with Binary Relations into Strings 36
4.2 Explicit Proofs of Some Closure Results 38
4.3 Summary of Work by Steinert-Threlkeld and Icard III 39

5 Extending Iteration Automata **42**
5.1 Automata for Type $\langle 1, 1, 2\rangle$ Regular Iterations 42
5.1.1 The Construction 42
5.1.2 Proving Correctness 48
5.2 Generalizing to Type $\langle 1, 1, \ldots, n\rangle$ Regular Iterations 49

5.2.1 Translating Models with n-ary Relations into Strings . . . 49
5.2.2 The Construction 51
5.2.3 Proving Correctness 57
5.3 Generalizing the Stack Construction for Regular Iterations 58

6 DCFL Iteration Closure and Iteration DPDA 63
6.1 Closure of DCFLs under Iteration 63
6.2 Automata for Deterministic Context-Free Iterations 68

7 Cumulation Automata 73
7.1 Automata for Type $\langle 1,1,2 \rangle$ Regular Cumulations 73
7.2 Generalizing to Type $\langle 1,1,\ldots,n \rangle$ Regular Cumulations 76

8 Toward A Novel Characterization of the Frege Boundary 81
8.1 Overview and Reformulation of Reducibility Results 83
8.2 Genuinely Polyadic Quantifier Languages are Not Context-Free . 89
8.3 Approaching a General Theorem for a Lower Limit 91

9 Practical Relevance and Future Work 95
9.1 Applications 95
9.1.1 Model Checking 95
9.1.2 Formal Learning Theory 99
9.2 Directions for Future Research 101
9.2.1 Generating FO Semantic Automata by Construction . . . 101
9.2.2 Further Extensions and the Role of Representations . . . 103

A Implementing Semantic Automata in Haskell with HaLeX 110

Chapter 1

Introduction

1.1 Motivations and Contributions

Nearly thirty years ago, van Benthem first introduced the notion of semantic automata, uniting generalized quantifier (GQ) theory and formal language theory in an elegant and powerful way. The basic idea is to use insights of GQ theory to identify a natural language quantifier with an automaton that, in a precise sense, recognizes the models in which it is true, letting us consider the quantifier as a procedure for checking whether it holds. This marriage enables the use of the Chomsky hierarchy as a measure of complexity of quantifiers, which "turns out to make eminent sense, both in its coarse and fine structure" [4]. Semantic automata have not only led to many observations interesting in their own theoretical right, but also to myriad insights in cognitive modeling and formal learning based on the idea of meaning as algorithm, as van Benthem himself predicted:

> ...the procedural perspective may also be viewed as a way of extending contemporary concerns in 'computational linguistics' to the area of semantics as well. Complexity and computability, with their background questions of *recognition* and *learning*, seem just as relevant to semantic understanding as they do to syntactic parsing [4].

A steady flow of research has been inspired by semantic automata in the following decades, but almost no work has been done toward broadening the model to address more than simple monadic quantifiers, though the definability of polyadic quantification is much-studied. Szymanik [50] provides a computational complexity (Turing machine-based) perspective on polyadic quantifiers, showing that some of those natural language constructions are polynomial-time closed and others are NP-hard. Those results prompt the question: can we

obtain stronger automata characterizations for some polyadic quantifiers? This thesis is inspired by a very recent (2013) answer to that question by Steinert-Threlkeld and Icard III [46], exploring semantic automata for iterated quantifiers. They show regular and context-free quantifier languages are closed under iteration; however, their proposed computational model is unnecessarily powerful.

Also in 2013, Kanazawa [27] answered an open question in [38] characterizing the class of quantifiers recognized by deterministic pushdown automata by their corresponding semilinear sets. As it turns out, it is rather difficult to form simple natural language quantifiers that go beyond this characterization. Given this new result, it is interesting to ask whether this natural subset of context-free languages is closed under quantifier iteration.

Our first main contribution is a different construction of iteration automata that is appropriately powerful, which was admittedly missing from the semantic automata landscape thus far. Further, we extend iteration automata (both our version and that presented in [46]) to accomodate any number of quantifiers. Second, we prove that deterministic context-free languages are closed under iteration. The thesis also includes constructions for cumulation automata (for any number of regular quantifiers) and iterations in which one or more quantifier is deterministic context-free but non-regular.

Iterated quantifiers are definable in terms of their monadic constituents and constitute a sort of default for producing complex quantifiers. The *Frege boundary* delineates just how much of polyadic quantification is attainable by multiple iterations, and there are indeed many natural language quantifiers beyond the boundary. Our final contribution is a first step toward establishing a correlate of the Frege boundary within the Chomsky hierarchy using the semantic automata framework.

We hope these contributions add momentum to the recent revival of interest in semantic automata, spurring further research into automata for polyadic quantifiers, and that the fruits of the algorithmic perspective on meaning may be brought to bear on practical applications related to multiquantifier constructions in natural language.

1.2 Themes

This thesis, and the idea of semantic automata in general, concerns topics in a highly inter-disciplinary area, at once in the intersection of formal semantics, computer science, and cognitive science, among other fields. As such there are various interrelated themes that frame our discussion and show up at least implicitly throughout the thesis. We think it useful to make the reader well aware of a few of these recurring motifs from the outset: the notion in computational

semantics of algorithmic meaning, considerations of complexity and determinism in natural language, the division between production and recognition in language understanding, compositionality, and the importance of representations.

1.2.1 Meaning as Algorithm

The idea of meaning as algorithm may be traced all the way back to Frege [16] and his notion of *Sinn*: the meaning of an expression is *the manner in which* its reference is determined. This notion lives on in computational semantics in the idea that the meaning of an expression is an *algorithm* or *procedure* for determining its extension in a model. This idea was first seriously formalized in Tichy's *Intension in terms of Turing machines* [55], an "attempt to base semantics directly on the concept of sense." Various possibilities for procedural semantics are discussed by van Benthem in *Towards a computational semantics* [5], including the semantic automata models for quantifiers that we further develop in this thesis.[1]

One of the biggest motivations to develop procedural semantics is the possibility of a psychologically plausible model of language. For example, Suppes claims:

> The basic and fundamental psychological point is that, with rare exception, in applying a predicate to an object or judging that a relation holds between two or more objects, we do not consider properties or relations as sets. We do not even consider them as somehow simply intensional properties, but we have procedures that compute their values for the object in question [47].

In cognitive science, Marr [33] has famously proposed that information processing systems, like the cognitive procedures employed in comprehending language, must be understood at three levels:

- Computational Theory (Level 1): What is the goal, why is it appropriate, and what is the logic of the strategy to carry it out?
- Representation and Algorithm (Level 2): How can the computational theory be implemented? How are the input and output represented, and how is one transformed into the other?
- Implementation (Level 3): How can the representation and algorithm be realized in a physical system?

Marr further claims that the highest level, the computational level, tells us most about the problem at hand, offering that "Trying to understand perception by studying only neurons is like trying to understand bird flight by studying only feathers." We need first, for example, an understanding of aerodynamics to understand why bird feathers are particularly good for flying. The nature of

[1] See [49] for a full run-through of the history of the algorithmic perspective on meaning.

flight constrains the kind of mechanism that may implement flight, but plenty of (in)animate things fly without feathers. The same goes for a cognitive task like verifying the truth of a quantified sentence.

The approach in this thesis perhaps straddles the computational and algorithmic level. Complexity differences between automata classes are reflected in measures of task difficulty, and even the more fine-grained structure of particular automata lead to accurate empirical hypotheses; however, we do not claim that people *actually* implement the algorithm represented by a given automata. Rather, semantic automata are idealized models of quantifier verification.[2] People may guess, approximate, or use some other strategy that is dependent upon the way the information is presented.

Steinert-Threlkeld puts it nicely:

> In principle, they [semantic automata] allow for a separation between abstract *control structure* involved in quantifier verification and innumerable other variables that the framework leaves underspecified ... This could be viewed as a modest distinction between *competence* and *performance* for quantifier expressions [46].
>
> ...the standard model-theoretic semantics of quantification could be seen as a potential computational, or level 1, theory. Semantic automata offer more detail about processing, but...less than one would expect by a full algorithmic story about processing (level 2), which would include details about order, time, salience, etc. Thus, we might see the semantic automata framework as aimed at level 1.5 explanation, in between levels 1 and 2, providing a potential bridge between abstract model theory and concrete processing details [46].

We discuss the fruitfulness of the semantic automata model in various practical applications in Section 9.1.

1.2.2 Determinism

If we pursue the paradigm of procedural semantics for its promise of the possibility of a cognitively tenable model of language processing, we might attend to this goal from the get-go by *not* positing *implausible* models. Since we deal with machine characterizations of procedures in this thesis, we may look to the divide between determinism and non-determinism for at least one distinction between the plausible and implausible[3]. Non-deterministic automata simply do not provide realistic algorithmic models as they are not implementable. Even

[2]However, see [13] for the first step toward realistic semantic automata, using probabilistic transitions to model verification error.

[3]Dealing with computational complexity, Szymanik [50] takes *tractability* as the natural notion of plausibility–in particular, polynomial-time computability (see van Rooij [42] for discussion of the P-cognition thesis).

when a non-deterministic automata has an equivalent deterministic counterpart, it is only the latter that is of practical interest to us, as a representation of the computational steps one may really take. Based on this entirely innocuous claim that people cannot utilize non-determinism, we only offer constructions of deterministic semantic automata in this thesis.[4]

As we will see at the end of Section 3.3, this assumption of determinism is incredibly natural in this context: the basic building blocks of natural language quantification are deterministic, and it takes a bit of effort to jump the hurdle into non-determinism. Still, squaring realistic assumptions about cognition with the existence of (compound) natural language determiners whose semantic automata models require non-determinism remains an interesting puzzle.

1.2.3 Production vs. Recognition

In formal language theory, every recognizing device has an equivalent generative device. For every kind of automata, there is a corresponding type of grammar such that each describes the same class of languages. The ability to translate between the two paradigms is extremely useful: one representation may be preferable in a context, or make it easier to obtain a particular result, or reveal a new insight.

Positioning the previous discussion of meaning as algorithm, in particular the meaning of quantifiers as automata, in the wider of context of formal language theory, the meaning of a quantifier should somehow be partially or alternatively constituted by a corresponding generative mechanism. These dual meanings are attested by the approach in formal *learning* theory, which seeks to explain how people gain semantic competence. Semantic competence consists in the ability to verify the truth of a sentence (comprehension) but also in the ability to produce correct utterances (production). These abilities align with automata as testing or model-checking devices and grammars as description generating devices.

In [18], Gierasimczuk calls attention to the issue that in using automata and grammars for modeling learning, we cannot treat the corresponding abilities of comprehension and production as equivalent. In short, production is *harder* for people, and comprehension is somehow prior: "Generating is more complicated than testing and the assumption of mutual reduciblity of these two competences seems unrealistic."

This is good news for us. In this thesis we are motivated by the possibility of modeling lifts of simple quantifiers as operations on automata by giving constructions *directly* from minimal automata to minimal automata, *without* having

[4]Note that we are *not* making the radical claim that human cognition never involves *probabilistic* procedures. See van Rooij [42] also for a discussion on the common confounding of non-deterministic and probabilistic procedures.

to translate to grammars. It may be easy to prove the existence of some automata by demonstrating the equivalent grammar, but this does not necessarily provide insight into the procedure. If we take the algorithmic meaning paradigm seriously and give due respect to the actual differences in generating and testing, it is accurate and valuable to keep to one side of the conceptual line dividing them.

While a proof by grammar and proof by automata indeed have the same force *qua* proof, we often give both sides of the story in this thesis. Grammars can sometimes seem more precise and guide the construction of automata with structural insight into the language (and, since the properties of these automata operations are interesting in their own right–divorced from their natural language applications–we are justified in taking advantage of the equivalence of grammars and automata). Nonetheless, our automata constructions themselves stand on their own.

1.2.4 Compositionality

A pervasive idea across many disciplines is the principle of compositionality: the meaning of a compound expression is determined by the meanings of its constitutent parts and their mode of combination. Two standard arguments for the compositionality of natural language are that it explains (1) how people can learn a language (a set of *infinite* meanings) while in fact internalizing only finitely many basic meanings, and (2) how people successfully communicate using novel expressions.[5] A comprehensive treatment of the history, justifications for, objections to, and formalization of compositionality can be found in [25]. Janssen points to its prevalence across several contexts such as, of course, in logic, in which "it is hardly ever discussed...and almost always adhered to" and in formal semantics, e.g. "the fundamental principle of Montague grammar." The basic idea behind a formal analysis of compositionality is simple: if E is a complex expressions built up from E_1 and E_2 (by some syntactic rule), then the meaning $M(E)$ of E is built up from $M(E_1)$ and $M(E_2)$ (by some semantic rule).

Thus, in keeping with compositionality, we would like the semantic automata giving the meaning of a polyadic quantifier to be built up from the semantic automata giving the meaning of the simpler monadic quantifiers composing it.[6] Moreover, the composition of those meanings should somehow reflect the way the quantifiers are combined in language: the principle is a claim not only about meaning, but about structure. Compare *The dog bit the man* and *The man bit the dog*, sentences containing exactly the same parts with exactly the same

[5] Pagin [39] claims that these arguments fail on the grounds that those capacities do not require *compositionality*, but rather *computability*, still ultimately concluding that compositionality simplifies linguistic calculations. This yields more evidence for the utility of a computational perspective on natural language.

meanings, but with very different overall meaning based on the way the parts are combined.

Since this thesis continues in the tradition of procedural semantics, in particular identifying quantifiers with automata, we must acknowledge that there will be multiple ways to combine simple algorithms into equivalent complex algorithms, in the sense that they produce the same outputs for all inputs.[7] But algorithms can be more or less similar, and there is no general definition of synonymy. This is an interesting hurdle for the extension and application of semantic automata. Is there one *right* model among the possibilities? If so, what are the criteria for choosing?

1.2.5 Representation

Finally, we mention the importance of attention to representations in establishing the difficulty of a language or problem. In his *Computational Complexity: A Conceptual Perspective*, Goldreich emphasizes:

> Indeed, the importance of representation is a central aspect of Complexity Theory. In general, Complexity Theory is concerned with problems for which the solutions are implicit in the problem's statement (or rather in the instance). That is, the problem (or rather its instance) contains all necessary information, and one merely needs to process this information in order to supply the answer...Thus, Complexity Theory clarifies a central issue regarding representation, that is, the distinction between what is explicit and what is implicit in a representation [22].

In this thesis we are not directly interested in the inherent time or space required to decide a problem, but the issue of representation plays a similar role in any theory of computation endeavor. Finding an appropriately powerful automaton model for a quantifier, represented as a language of string encodings of models, is another way of specifying the hardness of that quantifier. These encodings indeed contain all the information relevant to deciding whether strings are in the language of the quantifier, but some encodings make particular aspects of that information more or less explicit than others. Which aspects are useful in this decision process may vary with the type of quantifier. This theme becomes more and more prominent throughout the thesis, playing a large role in Chapters 7 and 8, and is indeed the final thought we reflect on, in Section 9.2.2.

[6] Also see Chapter 9.2 for a discussion of Clark's project in [10] showing how to construct the monadic quantifiers definable in first-order logic via operations on regular languages.

[7] Also from Suppes: "It has been a familiar point in philosophy since the last century that classes are abstractions of properties. The point relevant here is that properties stand in the same relation to procedures that classes stand to properties. For example, the property of a number being prime can be tested by quite different procedures and among this indefinitely large number of procedures some will of course be much more efficient or faster than others" [47].

1.3 Overview

The thesis begins by providing the necessary exposition for understanding the results in later chapters. Section 2.1 gives an overview of Generalized Quantifier Theory; Section 2.2 gives an overview of Formal Language and Automata Theory. (Readers familiar with these areas are invited to skip the prerequisite chapter). Chapter 3 illustrates the synthesis of these two fields in the form of semantic automata for monadic quantifiers and surveys the results so far obtained. This overview provides justification for further exploration in the same spirit into the realm of polyadic quantification and furnishes the reader with the background to not only understand but *appreciate* the results in later chapters.

Chapter 4 provides an overview of the work by Steinert-Threlkeld and Icard III in [46], establishing the necessary vocabulary for defining iteration automata and providing new and more precise proofs of their results.

In Chapter 5 we give our construction for the iteration of two finite automata corresponding to regular quantifiers and generalize this construction to handle iterations of arbitrary numbers of finite automata, with proofs of the correctness of the definitions in both cases. We also generalize the construction in [46], with an interesting outcome. Chapter 7 does the same for cumulative quantification, which is straightforwardly definable from iteration. Chapter 6 turns to deterministic context-free quantifier languages, proving they are closed under quantifier iteration and demonstrating a construction for the iteration of two automata where at least one is a deterministic pushdown automata.

In Chapter 8 we discuss the much (though not so recently) studied Frege boundary demarcating reducible and genuinely polyadic quantification, reformulating some results in that realm in terms of languages in a step toward locating the Frege boundary in the Chomsky hierarchy.

Section 9.1 gives additional motivation for this project and suggests its import for modeling in cognitive science and for formal learning theory. Lastly, Section 9.2 comments on a project of Clark ([9], [8]) in a similar spirit and gestures toward possible further extensions of the semantic automata model to capture quantification on the far side of the Frege boundary.

Note. This thesis is largely concerned with *iteration* of quantifiers, and unfortunately this may engender some confusion, as iteration already has a (different) meaning in the realm of formal languages (the Kleene star, or concatenation closure). The reader should take care to remember that when we use the term *iteration* in this thesis, it almost invariably refers to the *polyadic lift* discussed in Section 2.1 and the corresponding operation on languages and automata to be defined. We may sometimes refer to our notion as *quantifier iteration*, but in any case will not use *iteration* to refer to the Kleene star.

Chapter 2

Prerequisites

2.1 Generalized Quantifier Theory

2.1.1 Properties of Simple Monadic Quantifiers

Our summary of generalized quantifier theory draws heavily on Peters and Westerståhl's *Quantifiers in Language and Logic* [41]. A quick notational remark: we use "$|A|$" for both the cardinality of A (as a set) and the length of A (as a string), and thus ":" for set comprehension to enhance readability.

Generalized quantifier theory applied to natural language treats determiners as relations between the denotations of other constituents of a sentence. For example, every is the the inclusion relation: *Every student wrote a thesis* is true just in case every individual in the set of students is also in the set of thesis-writers. Counting quantifiers such as at least 3 put a restriction on the cardinality of the intersection of two sets: *At least 3 students received a mark of 10 on their theses* is true just in case the size of the intersection of the set of students and the set of people receiving a 10 is at least three. For example, we can write the meanings of the quantifiers in these examples as:

$$\begin{array}{lcl} \textsf{every} & = & \{(M,A,B) : A \subseteq B\} \\ \textsf{at least three} & = & \{(M,A,B) : \ |A \cap B| \geq 3\} \end{array}$$

These are simple examples, but a quantifier may denote a relation between any number of relations of any arity. Mostowski [37] first introduced the general notion of a unary quantifier, binding a single variable in a formula similarly to $\forall$ and $\exists$ in standard logic. Lindström extended this to arbitrary types.

Definition 2.1.1. [32] A Lindström quantifier Q of type $\langle n_1, \ldots, n_k \rangle$ is a class of models $\mathcal{M} = (M, R_1, \ldots, R_k)$ with the R_i n_i-ary that is closed under isomorphism.

This definition is the standard in logical settings, while the following appears more often in formal semantics.

Definition 2.1.2. A generalized quantifier assigns a k-ary relation Q_M to every set M such that if $(R_1, \ldots, R_k) \in \mathsf{Q}_M$ then R_i is $n_i - ary$ and is closed under bijections.

However, they are equivalent due to the following:

$$(M, R_1, \ldots, R_k) \in \mathsf{Q} \Leftrightarrow \mathsf{Q}_M(R_1, \ldots, R_k)$$

We leave out the subscript and just write $\mathsf{Q}(R_1, \ldots, R_k)$ or $\mathsf{Q}\ R_1, \ldots R_k$ omitting parentheses.

If all the n_i are 1, we say Q is monadic; otherwise it is polyadic. Typical natural language determiners are type $\langle 1, 1\rangle$ relativizations of type $\langle 1\rangle$ quantifiers.

Definition 2.1.3. If Q is of type $\langle n_1, \ldots, n_k\rangle$, then Q^{rel} has type $\langle 1, n_1, \ldots, n_k\rangle$ and is defined for $A \subseteq M$ and $R_i \subseteq M^{n_i}$ as:

$$(\mathsf{Q}^{\text{rel}})_M(A, R_1, \ldots, R_k) \Leftrightarrow \mathsf{Q}_A(R \cap A^{n_1}, \ldots, R \cap A^{n_k})$$

Again, we often omit the subscript. For instance, every is the relativization of the familiar $\forall$ from predicate logic. $\forall(B)$ says that all individuals in the universe are in B. But $\forall^{\text{rel}} =$ every restricts the domain of $\forall$ to a new unary relation (set of individuals) contained in M, allowing us to write every A B to mean that the elements of A are also elements of B. In general we have:

$$(\mathsf{Q}^{\text{rel}})_M(A, B) \Leftrightarrow \mathsf{Q}_A(A \cap B)$$

Natural language determiners are generally taken to satisfy certain semantic universals ([3]):

- Q of type $\langle 1, 1\rangle$ satisfies extensionality (EXT) if and only if for all $A, B \subseteq M$ and $M \subseteq M'$:

$$\mathsf{Q}_M(A, B) \Leftrightarrow \mathsf{Q}_{M'}(A, B)$$

- Q of type $\langle 1, 1\rangle$ is conservative (CONS) if and only if for all M and $A, B \subseteq M$:

$$\mathsf{Q}_M(A, B) \Leftrightarrow \mathsf{Q}_M(A, A \cap B)$$

- Q of type $\langle 1, 1\rangle$ satisfies isomorphism closure (ISOM)[1] if and only if for all $A, B \subseteq M$ and $A', B' \subseteq M'$, if $(M, A, B) \cong (M', A', B')$:

$$\mathsf{Q}_M(A, B) \Leftrightarrow \mathsf{Q}_{M'}(A', B')$$

EXT says that the domain can shrink or expand without affecting Q's value so long as A and B are untouched. CONS says that only the part of B that is also in A ($A \cap B$) is relevant to Q's value (since Q restricts the domain to its first argument). ISOM states a sort of topic-neutrality: the identities of the

[1] Note that isomorphism closure is part of our definition of generalized quantifier from the outset.

individuals are irrelevant, from which it follows that it is the cardinalities of the sets, and not their particular members, which determine Q's value.

Theorem 2.1.4. [41] A type $\langle 1,1\rangle$ quantifier satisfies CONS and EXT if and only if it is a relativization of a type $\langle 1\rangle$ quantifier.

Natural language determiners are almost invariably type $\langle 1,1\rangle$ relativizations. From a given type $\langle 1,1\rangle$ determiner, we can also recover a type $\langle 1\rangle$ quantifier, representing a noun phrase, by *freezing* the first argument[2]:

$$(Q^A)_M(B) \Leftrightarrow Q_{M\cup A}(A,B)$$

For instance, we may write *Every A is B* as the type $\langle 1,1\rangle$ determiner every applied to (A,B) or equivalently as the type $\langle 1\rangle$ noun phrase every^A applied to (B), true if the set of every element of A (i.e., A itself) is contained in B.

Quantifiers additionally satisfying ISOM (called CE quantifiers) have an equivalent representation as binary relations on numbers:

Definition 2.1.5. For Q a type $\langle 1,1\rangle$ CE quantifier, define Q^c by:

$$\mathsf{Q}^c(x,y) \Leftrightarrow \exists M \text{ and } A,B\subseteq M \text{ s.t. } |A-B|=x, |A\cap B|=y \text{ and } \mathsf{Q}_M(A,B)$$

Theorem 2.1.6. Let Q of type $\langle 1,1\rangle$ be a CE-quantifier. Then for all M and $A,B\subseteq M$:

$$\mathsf{Q}(A,B) \Leftrightarrow \mathsf{Q}^c(|A-B|,|A\cap B|)$$

Throughout this thesis, we will consider only quantifiers having these three semantic properties, which are indeed nearly universal for natural language determiners.[3]

Our earlier example determiners are given by the following binary relations:

$$\begin{array}{lcl} \mathsf{every}^c(x,y) & \Leftrightarrow & x=0 \\ \mathsf{at\ least\ three}^c(x,y) & \Leftrightarrow & y\geq 3 \end{array}$$

Due to this equivalence with relations on natural numbers, CE quantifiers have a nice geometric representation in the Tree of Numbers ([4]). We can identify Q with the pattern in the Tree determined by which pairs (x,y) are in Q^c. The Tree also provides an easy characterization of many invariance properties of simple quantifiers, a few of which we mention now.

We say Q is *monotone increasing* (*decreasing*) in its right argument, written MON↑ (MON↓) if and only if for all M, A, and $B\subseteq B'$ ($B'\subseteq B$), $\mathsf{Q}_M(A,B)$ implies $\mathsf{Q}_M(A,B')$. Left monotonicity, called *persistence* (*anti-persistence* and written ↑MON (↓MON), is defined similarly with respect to taking subsets and

[2]The use of $M\cup A$ makes this a global rather than local definition, but the difference does not matter here (we will always have $A\subseteq M$).

[3]This assumption puts aside, for example, proper names and, more controversially, *only*. *Only A's are B's* depends also on $B-A$.

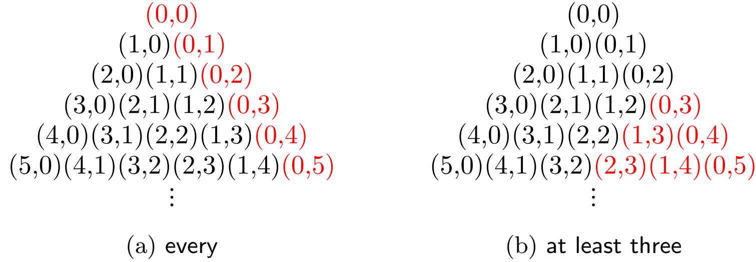


Figure 2.1: Interpreting quantifiers on the Tree of Numbers

supersets of A.[4] We say Q is (*right*) *continuous* if for all M, A, and $B' \subseteq B \subseteq B''$, $\mathsf{Q}_M(A, B')$ and $\mathsf{Q}_M(A, B'')$ imply $\mathsf{Q}_M(A, B)$.

Every CE quantifier is the relativization $\mathsf{Q}^{\mathrm{rel}}$ of a type $\langle 1 \rangle$ quantifier Q. Right montonicity behavior carries over from the monotonicity of Q. Persistence, or monotonicity in the restricted (left) argument, of $\mathsf{Q}^{\mathrm{rel}}$ does not have such a clear relation with Q *until* we consider its interpretation in the Tree of Numbers.

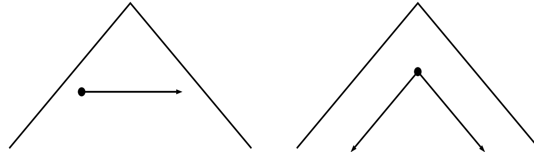


Figure 2.2: Right monotonicity and persistence

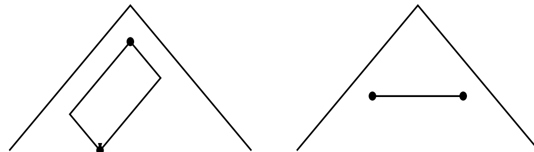


Figure 2.3: Left and right continuity

Figures 2.2 and 2.3 illustrate what we can infer about the Tree pattern of a quantifier possessing these properties. If Q is monotonic increasing in the right argument, then if a pair (x, y) is in Q^c, we know that every pair (x, y') with $y' > y$ is also in Q^c. If Q is persistent and (x, y) is in Q^c, then every pair in the downward triangle spanned by (x, y) is also in Q^c (for anti-persistence, it is the upward triangle). Moving down a level in the Tree corresponds to adding an individual to the domain; the downward triangle pattern it does not matter

[4] Often "monotone increasing (decreasing)" is assumed to refer to the right argument, and simply "monotone" is used to indicate that a quantifier is either monotone increasing or decreasing, as opposed to neither.

whether the individual is added to $A-B$ or $A \cap B$. For right continuity, if (x,y) and (x,y'), with $y' > y$, are in Q^c, then so is every pair in between. For left continuity, every pair in the rectangle determined by any two points in Q^c is also in Q^c.

2.1.2 Polyadic Lifts

Monadic quantifiers are sufficient to analyze simple sentences following the schema Q_1 *A are B*, as in *Every Olympian is an athlete.* However, natural language is full of examples of polyadic quantification:

(1) *Half the students passed every class.*

(2) *Three researchers published five papers in total.*

(3) *Not all twins are friends.*

(4) *Five hockey players punched each other.*

(5) *Some relative of each townsmen and some relative of each villager hate each other.*

We restrict the following definitions to the case that a *polyadic lift* is applied to at most two simple monadic quantifiers, but they may all be defined for an arbitrary number of quantifiers.

Sentence (1) is an example of *iteration.* Taking S as the set of students, C as the set of classes, and $P = \{(s,c) : s \text{ passed } c\}$, we can give the truth conditions of sentence (1) as:

$$(\mathsf{half} \cdot \mathsf{every})(S, C, P) \Leftrightarrow \mathsf{half}(S, \{s : \mathsf{every}(C, P_s)\})$$

where P_s is the set $\{c : (s,c) \in P\}$. The iteration of half and every yields a new quantifier $\mathsf{half} \cdot \mathsf{every}$ which takes two sets and a binary relation between them as arguments.

Definition 2.1.7. Let Q_1 and Q_2 both be of type $\langle 1,1 \rangle$. $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is the type $\langle 1,1,2 \rangle$ quantifier such that for all $A, B \subseteq M$ and $R \subseteq M^2$:

$$(\mathsf{Q}_1 \cdot \mathsf{Q}_2)(A, B, R) \Leftrightarrow \mathsf{Q}_1(A, \{a : \mathsf{Q}_2(B, R_a)\})$$

The iteration of three type $\langle 1,1 \rangle$ quantifiers creates a type $\langle 1,1,1,3 \rangle$ quantifier, and so forth.[5]

Iterations inherit some nice properties from their components. Zuber [60] observes the following facts (where the relevant notions of monotonicity and continuity of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ are obtained by adding or subtracting pairs in the relation argument):

[5] We can equivalently treat the iteration of two quantifiers as a type $\langle 2 \rangle$ quantifier by freezing the first two arguments: $(\mathsf{Q}_1^A \cdot \mathsf{Q}_2^B)(R)$.

- $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is monotone increasing if and only if Q_1 and Q_2 are both increasing or both decreasing
- $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is monotone decreasing if and only if one of Q_1 or Q_2 is increasing and the other decreasing
- $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is not monotonic if and only if one of Q_1 or Q_2 is not monotonic
- If $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is continuous, then Q_1 and Q_2 are continuous (if non-trivial)[6]

Sentence (2) is an example of *cumulation*, meaning that there are three researchers such that *all together*, they published five papers. Each researcher in the group worked on at least one paper, and each of the five papers was worked on by at least one of the researchers. Compare this to the iterative reading of this sentence, under which we expect each researcher published five papers separately, requiring fifteen total papers.

We denote the cumulation of Q_1 and Q_2 by $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)^{\mathsf{cl}}$ and observe that it can be defined in terms of iteration:

$$(\mathsf{Q}_1 \cdot \mathsf{Q}_2)^{\mathsf{cl}}(A, B, R) \Leftrightarrow (\mathsf{Q}_1 \cdot \mathsf{some})(A, B, R) \wedge (\mathsf{Q}_2 \cdot \mathsf{some})(B, A, R^{-1})$$

This says there is a Q_1-sized subset of A that participates in R and a Q_2-sized subset of B that participates in R^{-1}.

Note that we obtain precisely the same truth conditions for a cumulative reading whether the sentence is *Three researchers published five papers* or *Five papers were published by three researchers.* This means cumulation is an *independent lift.* Let lift be any operation taking two type $\langle 1,1\rangle$ quantifiers and producing a type $\langle 1,1,2\rangle$ quantifier. Then lift is independent, corresponding to the order-indifference of Q_1 and Q_2, if:

$$\text{For all } M \text{ and } A, B \subseteq M, R \subseteq M^2,\ \mathsf{lift}(\mathsf{Q}_1, \mathsf{Q}_2)_M \Leftrightarrow \mathsf{lift}(\mathsf{Q}_2, \mathsf{Q}_1)_M(B, A, R^{-1})$$

Iteration, on the other hand, is generally not order-indifferent. Compare *Every dog chased some cat* and *Some cat was chased by every dog.* In the former, *every* takes wide-scope, and the sentence may be true when every dog chases a different cat.

Sentence (3) illustrates *resumption.* Resumption has the effect of lifting Q to quantify over tuples instead of individuals. We define the resumption of Q, with universe M and $R_1, R_2 \subseteq M^2$ by:

$$Res^k(\mathsf{Q})_M(R_1, R_2) \Leftrightarrow \mathsf{Q}^k_M(R_1, R_2)$$

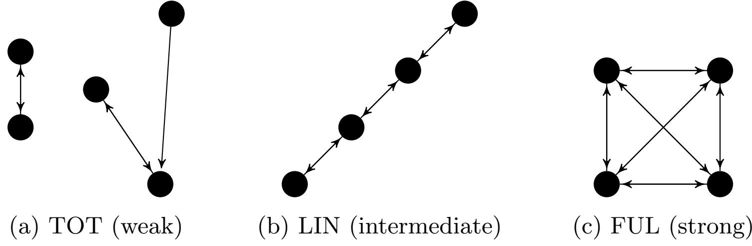
We can read the determiner in (3) as $Res^2(\mathsf{not\ all})$ ranging over pairs. Taking twins and $\mathsf{friends}$ to denote subsets of M^2 (sets of pairs), we have $Res^2(\mathsf{not\ all})(\mathsf{twins}, \mathsf{friends})$ if and only if $\mathsf{twins} - \mathsf{friends} \neq \varnothing$.

Sentence (4) exemplifies reciprocal quantification. Reciprocal quantifiers say there is a large-enough subset of a set whose members participate in a relation

[6] A quantifier is *trivial* if it is a universal or empty relation.

with one another. Reciprocal quantifiers have many possible interpretations varying in strength depending on the restrictions put on the relation. We can think of the domain A with relation R as the nodes and edge relation of a directed graph. Meanings depend on the following possible interpretations of R:

- FUL: Every pair in A participates in R directly (the graph is complete).
- LIN: Every pair in A participates in R directly or indirectly (the graph is connected).
- TOT: Every element in A participates in R directly with some element of A (every node has an outgoing edge).



(a) TOT (weak) (b) LIN (intermediate) (c) FUL (strong)

Figure 2.4: Models for different interpretations of reciprocals

These interpretations correspond to the following truth conditions[7] for the *strong*, *intermediate*, and *weak* readings, depending on whether the relation is FUL, LIN, or TOT, respectively (where $\mid A \mid$ is at least 2 and Q is MON↑) ([11],[50]):

- $\mathsf{Ram}_{\mathsf{S}}(\mathsf{Q})(A, R) \Leftrightarrow \exists X \subseteq A[\mathsf{Q}(A, X) \wedge \forall x, y \in X(x \neq y \Rightarrow R(x, y))]$
- $\mathsf{Ram}_{\mathsf{I}}(\mathsf{Q})(A, R) \Leftrightarrow \exists X \subseteq A[\mathsf{Q}(A, X) \wedge \forall x, y \in X(x \neq y \Rightarrow \exists$ sequence
  $$z_1, \ldots, z_l \in X : (z_1 = x \wedge R(z_1, z_2) \wedge \cdots \wedge R(z_{l-1}, z_l) \wedge z_l = y)]$$
- $\mathsf{Ram}_{\mathsf{W}}(\mathsf{Q})[A, R] \Leftrightarrow \exists X \subseteq A[\mathsf{Q}(A, X) \wedge \forall x \in X \exists y \in X(x \neq y \wedge R(x, y))]$

We also have $\mathsf{Ram}^{\vee}_{\mathsf{S}}(\mathsf{Q}), \mathsf{Ram}^{\vee}_{\mathsf{I}}(\mathsf{Q})$, and $\mathsf{Ram}^{\vee}_{\mathsf{W}}(\mathsf{Q})$ with $R(x, y)$ replaced by $R(x, y) \vee R(y, x)$ in the above definitions (for these *alternative* readings, think of an undirected graph). This yields at least six possible interpretations[8], with the strong meaning implying the intermediate meaning, and the intermediate implying the weak.

To explain how the meaning of a given reciprocal quantifier is chosen from this set of possibilities, Dalrymple et al. [11] propose the *strongest meaning hypothesis* (SMH), a pragmatic principle predicting that a sentence has the

[7] "Ram" indicates that reciprocal quantifiers are *Ramsey lifts*. The Ramseyfication of Q expresses that there is a Q-large subset whose members satisfy some formula.

[8] Dalrymple et al. [11] discuss many more potential readings; we present just these six, as formulated in [50], for simplicity.

logically strongest possible meaning allowed by the context. For example, the meaning of *Most of the children followed one another* is underspecified, except insofar as the strong meaning is ruled out since *follow* is an asymmetric relation. Different contextual information may determine different meanings. In *Most of the children followed one another through the door*, an alternative intermediate interpretation is called for, but in *Most of the children followed one another in a circle*, the regular intermediate interpretation is possible and logically stronger, and thus is predicted by SMH. Suppose the sentence were *Most of the children followed one another around the museum in small groups*: then it seems only the alternative weak reading is available.

An interesting perspective on the meaning of reciprocals is provided by the descriptive complexity results in [50]. Szymanik shows that the strong readings of reciprocal sentences in which the quantifier is counting or proportional are intractable (the problem of determining their truth in a model is NP-complete, since the Ramseyification of Q can be used to define the clique problem, which is famously NP-complete). He argues this entails that people will disprefer strong readings in such cases in favor of tractable ones based on the P-cognition thesis.[9] The variation in complexity of interpretations for reciprocals corroborates SMH but also predicts people will shift to weaker readings though an intractable strong meaning is consistent with the context. [43] presents empirical evidence for this claim.[10]

Sentence (5) is Hintikka's sentence, an example of Branching, or partially-ordered, quantification. The idea is that *some relative of each townsmen* ($\forall x_1 \exists y_1$) and *some relative of each villager* ($\forall x_2 \exists y_2$) are chosen independently, so the truth conditions cannot be given by any first-order sentence since the quantifiers would appear in some linear order, introducing scope dependencies. The hallmarks of branching quantification are noun phrases joined by *and* and relations that are in some sense reciprocal; however, the exact conditions are contentious and it is even controversial whether any natural language sentences really have branching meanings.[11] In general, we define the branching of two type $\langle 1,1\rangle$ MON$\uparrow$ quantifiers as:

$$Br^2(\mathsf{Q}_1,\mathsf{Q}_2)(A,B,R) \Leftrightarrow$$
$$\exists X \subseteq A \exists Y \subseteq B (\mathsf{Q}_1(A,X) \wedge \mathsf{Q}_2(B,Y) \wedge X \times Y) \subseteq R$$

Like cumulation, branching is an independent lift.

---

[9]See footnote 3, page 6 for a reference.

[10]The experiments confirm that intractable readings do exist in natural language, but that, contrary to the Strong Meaning Hypothesis, people strongly disprefer intractable readings and are more error-prone in those cases.

[11]See for example [19] in which Gierasimczuk and Szymanik challenge Hintikka's thesis (that branching quantifiers have no satisfactory linear representation) based on linguistic and logical observations and propose an alternative reading.

2.2 Formal Languages and Automata Theory

This exposition mostly follows Hopcroft and Ullman's *Introduction to Automata Theory, Languages, and Computation* [23], except for Section 2.2.3 which follows Sipser's *Introduction to The Theory of Computation* [44].

An *alphabet* Σ consists of a finite set of letters (symbols). A finite sequence of letters is a *word* or string. We sometimes refer to a portion of a given word as a *subword*. Σ^* denotes the set of all words over Σ. A *language* $\mathcal{L}$ is some set of strings contained in Σ^*. Its *complement*, written $\overline{\mathcal{L}}$ and sometimes $\neg\mathcal{L}$, is given by $\Sigma^* - \mathcal{L}$ (thus a language and its complement always share an alphabet).

2.2.1 Regular Languages and Finite Automata

Definition 2.2.1. A deterministic finite automaton (DFA) A is a five-tuple $(\mathcal{Q}, \Sigma, \delta, s, F)$ where:

- $\mathcal{Q}$ is a finite set of states
- Σ is an input alphabet
- δ is a function from $\mathcal{Q} \times \Sigma$ to $\mathcal{Q}$
- s, an element of $\mathcal{Q}$, is the start state
- F, a subset of $\mathcal{Q}$, is a set of final states

A DFA is often graphically represented as a set of nodes (the states of the machine, with the start state indicated by an ingoing arrow with no source, and the final states doubly circled) with labeled, directed edges between them (representing the transition function). An edge from q to p labeled a means that $\delta(q, a) = p$. We can extend δ to be defined for entire strings in the obvious way, setting $\delta(q, w) = \delta(\delta(q, a), v)$ where $w = av$ and a is a single symbol of Σ. The language of A is the set of strings w such that a *run* of A (a computation beginning in s, reading w and transitioning according to δ) ends in a final state:

$$L(A) = \{w : \delta(s, w) \in F\}$$

A non-deterministic finite automaton (NFA) has a transition *relation* rather than function, returning a set of states. The machine may have *epsilon* (ϵ) moves that don't consume any input symbol and more than one move per alphabet symbol in a single state. This reflects the idea that the NFA can non-deterministically "guess" its next move and keep track of every state it might be in. A string w is accepted in case there exists some run ending in a final state, so the language definition for A an NFA becomes:

$$L(A) = \{w : \delta(s, w) \cap F \neq \varnothing\}$$

NFA and DFA are provably equivalent (for every NFA there is a DFA accepting the same language), and we can obtain the latter from the former by following

the *subset construction.* For an NFA A with states $\mathcal{Q}$, its DFA A' has $\mathcal{P}(\mathcal{Q})$ for its set of states. Thus A' may have $2^{|\mathcal{Q}|}$-many states, an exponential increase, though many of those states will be equivalent. For every DFA accepting a language $\mathcal{L}$ there is a *minimal* DFA accepting $\mathcal{L}$ such that any equivalent DFA has at least as many states.

Definition 2.2.2. *Regular expressions* are algebraic descriptions of sets of strings. We say E is a regular expression over alphabet Σ if E is:

1. $a \in \Sigma$, ϵ, or $\varnothing$
2. $E_1 + E_2$, for E_1 and E_2 regular expressions
3. $E_1 E_2$, for E_1 and E_2 regular expressions
4. E_1^*, for E_1 a regular expression

As with finite automata, we have the *language* of a regular expression, denoted $L(E)$. $L(E_1 + E_2)$ is $L(E_1) \cup L(E_2)$, $L(E_1 E_2)$ is $L(E_1)L(E_2)$ (the concatenation)[12], and $L(E_1^*)$ is $L(E_1)^*$ (the Kleene star or concatenation closure).

Regular expressions and finite automata are equivalent ways of describing the regular languages.

Theorem 2.2.3. [30] If $\mathcal{L} = L(A)$ for some DFA A, then there is a regular expression E such that $\mathcal{L} = L(E)$. Finite automata and regular expressions generate exactly the same languages (regular languages).

Now we record some useful closure results for regular languages. By Theorem 2.2.3, demonstrating these results by producing a regular expression or finite automaton for the language are both acceptable, but sometimes one method is simpler. We very briefly sketch how these results can be demonstrated by one or the other approach. In the following results, let $\mathcal{L}_1$ and $\mathcal{L}_2$ be regular languages generated by regular expressions E_1 and E_2 and recognized by DFA A_1 and A_2.

Theorem 2.2.4. [30, 2] Regular languages are closed under the Boolean operations of union, intersection, and complementation.

Proof.

- For union closure, connecting a new start state s' to the start state s_1 of A_1 and s_2 of A_2 by ϵ-transitions yields an NFA recognizing $\mathcal{L}_1 \cup \mathcal{L}_2$
- Intersection closure is shown by the *product construction.* Taking $\mathcal{Q}_1 \times \mathcal{Q}_2$ as the set of states, $F_1 \times F_2$ as the set of accepting states, and transitioning from $\langle q, p \rangle$ to $\langle q', p' \rangle$ on symbol x if $\delta_1(q, x) = q'$ and $\delta_2(p, x) = p'$ yields a DFA that recognizes $\mathcal{L}_1 \cap \mathcal{L}_2$.
- For complementation closure, reversing the accepting and rejecting states (F_1 and $\mathcal{Q}_1 - F_1$) of A_1 yields a DFA recognizing $\overline{\mathcal{L}_1}$.

[12] The *concatenation* of L_1 and L_2 is the set of strings uv where $u \in L_1$ and $v \in L_2$.

It is easy to give a regular expression for the union: $E_1 + E_2$. For the other two, it is necessary to first construct the automaton and then extract the regular expression. □

Theorem 2.2.5. [30] Regular languages are closed under concatenation.

Proof. Connecting final states of A_1 to the start state of A_2 by ϵ-transitions yields an NFA recognizing $\mathcal{L}_1\mathcal{L}_2$. The corresponding regular expression is E_1E_2. □

A *substitution* s on $\mathcal{L}$ with alphabet Σ is a mapping of each $a \in \Sigma$ to a language $\mathcal{L}_a$. For $w = a_1 \cdots a_n \in \mathcal{L}$, $s(w)$ is the language of the concatenation $s(a_1) \cdots s(a_n)$. Then $s(\mathcal{L})$ is the union of $s(w)$ for all $w \in \mathcal{L}$.

Theorem 2.2.6. [2] Regular languages are closed under regular substitution.[13]

Proof. A proof by regular expression mimics the definition above: for $\mathcal{L}_1$ generated by E_1 with alphabet Σ and a substitution s mapping each $a \in \Sigma$ to a language $\mathcal{L}_a$ generated by E_a, replace each a in E_1 by E_a. This yields a regular expression generating $s(\mathcal{L})$.

The basic idea of the equivalent automaton construction is to replace every a-transition in A by an ϵ-transition to a distinct copy of A_a, and for every final state of A_a add an ϵ-transition to the target of the original a-transition.[14] □

2.2.2 Context-free Languages and Pushdown Automata

Now we ascend the Chomsky hierarchy to context-free languages (CFLs), which are strictly stronger than the class of regular languages (REG⊂CFL). Again we have dual formalisms for generation and recognition mechanisms.

Definition 2.2.7. A pushdown automaton (PDA) M is given by a six-tuple $(\mathcal{Q}, \Sigma, \Gamma, Z_0, \delta, s, F)$ where

- Γ is a stack alphabet
- Z_0 is a special symbol indicating the bottom of the stack
- δ is now a function from $\mathcal{Q} \times \Sigma \times \Gamma$ to $\mathcal{P}(\mathcal{Q} \times \Gamma)$

PDA extend the notion of NFA with a *stack* (the last-in-first-out data structure). The input to δ is not only the current state and input symbol but also the top of the stack, and the output of δ is not only a set of states but a set of pairs containing a state and potentially some manipulation of the stack contents: pushing a new symbol or popping the top symbol. If $\langle q, x, X, Y, p \rangle \in \delta$, meaning

[13] A substitution is regular if the substituted languages are regular.

[14] The interested reader can see Algorithm 4.2.7 of [36] for a complete description (indeed, apparently the only automaton proof for regular substitution in the literature at all). The work is a comprehensive dictionary of proof by automaton.

$\delta(q, x, X)$ contains (p, Y), we may write $(q, xw, X\beta) \vdash (p, w, Y\beta)$, where w is the remainder of the input word, the *instantaneous description* of M describing its step-by-step computatiosn. We write $\overset{*}{\vdash}$ for the transitive closure of $\vdash$.

A PDA may accept a string by final state or by empty stack. For the former we write

$$L(M) = \{w : (s, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \alpha)\}$$

where q is some state in F and α is any string in Γ^*, meaning that starting in s with empty stack and input w, after some number of steps M may end up in q with α on the stack and having read all of w. For the latter we write

$$N(M) = \{w : (s, w, Z_0) \overset{*}{\vdash} (q, \epsilon, \epsilon)\}$$

where q is any state at all. Since PDA allow non-determinism, these two acceptance conditions are equivalent.

PDA may also be represented graphically with nodes and labeled, directed edges specifying the symbol read and any changes made to the stack. An edge from q to p labeled $x, X/Y$ means that in p, if the PDA is reading x and has X on top of the stack, it may transition to p, replacing X by Y (i.e. $\langle q, x, X, Y, p\rangle \in \delta$).

Definition 2.2.8. A context-free grammar (CFG) G is given by the four-tuple (T, V, P, S) where:

- T is a set of terminal symbols
- V is a set of variables
- P is a set of production rules of the form $A \to \alpha$ with $A \in V$ and $\alpha \in (V \cup T)^*$
- S is a start symbol

If there are multiple rules whose left-hand side is A we may combine them into a single rule using "|" equivalent to "+" for regular expressions, e.g.:

$$A \to \alpha, A \to \beta, \ldots \Rightarrow A \to \alpha \mid \beta \mid \cdots$$

We write $\overset{*}{\to}$ for the transitive closure of $\to$ The language of a CFG G is the set of strings of terminals that can be derived from the start symbol:

$$L(G) = \{w \in T^* : S \overset{*}{\to} w\}$$

Theorem 2.2.9. [15] Context-free grammars and pushdown automata are equivalent: for every $PDA\ M$ there is a CFG G such that $L(G) = N(P)$. Thus defining a *context-free language* as the language of some CFG is commensurate with defining it as the language of some PDA.

Theorem 2.2.10. [2] Context-free languages are closed under substitution (with both regular and context-free languages).

Proof. The proof idea is the same as with regular languages: given a context-free language $\mathcal{L}$ with alphabet Σ and a substitution s mapping $a \in \Sigma$ to a context-free language $\mathcal{L}_a$: take the CFG G generating L and replace each a (each terminal in the production rules) with the start symbol S_a of G_a generating $\mathcal{L}_a$. This yields a CFG generating $s(\mathcal{L})$. The equivalent PDA construction is also very similar to the construction for finite automata.[15] □

Theorem 2.2.11. Context-free languages are closed under concatenation.

Proof. Given G_1 generating $\mathcal{L}_1$ and G_2 generating $\mathcal{L}_2$, combining them with a new start symbol S and rule $S \to S_1 S_2$ yields a CFG generating $\mathcal{L}_1\mathcal{L}_2$. □

CFLs are also closed under union (just combine G_1 and G_2 with a rule $S \to S_1 \mid S_2$), but are not closed under the remaining boolean operations of intersection and complementation (thus, sometimes performing these operations with CFLs produces languages in yet stronger classes higher up the hierarchy with *context sensitivity*, which we do not discuss further here). Suppose CFLs were closed under intersection. Let $\mathcal{L}_1 = \{a^n b^n c^m\}$ and $\mathcal{L}_2 = \{a^m b^n c^n\}$. These are both context-free since in each only two numbers must be matched. But their intersection $\mathcal{L} = \{a^n b^n c^n\}$ is canonically non-context-free. Since closure under complementation and union together yield intersection closure by De Morgan's laws, we know the former also cannot hold.

2.2.3 Deterministic Context-free Languages and Deterministic Pushdown Automata

Finally we turn to deterministic context-free languages (DCFLs), a proper subclass of context-free languages. As usual there are production and verification sides to the coin, but they are not entirely equal in this case.

Definition 2.2.12. A deterministic pushdown automaton (DPDA) M is given by a six-tuple $(\mathcal{Q}, \Sigma, \Gamma, Z_0, \delta, s, F)$ where:

- δ is a function from $\mathcal{Q} \times \Sigma \times \Gamma$ to $(\mathcal{Q} \times \Gamma) \cup \{\varnothing\}$ such that the following condition holds for every $q \in \mathcal{Q}$, $a \in \Sigma$, and $x \in \Gamma$:

  exactly one of $\delta(q,a,x), \delta(q,a,\epsilon), \delta(q,\epsilon,x)$ and $\delta(q,\epsilon,\epsilon)$ is non-empty.

This ensures that M always has exactly one move per configuration (is deterministic).

As with nondeterministic PDA, there are two notions of acceptance defining the language of a DPDA–by final state or by empty stack, which we again denote by $L(M)$ and $N(M)$ respectively–and in this case they diverge. If M accepts $\mathcal{L}$ by empty stack, we say $\mathcal{L}$ has the *prefix property*: if $w \in \mathcal{L}$, then there is no v such that $wv \in \mathcal{L}$. In this case, we can construct M' accepting $\mathcal{L}$ by final state

[15] And can again be found (uniquely, as far as we know) in [36].

as follows: M' simulates M on w; if M reads all of w and empties its stack, M' transitions to a final state. To convert in the other direction, we force $\mathcal{L}$ to have the prefix property by adding an endmarker to every string, forming $\mathcal{L}\dashv$. If $\mathcal{L} = L(M)$ for some M, then we can construct M' such that $\mathcal{L}\dashv = N(M')$ since M' can recognize the end of the input string and empty its stack if M would accept the non-endmarked string.

Theorem 2.2.13. [21] DCFLs are closed under complement. For every DPDA M recognizing a language $\mathcal{L}$, there is a DPDA M' recognizing $\neg\mathcal{L}$.

The construction of M' from M is not as easy as complemention for DFA: we cannot simply interchange final and rejecting states. Acceptance is defined as entering a final state after reading the input, but a DPDA may enter both final and non-final states after consuming the last input symbol (by making ϵ-moves); in such a case, inverting final and non-final states still results in acceptance. Briefly, the construction requires identifying the set of states that always consume an input symbol ("reading states"). By restricting the final states to this set, the DPDA can only change its accepting behavior if its actually reading input. Interchanging final and rejecting states within the set of reading states produces the complement automaton.

Deterministic context-free grammars are context-free grammars that have *forced handles*, which will be made precise shortly.[16] DCFGs are still generative devices, but to see what makes them deterministic we must take the reverse perspective and consider their production rules as *reduction* rules. If $u \to v$ is a step in a derivation expanding a variable in u, then we write $v \rightarrowtail u$ and say v *reduces* to u. If $u \rightarrowtail v$ where $u = xhy$ and $v = xTy$, then this reduction step is the reverse of the substitution $T \to h$. We call h the *handle* of u. A grammar has forced handles if and only if every reducing step $u \rightarrowtail v$ is uniquely determined by the prefix of u up to and including its handle.

Example 2.2.14. Consider the following deterministic grammar generating the endmarked language *same number of a's and b's*[17]:

$$\begin{array}{lcl} S & \to & T\dashv \\ T & \to & TaPb \mid TbMa \mid \epsilon \\ P & \to & PaPb \mid \epsilon \\ M & \to & MbMa \mid \epsilon \end{array}$$

and the following step in the reduction of a string in the language:

$$Ta\,\overbrace{PaPb}^{P}b\dashv \rightarrowtail TaPb\dashv$$

[16]Sipser defines a DCFG as a LR(0) grammar. A LR(k) grammar may be parsed from left to right with a *lookahead* of k. We will soon see that every DPDA can be minimally modified such that there is an equivalent DCFG. Since our approach in this thesis in a sense takes automata as primitive, we do not properly introduce the notion of a LR(k) grammar, of which DCFGs are a special case.

[17]The grammar shown is a correction from a list of errata in [44] at `math.mit.edu/~sipser/itoc-derrs3.1.html`.

This step is the reverse of using the production rule $P \to PaPb$ in the derivation of the string. Thus P is a handle of $TaPaPbb$, and is indeed forced as there is no other possible handle.

Now we explain the DK-test, a method to decide whether an arbitrary CFG is deterministic.[18] The automaton DK for a CFG $G = (T, V, P, S)$ is a DFA that simulates matching the handle of its input. For every rule $B \to u_1 u_2 \dots u_k$ there are $k+1$ *dotted rules*, one rule per way of placing a dot in the right-hand side (for example, from $A \to BC$ we obtain $A \to .BC$, $A \to B.C$ and $A \to BC.$). To construct the DK DFA, perform the following steps:

1. Put a dot at the initial point in all rules with S on the left-hand side and place these dotted rules in DK's start state.
2. If there are rules in the state where the dot is immediately followed by a variable C, place dots at the initial points in all rules with C on the left-hand side and add them to the state.
3. Repeat step (2) until no new dotted rules are obtained.
4. For any symbol c (terminal or variable) immediately following a dot, add a c-transition to a state containing dotted rules obtained by shifting the dot to immediately follow c in all the rules where the dot was before c.
5. Repeat steps (2) through (4) for all states until no new states are created.

The final states of DK are those that contain a completed rule (a rule ended by a dot). Note that there may be cycles in DK, i.e. step (4) does not always create a *new* state. The DK-test requires that every final state contains:

- exactly one completed rule,
- no rule where a terminal immediately follows the dot

For example, the DK automaton in Figure 2.5 shows that the grammar in Example 2.2.14 passes the test. Observe that every final state contains exactly one completed rule (having the dot at the end) and no other rule with a terminal symbol (a or b) following a dot.

Theorem 2.2.15. [31] A context-free grammar G is deterministic if and only if it passes the DK-test.

Theorem 2.2.16. [31] An endmarked language is generated by a DCFG if and only if it is deterministic context-free.

The proof of this theorem consists in showing the following two lemmas. We briefly explain the second for later reference.

Lemma 2.2.17. [31] Every DCFG has an equivalent DPDA.

[18] "DK" stands for *Donald Knuth*, who introduces LR(k) grammars in [31] and describes the following test. We present the test for DCFG, but it can be adapted to LR(k) grammars: "The essential reason behind this [is] that *the possible configurations of a tree below its handle may be represented by a regular (finite automaton) language.*"

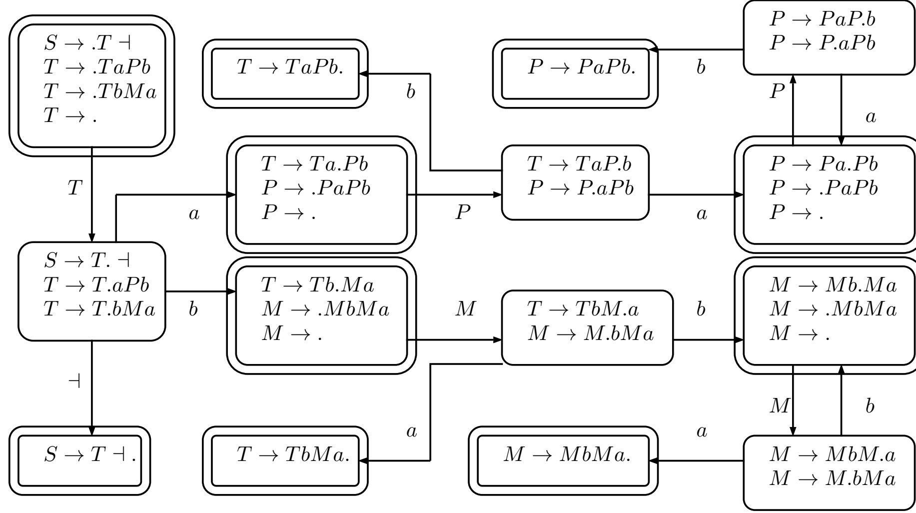


Figure 2.5: DK example for a grammar generating $L = \{w \dashv : \#_a(w) = \#_b(w)\}$

Lemma 2.2.18. [31] Every DPDA recognizing an endmarked language has an equivalent DCFG.

Proof. The DPDA $M = (\mathcal{Q}, \Sigma, \Gamma, Z_0, \delta, q_0, F)$ is altered to empty its stack and enter a new accept state q_{accept} if it would have accepted originally. This is where the endmark assumption comes in: M needs an endmarker to recognize the end of the input and implement this behavior. M is also modified to push or pop a (possibly arbitrary) symbol at every step. The DCFG G with start symbol $A_{q_o q_{\text{accept}}}$ is constructed with the following productions:

1-2. For $p, q, r \in \mathcal{Q}$, $u \in \Gamma$ and $a, b \in \Sigma \cup \{\epsilon\}$, if $\delta(r, a, \epsilon) = (s, u)$ and $\delta(t, b, u) = (q, \epsilon)$, add $A_{pq} \to A_{pr} a A_{st} b$

3 . For $p \in \mathcal{Q}$, add $A_{pp} \to \epsilon$

Each variable A_{pq} derives all and only the strings on which M goes from p with empty stack to q with empty stack. Finally, δ is extended with the variables of G so that M can also read valid strings, and the determinancy of G is proven using the *DK*-test and appeal to the determinancy of M. □

Chapter 3

Survey of Semantic Automata for Monadic Quantifiers

3.1 Models as Strings

Recall from Section 2.1.1 that if a quantifier Q satisfies CONS, EXT, and ISOM, then it has an equivalent representation as a binary relation on natural numbers Q^c such that

$$\mathsf{Q}(A, B) \Leftrightarrow \mathsf{Q}^c(\mid A - B \mid, \mid A \cap B \mid)$$

Since the truth of $\mathsf{Q}(A, B)$ depends only on the cardinalities of A and $A \cap B$, we can record all the information relevant to its evaluation as a string of 0's and 1's with one symbol per element a in A: if a is in $A \cap B$, record a 1, otherwise record 0 (a is in $A - B$). Formally, we can define the following translation function.

Definition 3.1.1. Let $\mathcal{M} = \langle M, A, B \rangle$ be a model, $\vec{a}$ an enumeration of A, and $n = \mid A \mid$. We define $\tau(\vec{a}, B) \in \{0, 1\}^n$ by

$$(\tau(\vec{a}, B))_i = \begin{cases} 0 & a_i \in A - B \\ 1 & a_i \in A \cap B \end{cases}$$

For example, applying τ to the model depcited in Figure 3.1, we take any enumerate $\vec{a}$ of A, record a 1 or 0 according to which case applies, and concatenate the individual digits. Taking the natural enumeration $(a_1, a_2, a_3, a_4, a_5)$ yields the string 00011, but any permutation of this string encodes the same information, namely that 2 $a's$ are B, and 3 are not. Note that b_1 and b_2 are irrelevant, as would be any individuals entirely outside of the sets A and B. That is, τ

encodes all the information relevant to the CE quantifiers we are here interested in.

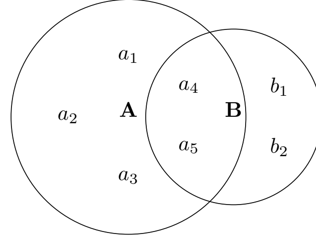


Figure 3.1: Example model to illustrate τ

For a string $s \in \{0,1\}^*$ generated by $\tau(\vec{a}, B)$, let $\#_0(s)$ denote the number of 0's in s and $\#_1(s)$ the number of 1's. Then we have

$$(\#_0(s), \#_1(s)) \in \mathsf{Q}^c) \Leftrightarrow (\mid A - B \mid, \mid A \cap B \mid) \in \mathsf{Q}^c \Leftrightarrow \mathsf{Q}(A, B)$$

Since we have a correspondence between models and strings, we can take the set of strings corresponding to the set of models where Q is true to constitute the *language* of Q:

$$\mathcal{L}_{\mathsf{Q}} = \{s \in \{0,1\}^* \mid (\#_0(s), \#_1(s)) \in \mathsf{Q}^c\}$$

For example:

$$\begin{array}{lcl} \mathcal{L}_{\text{every}} & = & \{s \in \{0,1\}^* : \#_0(s) = 0\} \\ \mathcal{L}_{\text{at least three}} & = & \{s \in \{0,1\}^* : \#_1(s) \geq 3\} \\ \mathcal{L}_{\text{some}} & = & \{s \in \{0,1\}^* : \#_1(s) > 1\} \end{array}$$

It is clear by inspecting our example string $s = 00011$ that $s \notin \mathcal{L}_{\text{every}}$, $s \notin \mathcal{L}_{\text{at least three}}$, but $s \in \mathcal{L}_{\text{some}}$.

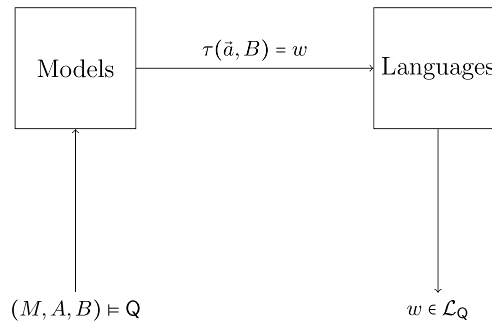


Figure 3.2: Correspondence between models and strings (diagram from [9])

3.2 Semantic Finite Automata

Note that all the results in this section concern quantifiers of type $\langle 1,1\rangle$ (equivalently, type $\langle 1\rangle$), so we often omit the explicit type information.

We will see how semantic finite automata *almost* correspond to first-order (FO) definable quantifiers. Definability in FO logic is captured by Ehrenfeucht-Fraïssé (EF) games.[1] A formula is FO definable on a finite model if there is an n such that no two models that are indistinguishable up to n disagree on the truth of the formula (and this may be demonstrated by players taking turns choosing elements from the respective models). To begin, we formalize this idea of indistinguishability, which means there is a partial isomorphism between the two models. First, for two sets:

$$X \sim_n Y \text{ if either } \mid X \mid = \mid Y \mid = l < n \text{ or } \mid X \mid, \mid Y \mid \geq n$$

Then define $(M, A, B) \sim_n (M', A', B')$ if the relevant sets are $\sim_n$ (e.g. $A - B$ and $A' - B'$, $A \cap B$ and $A' \cap B'$,...). Now we state the general theorem, applied to quantifiers.

Theorem 3.2.1. [4] On finite models, Q is FO definable if and only if for some fixed n:

$$(M, A, B) \sim_n (M', A', B') \text{ implies } \mathsf{Q}(A, B) \Leftrightarrow \mathsf{Q}(A', B')$$

We can interpret this characterization on the Tree of Numbers (recall from Section 2.1.1), from which it will then be clear that the corresponding languages are regular. For a quantifier Q with its representation in the Tree, after some finite upper triangle the pattern of pairs in the extension is completely predictable. This *Fraïssé threshold* occurs at level $2n$, and the pattern has the following properties:

- The point (n, n) determines the value of its downward triangle
- The points $(n - k, n + k)$ determine the values of their leftward diagonals
- The points $(n + k, n - k)$ determine the values of their rightward diagonals

To see why the Fraïssé threshold occurs at $2n$, note the following:

- In every point (l, m) in the downward triangle generated by (n, n), $k, m > n$, so the models corresponding to pairs in the triangle are $\sim_n$ to the model corresponding to (n, n).
- In every point (l, m) in the leftward diagonal generated by a point $(n - k, n + k)$, $k = n$ and $m > n$, so the models corresponding to pairs in the diagonal are $\sim_n$ to the model corresponding to $(n - k, n + k)$. (And clearly the same holds for $(n + k, n - k)$ to the right).

[1] See, e.g., [24] for further explanation EF games.

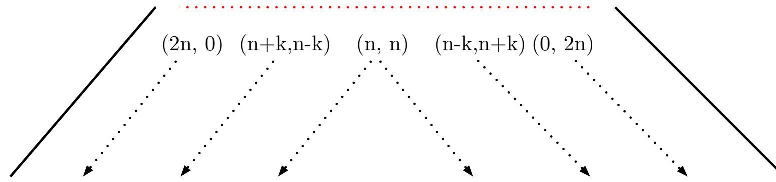


Figure 3.3: Fraïssé Threshold at level $2n$

Theorem 3.2.2. [4] The FO definable quantifiers are precisely those which can be recognized by permutation-invariant acyclic finite state machines.

To see that this holds, it suffices to observe that the Tree patterns corresponding to the languages of the automata are themselves regular. Since the upper triangle possibly containing exceptions to the threshold pattern is finite, it is regular. For the pattern below the threshold, note that a triangle under the point (i, j) is described by *at least i 0's and at least j 1's*, and a leftward diagonal band under (i, j) is described by *at least i 0's and exactly j 1's* (similarly for right diagonal bands). These are all given by regular expressions[2] (see Appendix A for examples).

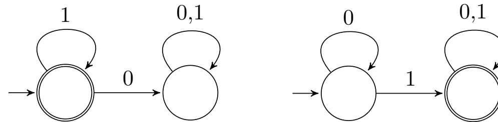


Figure 3.4: every and some

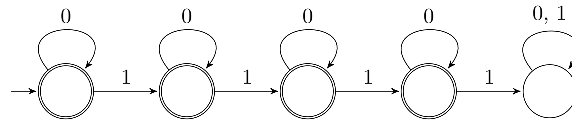


Figure 3.5: at most three

Acyclic finite automata accept only a proper subset of the regular quantifier languages. Define $D_n x\varphi$ if and only if the number of x satisfying φ is divisible by n. These formulae define the "counting modulo" quantifiers: the familiar *even* and *odd*, but also *divisible by three*, *divisible by seven*, etc. FO(D_ω) is first-order logic extended with D_n for all $n \geq 2$. Mostowski showed that the class of

[2]Indeed, the Tree itself can be transformed into an automata recognizing the quantifier whose pattern it holds. Take the nodes as the points of the upper triangle with 0-transitions from every (i, j) to $(i+1, j)$ and 1-transtions from (i, j) to $(i, j+1)$. For points on the threshold and to the left of (n, n), let them additionally loop on 0 and go to the right on 1 (opposite for points to the left of (n, n)), and let (n, n) loop on both symbols [4].

regular $\langle 1, 1\rangle$ quantifiers are exactly those definable in FO logic augmented with divisiblity quantifiers.

Theorem 3.2.3. [38] Finite automata accept the class of all monadic quantifiers definable in FO(D_ω).

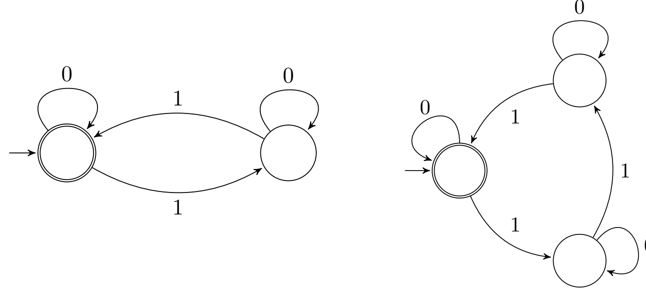


Figure 3.6: Parity quantifiers: even and divisible by 3

We can also use the notion of EF games to demonstrate that even is not FO definable. This requires showing that for every n, there are $(M, A, B) \sim_n (M', A', B')$ where $\mathsf{even}(A, B)$ but $\neg\mathsf{even}(A', B')$. But this is very easy: take two sufficiently large models such that $\mid A \cap B \mid = m \geq n$ and $\mid A' \cap B' \mid = m + 1$. Then they are $\sim_n$, but even must hold in one and not the other.

3.3 Semantic Pushdown Automata

To state results concerning PDA, we need to define a bit of terminology. Say a subset of $\mathbb{N}^n$ is a *linear set* if it is of the form

$$L(c; \{v_1, \dots, v_r\}) = \{c + k_1 v_1 + \cdots k_r v_r : k_i \in \mathbb{N}, 1 \leq i \leq r\}$$

where $c, v_i \in \mathbb{N}^n$. The vector c is the *offset*, and the v_i are *generators*. A *semilinear* set is a finite union of linear sets. *Presburger arithmetic* is the FO theory of the natural numbers with addition only (not multiplication). Every Presburger formula defines a set. If a set is Presburger-definable, we equivalently say it is FO additively definable. The FO additively definable sets are exactly the semilinear sets [20].

The *Parikh image* of a language $\mathcal{L}$ with alphabet $\Sigma = \{a_1, \dots, a_n\}$, denoted $\psi(\mathcal{L})$, is the set of vectors:

$$\{(\#_{a_1}(w), \dots, \#_{a_n}(w)) : w \in \mathcal{L}\}$$

For example, the Parikh image of the language $\mathcal{L}_1 = \{0^n 1^n : n \geq 0\}$ is the set containing $(0, 0), (1, 1), (2, 2), \dots$, i.e. the semilinear set $\psi(\mathcal{L}_1) = L((0, 0); \{(1, 1)\})$. The language $\mathcal{L}_2 = \{w \in \{0, 1\}^* : \#_0(w) = \#_1(w)\}$ has the same Parikh image as $\mathcal{L}_1$. We say the two are *Parikh equivalent*.

Theorem 3.3.1. [40] For every context-free language $\mathcal{L}$, $\psi(\mathcal{L})$ is semilinear.

Theorem 3.3.2. [4] Every PDA-computable quantifier is FO additively definable.

Given a quantifier Q that is PDA-computable, we know its language $\mathcal{L}_{\mathsf{Q}}$ is context-free. Thus the Parikh image of $\mathcal{L}_{\mathsf{Q}}$ is semilinear, i.e. FO additively definable. Parikh's theorem does not convert in general: consider the semilinear set $L((0,0,0);\{(1,1,1)\})$, which is the Parikh image of (for example) the non-context-free language $\{w \in \{0,1,2\}^* : \#_0(w) = \#_1(w) = \#_2(w)\}$. However, restricting to a binary alphabet (that is, type $\langle 1,1\rangle$ quantifiers), we have also the following result in the other direction:

Theorem 3.3.3. [4] Every FO additively definable binary quantifier is PDA-computable.

It follows from this theorem that type $\langle 1,1\rangle$ context-free languages are closed under complement, union, and intersection. If $\mathcal{L}_1$ and $\mathcal{L}_2$ are such languages, then they are FO additively definable. Since semilinear sets are closed under these operations, there are formulas defining $\neg\mathcal{L}_1$, $\mathcal{L}_1 \cup \mathcal{L}_2$, and $\mathcal{L}_1 \cap \mathcal{L}_2$, i.e. they are also context-free.

In [38], Mostowski gives a characterization of quantifiers accepted by DPDA by empty stack; however, he defines somewhat idiosyncratic acceptance conditions that do not align with the standard notion of empty stack acceptance (see Section 2.2.2). He states that the DPDA M accepts a string if the computation ends in a final state, and if *additionally* the stack is emtpy at the end, that M accepts the string by empty stack. This has the odd consequence that regular quantifiers are accepted by empty stack, entailing they have the prefix property, which is not in general the case (see Section 2.2.3). Furthermore, *no* monadic quantifier language has the prefix property (for any w in some $\mathcal{L}_{\mathsf{Q}}$, there is always an extension of w that is also in $\mathcal{L}_{\mathsf{Q}}$). It appears that the intended notion should allow the DPDA to transition from a configuration with empty stack (so the language needn't have the prefix property). Note that in the following results, we refer to this alternative meaning of empty stack acceptance. This meaning does indeed correspond to an interesting subclass of the DPDA-computable quantifiers.

There is an intuitive way to think of these quantifiers Q that are computable by DPDA that happen to have an empty stack when accepting $w \in \mathcal{L}_{\mathsf{Q}}$, but may also lead to empty stack configurations at some point mid-computation. The strings w in their languages have the following property:

$$w = uv \in \mathcal{L}_{\mathsf{Q}} \text{ and } u \in \mathcal{L}_{\mathsf{Q}} \Rightarrow v \in \mathcal{L}_{\mathsf{Q}}$$

For example, consider the language of exactly half and the string $w = 1011000011$. We can write $w = uv$ where $u = 101100$ and $v = 0011$ are both in the language; but also *any* v' belonging to Q following u would be in the language.

To prove his result, Mostowski defines an "almost-linear" set. First, for $a, b, c, d \in \mathbb{N}$ we define an $(a, b, c, d) - \textit{linear}$ quantifier $\mathsf{Lin}_{(a,b,c,d)} = \mathsf{Q}_R$ where $R = \{(x, y) : \exists z \in \mathbb{N} \text{ s.t. } (x, y) = (c, d) + z(a, b)\}$. Then Q is *almost-linear* if

$$\mathsf{Q} = \mathsf{Q}_{R_0} \cup \mathsf{Lin}_{(a,b,c_1,d_1)} \cup \cdots \mathsf{Lin}_{(a,b,c_k,d_k)}$$

where R_0 is a finite relation. That is, the set defining Q is a finite union of linear sets all with at most one, identical, non-trivial generator (a, b)–modulo a finite number of exceptions (R_0). The almost-linear quantifiers are the proportional quantifiers exactly b/(a+b) (modulo the initial offset from (c_i, d_i)).

Theorem 3.3.4. [38] The class of quantifiers recognized by DPDA by final state and empty stack is the union of:

- regular quantifiers
- intersections of regular and almost-linear quantifiers

Mostowski also states the following facts about almost-linear sets:

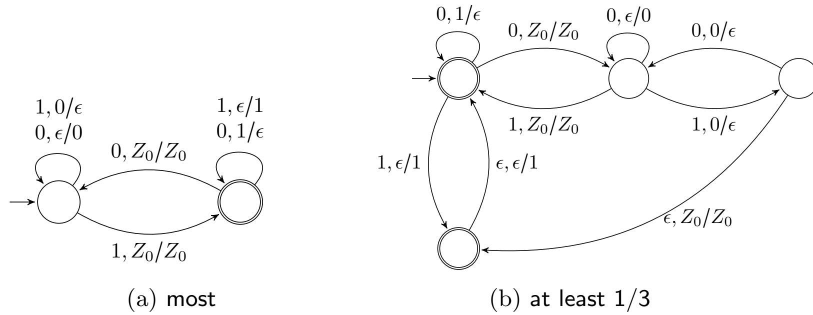
- The complement of an almost-linear set is never almost-linear.
- Finite unions of almost-linear sets are almost-linear only if they have the same ratio (generating vector).
- Intersection of almost-linear sets are either almost-linear or finite.

Thus the subclass of deterministic context-free quantifiers recognized by DPDA by this modified notion of empty stack are not in general closed under any Boolean operations.

Very recently, Kanazawa [27] answered the open question of how to characterize quantifiers recognized by DPDA by arbitrary stack.

Theorem 3.3.5. [27] Q is recognized by DPDA if and only if there are $k, l, m, n \in \mathbb{N}$ such that $\psi(\mathcal{L}_\mathsf{Q})$ is a finite union of linear sets each of which has one of the following as its set of generators:

$$\varnothing,\ \{(k, 0)\},\ \{(0, l)\},\ \{(m, n)\},\ \{(k, 0), (m, n)\}$$



(a) most (b) at least 1/3

Figure 3.7: Semantic DPDA examples [26]

The class of quantifiers recognzied by DPDA by arbitrary stack expands the class of proportional quantifiers defined by Mostowski with *at least* and *at most* as well as some amount of Boolean combinations (but not arbitrary combinations). Kanazawa provides examples showing that this class ranges from the mundane to the complicated

at least 2/3	:	$L((0,3);\{(0,1),(1,2)\})$
3 more than twice as many or less than twice as many	:	$L((0,3);\{(1,2)\})$ $\cup L((1,0);\{(1,0),(1,2)\})$ $\cup L((1,1);\{(1,0),(1,2)\})$

However, the following is not DPDA-recognizable:

more than 1/3 and less than 2/3	:	$L((1,1);\{(2,1),(1,2)\})$ $\cup L((2,2);\{(2,1),(1,2)\})$

because the generators are of the form $\{(m,n),(k,l)\}$. Kanazawa also gives the analogous characterization for regular quantifiers.

Proposition 3.3.6. [27] Q is recognized by finite automaton if and only if there are $k,l \in \mathbb{N}$ such that $\psi(\mathcal{L}_{\mathsf{Q}})$ is a finite union of linear sets each of which has one of the following as its set of generators:

$$\varnothing,\ \{(k,0)\},\ \{(0,l)\},\ \{(k,0),(0,l)\}$$

The fact that none of the generating vectors can be of the form (m,n) (one of the components is always a 0) corresponds to the fact that $\mathsf{Q}^c xy$ (the binary relation on $\mathbb{N}$) for a regular quantifier Q is always defined by either a constraint on one of x or y (exclusively) or independent constraints on both.[3] Regular quantifiers restrict the cardinality of one of their arguments; context-free quantifiers compare the cardinalities of their arguments.

Lastly, Kanazawa shows the following:

Theorem 3.3.7. [27] NPDA-computable quantifiers are finite Boolean combinations of DPDA-computable quantifiers.

Thus we have a formal statement of our earlier claim that natural language quantification is in some way essentially deterministic. Every determiner requiring non-determinism is in fact decomposable into simpler deterministic determiners.

[3]E.g. the independent constraints $x > 2$ *and* $y = 5$ defines a relation corresponding to a regular quantifier, while the dependent constraint $x = 3y$ defines a context-free relation.

Chapter 4

A Summary and Reformulation of Known Results for Iteration Automata

Shane Steinert-Threlkeld and Thomas Icard III made the first foray into semantic automata for polyadic quantifiers with their paper "Iterating Semantic Automata" in 2013 [46]. They show both that regular quantifier languages (and thus the equivalent DFA) and context-free quantifier languages (and thus the equivalent PDA) are closed under the iteration operation; however, their constructions of the corresponding iterated automata are disconnected from these results. These closure results of course indicate that for any two DFA-recognizable quantifiers, there exists a DFA recognizing their iteration, and likewise for two PDA-recognizable quantifiers. Their construction generates automata that have one more stack than necessary (a PDA for iterations of two DFA and a two-stack PDA when one or more PDA is involved in the iteration). Though these superfluous stacks are not used to their full potential, the models are nonetheless too powerful. The authors are of course aware of this fact, but unaware of the equally easy and algorithmic constructions for the minimally-necessary powerful automata.

First we present a convention, established in [46], for translating models with binary relations into strings. Then we reprove the results that regular and context-free languages are closed under quantifier iteration in a cleaner fashion that makes explicit the intuitions grounding their proofs. Finally, we summarize their automata construction for later reference.

4.1 Translating Models with Binary Relations into Strings

In order to talk about the language accepted by an automaton for an iterated quantifier, we need a way of translating models with *relations* into strings. For now we assume the relation is binary and later generalize the translation and construction in Section 5.2. The idea is simple: given a binary relation R with domain A and range B, look in turn at every element a of A and record, for each element b of B, whether or not a is in the relation R with b. To keep the substrings generated by each a distinguishable, we introduce a new separator symbol $\boxdot$.

Example 4.1.1. A quick example will make the definition to follow more intuitive. Figure 4.1 depicts a model with sets A and B and a relation R between the two. This could represent, for example, a set of aunts, a set of books, and the information regarding which aunts read which books. To translate this model into a string, we look at the elements of A in some order (the indices yield a natural enumeration) and examine which elements of B they connect to: a_1 R's every element of B, so we write $111\boxdot$; a_2 R's only the first element, so we write $100\boxdot$; a_3 R's the last two elements, so we write $011\boxdot$. Concatenating these three substrings yields $111 \boxdot 100 \boxdot 011\boxdot$, which is the string representation of the model.

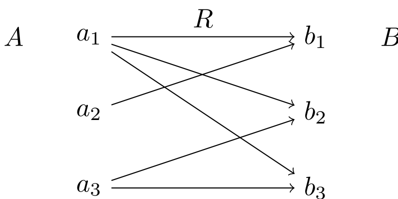


Figure 4.1: Example for Definition 4.1.2

Definition 4.1.2. Let $\mathcal{M} = \langle M, A, B, R\rangle$ be a model, $\vec{a}$ and $\vec{b}$ enumerations of A and B, and let $n =\mid A\mid$. Define a new translation function τ_2 which takes two sets and a binary relation as arguments:

$$\tau_2(\vec{a}, \vec{b}, R) = (\tau(\vec{b}, R_{a_i})\boxdot)_{i\leq n}$$

where $R_{a_i} = \{b \in B : (a_i, b) \in R\}$ is the set of b in B in the relation R with a_i. That is, for each a_i, τ computes a substring with a separator symbol $\boxdot$ appended to the end, recording a 1 if b_j is in R_{a_i} and a 0 otherwise. The final string is the concatenation of all these substrings.

Now languages of iterated type $\langle 1, 1, 2\rangle$ quantifiers are straightforward extensions of languages in the monadic type $\langle 1, 1\rangle$ case. Recall that a quantifier Q_1 is equivalently a binary relation Q_1^c between the number of 1's and 0's in the strings of its language. For quantifiers of the form $\mathsf{Q}_1 \cdot \mathsf{Q}_2$, we let subwords

(sequences of 1's and 0's separated by ⧈'s) in the language of Q_2 replace 1's and subwords in the complement of the language of Q_2 replace 0's as the units upon which Q_1^c is defined. Whether or not a subword is in the language of Q_2 is just an instance of the simple monadic case.

To see that this is the correct intuition, recall the definition of binary iterations:

$$(\mathsf{Q}_1 \cdot \mathsf{Q}_2)(A, B, R) \Leftrightarrow \mathsf{Q}_1(A, \{a : \mathsf{Q}_2(B, R_a)\})$$

If we denote the set $\{a : \mathsf{Q}_2(B, R_a)\}$ by X, then a single a (yielding a 1 by τ) is in $A \cap X$ if and only if Q_2-many b are in $B \cap R_a$ (yielding a string in the language of Q_2 by τ). Thus we also have that a is in $A - X$ (yielding a 0) if and only if it's not the case that Q_2-many b are in $B \cap R_a$ (yielding a string not in the language of Q_2–equivalently, a string in the complement). Since they are equivalent, we write $w \notin \mathcal{L}_{\mathsf{Q}_2}$ and $w \in \mathcal{L}_{\neg\mathsf{Q}_2}$ interchangably throughout.

Definition 4.1.3. Let Q_1 and Q_2 be quantifiers of type $\langle 1, 1\rangle$. We define the language of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ by

$$\begin{aligned} \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2} \quad = \quad & \{w \in (w_i ⧈)^* : w_i \in \{0,1\}^* \text{ and} \\ & (|\{w_i : w_i \notin \mathcal{L}_{Q_2}\}|, |\{w_i : w_i \in \mathcal{L}_{\mathsf{Q}_2}\}|) \in {\mathsf{Q}_1}^c\}. \end{aligned}$$

Example 4.1.4. The language of the iterated quantifier $\mathsf{some} \cdot \mathsf{every}$ still ultimately reduces to a numerical constraint on the number of 1's and 0's in strings of the language:

$$\begin{aligned} s \in \mathcal{L}_{\mathsf{some}\cdot\mathsf{every}} \quad & \Leftrightarrow \quad (|\{w_i : w_i \notin \mathcal{L}_{\mathsf{every}}\}|, |\{w_i : w_i \in \mathcal{L}_{\mathsf{every}}\}|) \in \mathsf{some}^c \\ & \Leftrightarrow \quad |\{w_i : w_i \in \mathcal{L}_{\mathsf{every}}\}| > 0 \\ & \Leftrightarrow \quad |\{w_i : (\#_0(w_i), \#_1(w_i)) \in \mathsf{every}^c\}| > 0 \\ & \Leftrightarrow \quad |\{w_i : \#_0(w_i) = 0\}| > 0 \end{aligned}$$

By a similar derivation we get:

$$s \in \mathcal{L}_{\mathsf{every}\cdot\mathsf{some}} \Leftrightarrow |\{w_i : \#_1(w_i) = 0\}| = 0$$

The string from Example 4.1.1, 111 ⧈ 100 ⧈ 011⧈, is a member of both these languages, indicating that the sentences *Every A R some B* and *Some A R every B* are both true in the model depicted by Figure 4.1.

In the following sections we often speak of *words in the language of* Q_2 without explicitly stating whether we mean words in $\{0, 1\}$ or words ending in ⧈. When considering these strings as input for iteration automata, we will make reference to the *well-formedness* of a string. Call a string *well-formed* if it ends in ⧈. The reader may wonder why we don't define well-formedness in terms of individual subwords. To explain this, we must point out that the language accepted by an iteration automaton is in a sense bigger than the number of relations whose translation it accepts. Observe:

- For a model $\mathcal{M} = (M, A, B, R)$ with $n = |A|$ and $m = |B|$, τ_2 generates strings of the form $((1 + 0)^m ⧈)^n$.

- $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ contains strings of the form $((1+0)^* \boxdot)^*$. This of course includes every string with the appropriate constraints on 1's and 0's for fixed n and m, but also every other string with the same restrictions but with subwords of varying length (even of length zero).

In defining iteration automata, there is no way to enforce a rule that every $(m+1)^{st}$ symbol should be a $\boxdot$ (at least, not without superfluous computing power). Ultimately a subword is a subword *because* it ends in $\boxdot$, and the only way to ensure that a string consists of well-formed subwords is to require that it end in $\boxdot$.

Question 4.1.5. How much depends on the translation function? This definition of τ_2 is the natural extension of τ, but the above-mentioned mismatch between models with binary relations and iterated languages already suggests a potential problem for evaluating polyadic lifts that depend on more than the number of 1's and 0's in independent subwords. How complex are some lifts given τ_2, and how much less complex do they become when we add more discriminatory power to τ_2? We revisit these and related questions when we discuss cumulation automata and the Frege boundary.

4.2 Explicit Proofs of Some Closure Results

Steinert-Threlkeld and Icard III argue the closure of regular and context-free languages under quantifier iteration via arguments from regular expressions and context-free grammars, respectively. Their intuitive justification does not outright mention the general closure of regular and context-free languages under substitution; however, it is informative to deliberately state the fact that quantifier iteration *just is* an instance of substitution, from which these closure results follow straightforwardly. In our reformulations of their proofs, we define substitutions directly on languages.

Theorem 4.2.1. Let $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_2}$ be languages of type $\langle 1,1 \rangle$ regular quantifiers with alphabets $\Sigma_1 = \Sigma_2 = \{0,1\}$. $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ is a regular language.

Proof. Define a substitution s on $\mathcal{L}_{\mathsf{Q}_1}$ by the following:

- $s(0) = \mathcal{L}_{\neg \mathsf{Q}_2} \boxdot$
- $s(1) = \mathcal{L}_{\mathsf{Q}_2} \boxdot$

Claim: $s(\mathcal{L}_{\mathsf{Q}_1}) = \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$

Proof: This is immediately clear from the substitution. For $w = (w_i \boxdot)^*$, $w \in \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ if and only if $(|\, \{w_i : w_i \in \mathcal{L}_{\neg \mathsf{Q}_2}\}\,|, |\, \{w_i : w_i \in \mathcal{L}_{\mathsf{Q}_2}\}\,|) \in \mathsf{Q}_1^c$, if and only if $w = s(w')$ where $(\#_0(w'), \#_1(w')) \in \mathsf{Q}_1^c$, if and only if $w \in s(\mathcal{L}_{\mathsf{Q}_1})$. ∎

Thus s is the appropriate substitution. Since regular languages are closed under complement (Theorem 2.2.4), $\mathcal{L}_{\neg \mathsf{Q}_2}$ is regular, and since regular languages are

closed under concatenation (Theorem 2.2.5), $\mathcal{L}_{(\neg)\mathsf{Q}_2}\boxdot$ is regular. Thus s defines a regular substitution, so by regular substitution closure (Theorem 2.2.6), $s(\mathcal{L}_{\mathsf{Q}_1}) = \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ is a regular language. □

Theorem 4.2.2. Let $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_2}$ be languages of type $\langle 1, 1 \rangle$ context-free quantifiers with alphabets $\Sigma_1 = \Sigma_2 = \{0, 1\}$. $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ is a context-free language.

Proof. We use the same substitution s on $\mathcal{L}_{\mathsf{Q}_1}$:

- $s(0) = \mathcal{L}_{\neg\mathsf{Q}_2}\boxdot$
- $s(1) = \mathcal{L}_{\mathsf{Q}_2}\boxdot$

Claim: $s(\mathcal{L}_{\mathsf{Q}_1}) = \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$

Proof: The argument in Claim 4.2 holds here. ■

Since context-free *quantifier* languages are closed under complement (Theorem 3.3.3), $\mathcal{L}_{\neg\mathsf{Q}_2}$ is context-free, and since context-free languages are closed under concatenation (Theorem 2.2.11), $\mathcal{L}_{(\neg)\mathsf{Q}_2}$ is context-free. Thus s defines a context-free substitution, so by context-free substitution closure (Theorem 2.2.10), $s(\mathcal{L}_{\mathsf{Q}_1}) = \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ is a context-free language.

□

4.3 Summary of Work by Steinert-Threlkeld and Icard III

There is great theoretical interest in identifying the least-powerful automata recognizing iterated quantifiers. The duality between formal languages and automata is what makes the field useful and interesting in the first place. However, automata are also interesting in that they often present an intuitive algorithm for string membership, and this is one reason for investigating semantic automata in particular. The automata defined in [46], henceforth referred to as "stack" versions and denoted by, for example, $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)_{\mathsf{stack}}$, yield genuinely different algorithms than the minimal[1] constructions we give in later sections. Since we anticipate that the stack and minimal versions of iteration automata likely share explanatory power with respect to describing the algorithms human beings may *actually* perform in model-checking tasks[2], it is useful to discuss this work here for reference in the practical discussion later on.

Now we give a high-level description of the stack construction for iteration automata. The construction takes two DFA, Q_1 and Q_2, as input (we assume this for simplicity; [46] also only gestures at how to extend the formal definitions

[1] We must qualify the claim that the constructions given are minimal, since there is no such notion for PDA.

[2] See for example [51] for a recent empirical comparison of the two and Section 9.1.1 for a discussion.

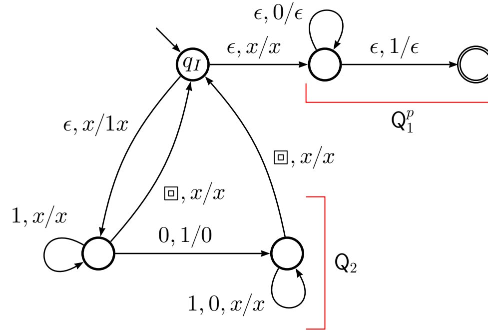


Figure 4.2: $(\mathsf{some} \cdot \mathsf{every})_{\mathsf{stack}}$

for iterations involving one or more PDA). Q_2 runs on every subword of the input. If the computation ends in a final state, a 1 is on the stack, and otherwise a 0 is on the stack. This is achieved by initially pushing a 1 (0) if the start state of Q_2 is an accepting (rejecting) state, and swapping the 1 for 0 (0 for 1) whenever the automaton transitions from an accepting to a rejecting state (vice versa). After reading the entire string, the automaton has recorded on its stack how many subwords were and were not in the language of Q_2. Recalling the definition of iterated quantifier languages, if this sequence of 1's and 0's is in the language of Q_1, then the original string is in the language of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$. In order to "read" the stack, the automaton transitions to a modified *pushdown reader* Q_1^P, which behaves exactly as Q_1 by popping 1's and 0's from the stack as if it were reading them from an input string. The iteration automaton $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)_{\mathsf{stack}}$ accepts if and only if the pushdown reader is in a final state once the stack has been "read" (emptied).

Let's look at an example (see Figure 4.2). Recall again the sample string 111 $\boxdot$ 100 $\boxdot$ 011$\boxdot$. Given this string, $(\mathsf{some} \cdot \mathsf{every})_{\mathsf{stack}}$ first processes 111$\boxdot$ by non-deterministically transitioning from q_I to the start state of every, pushing a 1 onto the stack since this is also an accepting state. Since every loops in this final state on 111, the automaton transitions back to q_I on $\boxdot$ with a 1 on its stack. Now every processes 100. A 1 is again pushed to the stack, but replaced by a 0 since every transitions to and remains in a rejecting state on this string, so the stack contents are [01] when the automaton transitions back to q_I. The same happens on the final subword, resulting in the automaton being in q_I with stack contents [001] after seeing the whole string. Now the automaton non-deterministically transitions to the pushdown reader some^P, which loops in the first state while popping the first two 0's, but transitions to a final state popping the 1. Thus the original string is accepted.

The non-determinism in this strategy involved turns out to be unavoidable given the chosen string representation. A PDA is deterministic if every configuration allows transition to at most one configuration. As it stands, the definition of

iteration automaton in [46] has two ϵ-transitions, both ignoring the stack, from q_I. We could rectify this by letting the automaton consume the first symbol of a subword when transitioning to Q_2 from q_I and adjust δ accordingly, but their would still be a choice between ϵ and non-ϵ input in q_I. The issue can be completely avoided by both performing the just-mentioned amendment *and* adding another symbol to the alphabet marking the end of the entire string (which is not so terrible).

Fact 4.3.1. [46] $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)_{\mathsf{stack}}$ has $1+ \mid \mathcal{Q}_1 \mid + \mid \mathcal{Q}_2 \mid$ states.

This is easy to see, as the construction always consists in taking exactly one copy each of Q_1 and Q_2 and linking them together with a new start state.

Chapter 5

Extending Iteration Automata

In this chapter we present our original constructions for the iteration of deterministic finite automata. The first section handles automata for type $\langle 1, 1, 2 \rangle$ iterated regular quantifiers, starting with some intuitive motivation and explanation for the definition we choose, then giving the exact construction with some examples, and finally proving its correctness. The next section generalizes this construction to type $\langle 1, 1, \ldots, n \rangle$ iterated regular quantifiers. We establish new vocabulary for translating n-ary relations and for the languages of such quantifiers, give the construction with examples, and prove its correctness; all of this extends naturally from the two-quantifier case. Lastly, we sketch how to generalize the stack construction in [46] for regular iterated quantifiers.

5.1 Automata for Type $\langle 1, 1, 2 \rangle$ Regular Iterations

5.1.1 The Construction

The juxtaposition of the proof given in [46] for the closure of regular languages under quantifier iteration with the construction they ultimately give was largely the inspiration for the algorithm we present in this section. We repeat their definition of iterated regular expressions now:

Definition 5.1.1. Let E_1 and E_2 be regular expressions in $\{0, 1\}$. Define the iterated regular expression $It(E_1, E_2)$ by $E_1[0/(E_2^c \boxdot),\ 1/(E_2 \boxdot)]$, where E^c denotes a regular expression for the complement of the language generated by E.

As we have previously seen, regular expressions and finite state automata generate exactly the same class of languages. The standard proof of one direction of this result offers a method of generating the minimal DFA from the iterated regular expression, namely constructing a non-deterministic finite automata from the expression and then going through the subset construction. However, we want to give a construction based on the properties of the two iterated *automata*, similarly to Steinert-Threlkeld and Icard III. In [46], it is very clear how $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)_{\mathsf{stack}}$ is built from Q_1 and Q_2 themselves. Since we are interested in semantic automata as potential models of real algorithms being used by people, we should take the simple automata for monadic quantifiers as primitive units of reasoning rather than regular expressions. That is, we should expect that people are somehow piecing together two *algorithms* rather than directly using the corresponding regular expressions. However, toward the goal of reflecting the compositionality of language (see Section 1.2.4), we should also take the substitutional structure of iterated regular expressions as indicative of how the meaning of iterated automata should be built from the meaning of their component automata.

It is the existence of a construction directly from minimal DFA for monadic quantifiers to minimal DFA for iterated quantifiers (with no intermediate translation through regular expressions and NFA) that Steinert-Threlkeld and Icard III doubted when saying, "On the one hand, the PDA construction [stack version] provides a general method for generating a machine for the iteration of any two quantifiers. There appears to be no such analogously general mechanism for generating minimal DFAs." In this section we show there is such a general mechanism, which is easy to use and depicts iterations in a much more compositional manner.

The definitions of iterated regular expressions and languages of iterated quantifiers already suggest how to go about constructing iterated automata from the monadic building blocks. For the regular expression $It(E_1, E_2)$, just replace 1's in the first by an endmarked expression for the second, and 0's in the first by an endmarked expression for the complement of the second. For languages, just replace 1's in the first by entire words in the language of the second, and 0's by entire words in the complement of the language of the second. To complete the trinity, we must ask ourselves, *what is the analogous notion in terms of automata*? Quite simply, *1-transitions* of the first automaton should be replaced by *accepting runs* of the second automaton and *0-transitions* replaced by *rejecting runs*.[1]

Nonetheless, the computation performed by our iteration automata is still similar to that performed by the stack version. Every subword is still processed by Q_2, but instead of "remembering" the outcome of all these subprocesses and

[1] Actually, the precise details of the construction to follow were extrapolated from the results of automatically generating many different minimal iteration DFA using a program called HaLeX, a Haskell implementation of regular expressions, finite automata, and common related algorithms. See Appendix A for an overview of our use of HaLeX.

finally running Q_1 on the result, our automata involve a "real-time" update of Q_1 after each subrun of Q_2. The main idea is to start with Q_1 as the backbone of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ and then replace each of its states with a copy of Q_2. To make things easier, imagine these copies are indexed by the state they replace. From here on we refer to such copies by Q_2^q and refer to their components in the obvious way (e.g. $\mathcal{Q}_2^q, s_2^q, \delta_2^q, F_2^q$). If some copy Q_2^q ends in a final state seeing some subword, then the machine should behave as if Q_1 had seen a 1. Suppose q would transition to p on a 1. This means that every final state of Q_2^q should transition to the start state of Q_2^p on ⧈ (as this marks the end of the subword). Similarly, every rejecting state of Q_2^q should have a ⧈-transition to Q_2^r, where r is the state that q would transition to on a 0. $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ has the same start and final states as Q_1.

Before giving the formal definition, let us dig a little deeper with an example. Consider the automaton exactly three · every depicted in Figure 5.2. The vestiges of the original state-space of exactly three is clearly visible as the "spine" of the automaton, enclosed in the darker dashed box, but the original states have been replaced by copies of every, enclosed in the ligher dashed boxes. There are three exceptions to this simple replacement scheme:

(i) Final states: Notice that the final state of exactly three remains externally linked up with a copy of every. This is so that the automaton cannot erroneously accept a word ending in 111, for example, which is not well-formed.

(ii) Terminal states: Notice that the terminal state of exactly three doesn't seem to have been replaced at all. If the automaton reaches state q_5, the rest of the input is irrelevant. The automaton can *only* reject at this point, hence the looping on every symbol.

(iii) Final, terminal states: The current example does not exhibit this exception, but when an automaton reaches a state that is both final and terminal, it should accept irrespective of the remaining input *so long as* it is well-formed. Such states q require at most one extra state p to go to in which to loop on 0 and 1 and then return to q on ⧈. If q has a predecessor state r with equivalent behavior, then the extra state is unnecessary (as q may just go to r on 0 and 1).

As the state-space is given by the replacement scheme, and the 1 and 0-transitions are given by the copies of Q_2, all that remains is to specify the ⧈-transitions, which are determined by the 1 and 0-transitions in the original Q_1. Consider the states q_1 and q_1', together comprising Q_2^q, replacing q in exactly three. Since q_1 is an accepting state in every, a subrun ending there is analogous to a 1. Thus the ⧈-transition from q_1 should mimic the 1-transition of q to q', going to the start state of the copy of Q_2 replacing q'. Similarly, q_1' should mimic the 0-behavior of q, which is to loop, meaning q_1' should return to the start state of the Q_2 copy of which it is a member.

The final state q_4 is not itself a member of any copy of Q_2, but its ⧈ transitions

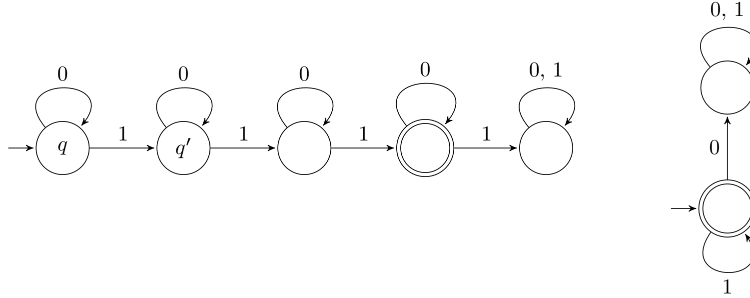


Figure 5.1: exactly three and every

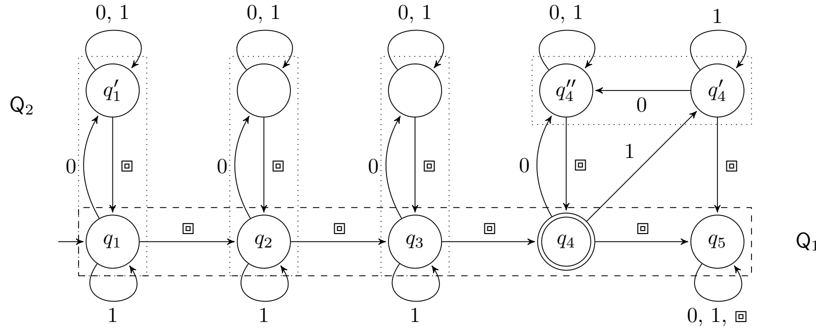


Figure 5.2: exactly three·every

are still decided by the start state of its associated copy of Q_2. If ⧈ is seen in such a state, this means the current subword was empty; this works because ϵ is in the language of Q_2 if and only if s_2 is final, so s_2 appropriately determines ⧈-behavior.

Definition 5.1.2. Previously we likened the state space of Q_1 to the spine or essential structure of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$. Here we make this idea precise by describing a mapping between $\mathcal{Q}_1$ and a subspace of $\mathcal{Q}$ (the state-space of the iteration automaton). Define a bijection $f : \mathcal{Q}_1 \to \mathcal{Q}$ by the following:

$$f(q) = \begin{cases} q_F & q \in F_1, \text{ non-terminal} \\ s_2^q & q \notin F_1, \text{ non-terminal} \\ q_{FT} & q \in F_1, \text{ terminal} \\ q_T & q \notin F_1, \text{ terminal} \end{cases}$$

For example, if a state in $\mathcal{Q}$ has the subscript F, then the corresponding state in $\mathcal{Q}_1$ must have been both final and non-terminal. Not every s_2^q in $\mathcal{Q}$ is $f(q)$ for some $q \in \mathcal{Q}_1$, but if $q \in \mathcal{Q}_1$ is non-final and non-terminal, it will be merged with the start state of its copy of Q_2. This state mapping is mostly useful for defining the transition function for iteration automata. Using this mapping to

convert between q and $f(q)$ and vice versa, we can be sensitive to the above-mentioned exceptions in the replacement scheme while still using δ_1 to define δ. When the value of f for some q is clear, we may write it directly, e.g. "q_{FT}" in lieu of "$f(q)$," and similarly for f^{-1}. Sometimes we write, e.g., q_T to mean a specific state, and other times to mean the set of all states that are $f(q)$ for some non-final, terminal state q. The intention should be clear from the context.

Definition 5.1.3. Let $\mathsf{Q}_1 = (\mathcal{Q}_1, \Sigma_1, \delta_1, s_1, F_1)$ and $\mathsf{Q}_2 = (\mathcal{Q}_2, \Sigma_2, \delta_2, s_2, F_2)$ be DFAs accepting the monadic quantifier languages $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_2}$, respectively. The iteration DFA $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is given by:

- $\mathcal{Q}$: $\bigcup_{q \in \mathcal{Q}_1} \begin{cases} \mathcal{Q}_2^q \cup \{q_F\} & q \in F_1, \text{ non-terminal} \\ \mathcal{Q}_2^q & q \notin F_1, \text{ non-terminal} \\ \{e_q, q_{FT}\} & q \in F_1, \text{ terminal} \\ \{q_T\} & q \notin F_1, \text{ terminal} \end{cases}$

  Here e_q is the (potentially unnecessary) state added to make sure input seen in q_{FT} is well-formed.
- $\Sigma = \{0, 1, ⊡\}$
- Transition function:
  - For $p \in \mathcal{Q}_2^q$: $\delta(p, x) = \begin{cases} \delta_2^q(p, x) & x \in \{0,1\} \\ f(\delta_1(f^{-1}(p), 1)) & x = ⊡, p \in F_2^q \\ f(\delta_1(f^{-1}(p), 0)) & x = ⊡, p \notin F_2^q \end{cases}$
  - For $q \in \{q_F\}$: $\delta(q, x) = \delta_2^q(s_2^q, x)$
  - For $q \in \{q_T\}$: $\delta(q, x) = q$
  - For $q \in \{q_{FT}\}$: $\delta(q, x) = \begin{cases} e_q & x \in \{0,1\} \\ q & x = ⊡ \end{cases}$
  - For $p \in \{e_q\}$: $\delta(p, x) = \begin{cases} e_q & x \in \{0,1\} \\ q & x = ⊡ \end{cases}$
- $s = f(s_1)$
- $F = \{f(q) \mid q \in F_1\}$

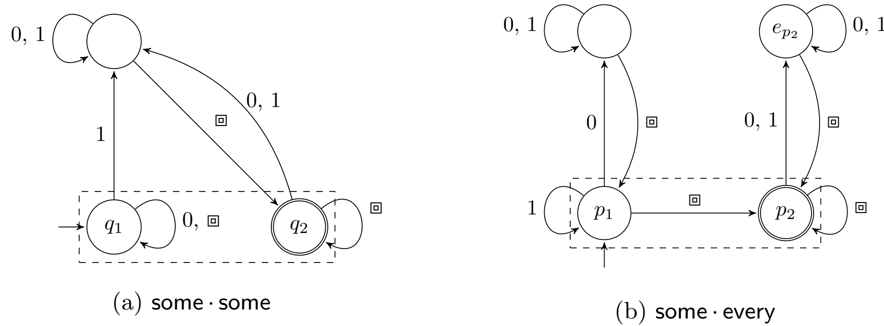
As remarked earlier, the e_q may not be necessary, but whether they are needed is easily seen after δ has been specified. Once the above construction is completed, one must inspect the state p such that $\delta(p, ⊡) = q$, for each q in q_{FT}. If p also loops on 0 and 1, then e_q can be removed, and δ amended such that q transitions to p on ⊡.

The definition of $\mathcal{Q}$ makes obvious the following upper bound on the size of iteration DFAs:

Fact 5.1.4. The state space of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is at most

$$\sum_{q \in \mathcal{Q}_1} \begin{cases} \mid \mathcal{Q}_2 \mid +1 & q \in F_1, \text{ non-terminal} \\ \mid \mathcal{Q}_2 \mid & q \notin F_1, \text{ non-terminal} \\ 2 & q \in F_1, \text{ terminal} \\ 1 & q \in F_1, \text{ non-terminal} \end{cases}$$

This fact gives us the *state complexity* of iteration DFA, which is a worst-case notion of state-space size[2], and thus an upper bound.[3] Since our definition uses cases, it is generally tight for any language. Iteration DFA may have fewer states if, as discussed above, extra states e_q are not needed to ensure well-formedness of the input (see Figure 5.3). Thus, the size of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is generally within $m = \mid \{q_{FT}\} \mid$ of this upper bound (and m is at most 1 for regular quantifiers[4]). It is also possible for unforseen state equivalences to occur (see, for example, Figure 5.4), but such minimizations are similarly restricted to "end behavior."



(a) $\mathsf{some} \cdot \mathsf{some}$

(b) $\mathsf{some} \cdot \mathsf{every}$

Figure 5.3: In (a), the terminal final state q_2 of the outer some can $0,1$-transition to the terminal state of the embedded some since that state is equivalent to the potential e_{q_2}. In (b), our definition correctly predicts the necessity of the additional e_{p_2}.

---

[2]See [17] for an overview of the concept. "The *state complexity* of a regular language L ...is the number of states of its minimal DFA," and "[t]he *state complexity of an operation* (or *operational state complexity*) on regular languages is the worst-case complexity of a language resulting from the operation, considered as a function of the state complexities of the operands."

[3]In [45] (2014), Steinert-Threlkeld also gives a definition (developed independently) of iteration DFA for type $\langle 1,1,2 \rangle$ regular iterations (as opposed to the stack version $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)_{\mathsf{stack}}$ explained earlier). His definition uses the cross-product of $\mathcal{Q}_1$ and $\mathcal{Q}_2$ for the state space with an "unrolled" version of Q_2 with an extra state if s_2 is final. This leads to a state complexity of $\mid \mathcal{Q}_1 \mid \cdot \mid \mathcal{Q}_2 + 1 \mid$. Using the function f to distinguish different cases in defining the state space, we achieve a smaller state complexity that only reaches $\mid \mathcal{Q}_1 \mid \cdot \mid \mathcal{Q}_2 + 1 \mid$ in case Q_1 is a trivial language (having only final states).

[4]Note though that our definition, along with that of [45], is fully general, applicable to any two binary regular languages.

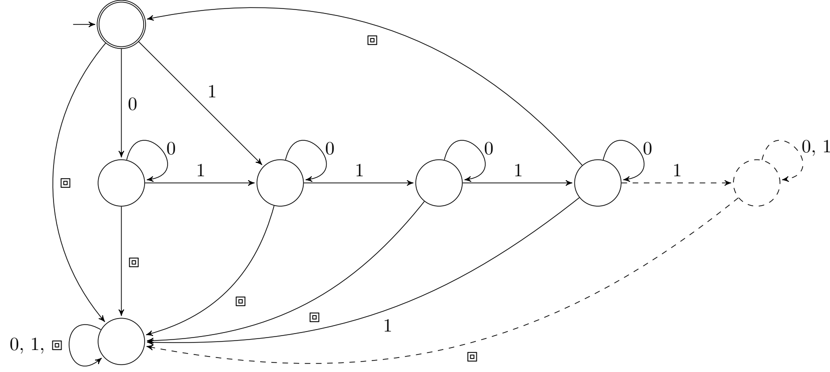


Figure 5.4: every·exactly three. The solid lines indicate the minimal DFA; the dashed lines indicate the output of the construction, with a full copy of exactly three.

5.1.2 Proving Correctness

We now make explicit the correctness of the definition of iteration automata by showing that the language accepted by the automaton constructed from the automata for regular quantifiers Q_1 and Q_2 and the language $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$ are equivalent. To prove this, we first introduce a bit more helpful terminology and a preliminary lemma. Define a function $g : \{0,1\}^* \to \{0,1\}$ by:

$$g(w) = \begin{cases} 0 & w \notin \mathcal{L}_{\mathsf{Q}_2} \\ 1 & w \in \mathcal{L}_{\mathsf{Q}_2} \end{cases}$$

so g is the characteristic function of $\mathcal{L}_{\mathsf{Q}_2}$, and let $g'(w) = g(w_i)_{i \leq \#_{\boxdot}(w)}$, where $w \in (w_i \boxdot)^*$. For example, letting $\mathsf{Q}_2 = \mathsf{every}$, we can calculate $g'(111 \boxdot 101 \boxdot) = g(111)g(101) = 10$.

Using g and the f defined in Definition 5.1.2, in the following lemma we prove the intuition grounding our construction in the first place: that transitions on words in the language of Q_2 and its complement in $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ are somehow equivalent to transitions on 1's and 0's in Q_1. See Figure 5.5 for an illustration of this idea. Given this correspondence, the desired result will be easy to see.

Lemma 5.1.5. For $w_i \in \{0,1\}$ and $p = f(q)$ for some $q \in \mathcal{Q}_1$, $\delta(p, w_i \boxdot) = f(\delta_1(f^{-1}(p), g(w_i)))$. This means that the state $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ reaches from p reading $w_i \boxdot$ is the result of applying f to the state that Q_1 reaches from $f^{-1}(p)$ reading $g(w_i)$.

Proof. There are four cases to consider, depending on what kind of state p is:

(i) s_2^q: Suppose $w_i \in \mathcal{L}_{\mathsf{Q}_2}$. Then $\delta(s_2^q, w_i) = p$ where $p \in F_2^q$, and $\delta(p, \boxdot) = f(\delta_1(q, 1))$, which is precisely $f(\delta_q(f^{-1}(s_2^q), g(w_i)))$. The case for $w_i \notin \mathcal{L}_2$

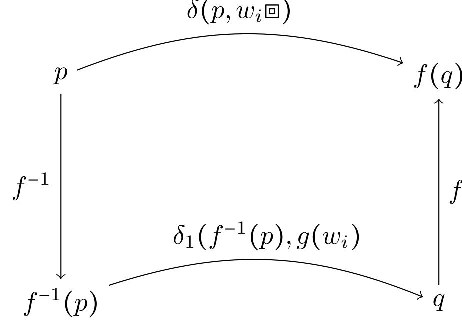


Figure 5.5: Diagram for Lemma 5.1.5

is symmetric.

(ii) q_F: Suppose $w_i \neq \epsilon$, so $w_i = xw'_i$ where $x \in \{0,1\}$. Then $\delta(q_F, x) \in \mathcal{Q}_2^q$, and this collapses to case (i). Suppose $w_i = \epsilon$, and $\epsilon \in \mathcal{L}_{\mathsf{Q}_2}$, so $g(w_i) = 1$. Then $\delta(q_F, \boxdot) = \delta(s_2^q, \boxdot) = f(\delta_1(q, 1))$, since $s_2^q \in F_2^q$ (and similarly if $\epsilon \notin \mathcal{L}_{\mathsf{Q}_2}$).

(iii) q_{FT}: Since $q = f^{-1}(q_{FT})$ is terminal, $\delta_1(q, g(w_i)) = q$. If $w_i = \epsilon$, then $\delta(q_{FT}, \boxdot) = q_{FT} = f(q)$. If not, then $\delta(q_{FT}, w_i) = e_q$, and $\delta(e_q, \boxdot) = q_{FT} = f(q)$.

(iv) q_T: Again, $q = f^{-1}(q_T)$ is terminal, so $\delta_1(q, g(w_i)) = q$, and $\delta(q_T, w_i \boxdot) = q_T = f(q)$.

□

Theorem 5.1.6. The language accepted by $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ is $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$.

Proof. It follows from Lemma 5.1.5 that if w is a string of the form $(w_i \boxdot)^*$, $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ accepts w visiting a sequence of states $s_2^{s_1} q_1 \cdots q_n$ (with $q_i = f(q)$ for some $q \in \mathcal{Q}_1$, and possibly repeating) if and only if Q_1 accepts the string $g'(w)$ visiting the sequence of states $s_1 f^{-1}(q_1) \cdots f^{-1}(q_n)$. That is, $w \in L(\mathsf{Q}_1 \cdot \mathsf{Q}_2)$ if and only if $g'(w) \in \mathcal{L}_{\mathsf{Q}_1}$. But $g'(w) \in \mathcal{L}_{\mathsf{Q}_1}$ if and only if $(\#_0(g(w)), \#_1(g(w))) \in \mathcal{Q}_1^c$, if and only if, by definition, $(|\{w_i : w_i \notin \mathcal{L}_{\mathsf{Q}_2}\}|, |\{w_i : w_i \in \mathcal{L}_{\mathsf{Q}_2}\}|) \in \mathsf{Q}_1^c$, which is the definition of membership for $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$. □

5.2 Generalizing to Type $\langle 1, 1, \ldots, n \rangle$ Regular Iterations

5.2.1 Translating Models with n-ary Relations into Strings

So far we have only considered iterations of type $\langle 1, 1, 2 \rangle$, representing sentences such as *Five students passed every exam* or *No professors failed all the students.*

But natural language contains higher-order iterations as well. For example, *Some organization sent five representatives to every meeting* can be treated as the type $\langle 1,1,1,3\rangle$ quantifier some·five·every applied to the ternary relation *send*. Even higher order iterations occur, such as *Two charities requested every person donate exactly three things to every needy family*, a type $\langle 1,1,1,1,4\rangle$ iteration. Certainly there is a limit to the number of embedded quantifiers people produce and comprehend in natural language, but in principle iterations of any type $\langle 1,1,1,\dots,n\rangle$, where the number of 1's is n, are possible. In this section we show how to generalize the above definitions and automata construction for type $\langle 1,1,2\rangle$ iterated quantifiers in a straight-forward, recursive fashion.

As with type $\langle 1,1,2\rangle$ iterations, we first define the string translations of relations of arbitrary arity, which will be the members of iterated quantifier languages and automata input strings. As usual the idea is best introduced by an example.

Example 5.2.1. To transcribe the ternary relation depicted in Figure 5.6, we look, for each (a,b) tuple, at every c and record whether c is in the relation R with a and b, i.e. whether (a,b,c) is in R. The substrings generated by an (a,b) pair are demarcated by ⊡, and the substrings consisting of the (a,b)-substrings for a given a and every b are distinguished by some new symbol, say ⊠.

To be precise, we want to calculate:

$$\tau_3(\vec{a},\vec{b},\vec{c},R) = (\tau_2(\vec{b},\vec{c},R_{a_i})\boxtimes)_{i\leq|A|} = (((\tau(\vec{c},R_{a_ib_j})\boxdot)_{j\leq|B|})\boxtimes)_{i\leq|A|}$$

which breaks down into the following steps:

$i=1$ $((\tau(\vec{c},R_{a_1b_j})\boxdot)_{j\leq|B|})\boxtimes$

$j=1$ $\tau(\vec{c},R_{a_1b_1})\boxdot \Rightarrow R_{a_1b_1}=\{c_1,c_2\} \Rightarrow 110\boxdot$

$j=2$ $\tau(\vec{c},R_{a_1b_2})\boxdot \Rightarrow R_{a_1b_2}=\{c_1,c_3\} \Rightarrow 101\boxdot$

$j=3$ $\tau(\vec{c},R_{a_1b_3})\boxdot \Rightarrow R_{a_1b_3}=\varnothing \Rightarrow 000\boxdot \Rightarrow$ concatenate and add ⊠

$i=2$ $\Rightarrow 100\boxdot 000\boxdot 101\boxdot\boxtimes$

$i=3$ $\Rightarrow 000\boxdot 010\boxdot 010\boxdot\boxtimes$

The concatenation of these strings for $1\leq i\leq 3$, $110\boxdot 101\boxdot 000\boxdot\boxtimes 100\boxdot 000\boxdot 101\boxdot\boxtimes 000\boxdot 010\boxdot 010\boxdot\boxtimes$, encodes the relation.

Definition 5.2.2. Let $\mathcal{M}=(M,A_1,\dots,A_n,R)$ be a model with R an n-ary relation consisting of tuples $(a_1,\dots,a_n)$ where each a_i is from the set A_i. We define the function τ_n for translating n-ary relations to strings by:

$$\tau_n(\vec{a_1},\dots,\vec{a_n},R) = \tau_{n-1}((\vec{a_2},\dots,\vec{a_n},R_{a_{1_i}})\boxdot)_{i\leq|A_1|}$$

and applying the definition until arriving at the base case yields:

$$\tau_n(\vec{a_1},\dots,\vec{a_n},R) = (((\tau(\vec{a_n},R_{a_{1_{\sigma_1}},\cdots,a_{n-1_{\sigma_{n-1}}}})\boxdot_1)_{\sigma_{n-1}\leq|A_{n-1}|}\cdots)\boxdot_{n-1})_{\sigma_1\leq|A_1|}$$

where $R_{a_1\cdots a_k}=\{(a_{k+1},\dots,a_n):(a_1,\dots,a_k,a_{k+1},\dots,a_n)\in R\}$. The strings generated are of the form $(\cdots(((0+1)^{|A_n|}\boxdot_1)^{|A_{n-1}|}\boxdot_2)\cdots\boxdot_{n-1})^{|A_1|}$. In general,

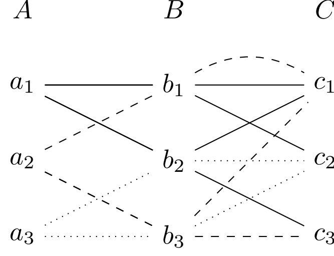


Figure 5.6: Model with ternary relation

translating an n-ary relation requires $n+1$ symbols: 0, 1, and $n-1$ distinct separator symbols.

Definition 5.2.3. Let $\mathsf{Q}_1, \ldots, \mathsf{Q}_{n+1}$ be type $\langle 1,1 \rangle$ quantifiers. Let $w_{\boxdot_n}$ denote strings of the form $((\cdots((w_i \boxdot_1)^* \boxdot_2)^* \cdots) \boxdot_n)^*$, where $w_i \in \{0,1\}^*$. Define the language of the iterated quantifier $\mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n$ by:

$$\begin{aligned}\mathcal{L}_{\mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n} = \{w \in w_{\boxdot_n} : (| \{w_{\boxdot_{n-1}} : w_{\boxdot_{n-1}} \notin \mathcal{L}_{\mathcal{Q}_2 \circ \cdots \circ \mathsf{Q}_n}\} |, \\ | \{w_{\boxdot_{n-1}} : w_{\boxdot_{n-1}} \in \mathcal{L}_{\mathcal{Q}_2 \circ \cdots \circ \mathsf{Q}_n}\} |) \in \mathsf{Q}_1^c\}\end{aligned}$$

Example 5.2.4. Now we see the definition in action to derive the language of $\mathsf{every} \cdot \mathsf{two} \cdot \mathsf{some}$.

$$\begin{aligned} s \in \mathcal{L}_{\mathsf{every \cdot two \cdot some}} &\Leftrightarrow (| \{w_{\boxdot_1} : w_{\boxdot_1} \notin \mathcal{L}_{\mathsf{two \cdot some}}\} |, | \{w_{\boxdot_1} : w_{\boxdot_1} \in \mathcal{L}_{\mathsf{two \cdot some}}\} |) \in \mathsf{every}^c \\ &\Leftrightarrow | \{w_{\boxdot_1} : w_{\boxdot_1} \notin \mathcal{L}_{\mathsf{two \cdot some}}\} | = 0 \\ &\Leftrightarrow | \{w_{\boxdot_1} : (| \{w_i : w_i \notin \mathcal{L}_{\mathsf{some}}\} |, | \{w_i : w_i \in \mathcal{L}_{\mathsf{some}}\} |) \notin \mathsf{two}^c\} | = 0 \\ &\Leftrightarrow | \{w_{\boxdot_1} : | \{w_i : w_i \in \mathcal{L}_{\mathsf{some}}\} | < 2\} | = 0 \\ &\Leftrightarrow | \{w_{\boxdot_1} : | \{w_i : (\#_1(w_i), \#_0(w_i)) \in \mathsf{some}^c\} | < 2\} | = 0 \\ &\Leftrightarrow | \{w_{\boxdot_1} : | \{w_i \mid \#_1(w_i) > 0\} | < 2\} | = 0 \end{aligned}$$

Recall the string from Example 5.2.1 and check that it is in this language: every $w_{\boxdot_1}$ has at least two w_i with $\#_1(w_i)$ greater than zero (so the number of $w_{\boxdot_1}$ with fewer than two w_i with $\#_1(w_i)$ greater than zero is zero).

$$\overbrace{\underbrace{110 \boxdot 010 \boxdot 000 \boxdot}_{w_{\boxdot_1}} \boxtimes \underbrace{100}_{w_1} \boxdot \underbrace{000}_{w_2} \boxdot \underbrace{101}_{w_3} \boxdot \boxtimes 000 \boxdot 010 \boxdot 010 \boxdot \boxtimes}^{w_{\boxdot_2}}$$

5.2.2 The Construction

For higher-level iterations, the well-formedness of input requirement calls for a simplified state-replacement scheme, since it is no longer the case that a state q may merge with the start symbol of an embedded copy of some A_i^q. Recall that in the $\langle 1,1,2 \rangle$ case, well-formedness effectively requires that the string end in $\boxdot_1$, since a string of 1's and 0's ended by a single $\boxdot_1$ is by default a well-formed

string, even though most interesting models will have more than one element in the domain of the relation. This condition still exists for proper translations of n-ary relations: the string must at least end in $\boxdot_{n-1}$; but now there is much more internal structure. An occurrence of $\boxdot_i$ can never be followed by $\boxdot_j$ where $j > i + 1$ (similarly, 1's and 0's can only be followed by 1's, 0's, and $\boxdot_1$). Thus only states representing the outermost quantifier may merge with the start state of an embedded automaton (mergers can only occur in the first iteration). In an automaton accepting $\boxdot_1$ strings, empty w_i may only be followed by $\boxdot_1$, but $\boxdot_1$ is the only separator symbol anyway; however, an automaton accepting $\boxdot_2$ strings can only allow empty $\boxdot_1$ "words," and must be able to distinguish the rejecting situation that an empty w_i is followed by $\boxdot_2$ instead of $\boxdot_1$.

For this reason, it is more natural to define higher-order iteration automata $\mathsf{Q} \cdot \mathsf{Q}'$ recursively where Q is $(n-1)$-ary, rather than where Q' is $(n-1)$-ary, even though iteration is associative. Proceeding from left to right, iterating with additional automata adds additional layers of structure to the existing skeleton of the automaton. Proceeding from right to left, an additional iteration erases the basic structure of the existing automaton, and it is not straightforward to define the minimal DFA when embedding the more complex automaton under the monadic one.

Before giving a formal definition, we illustrate the idea with an example showing the progressive construction of two · every·exactly three·every.

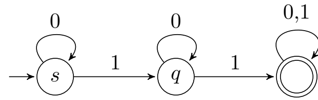


Figure 5.7: two

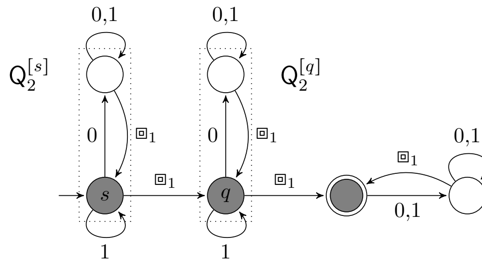


Figure 5.8: two · every

The jump from two (Figure 5.7) to two · every (Figure 5.8) follows precisely the definition presented in the previous section. The two non-terminal, non-final states of two, s and q, merge with their copies of every, $\mathsf{Q}_2^{[s]}$ and $\mathsf{Q}_2^{[q]}$, and the terminal final state of two is hooked up to a new state serving only to check the well-formedness of the remaining input. Shaded states represent the state-

space of the automaton from the previous iteration, constituting the spine of the current iteration. The only difference is we now index embedded copies of automata with a *list* of states, call it "H" for "history."

To construct two · every·exactly three (Figure 5.9) we start with the state-space of two · every (the six shaded states) and map every $⧈_1$ transition to a $⧈_2$ transition. Now we embed copies of exactly three in the existing copies of every. For example, the start state of every, s, is hooked up to its embedded copy of exactly three, $\mathsf{Q}_3^{[s]}$, according to the transitions of *its* start state p on 0's, 1's, and $⧈_1$. This of course incidentally defines the same transitions for the start state of two, which is identical to the start state of every (thus the size of the history also remains the same, consisting only of s). The rejecting state of every is terminal with respect to the subprocess it computes, so only needs to check the well-formedness of the input seen up until the next $⧈_2$. The state we would add to accomplish this is equivalent to the terminal state of exactly three. Lastly, the final state which simply checks input must be modified to process a more complicated structure. This modification is entirely systematic: since this subspace is only checking the *form* of the input, we map $⧈_1$ to $⧈_2$ but also $0, 1$ (considered here as $⧈_0$) to $⧈_1$; then add a state looping on $⧈_0$, and transitions from every $⧈_i$-looping state to the $⧈_j$-looping state on $⧈_j$, where $j \leq i + 1$. Notice there are no transitions drawn for $⧈_2$ in the newly embedded states. A $⧈_2$ immediately following a 1 or 0 would be ill-formed; an undrawn "dead" state is the target of these implicit transitions. In general, in an iteration of n automata, Q_1 will have transitions defined for every separating symbol (of which there are $n - 1$), and the ith-level embedded automata Q_i will be defined for separating symbols up to $⧈_{n-i+1}$.

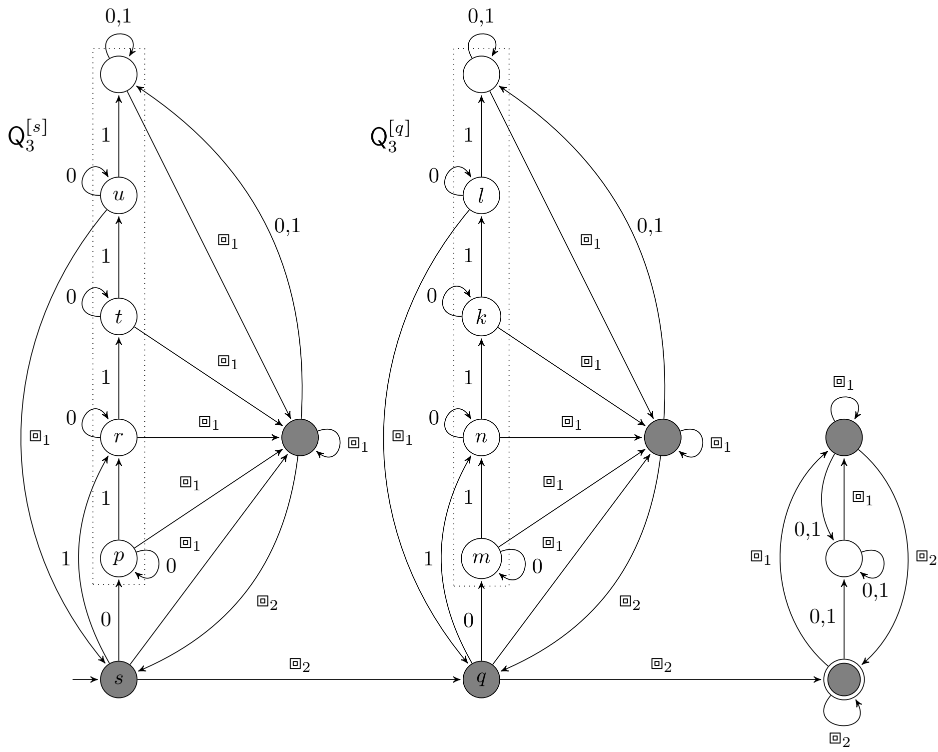


Figure 5.9: two · every·exactly three

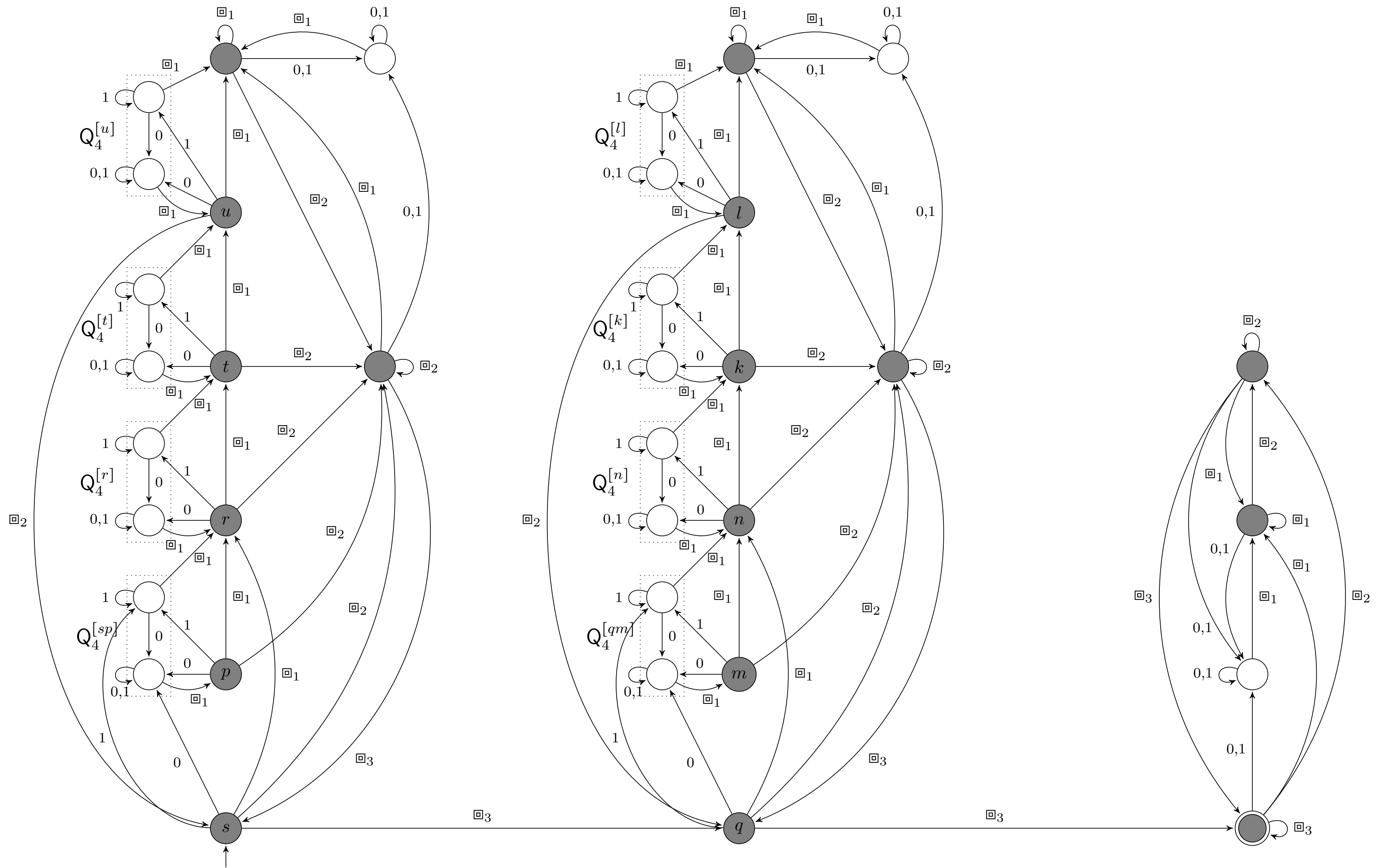


Figure 5.10: two · every·exactly three·every

For the final step in the iteration (Figure 5.10) we again take the state-space of two · every·exactly three, map $\boxdot_1$ to $\boxdot_2$ and $\boxdot_2$ to $\boxdot_3$, and create a copy of every for every state in an embedded copy of exactly three. For example, p gets $\mathsf{Q}_4^{[sp]}$ and r gets $\mathsf{Q}_4^{[r]}$. The former requires both p and s in its history because s's transitions on 0,1, and $\boxdot_1$ depend on p's transitions. Note that even though p, r, and t are non-final states in exactly three, they do not merge with the start states of automata embedded under them. This is the case for every step of constructing an iteration automaton, except the initial step. Finally, three more states are added to allow the cycles checking input to process strings with an additional level of structure.

Definition 5.2.5. Let $\mathsf{Q}_1, \dots, \mathsf{Q}_n$ denote DFA recognizing regular quantifier languages $\mathcal{L}_{\mathsf{Q}_1}, \dots, \mathcal{L}_{\mathsf{Q}_n}$. If $n = 2$, construct $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ according to Definition 5.1.3. If $n \geq 3$, construct $\mathsf{Q}' \cdot \mathsf{Q}_n = (\mathcal{Q}, \Sigma, s, \delta, F)$ from $\mathsf{Q}' = \mathsf{Q}_1 \cdot \dots \cdot \mathsf{Q}_{n-1} = (\mathcal{Q}'_1, \Sigma', s', \delta', F')$ and $\mathsf{Q}_n = (\mathcal{Q}_n, \Sigma_n, s_n, \delta_n, F_n)$ as follows, where Q_i^H denotes a copy of Q_1 indexed to a list of states $H = [h_1, \dots, h_m]$:

- $\mathcal{Q} = \mathcal{Q}' \cup \bigcup_{\{\mathsf{Q}_{n-1}^H\}} \bigcup_{q \in \mathcal{Q}_{n-1}^H} \begin{cases} \mathsf{Q}_n^{H+[q]} & q \text{ is non-terminal}, q = s_{n-1}^H \\ \mathsf{Q}_n^{[q]} & q \text{ is non-terminal}, q \neq s_{n-1}^H \end{cases}$
- $\Sigma = \Sigma' \cup \{\boxdot_{n-1}\}$
- $s = s'$
- δ :
  - $\delta'[\boxdot_i / \boxdot_{i+1}]$
  - $q \in \mathsf{Q}_n^H, H = [\dots, p] : \delta(q, x) = \begin{cases} \delta'(p, 1) & q \notin F_n^H, x = \boxdot_1 \\ \delta'(p, 0) & q \in F_n^H, x = \boxdot_1 \\ \delta_n(q, x) & x \in \{0, 1\} \end{cases}$
  - $p \in H$ of Q_n^H: $\delta(p, x) = \delta(s_n^H, x)$ for $x \in \{0, 1, \boxdot_1\}$
- $F = F'$

The above handles the embedding of Q_n; now we discuss how to make the iteration automaton capable of correctly rejecting words that are not of the form $w\boxdot_{n-1}$.

For every distinct subset A of $\mathcal{Q}'$ that collectively previously checked words for the form $w_{\boxdot_i}$, A now consists of a $\boxdot_j$-looping state for $1 \leq j \leq i+1$ (since we have already mapped all $\boxdot_j$ to $\boxdot_{j+1}$, and for states in A we consider $0, 1$ as $\boxdot_0$). Add a new state s to A that loops on $\{0, 1\} = \boxdot_0$. For every $a \in A$: if a loops on $\boxdot_j$, then add a $\boxdot_k$-transition from a to the $\boxdot_k$-looping state in A for $0 \leq k \leq j+1$.

For every terminal state q in every Q_{n-1}^H, add a state e_q and transitions such that q returns to itself reading words of the form $w_{\boxdot_1}$ (unless an equivalent state exists).

If $n = 3$ or there was no terminal, non-final state in Q_1, add a dead state d looping on every symbol in Σ and transitions to d for every transition undefined to this point. Otherwise, such a d exists; add transitions to d for every undefined transition.

Fact 5.2.6. Let $\mathsf{Q}' = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_{n-1}$, let $\mathcal{Q}'_T$ denote the set of states in $\mathcal{Q}'$ that were terminal in their respective Q_i, and let m denote the number of embedded copies of Q_{n-1}. The state complexity of $\mathsf{Q} = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n$ is

$$|\,\mathcal{Q}'\,| + \sum_{i}^{x} |\,\mathcal{Q}_{n-1}\,| \cdot |\,\mathcal{Q}_n\,| + |\,\mathcal{Q}'_T\,|$$

The maximum number of states of Q is the size of Q', plus the size of all the embedded copies of Q_n, plus the number of distinct input-checking subspaces (which is at most the number of terminal states added along the way).

5.2.3 Proving Correctness

Now we prove the correctness of the general construction for iteration DFA given in the preceding section. This proof largely has the same flavor as the proof of correctness for the case with only two automata, but we modify those notions for use in an inductive proof. Given an iteration DFA $\mathsf{Q} = \mathsf{Q}' \cdot \mathsf{Q}_n$, where $\mathsf{Q}' = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_{n-1}$, we again define a characteristic function for the language of the right-most quantifier Q_n and use this to prove a lemma relating transitions in Q to transitions in Q', from which the desired result will easily follow.

To prove this result inductively, we also require a modified definition of an iterated language. First recall the definition of $g : \{0,1\}^* \to \{0,1\}$ given by:

$$g(w_i) = \begin{cases} 0 & w_i \notin \mathcal{L}_{\mathsf{Q}_n} \\ 1 & w_i \in \mathcal{L}_{\mathsf{Q}_n} \end{cases}$$

Now instead of using exactly the same g' extending this to higher-type strings as previously defined, we need a new function $h : w_{\boxdot_n} \to w_{\boxdot_{n-1}}$ that can accomodate strings with additional $\boxdot$ symbols. Given a string w, calculate $h(w)$ by sequentially running through the entire string, applying g to subwords of 0's and 1's, erasing $\boxdot_1$, changing $\boxdot_i$ to $\boxdot_{i-1}$, and concatenating these results in the same order. For example, for $n = 3$ and $\mathsf{Q}_3 = \mathsf{some}$, we have:

$$h(01\,\boxdot_1\,11\,\boxdot_1\,\boxdot_2 11\,\boxdot_1\,00\,\boxdot_1\,\boxdot_2) = 11\,\boxdot_1\,10\boxdot_1$$

Using h we obtain the following definition of iterated languages:

$$\mathcal{L}_{\mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n} = \{w : w \text{ has the form } w_{\boxdot_n} \text{ and } h(w) \in \mathcal{L}_{\mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_{n-1}}\}$$

Lemma 5.2.7. Let $\mathsf{Q}' = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_{n-1}$ and $\mathsf{Q} = \mathsf{Q}' \cdot \mathsf{Q}_n$, where $n \geq 3$. For every $q \in \mathcal{Q}'$, $\delta(q, w\boxdot_1) = \delta'(q, g(w))$.

Proof. There are three main cases depending on the identity of q.

(i) $q \in \mathcal{Q}^H_{n-1}$

(a) q is terminal for Q_{n-1}: Then we explicitly construct q's transitions in Q such that q returns to itself reading words of the form $w_{\boxdot_1}$, so the Lemma clearly holds.

(b) q is non-terminal for Q_{n-1}: Then q is the most recent (or only) addition to H' for some $\mathsf{Q}^{H'}_n$. Thus $\delta(q, w_i\boxdot_1) = \delta(\delta(s^{H'}_n, w_i) = p, \boxdot_1) = \delta'(q, g(w_i))$, since $p \in F^{H'}_n \Leftrightarrow w_i \in \mathcal{L}_{\mathsf{Q}_n}$. (Note that this holds even if $w_i = \epsilon$).

(ii) $q = s^H_i$ for $i < n-1$: Then q was added to H' for some $\mathsf{Q}^{H'}_{i+1}$ and is thus in H' for some $\mathsf{Q}^{H''}_n$. So we have $\delta(q, w_i\boxdot_1) = \delta(\delta(s^{H''}_n, w_i) = p, \boxdot_1) = \delta'(r, g(w_i))$, since $p \in F^{H''}_n \Leftrightarrow w_i \in L_{\mathsf{Q}_n}$. Though $r \neq q$, r and q are both in H'', so $\delta'(r, g(w_i)) = \delta'(q, g(w_i))$.

(iii) Otherwise: If q is any other state, its transitions on $\{0, 1, \boxdot_1\}$ only serve to check well-formedness of the string, and the Lemma holds by construction.

□

Theorem 5.2.8. Let $\mathsf{Q} = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n$ be constructed from $\mathsf{Q}_1, \ldots, \mathsf{Q}_n$ accepting regular quantifier languages $\mathcal{L}_{\mathsf{Q}_1}, \ldots, \mathcal{L}_{\mathsf{Q}_n}$ according to Definition 5.2.5. The language of Q is $\mathcal{L}_{\mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n}$.

Proof. Let $\mathsf{Q}' = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_{n-1}$. Since we have that $w \in \mathcal{L}_{\mathsf{Q}' \cdot \mathsf{Q}_n}$ if and only if $h(w) \in \mathcal{L}_{\mathsf{Q}'}$ (by definition) if and only if Q' accepts $h(w)$ (by hypothesis), we only need to show that $\mathsf{Q}' \cdot \mathsf{Q}_n$ accepts w if and only if Q' accepts $h(w)$. This is clear from the previous Lemma: Q transitions from a state q to a state p reading some $w_i\boxdot_1$ if and only if Q' does the same reading the symbol $g(w_i)$, and Q transitions from q to p reading $\boxdot_i$ $(i \geq 2)$ if and only if Q' does the same reading $\boxdot_{i-1}$ (by construction); but these modifications of the symbols read are exactly the result of applying h. □

5.3 Generalizing the Stack Construction for Regular Iterations

Steinert-Threlkeld and Icard III do not discuss how their stack construction generalizes beyond automata for type $\langle 1, 1, 2 \rangle$ iterations. Recall that in the simple case, the stack automaton runs through the entire input, converting each subword of the form $(0+1)^*\boxdot$ into a 0 or 1 on the stack. The stack then has the form of a word in the language of a type $\langle 1, 1 \rangle$ quantifier. The generalization of this strategy is to take input in the form of a word in the language of a type $\langle 1, 1, \ldots, n \rangle$ quantifier (with n 1's), i.e. some $w_{\boxdot_{n-1}}$, and reduce it to a word in the language of a type $\langle 1, 1, \ldots, n-1 \rangle$ quantifier (with $n-1$ 1's), i.e. some

$w_{\boxdot_{n-2}}$, on the stack, by turning the subwords w_i into 1's or 0's based on the nth quantifier.

This means that when more than two quantifiers are involved, two stacks are necessary to implement this strategy. This means that, technically, these PDA are *Turing universal*, though the use of the two stacks is actually extremely restricted.[5] The nth quantifier, last in the iteration but first in the computation, reads the string of the form $w_{\boxdot_{n-1}}$ in the usual way, and computes a stack containing some $w_{\boxdot_{n-2}}$. Then the $(n-1)^{st}$ quantifier reads this first stack and computes some $w_{\boxdot_{n-3}}$ on the second stack. After this step, the first stack is empty, so in the next round of word-reduction, the second stack is read while the first stack is pushed.

It is quite interesting that, though DFA iteration automata somehow better exemplify the compositional nature of language in their very structure, the stack versions of iteration automata better exhibit quantifiers as *relation reducers*.[6] In the simple case $(\mathsf{Q}_1^A, \mathsf{Q}_2^B)(R^2)$, applying Q_2^B to R^2 yields a unary relation, the set of $a \in A$ such that $\mathsf{Q}_2^B(R_a)$ is (the 0-ary relation) true. For the ternary iteration $(\mathsf{Q}_1^A, \mathsf{Q}_2^B, \mathsf{Q}_3^C)(R^3)$, applying Q_3^C to R^3 yields a binary relation, the set of (a, b) such that Q_3 holds of $R_{a,b}$. In general, applying an m-ary quantifier to an $(m+n)$-ary relation yields an n-ary relation.

We can now also compare the number of computational steps, measured as the number of non-trivial symbols (those in $\{0, 1\}$) that must be read, taken by iteration DFAs and their stack versions.

Fact 5.3.1. The number of non-trivial computational steps taken by an iteration DFA reading input w is $\#_{0,1}(w)$.

Fact 5.3.2. The number of non-trivial computational steps taken by the stack version of an iteration automaton reading input w is exactly $|\, w \,|$.

Proof. Since $\#_{0,1}(w) = |\, w \,| - \sum_{i=1}^{n-1} \#_{\boxdot_i}(w)$, we need to account for the extra $\#_{\boxdot_i}(w)$-many steps taken by the stack version. The automaton repeatedly reduces words from the form $w_{\boxdot_i}$ to $w_{\boxdot_{i-1}}$, so we have:

$$(\cdots(((0+1)^{|A_i|}\boxdot_1)^{|A_{i-1}|}\boxdot_2)\cdots\boxdot_{i-1})^{|A_1|} \Rightarrow (\cdots(((0+1)^{|A_{i-1}|}\boxdot_1)^{|A_{i-2}|}\boxdot_2)\cdots\boxdot_{i-2})^{|A_1|}$$

The number of each symbol is shifted down by a factor of $|\, A_i \,|$. Each $w_{\boxdot_i}$ has $\prod_{j=1}^{i} |\, A_j \,|$ 1's or 0's that must be read. Summing over all inputs the automaton

[5] A Universal Turing Machine (UTM) can simulate an arbitrary Turing machine.

[6] Note also that [46] suggests how to compute type $\langle 1, 1, 2 \rangle$ iterations involving at least one PDA-computable quantifier, using a machine with two stacks. A generalization of PDA stack automata would then require *three* stacks, at least on the face of it–since they could of course be defined with just two or one, but not while so neatly exemplifying the iterated rewriting of the stack into the translation of a relation of lower arity.

must read (the initial string and all the stack rewrites), we have $\sum_{i=1}^{n}\prod_{j=1}^{i} \mid A_j \mid$ 1's or 0's read in total, which is exactly the length of the original input w (including 1's,0's, and every $\boxdot_i$). $\square$

Rather than a detailed definition, we give a high-level explanation of how to generalize stack iteration automata (refer to Figure 5.11). The version given by Steinert-Threlkeld and Icard III for type $\langle 1, 1, 2\rangle$ iterations remains the base-case of the construction. Given a stack iteration automaton recognizing some type $\langle 1, 1, \ldots, n-1\rangle$ quantifier (call this $\mathsf{Q}^{n-1}_{\mathsf{stack}}$), we construct the stack iteration automaton for a type $\langle 1, 1, \ldots, n\rangle$ quantifier by adding a new start state s_{n-1} with ϵ transitions to Q_n and the start state s_{n-2} of $\mathsf{Q}^{n-1}_{\mathsf{stack}}$. Each Q_i behaves exactly as in the simple case, reading a word of type $w_{\boxdot_{i-1}}$ from one stack (unless it is Q_n, which reads the original input string), and writing a 1 or 0 to the other stack depending on whether individual subwords in $(0+1)^*$ are in its language. Every time Q_i transitions back to s_{i-1} on $\boxdot_1$, there may be some number of separating symbols still to be seen before the next subword in $(0+1)^*$. Each s_{i-1} has transition rules for rewriting the $\boxdot_m$ on the input stack as $\boxdot_{m-1}$ on the other stack, for $1 \le m \le i-1$. This is how words are reduced at each step. Every starting state also has transitions to reverse the input stack (since stacks are last-in-first-out data structures, the word written in the previous computation is always backward). The machine must use the non-deterministic transitions as intended–going to the Q_i in the correct order and fully reversing the stack before going to the next Q_{i-1}–in order to accept a string without hitting some undefined configuration along the way. This means there exists an accepting computation for some input if and only if that string is in the language, which is the acceptance condition for nondeterministic PDA.

Example 5.3.3. To get a feel for how stack iteration automata operate in the general case, we give an overview of the steps taken in an example run of $(\mathsf{every}\cdot\mathsf{some}\cdot\mathsf{two}\cdot\mathsf{every})_{\mathsf{stack}}$:

$$\text{Input word: } \begin{array}{l} 111\boxdot_1 111\boxdot_1 111\boxdot_1 \boxdot_2 000\boxdot_1 000\boxdot_1 111\boxdot_1 \boxdot_2\boxdot_3 \\ 000\boxdot_1 111\boxdot_1 111\boxdot_1 \boxdot_2 000\boxdot_1 111\boxdot_1 111\boxdot_1 \boxdot_2\boxdot_3 \end{array}$$

$\mathsf{Q}_4 = \mathsf{every}$ turns words $(0+1)^3\boxdot_1$ into 1's or 0's on stack 2. s_3 writes $\boxdot_1$ to stack 2 reading $\boxdot_2$, and writes $\boxdot_2$ to stack 2 reading $\boxdot_3$:

$$\begin{array}{l} \text{stack } 1 = [\ ] \\ \text{stack } 2 = [111\boxdot_1 001\boxdot_1 \boxdot_2 011\boxdot_1 011\boxdot_1 \boxdot_2]^{\text{top}} \end{array}$$

Before guessing to advance to s_2, s_3 reverses the contents of stack 2 by pushing every symbol to stack 1:

$$\begin{array}{l} \text{stack } 1 = {}^{\text{top}}[111\boxdot_1 001\boxdot_1 \boxdot_2 011\boxdot_1 011\boxdot_1 \boxdot_2] \\ \text{stack } 2 = [\ ] \end{array}$$

$\mathsf{Q}_3 = \mathsf{two}$ turns each word $(0+1)^3\boxdot_1$ in stack 1 into a 1 or 0 on stack 2. s_2 reads the $\boxdot_2$ on stack 1 and writes $\boxdot_1$ to stack 2. Again, s_2 reverses the stack contents

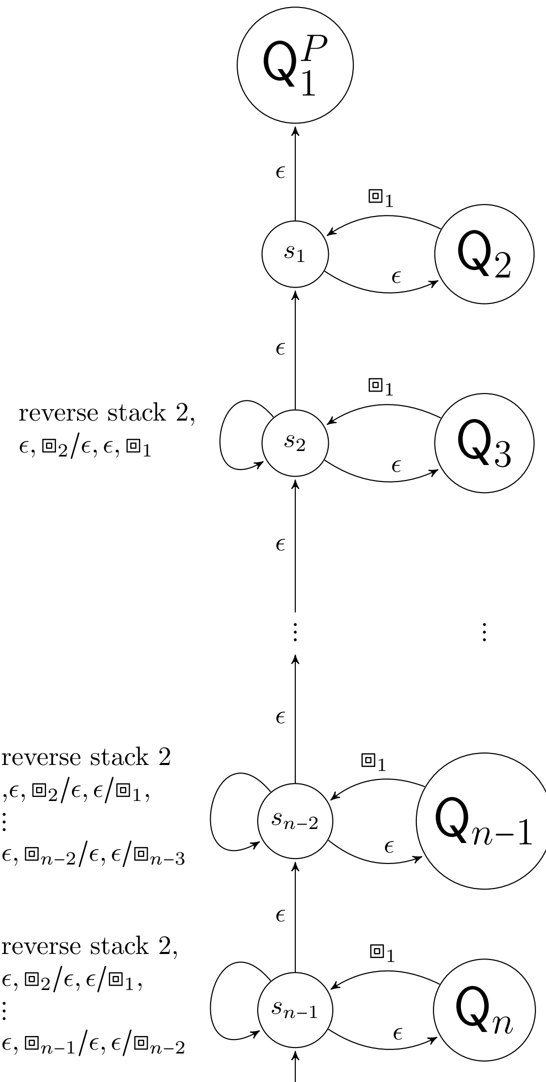


Figure 5.11: Schema for generalizing stack iteration automata: $\mathsf{Q}_{\mathsf{stack}}^{n-1}$

once stack 1 is empty, making stack 2 the empty one, before moving to s_1:

$$\text{stack } 1 = {}^{\text{top}}[10 \boxdot_1 11 \boxdot_1]$$
$$\text{stack } 2 = [\,]$$

Then $\mathsf{Q}_2 = \mathsf{some}$ turns each word $(0+1)^2 \boxdot_1$ in stack 1 into a 1 or 0 on stack 2:

$$\text{stack } 1 = [\,]$$
$$\text{stack } 2 = [11]^{\text{top}}$$

Finally s_1 transitions to $Q_1^P = \mathsf{every}$, which reads the contents of stack 2 and accepts. Since the input stack contains no $\boxdot$'s, there is no need to reverse the contents.

Fact 5.3.4. It is easily seen from the schema given in Figure 5.11 that the size of the state space of $\mathsf{Q}^n_{\mathsf{stack}}$ is $(\sum_{i=1}^{n} \mid \mathcal{Q}^i_{\mathsf{stack}} \mid +1) - 1$, or alternatively, $\mid \mathcal{Q}^{n-1}_{\mathsf{stack}} \mid + \mid \mathcal{Q}_n \mid +1$.

Summary and Open Questions

In this chapter we saw how to construct minimal iteration DFA from two or more minimal DFA and observed that the sizes of the resulting state-spaces are roughly multiplicative in the size of the input automata. We also saw that the stack construction generalizes in a very interesting way. The size of these stack automata is always only additive in the size of the input automata, yet it is always the case that more computational steps must be taken compared to a run of the minimal iteration DFA. These facts motivate questions regarding which model of computation is preferable, which we revisit in Section 9.1.1. In a more theoretical vein:

Question 5.3.5. To which other polyadic constructions can we extend the semantic automata model?

Question 5.3.6. Are there possible semantic automata models of polyadic quantifiers for which generalization is not so well-behaved? For example, is there a lift for which a single application (resulting in a type $\langle 1, 1, 2 \rangle$ quantifier with a $\boxdot_1$ language) is computationally feasible, but n-many applications (resulting in a type $\langle 1, 1, \ldots, n \rangle$ quantifier with a $\boxdot_{n-1}$ language) crosses some computational boundary?

Of course, if we allow two-stack machines as in generalized stack automata, we could probably recognize any polyadic construction we wished. The interesting question is: can this be done in a *compositional* way? In Chapter 7, we show that modulo a strengthening of the translation function, we can construct simple automata to account for cumulation. Surely there is great opportunity for further research into this question.

Chapter 6

Closure of DCFLs Under Iteration and Constructions for Iteration DPDA

Having seen in Chapter 4 that regular and context-free quantifier languages are closed under iteration, and given the recent result of Kanazawa, it is natural and relevant to ask whether *deterministic* context-free quantifier languages are closed under iteration. In this section we provide a proof answering this open question in the affirmative.[1] The result is not obvious since DCFLs do not enjoy general substitution closure. With this fact established, we give constructions of iteration DPDA for type $\langle 1, 1, 2 \rangle$ deterministic context-free iterated quantifiers where at least one of the inputs is a DPDA. In addition to very general considerations of the relevance of determinism to human cognition, there are other good reasons to restrict our automata constructions to deterministic models of computation (see Section 1.2.2).

6.1 Closure of DCFLs under Iteration

The proof proceeds by the following steps: given two DPDA recognizing type $\langle 1, 1 \rangle$ quantifiers Q_1 and Q_2, we construct the equivalent grammars G_1, G_2, and $\overline{G_2}$ and compose them in an iterated grammar $G_1 \cdot G_2$ in the obvious way, subbing the the start symbols of the latter two for 1's and 0's in the first. Then we use the DK-test[2] to show that this iterated grammar is itself deterministic, from

[1] This closure result was obtained independently by Shane Steinert-Threlkeld and a proof sketch appears in [45]. See a footnote in Section 6.2 for a remark on those results.

[2] See Section 2.2.3 for an explanation of the DK-test; the subject is treated in detail in [44].

which it follows that the language it generates (the language of the iteration of the two original quantifiers) is deterministic context-free.

G_2 and $\overline{G_2}$ must be such that, when we take the union of their components to construct the iterated grammar, every string of the form $(0+1)\boxdot$ has a unique reduction. To ensure this, we first prove the following lemma showing that we can modify any DPDA so that it yields a DCFG for both its accepted language and its complement.

Lemma 6.1.1. For every DPDA P recognizing some $\mathcal{L}_{\mathsf{Q}}$ that is the language of a deterministic context-free type $\langle 1, 1 \rangle$ quantifier Q, there is a DPDA P' with the following properties:

1. P' has a single accept state q_{accept} such that $(q_0, w\boxdot, \epsilon) \overset{*}{\vdash} (q_{\text{accept}}, \epsilon)$ if and only if $w \in \mathcal{L}_{\mathsf{Q}}$
2. P' has a state q_{reject} such that $(q_0, w\boxdot, \epsilon) \overset{*}{\vdash} (q_{\text{reject}}, \epsilon)$ if and only if $w \in \mathcal{L}_{\neg\mathsf{Q}}$
3. P' exclusively pushes or pops a symbol on every transition

That is, given P recognizing the language of Q, we can construct another DPDA that in a sense recognizes both Q and $\neg\mathsf{Q}$ by empty stack given an endmarker.

Proof. Follows from sequential modification according to Theorems 2.2.13 and 2.2.18 (see Figure 6.1 for an illustration of the final result). The first series of modifications produces a set of reading states R and a new set of final states F contained in R such that $R - F$ is accepting for the complement of P. The second series of modifications adds the new accept state q_{accept} and modifies the transition function such that the automaton empties its stack and goes to q_{accept} if it enters a state in F after reading $\boxdot$, satisfying (1). We do the same for a new state q_{reject} and $R - F$, satisfying (2). Satisfying (3) requires expanding some states, but does not affect the extension of R or F (or $R - F$). □

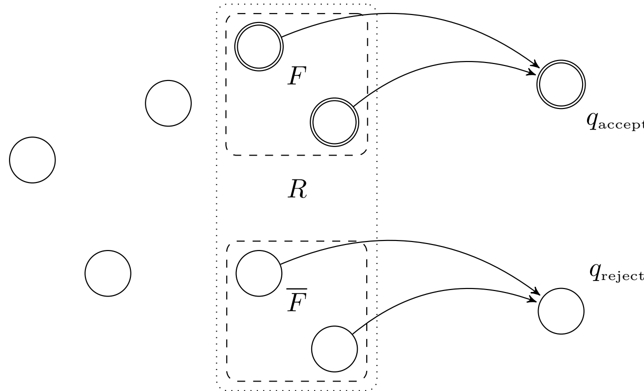


Figure 6.1: End result of combining complementation and grammar construction

Theorem 6.1.2. Deterministic context-free languages are closed under quantifier iteration.[3]

Proof. Let $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_2}$ be the languages of type $\langle 1,1\rangle$ deterministic context-free quantifiers Q_1 and Q_2. Then there exists DPDA A_1 and A_2 recognizing these languages. By Theorem 2.2.18 there is A_1' recognizing $\{w\boxtimes : w \in \mathcal{L}_{\mathsf{Q}_1}\}$ such that we may construct an equivalent DCFG $G_1 = (T_1, V_1, P_1, S_1)$. By Lemma 6.1.1, there is A_2' such that we may construct both the DCFG $G_2 = (T_2, V_2, P_2, S_2)$ and $\overline{G_2} = (T_2, V_2, P_2, \overline{S_2})$, where $S_2 = q_0 q_{\text{accept}}$ and $\overline{S_2} = q_0 q_{\text{reject}}$ (where q_0 is the start state of A_2'). Thus G_2 generates $\{w\boxdot : w \in \mathcal{L}_{\mathsf{Q}_2}\}$ and $\overline{G_2}$ generates $\{w\boxdot : w \in \mathcal{L}_{\neg\mathsf{Q}_2}\}$. G_2 and $\overline{G_2}$ have exactly the same terminals, variables, and productions since they are extracted from the same automaton; they differ only on their start symbols.

Define the iterated grammar $G_1 \cdot G_2 = (T_1, V, P, S_1)$ by:

- $V = V_1 \cup V_2$ (where V_1 and V_2 are disjoint)
- $P = P_1[0/\overline{S_2}, 1/S_2] \cup P_2$

The terminal symbols 1 and 0 in the production rules of G_1 are replaced by S_2 and $\overline{S_2}$ respectively. We may refer to $P - P_2$ by P_1' for shorthand. Since the above definition is the grammar counterpart of the language substitutions we defined to prove regular and context-free closure in Section 4.2, it follows from those arguments and the equivalence of *end-marked* DCFGs with DCFLs that $G_1 \cdot G_2$ generates the language of $\mathsf{Q}_1 \cdot \mathsf{Q}_2$, modulo an extra, distinct separator symbol ($\boxtimes$) at the end of the string.

Claim: $G_1 \cdot G_2$ is a DCFG.

Proof: We show $G_1 \cdot G_2$ satisfies the DK-test. Then it will follow from Theorem 2.2.15 that $G_1 \cdot G_2$ is deterministic.

Modify A_1' and A_2' by adding all the variables of G_1 and G_2 to their respective alphabets, and setting $\delta_i(p, V_{pq}, \epsilon) = (q, \epsilon)$ (and every other transition involving V_{pq} to $\varnothing$).[4] Additionally, replace 1 and 0 transitions in A_1' by S_2 and $\overline{S_2}$. This allows A_1' and A_2' to read valid strings generated by G_1 (with P_1' for P_1) and $(G_2 \cup \overline{G_2})$, respectively.

We recall the steps for creating grammar production rules for reference in the proof:

1-2. For each $p, q, r, s, t \in \mathcal{Q}, u \in \Gamma$, and $a, b \in \Sigma_\epsilon$, if $\delta(r, a, \epsilon) = (s, u)$ and $\delta(t, b, u) = (q, \epsilon)$, put the rule $A_{pq} \to A_{pr} a A_{st} b$ in G.

3. For each $p \in \mathcal{Q}$, put the rule $A_{pp} \to \epsilon$ in G.

[3] Note that, though we state this proof in terms of quantifier languages, it applies to the "quantifier iteration" of any two binary DPDA-recognizable languages.

[4] Note that $A_{q_0 q_{\text{accept}}}$ still refers to the start symbol of G_2 (S_2), and not the start symbol of $G_1 \cdot G_2$ (in the proof, we will not need to refer to S_1.)

Construct the DFA $DK_{G_1 \cdot G_2}$. Suppose it enters a final state on input z. This state contains at least one completed rule, R, having one of two forms:

1. $A_{pq} \to A_{pr} a A_{st} b.$

2. $A_{pp} \to .$

We show the accept state must contain neither

a. another completed 1-2 or ϵ rule, nor

b. a dotted 1-2 rule with a terminal immediately following the dot

This yields four main cases:[5]

1a. $R = A_{pq} \to A_{pr} a A_{st} b.$

I. $R \in P_2$:

i. $(*)$ There is no other completed type 1-2 rule. For every other rule R' in this accept state, z ends with the symbols preceding the dot in R'. If R and R' agree on their right hand sides, they must agree on their left hand sides, so R and R' are the same.

ii. If there is a completed ϵ rule $T = A_{xx} \to .$, it derives from a type 1-2 rule $T' = A_{wu} \to A_{wv} c . A_{yz} d$ (where $A_{yz} = A_{xx}$ or leads to adding T) with the dot before the second variable (before the *second* variable since it could not have been *added* to the state, as R is complete; before the second *variable* since it must necessitate adding T).

A. $(*)$ $T, T' \notin P_2$ as this would entail a simultaneous pushing and popping move in A'_2: by the definition of (1-2) rules, at the end of R A'_2 popped a symbol, and before the second variable of T (after the first terminal) A'_2 pushed a symbol. Recall that A'_2 was modified such that pushing and popping are always separate moves.

B. Then suppose T, T' are from P'_1. R and T must have both appeared in the same predecessor state of this final state and advanced their right hand side by the same symbol, implying one of the following holds:

- $c = b = \epsilon$, so $A_{wv} = A_{st}$, but $V_1 \cap V_2 = \varnothing$
- $b = \epsilon, c = A_{st} = S_2 = A_{q_0 q_{\text{accept}}}$, but this means there is a transition out of q_{accept} in A'_2 (see the description of 1-2 rules: if $A_{q_0 q_{\text{accept}}}$ appears in the right hand side of a rule in P_2, then A'_2 has a transition for q_{accept}).
- $c = \epsilon, b = A_{wv}$, but $b \in \{0, 1, ⧈\}$

[5] We follow the outline of the proof of Theorem 2.2.15 in [44]; identical subcases are marked by $(*)$. When some symbol may be either S_2 or $\overline{S_2}$, we assume the former without loss of generality.

which in any case is a contradiction.

II. $R \in P_1'$: This case is symmetric.

2a. $R = A_{pp} \to \bullet$

I. $R \in P_2$: We show there is no other ϵ rule (the possibility of a completed 1-2 rule is already handled by **1a.**).

i. $(*)$ There is no rule $T = A_{xx} \to \bullet \in P_2$, since this would entail that A_2' may end up in both p and x in reading z.

ii. Suppose there is some such rule T from P_1'. In order for R to be in the same state as T, there must be some rule $R' \in P_1'$, $A_{qr} \to A_{qs} \bullet S_2 A_{tu} b$ (with the dot after the *first* variable, w.l.o.g.). This means T must derive from some rule $T' \in P_1'$ of the form $A_{qw} \to A_{qs} \bullet A_{xx} d$, since the dot must be preceded by A_{qs} and followed by a variable (either equal to A_{xx}, or requiring the addition of a rule whose right hand side starts with A_{xx}, or and so forth). But this means that both $\delta_1(s, 1, \epsilon)$ and $\delta_1(s, \epsilon, \epsilon)$ are non-empty (recall, $R' \in P_1'$ means there was $R^* = A_{qr} \to A_{qs} 1 A_{tu} b$ in P_1), meaning A_1' is not a deterministic PDA, which is a contradiction.

II. $R \in P_1'$: This case is symmetric.

1b. $R = A_{pq} \to A_{pr} a A_{st} b \bullet$

I. $R \in P_2$:

i. $(*)$ Suppose there is a rule $T \in P_2$ with a dot before a terminal. Then A_2' does not pop its stack after reading z; but from R we know that it does.

ii. Suppose there is $T \in P_1'$ with a dot before a terminal (which must be $\boxtimes$). Let the dot be after the first variable w.l.o.g., so $T = A_{wu} \to A_{wv} \bullet \boxtimes A_{xy} d$. From R we know $DK_{G_1 \cdot G_2}$ just read $b \in \{0, 1, \boxdot\}$ or A_{st} $(b = \epsilon)$. Since A_{wv} cannot be a terminal, we have $A_{wv} = A_{st}$, but $V_1 \cap V_2 = \varnothing$.

II. $R \in P_1'$:

i. $(*)$ There is no rule $T \in P_1'$ with a dot before a terminal (see previous item).

ii. Suppose there is such a $T \in P_2$. From R we know that $DK_{G_1 \cdot G_2}$ either just read $b \in \{S_2, \boxtimes\}$ or A_{st} $(b = \epsilon)$. T can't have been added in this state (see argument in **1a.**) and can't contain A_{st} or $\boxtimes$, so $DK_{G_1 \cdot G_2}$ read S_2 and T is a rule in P_2 with a dot following S_2 on its right hand side; but this entails a transition in q_{accept} in A_2'.

2b. $R = A_{pp} \to \bullet$

I. $R \in P_2$:

i. ($*$) There is no dotted rule $T \in P_2$ with a terminal following the dot. Suppse there were: then z ends with some variable A_{st}, and after reading z A'_2 is prepared to read a non-ϵ terminal a. R must derive from some rule 1-2 rule $R' = A_{sr} \to A_{st}c \centerdot A_{wu}d \in P_2$ with the dot preceding the second variable (see **1a.**). Since z ends in A_{st}, c must be ϵ. Thus after reading z, A'_2 is also prepared to read an ϵ input in addition to $a \neq \epsilon$, which contradicts its determinism.

ii. Suppose there is $T \in P'_1$ with a terminal following the dot (which must be $\boxtimes$). Let the dot come after the first variable w.l.o.g., so $T = A_{wu} \to A_{wv} \centerdot \boxtimes A_{xy}d$. R must derive from some $R' = A_{mn} \to A_{ml}a \centerdot A_{nk}b \in P_2$ (where either $A_{nk} = A_{pp}$ or A_{nk} leads to adding R). This leads to a contradiction as in case **1b.I(ii)**.

II. $R \in P'_1$:

i. ($*$) There is no dotted rule $T \in P'_1$ with a terminal following the dot (above item).

ii. Suppose there is some such $T \in P_2$, either of the form $A_{wu} \to A_{wv} \centerdot cA_{xy}d$ or $A_{wu} \to A_{wv}cA_{xy} \centerdot d$ (assume the former without loss of generality). R must derive from some $R' = A_{mn} \to A_{ml}a \centerdot A_{nk}b$ (where $A_{nk} = A_{pp}$ or leads to adding R) in P'_1 with the dot preceding the second variable, implying one of the following:

- $a = A_{wv} = S_2$, which means there is a transition out of q_{accept} in A_2.
- $a = \epsilon$, so $A_{ml} = A_{wv}$, but $V_1 \cap V_2 = \varnothing$

which in any case is a contradiction.

■

Since $G_1 \cdot G_2$ is a DCFG, it follows from Theorem 2.2.17 that there is an equivalent DPDA recognizing the same language. □

6.2 Automata for Deterministic Context-Free Iterations

Now we define semantic automata for type $\langle 1,1,2 \rangle$ deterministic context-free iterations. Recall that we use the notation $\langle q, x, \alpha, \beta, q' \rangle$ for $\delta(q, x, \alpha) = (q', \beta)$ for (D)PDA. Similarly, if δ is the transition function of a DFA, we will write $\langle q, x, q' \rangle$ if $\delta(q, x) = q'$.

Definition 6.2.1. Let $\mathsf{Q}_1 = (\mathcal{Q}_1, \Sigma_1, \Gamma_1, \delta_1, s_1, F_1)$ be any DPDA recognizing a deterministic context-free quantifier language $\mathcal{L}_{\mathsf{Q}_1}$. Let $\mathsf{Q}_2 = (\mathcal{Q}_2, \Sigma_2, \Gamma_2, \delta_2, s_2,$

$q_{\text{accept}}, q_{\text{reject}})$ be a DPDA modified according to Lemma 6.1.1 recognizing an end-marked deterministic context-free quantifier language $\mathcal{L}_{\mathsf{Q}_2}$. Define the iteration DPDA $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ by:

- $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2$
- $\Sigma = \Sigma_1 \cup \Sigma_2$
- $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \mathcal{Q}_1$
- $$\begin{aligned} \delta \;=\; & \delta_2 & (1)\\ \cup\; & \{\langle q, \epsilon, \alpha, \beta, q'\rangle : \langle q, \epsilon, \alpha, \beta, q'\rangle \in \delta_1\} & (2)\\ \cup\; & \{\langle q, \epsilon, \alpha, q\alpha, s_2\rangle : (q, x, \alpha) \in \text{dom}(\delta_1) \text{ and } x \in \{0,1\}\} & (3)\\ \cup\; & \{\langle q_{\text{accept}}, \epsilon, q\alpha, \beta, q'\rangle : \langle q, 1, \alpha, \beta, q'\rangle \in \delta_1\} & (4)\\ \cup\; & \{\langle q_{\text{reject}}, \epsilon, q\alpha, \beta, q'\rangle : \langle q, 0, \alpha, \beta, q'\rangle \in \delta_1\} & (5) \end{aligned}$$
- $s = s_1$
- $F = F_1$

We take the states of Q_1 and the states of Q_2 and connect them in the following way: for every transition in δ_1 in which some state q reads a symbol, we replace that transition with an ϵ transition to the start state of Q_2 and push q to the stack. Thus all subwords $w_i⊡$ of the input are processed by Q_2; in any case, Q_2 empties its stack up to q and ends up in one of q_{accept} or q_{reject}, and transitions back into Q_1–with the new state and new stack contents decided by q.[6]

Of course, natural language iterations often involve a mixture of regular and context-free quantifiers:

(i) *A third of the students answered every question correctly.*

(ii) *Fewer than five students attended more than half of the presentations.*

Whether the DPDA-computable quantifier is outermost (as in (i)) or embedded (as in (ii)), we can still utilize the stack to record the progress of the computation (state of the outermost machine) before beginning a subcomputation with the embedded machine, avoiding the need to create multiple copies of the latter. The following two definitions make this precise; note the only significant departure from Definition 6.2.1 is with respect to δ, and in any case the resulting iteration automaton is a DPDA. For simplicity we do not pursue the "minimal" DPDA by distinguishing between terminal and non-terminal states of the outer quantifier. Note that since there is no state merging and the transitions from states in $\mathcal{Q}_1$ to states in $\mathcal{Q}_2$ are all ϵ transitions, the definition works for words of the form $(w_i⊡)^*$ even if some w_i are ϵ.

---

[6]The proof sketch of DCFL iteration closure in [45] is by DPDA construction and indicates that the construction proceeds similarly to the DFA case. Since simply complementing the accepting states of a given DPDA may not result in the correct behavior (as it may continue to transition between accepting and rejecting states after reading the input), the correctness of our definitions in this section relies on the modifications described in Lemma 6.1.1. The cases are not necessarily similar (that is, we can not necessarily use a single DPDA Q_2 to decide if $w \in L_{\mathsf{Q}_2}$ or $w \in L_{\neg\mathsf{Q}_2}$) without the kind of normal form for DPDA we describe here.

Definition 6.2.2. Let $\mathsf{Q}_1 = (\mathcal{Q}_1, \Sigma_1, \Gamma_1, \delta_1, s_1, F_1)$ be a DPDA recognizing a deterministic context-free quantifier language $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathsf{Q}_2 = (\mathcal{Q}_2, \Sigma_2, \delta_2, s_2, F_2)$ a DFA recognizing a regular quantifier language $\mathcal{L}_{\mathsf{Q}_2}$. Define the DPDA $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ by:

- $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2$
- $\Sigma = \Sigma_1 \cup \{\boxdot\}$
- $\Gamma = \Gamma_1 \cup \mathcal{Q}_1$
- $$\begin{aligned} \delta \;=\; & \{\langle q, x, \alpha, \alpha, q' \rangle : \langle q, x, q' \rangle \in \delta_2\} && (1') \\ \cup\; & \{\langle q, \epsilon, \alpha, \beta, q' \rangle : \langle q, \epsilon, \alpha, \beta, q' \rangle \in \delta_1\} && (2') \\ \cup\; & \{\langle q, \epsilon, \alpha, q\alpha, s_2 \rangle : (q, x, \alpha) \in \mathrm{dom}(\delta_1) \text{ and } x \in \{0,1\}\} && (3') \\ \cup\; & \{\langle p, \boxdot, q\alpha, \beta, p' \rangle : p \in F_2 \text{ and } \langle q, 1, \alpha, \beta, p' \rangle \in \delta_1\} && (4') \\ \cup\; & \{\langle p, \boxdot, q\alpha, \beta, p' \rangle : p \notin F_2 \text{ and } \langle q, 0, \alpha, \beta, p' \rangle \in \delta_1\} && (5') \end{aligned}$$
- $s = s_1$
- $F = F_1$

Definition 6.2.3. Let $\mathsf{Q}_1 = (\mathcal{Q}_1, \Sigma_1, \delta_1, s_1, F_1)$ be a DFA recognizing a regular quantifier language $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathsf{Q}_2 = (\mathcal{Q}_2, \Sigma_2, \delta_2, s_2, q_{\text{accept}}, q_{\text{reject}})$ a DPDA modified according to Lemma 6.1.1 recognizing an endmarked deterministic context-free quantifier language $\mathcal{L}_{\mathsf{Q}_2}$. Define the DPDA $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ by:

- $\mathcal{Q} = \mathcal{Q}_1 \cup \mathcal{Q}_2$
- $\Sigma = \Sigma_2$
- $\Gamma = \Gamma_2 \cup \mathcal{Q}_1$
- $$\begin{aligned} \delta \;=\; & \delta_2 && (1") \\ \cup\; & \{\langle q, \epsilon, \epsilon, q, s_2 \rangle : (q, x) \in \mathrm{dom}(\delta_1)\} && (2") \\ \cup\; & \{\langle q_{\text{accept}}, \epsilon, q, \epsilon, q' \rangle : \langle q, 1, q' \rangle \in \delta_1\} && (3") \\ \cup\; & \{\langle q_{\text{reject}}, \epsilon, q, \epsilon, q' \rangle : \langle q, 0, q' \rangle \in \delta_1\} && (4") \end{aligned}$$
- $s = s_1$
- $F = F_1$

Since the DFA Q_1 only has 0,1 transitions, there is no need for a corresponding (2") case.

Claim 6.2.4. Each automata $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ yielded by Definitions 6.2.1, 6.2.2 and 6.2.3 is deterministic.

Proof. First we show this holds when both Q_1 and Q_2 are DPDA. We show there is only one move per configuration in δ by examining each part (1)-(5) of the definition:

(1) δ_2 has at most one move per configuration.

(2) δ_1 has at most one move per configuration.

(3) A transition of this type is added if q has 0,1 moves in δ_1 with α on the stack. This means q does not have an ϵ move with α on the stack in δ_1 (or a transition with both ϵ input and stack). Thus, replacing 0,1 with ϵ with α on the stack leaves q with one choice in δ.

(4) In δ_2, q_{accept} has no moves by construction, and δ_1 is deterministic, so there is exactly one move in δ for configuration $(p, \epsilon, q\alpha)$.

(5) The same argument in (4) applies for q_{reject}.

The same arguments suffice to see that the δ given by (1')-(5') and (1")-(5") are deterministic, with the additional minor observations: for (1'), adding an inert stack component does not affect choice; for (4'/5'), the states $p \in \mathcal{Q}_2$ have no ⊡ transitions in δ_2. □

To see the correctness of these automata definitions, we again prove a lemma relating transitions on ⊡-ended words in $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ to transitions on individual symbols in Q_1.

Lemma 6.2.5. Let g be the characteristic function of $\mathcal{L}_{\mathsf{Q}_2}$. For $w_i \in \{0,1\}^*$ and $q \in \mathcal{Q}_1$, $\delta(q, w_i⊡, \alpha) = \delta_1(q, g(w_i), \alpha)$.

Proof. There are three cases; one per definition.

(i) Let $\mathsf{Q}_1, \mathsf{Q}_2$ both be DPDA. Assume w.l.o.g. that q has 0,1-transitions in δ_1 (otherwise there is an ϵ-transition to some q', in both δ_1 and δ_2, with the same effect on the stack (2)). Then in δ, q has an ϵ-move to s_2 with q pushed to the stack (3). Since q is not in Γ_2, this is effectively an empty stack to δ_2, so by (1) and Lemma 6.1.1 we have that $\delta(s_2, w_i⊡, q\alpha)$ goes to $(q_{\text{accept}}, q\alpha)$ if $g(w_i) = 1$ or $(q_{\text{reject}}, q\alpha)$ if $g(w_i) = 0$. By (4) and (5), there is an ϵ-move to $\delta_1(q, g(w_i), \alpha)$.

(ii) Let Q_1 be a DPDA and Q_2 a DFA. Again assume w.l.o.g. that q has 0,1-moves (2'). By (3'), there is an ϵ-move to s_2 with q pushed to the stack. By (1'), $\delta(s_2, w_i, q\alpha) = (p, q\alpha)$ (as the stack is left untouched), where $p \in F_2$ if and only if $g(w_i) = 1$. Finally, by (4'/5'), there is a ⊡-move to $\delta_1(q, g(w_i), \alpha)$.

(iii) Let Q_1 be a DFA and Q_2 a DPDA. In this case we take $\alpha = \epsilon$ and show that $\delta(q, w_i⊡, \epsilon) = (\delta_1(q, g(w_i)), \epsilon)$. By (3"), there is first an ϵ-move to s_2 with q pushed to the stack. As in case (i), Q_2 goes from s_2 with effectively empty stack (q on top) to q_{accept} if $g(w_i) = 1$ or q_{reject} if $g(w_i) = 0$ with effectively empty stack (q on top). By (4"/5"), there is an ϵ-move to $(\delta_1(q, g(w_i)), \epsilon)$.

□

Theorem 6.2.6. The language accepted by the DPDA $\mathsf{Q}_1 \cdot \mathsf{Q}_2$, generated by any of the three preceding definitions, is $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{Q}_2}$.

Proof. This follows from the preceding lemma and the argument for Theorem 5.1.6 for the two-DFA case. □

We do not give a formal definition for reasons of space, but these constructions can be generalized to iterations of an arbitrary number of DFA and DPDA by (1) using $\boxdot_i$ as an indicator for the ith embedded DPDA to empty its stack and (2) expanding the stack alphabet with each new embedding to maintain a history (as in Definition 5.2.5 for generalized iteration DFA).

Summary and Open Questions

In this chapter we saw a proof of the closure of deterministic context-free languages under quantifier iteration and definitions for constructing iteration DPDA when one or more of the quantifiers is DPDA-computable.

Are there other natural classes of quantifiers for which we can investigate iteration closure? For instance:

Question 6.2.7. Are context-sensitive languages (for example, *the same number of a's,b's, and c's*, which is type $\langle 1, 1, 1\rangle$) closed under quantifier iteration? The answer is not immediate; like DCFLs, context-sensitive languages are not closed under substitution.

Mostowski [38] identifies a subset of quantifiers that are accepted by DPDA by *both* final state and empty stack (recall: this means the DPDA is in a final state and the stack is also empty; not the usual notion of empty stack). This class more-or-less corresponds to exact proportional quantifiers (for example, exactly 1/3). Is *this* subset of DCFLs closed under quantifier iteration? This question is actually easy to answer.

Fact 6.2.8. This natural proper subset of deterministic context-free quantifier languages is *not* closed under iteration.

Proof. This follows immediately from almost-linear quantifiers lacking complement closure (indeed, the complement of an almost-linear quantifier is *never* almost-linear).[7] □

[7] See Section 3.3 for Mostowski's results and the statement of this fact.

Chapter 7

Cumulation Automata

[46] mentions in a footnote the possibility of defining cumulation automata as the sequential composition of iteration automata. The contribution of this chapter is to give precise definitions of automata both for type $\langle 1, 1, 2 \rangle$ regular cumulations as well as for type $\langle 1, 1, \ldots, n \rangle$ regular cumulations based on that suggestion. This requires a modification of the translation function to record both R and R^{-1}, which turns out to substantially simplify the difficulty of the language, indicating that the choice of model representation is an integral factor of semantic automata complexity. Cumulation constitutes an interesting extension of semantic automata because the quantifiers resulting from this lift are irreducibly polyadic. They are somehow *on or around* the Frege boundary since they *are* definable from iterations, but are not themselves iterations.

7.1 Automata for Type $\langle 1, 1, 2 \rangle$ Regular Cumulations

Recall that the cumulation $(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}(A, B, R)$ can be defined by

$$(\mathsf{Q}_1 \cdot \mathsf{some})(A, B, R) \wedge (\mathsf{Q}_2 \cdot \mathsf{some})(B, A, R^{-1})$$

Here we cannot interpret the "$\wedge$" as intersection, as we might in the case of the language of "More than two and less than four." The first iteration is evaluated using a string generated by τ_2 given R, and the other using R^{-1}. To extend the semantic automata model to cumulation, we need a way to combine DFAs $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}_2 \cdot \mathsf{some}$ into a single automaton that accepts a single input combining the strings they should evaluate. What we need is the sequential composition of these iteration automata, accepting concatenations of strings from the two languages.

Recall that the concatenation of $\mathcal{L}_1$ and $\mathcal{L}_2$ is $\mathcal{L}_1\mathcal{L}_2 = \{uv \mid u \in \mathcal{L}_1, v \in \mathcal{L}_2\}$, and regular languages are closed under concatenation (Theorem 2.2.5). Sequential composition is the automata operation corresponding to concatenation, illustrated in Figure 7.1, consisting of connecting final states of N_1 to the start state of N_2 by ϵ-transitions.

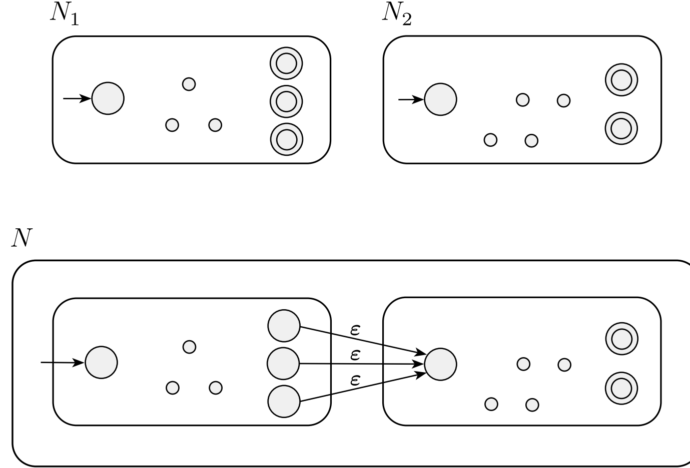


Figure 7.1: Sequential composition of automata ([44])

Definition 7.1.1. Let $\mathcal{M} = \langle M, A, B, R\rangle$ be a model with $\vec{a}$ and $\vec{b}$ any enumerations of A and B. Define the translation function τ_2^{cl} which takes two sets and a binary relation as arguments:

$$\tau_2^{\mathsf{cl}}(\vec{a}, \vec{b}, R) = \tau_2(\vec{a}, \vec{b}, R) \boxtimes \tau_2(\vec{b}, \vec{a}, R^{-1})$$

Definition 7.1.2. Let Q_1 and Q_2 be quantifiers of type $\langle 1, 1\rangle$. Define the language $(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}$ by:

$$\mathcal{L}_{(\mathsf{Q}_1,\mathsf{Q}_2)^{\mathsf{cl}}} = \{w_1 \boxtimes w_2 : w_1 \in \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{some}}, w_2 \in \mathcal{L}_{\mathsf{Q}_2 \cdot some}\}$$

By separating the words in $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{some}}$ and $\mathcal{L}_{\mathsf{Q}_2 \cdot \mathsf{some}}$ with a distinguished $\boxtimes$ symbol, we avoid the problem of the automaton having to "guess" when it has seen the end of the first word and the beginning of the second. Such non-determinism is not really an issue since NFAs and DFAs both generate exactly the regular languages; however, determinism is easily retained in this fashion. Moreover, these separator symbols may be necessary for cumulation automata to mean what we intend them to mean (see the discussion following Question 7.2.11 in the summary).

Example 7.1.3. Consider the sentence *Three cinephiles watched five movies*, on the reading that the three cinephiles, all together, watched a sum total of five movies. To translate the model in Figure 7.2, we calculate $\tau_2^{\mathsf{cl}}(\vec{c}, \vec{m}, W)$.

$\tau_2(\vec{c}, \vec{m}, W)$ yields the string 11110 ⧈ 10100 ⧈ 00001⧈ and $\tau_2(\vec{m}, \vec{c}, W^{-1})$ yields the string 110 ⧈ 100 ⧈ 110 ⧈ 100 ⧈ 001⧈ (imagine looking at the mirror image of the model). Concatenating these with ⊠ in the middle results in:

11110 ⧈ 10100 ⧈ 00001 ⧈ ⊠110 ⧈ 100 ⧈ 110 ⧈ 100 ⧈ 001⧈

Since the pre-⊠ portion of the string is in $\mathcal{L}_{3 \cdot \mathsf{some}}$ and the post-⊠ portion is in $\mathcal{L}_{5 \cdot \mathsf{some}}$, the whole string is in $\mathcal{L}_{(3,5)^{\mathsf{cl}}}$.

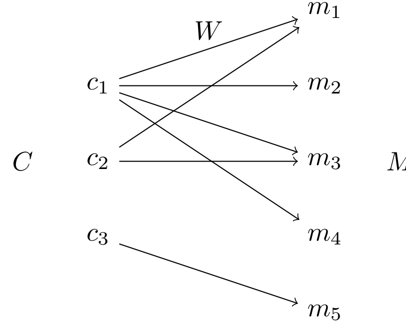


Figure 7.2: Model for Example 7.1.3

Definition 7.1.4. Let Q_1 and Q_2 be DFAs accepting the monadic quantifier languages $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_2}$, respectively. Construct the iteration DFA $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}_2 \cdot \mathsf{some}$. Denote these by A_1 and A_2, respectively. The cumulation DFA $(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}$ is given by:

- $\mathcal{Q} = \mathcal{Q}_{A_1} \cup \mathcal{Q}_{A_2}$
- $\Sigma = \Sigma_{A_1} \cup \{⊠\}$
- $\delta(q, x) = \begin{cases} \delta_{A_1}(q, x) & q \in \mathcal{Q}_{A_1}, x \neq ⊠ \\ s_{A_2} & q \in F_{A_1}, x = ⊠ \\ \delta_{A_2}(q, x) & q \in \mathcal{Q}_{A_2}, x \neq ⊠ \end{cases}$
- $s = s_{A_1}$
- $F = F_{A_2}$

Theorem 7.1.5. The language accepted by the DFA $(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}$ is $\mathcal{L}_{(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}}$

Proof. Let A_1 and A_2 denote the DFA $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}_2 \cdot \mathsf{some}$.

(⊆) Suppose $w \in \mathcal{L}_{(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}}$, so $w = w_1 ⊠ w_2$ where $w_1 \in L(A_1)$ and $w_2 \in L(A_2)$. By definition $\delta(s, w_1) = f \in F_{A_1}$, $\delta(f, ⊠) = s_{A_2}$, and $\delta(s_{A_2}, w_2) = f' \in F_{A_2}$, so $\delta(s, w) \in F$.

(⊇) Suppose $\delta(s, w) \in F$. By construction, any path from s to $f \in F$ consists of $s_{A_1} \cdots f s_{A_2} \cdots f'$ where $f \in F_{A_1}$ and $f' \in F_{A_2}$. Clearly w must consist of some string $w_1 \in L(A_1)$, followed by ⊠, followed by some string $w_2 \in L(A_2)$.

□

Fact 7.1.6. The state complexity of cumulation is twice the state complexity of iteration. More specifically, the size of $(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}$ is at most the size of $\mathsf{Q}_1 \cdot \mathsf{some}$ plus the size of $\mathsf{Q}_2 \cdot \mathsf{some}$. This upper bound is not reached in case both $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}_2 \cdot \mathsf{some}$ have q_T states. These can be merged since if either automaton reaches a terminal non-final state, the cumulation automaton will not accept (a terminal non-final state of either iteration automaton is of course also a terminal non-final state of the whole cumulation automaton).

Theorem 7.1.7. Regular languages are closed under cumulation (using the language definition developed for this chapter allowing inverse relations).

Proof. This result follows from the closure of regular languages under quantifier iteration and concatenation. □

Finally, we also sketch the corresponding result for deterministic context-free languages. Since cumulation automata are essentially a simple sequential composition of iteration automata, this follows from the results of the previous chapter, and it is clear that our definitions of cumulation DFA can easily be extended to cumulation DPDA.

Theorem 7.1.8. Deterministic context-free languages are closed under cumulation.

Proof. Let $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_1}$ be binary deterministic context-free languages recognized by Q_1 and Q_2. Since $\mathcal{L}_{\mathsf{some}}$ is regular, and thus a DCFL, by Theorem 6.1.2, $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{some}}$ and $\mathcal{L}_{\mathsf{Q}_2 \cdot \mathsf{some}}$ are also DCFL, and we can construct their automata $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}.\mathsf{some}$ using Definition 6.2.2. DCFL are not in general closed under concatenation; however, $\mathcal{L}_{(\mathsf{Q}_1, \mathsf{Q}_2)^{\mathsf{cl}}}$ is defined by concatenating $\mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{some}}$ and $\mathcal{L}_{\mathsf{Q}_2 \cdot \mathsf{some}}$ with a distinguished symbol ⊠, it is clear that connecting $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}_2 \cdot \mathsf{some}$ with a ⊠ transition as in Definition 7.1.4 (and using ⊠ as a marker for $\mathsf{Q}_1 \cdot \mathsf{some}$ to empty its stack) yields a deterministic PDA recognizing the correct language. □

7.2 Generalizing to Type $\langle 1, 1, \ldots, n \rangle$ Regular Cumulations

Just as with iterations in the previous chapter, we can define higher-type cumulations from their components. Since cumulation is an independent lift, the way we combine the constitutive parts is quite simple: given a cumulation of quantifiers Q_1 through Q_n and a relation R on sets A_1 through A_n, we conjoin a series of independent requirements on each sequential pair. Basically, we can determine the truth of the entire cumulation just by looking individually at the

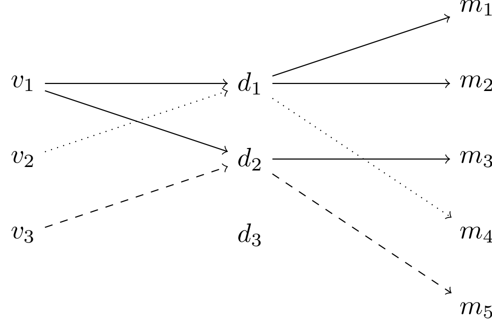


Figure 7.3: Model for Example 7.2.2

relation between each "adjacent" set A_i and A_{i+1}. To formalize the notion of a "part" of a relation, we define the following:

Definition 7.2.1. Let R be an n-ary relation on sets $A_1, \ldots, A_n$.

$$R \restriction_{A_i, A_{i+1}} = \{(a_i, a_{i+1}) : a_i \in A_i, a_{i+1} \in A_{i+1}, \exists a_j \in A_j \text{ s.t. } (\ldots, a_i, a_{i+1}, \ldots) \in R\}$$

Example 7.2.2. Consider the cumulative reading of the sentence *Exactly three veterinarians gave at most three dogs at least five milkbones*, which we formalize as $(=3, \leq 3, \geq 5)^{\mathsf{cl}}(V, D, M, G)$ (see Figure 7.3). This is true in a model if exactly three veterinarians (V), at most three dogs (D), and at least five milkbones (M) participate in the giving relation (G). We can break this down into the requirements that (1) exactly three veterinarians and at most three dogs are in the relation and (2) at most three dogs and at least five milkbones are in the relation, meaning $(=3, \leq 3)^{\mathsf{cl}}(V, D, G_{\restriction_{V,D}}) \wedge (\leq 3, \geq 5)^{\mathsf{cl}}(D, M, G_{\restriction_{D,M}})$. Each conjunct further decomposes into independent iterations: $(=3 \cdot \mathsf{some})$ $(V, D, G \restriction_{V,D})$ $\wedge(\leq 3 \cdot \mathsf{some})$ $(D, M, G^{-1} \restriction_{D,V})$ $\wedge(\leq 3 \cdot \mathsf{some})$ $(D, M, G \restriction_{D,M})$ $\wedge(\geq 5 \cdot \mathsf{some})(M, D, G^{-1} \restriction_{M,D})$. To evaluate these, we must look at four different strings:

$$\begin{aligned}
\tau_2(V, D, G \restriction_{V,D}) &= 110 \boxdot 100 \boxdot 010 \boxdot \\
\tau_2(D, V, G^{-1} \restriction_{D,V}) &= 110 \boxdot 101 \boxdot 000 \boxdot \\
\tau_2(D, M, G \restriction_{D,M}) &= 11010 \boxdot 00101 \boxdot 00000 \boxdot \\
\tau_2(M, D, G^{-1} \restriction_{M,D}) &= 100 \boxdot 100 \boxdot 010 \boxdot 100 \boxdot 010 \boxdot
\end{aligned}$$

If each of these strings is accepted by its respective iteration automaton, then the cumulation is true in the model. As in the $\langle 1, 1, 2 \rangle$ case, we combine the strings generated from the same two sets by gluing them together with a $\boxtimes$ symbol, and then glue *these* together with a new $\oplus$ symbol.

Definition 7.2.3. Let $\mathcal{M} = \langle M, A_1, \ldots, A_n, R \rangle$ be a model with R an n-ary relation consisting of tuples $(a_1, \ldots, a_n)$ with each a_i from A_i. Define the translation function τ_n^{cl}:

$$\tau_n^{\mathsf{cl}}(\vec{a_1}, \ldots, \vec{a_n}, R) = \tau_{n-1}^{\mathsf{cl}}(\vec{a_1}, \ldots, \vec{a_{n-2}}, R) \oplus \tau_2^{\mathsf{cl}}(\vec{a_{n-1}}, \vec{a_n}, R \restriction_{a_{n-1}, a_n})$$

Again, separating the subwords of each cumulation language is an easy way to preserve determinism. It is not absolutely necessary that we introduce a new symbol ($\oplus$), as it was in the $\langle 1,1,2 \rangle$ case ($\boxtimes$). Rather, it helps to maintain a conceptual distinction between the cumulations that combine to form $\mathcal{L}_{(\mathsf{Q}_1,\ldots,\mathsf{Q}_n)^{\mathsf{cl}}}$ and the iterations that combine to form *them*.

With relations of arity three and higher, it is possible to have "overlap" in a restricted part of the relation: for example, if R contains the triples (a_1, b_1, c_1) and (a_2, b_1, c_1), then $R \upharpoonright_{B,C}$ in a way contains the pair (b_1, c_1) twice. However, observe that when this part of the relation is eventually encoded into 0's and 1's by τ, we achieve the same result as if only one a were in the relation with b_1 and c_1. Furthermore, this is precisely the result we *want* to achieve. This may seem odd, for two related potential reasons:

(i) In natural language, the most common constructions involving a ternary relation use verbs such as *give*, *send*, etc., and this somehow enforces a linguistic prejudice for having distinct elements in the third spot.

(ii) Perhaps it is not due to the specific verbs we are used to seeing in these constructions, but moreso a dispreference for reading a sentence as a higher-arity cumulation in general. It would be interesting to see what kind of readings people are willing to attribute to sentences containing ternary, quaternary, etc., relations.

Definition 7.2.4. Let $\mathsf{Q}_1, \ldots, \mathsf{Q}_n$ be type $\langle 1,1 \rangle$ quantifiers. Define the language of $(\mathsf{Q}_1, \ldots, \mathsf{Q}_n)^{\mathsf{cl}}$ by:

$$\mathcal{L}_{(\mathsf{Q}_1,\ldots,\mathsf{Q}_n)^{\mathsf{cl}}} = \{w_1 \oplus \cdots \oplus w_n : w_i \in \mathcal{L}_{(\mathsf{Q}_i,\mathsf{Q}_{i+1})^{\mathsf{cl}}}\}$$

Definition 7.2.5. Let $\mathsf{Q}_1, \ldots, \mathsf{Q}_n$ be type $\langle 1,1 \rangle$ quantifiers. Define their type $\langle 1,1,\ldots,n \rangle$ cumulation by:

$$(\mathsf{Q}_1, \ldots, \mathsf{Q}_n)^{\mathsf{cl}}(A_1, \ldots, A_n, R) \Leftrightarrow \bigwedge_{i=1}^{n-1} (\mathsf{Q}_1, \mathsf{Q}_{i+1})^{\mathsf{cl}}(A_i, A_{i+1}, R \upharpoonright_{A_i, A_{i+1}})$$

Definition 7.2.6. Let $\mathsf{Q_1}$ through $\mathsf{Q_n}$ be DFAs accepting the monadic quantifier languages $\mathcal{L}_{\mathsf{Q_1}}$ through $\mathcal{L}_{\mathsf{Q_n}}$, respectively. Construct $A_i^{i+1} = (\mathsf{Q}_i, \mathsf{Q}_{i+1})^{\mathsf{cl}}$ for $1 \le i \le n-1$. The cumulation DFA $(\mathsf{Q}_1, \ldots, \mathsf{Q}_n)^{\mathsf{cl}}$ is given by:

- $\mathcal{Q} = \bigcup_{i=1}^{n-1} \mathcal{Q}_i^{i+1}$
- $\Sigma = \Sigma_1^2 \cup \{\oplus\}$
- $\delta(q,x) = \begin{cases} \delta_i^{i+1}(q,x) & q \in \mathcal{Q}_i^{i+1}, x \ne \oplus \\ s_i^{i+1} & q \in F_{i-1}^i, x = \oplus \end{cases}$
- $s = s_1^2$
- $F = F_{n-1}^n$

Theorem 7.2.7. The language accepted by the DFA $(\mathsf{Q}_1, \ldots, \mathsf{Q}_n)^{\mathsf{cl}}$ is $\mathcal{L}_{(\mathsf{Q}_1,\ldots,\mathsf{Q}_n)^{\mathsf{cl}}}$.

Proof. Let A_i^{i+1} denote $(\mathsf{Q}_i, \mathsf{Q}_{i+1})^{\mathsf{cl}}$.

(⊆) Suppose $w \in \mathcal{L}_{(\mathsf{Q}_1,\ldots,\mathsf{Q}_n)^{\mathsf{cl}}}$. Then $w = w_1 \oplus \cdots \oplus w_n$ with $w_i \in L(A_i^{i+1})$. Thus $\delta(s_i^{i+1}, w_i) = f_i^{i+1} \in F_i^{i+1}$ for each $i \leq n-1$, and $\delta(f_i^{i+1}, \oplus) = s_{i+1}^{i+2}$ for each $i \leq n-2$, so $\delta(s, w) \in F$.

(⊇) Suppose $\delta(s, w) \in F$. By construction, any path from s to some $f \in F$ consists of $s_1^2 \cdots f_1^2 \oplus \cdots \oplus s_{n-1}^n \cdots f_{n-1}^n$, with each $f_i^{i+1} \in F_i^{i+1}$. Clearly w must consist of $W_i \in L(A_i^{i+1})$ followed by $\oplus$ (with no $\oplus$ after w_n).

□

Summary and Open Questions

In this chapter we saw how to construct automata recognizing the cumulation of two or more regular quantifiers. We had to use a different definition for our translation function, allowing both R and R^{-1}. In Section 4.1 we asked how adding power to the translation function changes the complexity of a polyadic quantifier. The results of this chapter yield at least this much of an answer: *if* we use τ_2^{cl} (τ_n^{cl}) in lieu of τ_2 (τ_n), cumulations of regular quantifiers are DFA-computable. Defining this new τ_2^{cl} is the natural definition to analyze the cumulation $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)^{\mathsf{cl}}$ as the sequential composition of $\mathsf{Q}_1 \cdot \mathsf{some}$ and $\mathsf{Q}_2 \cdot \mathsf{some}$, since the latter must use the inverse relation.

Question 7.2.8. Are cumulations of regular quantifiers still DFA-computable if we use the original τ_2? This would require counting Q_2-many 1's in distinct positions in subwords. Already for cumulation, the "easy" irreducible lift, we encounter the need to identify symbols in different subwords, which does not make sense if we cannot enforce a uniform length restriction on subwords.

Question 7.2.9. If cumulations of regular quantifiers are not DFA-computable using τ_2, how much more computing power is required? Can we adequately count those 1's with a stack, or are cumulations not even context-free?

More generally:

Question 7.2.10. What kind of automata are needed to analyze the polyadic lifts we have yet to discuss in detail (resumptions, reciprocals, branching, etc.)? Does extending τ_2 go some distance toward reining them in?

We hope to shed some light on these questions in the next chapter.

Finally, on a more technical note:

Question 7.2.11. We introduced extra separator symbols in defining τ_2^{cl}. Are they strictly necessary to define cumulation DFA? Can we give a construction from two or more minimal DFA to a *minimal* cumulation DFA without these extra symbols? Would this change affect the cumulation closure of DCFLs?

It's not clear to us that such a general definition, in the spirit of the other constructions of this thesis, can be given. Definitively deciding that this is or is not the case may be an interesting problem.[1] However, cumulation automata without transitions for separators indicating the difference between the translation of R and the translation of R^{-1} do not really have the model-recognition power that we desire. They would accept all the strings that are actually translations of models in which the cumulation is true, but also possibly more: there would be no way to enorce that a subword corresponding to part of R^{-1} cannot count as a witness of $\mathsf{Q}_1{\cdot}\mathsf{some}$. This question of how tightly the language of a semantic automata should correspond to all and only the models in which the quantifier is true is also taken up again in the next chapter.

[1]If one did undertake such an endeavor, HaLeX, the tool discussed in Appendix A, would likely be indispensable for generating a multitude of correct examples from which to generalize.

Chapter 8

Toward A Novel Characterization of the Frege Boundary

In this chapter we propose that the main project of this thesis, extending the semantic automata model, provides an interesting medium for reconsidering results about quantification concerning the Frege boundary. The Frege boundary demarcates the line between reducible and irreducible polyadic quantifiers. Historically, the boundary bears the appellation of Frege because he introduced the familiar notion of quantification to modern logic. He was the first to give any satisfactory analysis of multiple quantification, by simply taking every instance of multiple quantification to be an iteration. Van Benthem [6] calls this "solving the problem by ignoring it," since on this view we can give an account of any polyadic quantifier in terms of simple monadic quantifiers. Thus these polyadic quantifiers that can be analyzed as iterations of monadic quantifiers are deemed *reducible*, or simply *Fregean*. Those that can be given no such analysis are *irreducible* or *non-Fregean*, and may be considered *genuinely polyadic*.

Recall the overview of polyadic quantification in Section 2.1.2. Iterations represent a kind of default, the "bread and butter" of multiple quantification in natural language. In logical notation, $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)(A, B, R)$ is simply $\mathsf{Q}_1 a \mathsf{Q}_2 b R(a, b)$. It's as "easy" as prefixing Q_2 by Q_1, and thus iteration is *monadically definable*: this is the sense in which the lift is not taken to be genuinely polyadic. The other lifts we looked at–cumulation, resumption, reciprocals, branching, and let us now add constructions containing *same* and *different*, among others–are generally not reducible to iterations. See Westerståhl [56] for the results that cumulation, resumption, and branching are only iterations for very simple choices of monadic quantifiers, like some and every. For example, the cumulation $(\mathsf{Q}_1, \ldots, \mathsf{Q}_n)^{\mathsf{cl}}$ is an n-ary iteration if and only if, on every model where

it is non-trivial, Q_2 through Q_n are all some. Consider $(\mathsf{five}, \mathsf{some})^{\mathsf{cl}}$, true if both $\mathsf{five} \cdot \mathsf{some}$ holds of R *and* $\mathsf{some} \cdot \mathsf{some}$ holds of R^{-1}: the first conjunct already implies the second. In general, if we write the truth conditions of an irreducible polyadic quantifier in terms of the relation R, they somehow depend on the set R_a for multiple a "at once," not fitting the iteration definition $\mathsf{Q}_1(A, \{a : \mathsf{Q}_2(B, R_a)\})$.

Now that we may discuss polyadic quantification from an automata and formal language perspective, we find there is actually an as-yet unanswered question with respect to the well-studied Frege boundary, namely: *where is the Frege boundary located in the Chomsky hierarchy*? In this chapter we take the first steps toward locating the analog of this boundary for *languages* of genuinely polyadic quantifiers.

Linking the Frege boundary to the Chomsky hierarchy, given the practical repercussions of identifying the automata complexity of a natural language quantifier (see Section 9.1), may further elucidate how we might interpret the original results. The boundary tells us whether some polyadic quantifier is essentially monadic, but doesn't have anything to say about how complex those constitutents are, or how their complexity may contribute to that of the iteration[1], but if some irreducible quantifier language is particularly easy or particularly difficult to recognize or learn (as demonstrated by its corresponding automata model), we have independent reason to care that natural language contains genuinely polyadic quantification.

First we present an overview of the evolution of Frege boundary characterizations, alongside reformulations of those results in terms of the vocabulary for polyadic quantifier languages that we have developed in this thesis. This makes it possible to talk about the languages and automata of quantifiers as themselves reducible or irreducible, and also reveals some assumptions and difficulties underlying the very project of extending the semantic automata model to the polyadic realm. We motivate the claim that, under those assumptions, irreducible languages are not context-free (a very conservative lower bound). We discuss the difficulty of finding a general theorem characterizing such languages that does not make so many assumptions, using our reformulations to discuss the problems faced by such a search in precise language.

[1]Says van Benthem, "The central issue is whether or not certain polyadic patterns have a natural decomposition into their unary components–and not so much whether they are *first-order definable*. Indeed, not all Fregean iterations are first-order, nor all genuine polyadic higher-order" [6].

8.1 A Brief History of the Boundary and Reformulation of Reducibility Results

To facilitate discussing the Chomsky analogue of the Frege boundary in a precise way, we now translate the notions used in results about reducible quantifiers into notions about languages. Study of the Frege boundary began around the same time as that of semantic automata. The first characterization, from van Benthem [6], appeals to the invariance properties of relations of which reducible quantifiers hold. A characterization of a different flavor, based on the behavior of a quantifier on relations that are cross-products, comes from Keenan [29].[2] This is generalized and given some intuitive oomph by Dekker [12], and generalized yet further by van Eijck [14].[3]

Membership above the Frege boundary is determined by contrast with membership below the boundary: a quantifier equivalent to one or more iterations of simple quantifiers is Fregean, and everything else is irreducible. Thus we have the following for the formal definition of reducibility:

Definition 8.1.1. ([29],[12]) For a quantifier Q of type $\langle 2 \rangle$, call Q reducible if and only if there are type $\langle 1 \rangle$ quantifiers Q_1 and Q_2 such that $\mathsf{Q} = \mathsf{Q}_1 \cdot \mathsf{Q}_2$. For Q of type $\langle n \rangle$, call Q n-reducible if and only if there are type $\langle 1 \rangle$ quantifiers $\mathsf{Q}_1, \ldots, \mathsf{Q}_n$ such that $\mathsf{Q} = \mathsf{Q}_1 \circ \cdots \circ \mathsf{Q}_n$.

We will define reducibility for languages as being equivalent to the right kind of substitution. Recall (Section 5.2.1) that we say a string is of the form $\boxdot_n$ if it is in $((\cdots(((0+1)^*\boxdot_1)^*\boxdot_2)^*\cdots)\boxdot_n)^*$. By extension, we say a language is of the form $\boxdot_n$ if all of its members are of the form $\boxdot_n$.

Definition 8.1.2. Say $\mathcal{L}_{\mathsf{Q}}$ of the form $\boxdot_1$ is reducible if $\mathcal{L}_{\mathsf{Q}} = s(\mathcal{L}_{\mathsf{Q}_1})$ where $s(0) = \neg\mathcal{L}_{\mathsf{Q}_2}\boxdot_1$ and $s(1) = \mathcal{L}_{\mathsf{Q}_2}\boxdot_1$, with $\mathcal{L}_{\mathsf{Q}_1}$ and $\mathcal{L}_{\mathsf{Q}_2}$ binary quantifier languages. Say $\mathcal{L}_{\mathsf{Q}}$ of the form $\boxdot_{n-1}$ is n-reducible if $\mathcal{L}_{\mathsf{Q}} = s(\mathcal{L}_{\mathsf{Q}_1\circ\cdots\circ\mathsf{Q}_{n-1}})$ where $s(0) = \neg\mathcal{L}_{\mathsf{Q}_n}\boxdot_1$, $s(1) = \mathcal{L}_{\mathsf{Q}_n}\boxdot_1$, and $s(\boxdot_i) = \boxdot_{i+1}$, with $\mathcal{L}_{\mathsf{Q}_n}$ a binary quantifier language and $\mathcal{L}_{\mathsf{Q}_1\circ\cdots\circ\mathsf{Q}_{n-1}}$ $(n-1)$-reducible.

Steienert-Threlkeld [45] shows that for regular (deterministic context-free) $\mathcal{L}_{\mathsf{Q}}$ of the form $\boxdot_1$, it is decidable whether the language is reducible to the iteration of two binary regular (deterministic context-free) languages. He further conjectures that this problem is *undecidable* if $\mathcal{L}_{\mathsf{Q}}$ is (non-deterministic) context-free, since the positive results depends on the decidability of testing language equivalence.[4]

---

[2] See also [28] for a kind of toolkit of theorems that are precursors to his more general characterization.

[3] For this section we break with the variable-naming notation in the literature for readability and consistency with the rest of the thesis. However, note that we do switch between quantifiers of type $\langle 2 \rangle$ (or $\langle n \rangle$) and $\langle 1,1,2 \rangle$ (or $\langle 1,\ldots,1,n \rangle$) in reformulating the results, but recall that for CE quantifiers, we can convert from one to the other by relativization or freezing (Section 2.1.1).

[4] We do not go into more detail since we are here interested in a different question: given

We proceed chronologically, beginning with van Benthem's characterization, based on the properties of *logicality* and *right-orientation*:

Theorem 8.1.3. ([6]) On any finite universe, a binary quantifier Q is a *right complex* (a Boolean combination of of iterations) if and only if it is both logical and right-oriented.

A quantifier is logical if closed under permutations of individuals: $R \in \mathsf{Q}$ if and only if $\pi(R) \in \mathsf{Q}$. If $S = \pi(R)$, we write $S \approx R$ and say Q is closed under $\approx$. A quantifier is right-oriented if closed under $\sim$, where we write $R \sim S$ if for all x, $| R_x |=| S_x |$. This corresponds to preserving the entire arrow pattern of a relation, and preserving the outgoing arrow pattern of a relation, respectively.[5]

Figure 8.1: Illustration for Example 8.1.4

Example 8.1.4. See the relations depicted in Figure 8.1. Since $| R_{a_1} |=| S_{a_1} |$, $| R_{a_2} |=| S_{a_2} |$, and $| R_{a_3} |=| S_{a_3} |$, we have $R \sim S$. This means that if a binary iteration (and more generally, a right complex) holds of R, it must also hold of S (and vice versa). Observe that for instance the iterations $\mathsf{every} \cdot \mathsf{some}$ and $\mathsf{three}{\cdot}\mathsf{exactly\ one}$ hold of both R and S, while *every A R's different B* holds of R but not S and *every A R's the same B* holds of S but not R.

In order to state the invariance properties of logicality and right-orientation in terms of languages, we must assume that subwords have the same length. Hence we give a more restricted notion of quantifier language, which amounts to requiring that every string is actually the translation of some model.

Definition 8.1.5. Let Q be a type $\langle 1,1,2 \rangle$ quantifier. Let $\mathcal{L}_{n,m}$ be the set of strings $\tau_2(\vec{a}, \vec{b}, R)$ for $(A, B, R) \in \mathsf{Q}$ such that $n =| A |$ and $m =| B |$. Then strings of $\mathcal{L}_{n,m}$ are of the form $(w_i \boxdot)^n$ with $| w_i |= m$ for all i. The language of Q is:

$$\mathcal{L} = \bigcup \mathcal{L}_{n,m} \text{ for } n, m \in \mathbb{N} \cup \{0\}$$

Definition 8.1.6. Let $w \in \mathcal{L}_{n,m}$ for some n and m. Set $w' \approx w$ if there exists $\pi = \pi_n \cup \pi_m$ (where π_n and π_m are independent permutations on $\{1, \ldots, n\}$ and $\{1, \ldots, m\}$)[6]

that a language is *not* reducible, how hard is it?

[5]Throughout we assume permutations apply to the domain and range of the relation separately, which we continue to denote with A and B, for ease of definition, but it's not essential.

[6]The idea is that the domain and range of the translated relation do not change. For, e.g., reciprocals, which are defined on a single set, there should be a single permutation.

such that for all w_i:

$$w_i[j] = w'_{\pi_n(i)}[\pi_m(j)]$$

Say $\mathcal{L}_{n,m}$ is logical if closed under $\asymp$, and $\mathcal{L}$ is logical if $\mathcal{L}_{n,m}$ is closed under $\asymp$ for all n and m. Logicality for languages of polyadic quantifiers does not quite amount to permutation closure, as it did in the monadic case. Instead, entire ⊡-ended chunks may be moved around, and individual symbols within *each* ⊡-ended chunk may be moved around (uniformly). Call the former an *outer* permutation, corresponding to a shuffle of individuals in A, and the latter an *inner* permutation, corresponding to a shuffle of individuals in B.

Definition 8.1.7. Let $w \in \mathcal{L}_{n,m}$ for some n and m. Set $w' \smile w$ if there exists $\pi : \{1, \ldots, n\} \to \{1, \ldots, n\}$ such that $\#_1(w'_i) = \#_1(w_{\pi(i)})$ and $\mid w'_i \mid = m$ for all $i \leq n$. Say $\mathcal{L}_{n,m}$ is right-oriented if closed under $\smile$, and $\mathcal{L}$ is right-oriented if $\mathcal{L}_{n,m}$ is closed under $\smile$ for every n and m.

Example 8.1.8. The models in Figure 8.1 yield the strings w = 100⊡010⊡001⊡ and v = 100⊡100⊡100⊡ respectively. The string w is in $\mathcal{L}_{3,3}$ for the language of three · exactly one (for instance). Since that is a logical quantifier, we know that, for example, w' = 010⊡001⊡100⊡ is also in $\mathcal{L}_{3,3}$ since each w'_i is obtained from w_i by the inner permutation $\pi : i \to (i \bmod 3) + 1$ with the identity mapping for the outer permutation (for example). Thus $w' \asymp w$. Moreover, since three·exactly one is reducible by definition, we know v is in $\mathcal{L}_{3,3}$ *even though* there are no permutations by which v is obtained from w. $\mathcal{L}_{3,3}$ is closed under $\smile$, and each v_i has the same number of 1's as w_i.

But now consider these same strings and the language of (three, *same*), which is true of a model (A, B, R) if and only there are a_1, a_2, a_3 and b such that (a_1, b), (a_2, b), and (a_3, b) are in R.[7] This quantifier is still logical, and thus does not distinguish between w and w' (rejecting both), but it *does* distinguish between w and v, hence lacking right-orientation, due to the requirement that b be identical in the three witnesses.

Van Benthem's theorem holds for *local* (on a particular finite universe) definability, but can be used to refute definability on *any* universe.[8] We illustrate the difference between unary compounds and iteration with an example from Westerståhl [56].

Example 8.1.9. Res^2(at least n) is a right unary complex, but not an iteration, for all $n \geq 2$. This type $\langle 2 \rangle$ resumption expresses that a relation R contains at least n pairs:

Res^2(at least n)$(R) \Leftrightarrow$

[7] The exact semantics are debatable, but constructions containing *same* can take on a wide variety of meanings while remaining irreducible.

[8] Thus this characterization cannot be used to show that a given resumptive quantifier or constructions with *the same number of* are irreducible, since these are right-oriented, and indeed definable on particular universes by enumerating the possibilities, while not being generally expressible on arbitrary universes.

$$\bigvee_{1 \le k \le n-1} \Big((\mathsf{exactly\ k \cdot some})(R) \wedge \mid R \mid \ge n \Big) \vee (\mathsf{at\ least\ n \cdot some})(R)$$

If the second disjunct holds, then we know R has at least n pairs because there are at least n different individuals in its domain. If the first disjunct holds, then there are $k < n$ different individuals in the domain of R, and some of them must be in the relation with multiple individuals in the domain–so we must write this disjunct as a unary complex. For each k, we simply enumerate the possible iterations, which are all the partitions of n with k parts. Thus we replace the first disjunct with $(\mathsf{exactly\ k \cdot some}) \wedge (\psi_1 \vee \cdots \vee \psi_r)$ for each k, where the ψ_i are the right complexes enumerating the possibilities. For example, for $n = 10$ and $k = 6$, one of the partitions is 3+3+1+1+1+1, so one of the *psi*'s will be $(\mathsf{at\ least\ 2 \cdot at\ least\ 3})$ (note that $(\mathsf{at\ least\ 4 \cdot at\ least\ 1})$ is implied by $(\mathsf{exactly\ 6 \cdot some})$, so we don't have to explicitly list that to know that R has at least 10 pairs in this case).

Right-orientation implies logicality (if $R \approx S$, then also $R \sim S$, so if Q is closed under $\sim$ it is also closed under $\approx$). We observe that the same holds for languages: if $w \approxeq w'$, then of course $w \smile w'$, so if $\mathcal{L}$ is closed under $\smile$ we can conclude it is closed under $\approxeq$.

Indeed, it is easy to see that our definitions of invariance conditions on languages fully capture the corresponding conditions on quantifiers.

Claim 8.1.10. Q is closed under $\approx$ ($\sim$) if and only if its language $\mathcal{L}$ is closed under $\approxeq$ ($\smile$).

Proof. We show for $w = \tau_2(\vec{a}, \vec{b}, R)$ and $w' = \tau_2(\vec{a'}, \vec{b'}, R')$, (i) $w \approxeq w' \Leftrightarrow R \approx R'$ and (ii) $w \smile w' \Leftrightarrow R \sim R'$.

(i) Construct the permutation on A and B from the permutations on n and m and vice versa using the indices from the enumerations $\vec{a}$ and $\vec{b}$.

(ii) This follows from the observation that by the definition of τ_2 and τ, $\#_1(w_{\pi(i)}) = \mid R_{\pi(a_i)} \mid = \mid R'_{a'_i} \mid = \#_1(w'_i)$.

□

Note that cumulations are *not* right complexes. They are neither right-oriented, nor left-oriented, but right-and-left oriented: Q has this property if $R \in \mathsf{Q}$, $\mid R_x \mid = \mid S_x \mid$, *and* $\mid {}_yR \mid = \mid {}_yS \mid$ for all x, y, then $S \in \mathsf{Q}$.[9] Replacing right-orientation by right-and-left orientation (left-orientation) in van Benthem's theorem yields reducibility to *unary* (*left*) complex.

Keenan provides a characterization that also applies to non-logical quantifiers and relies on the interesting observation that if two reducible quantifiers behave

[9] ${}_yR$ is the set of x such that $(x, y) \in R$, i.e. R_y^{-1}.

the same on relations that are cross-products, they actually behave the same on every relation (i.e. are equivalent).[10]

Theorem 8.1.11. (Type ⟨2⟩ Reducibility Equivalence, [29]) For reducible type ⟨2⟩ quantifiers Q and Q', $\mathsf{Q} = \mathsf{Q}'$ if and only if for all subsets A, B of M, $\mathsf{Q}(A \times B) = \mathsf{Q}'(A \times B)$.

The following equivalent statement of the theorem provides a test for reducibility: if $\mathsf{Q}(A \times B) = \mathsf{Q}'(A \times B)$ for all $A, B \in \mathcal{P}(M)$, and we know $\mathsf{Q}' = \mathsf{Q}_1 \cdot \mathsf{Q}_2$, then Q is reducible if and only if $\mathsf{Q} = \mathsf{Q}_1 \cdot \mathsf{Q}_2$.

Dekker then generalizes this to quantifiers of arbitrary arity:

Theorem 8.1.12. (Type $\langle n \rangle$ Reducibility Equivalence, [12]) For type $\langle n \rangle$ quantifiers Q and Q' that are n-reducible, $\mathsf{Q} = \mathsf{Q}'$ if and only if for all subsets $A_1, \dots, A_n$ of M, $\mathsf{Q}(A_1 \times \dots \times A_n) = \mathsf{Q}'(A_1 \times \dots \times A_n)$.

Again we can restate the theorem to conclude that if Q and Q' have the same behavior on cross-products and Q' is reducible, then Q is reducible only if it equals Q'. Dekker also defines Q to be *invariant for sets in products* if $\mathsf{Q}(A_1 \times \dots \times A_n)$ and $\mathsf{Q}(A'_1 \times \dots \times A'_n)$ imply $\mathsf{Q}(A_1 \times \dots \times A'_i \times \dots \times A_n)$ and shows Q is invariant for sets if and only if it is product equivalent to some $\mathsf{Q}' = \mathsf{Q}_1 \circ \dots \circ \mathsf{Q}_n$.[11] Furthermore, the proof actually constructs the Q_i, widening the applicability of the Keenan-style reducibility test by removing the problem that "maybe one has not tried hard enough" to find the product-equivalent iteration for comparison.

Example 8.1.13. Consider the sentence *Every professor wrote the same number of recommendation letters*, formalized as $(\mathsf{every}^P, \textit{same number}^L)(W)$. This is product-equivalent to $(\mathsf{every}^P \cdot \mathsf{every}^L)(W)$, since when W is a cross-product relation, every p is always connected to every l, and thus incidentally every p is connected to the same number of l. Since these quantifiers are not the same (take a model in which every p is connected to the same number of l, but $|W_p| < |L|$), $(\mathsf{every}, \textit{same number})$ is not reducible to *any* two unary quantifiers.

Dekker nicely sums up what cross-product characterizations tell us about iterations:

> Not only is this a new and welcome generalization, it also gives some insight into the intimate relation between (n)-reducible type $\langle n \rangle$ quantifiers and n-ary product relations. If type $\langle n \rangle$ quantifier F^n is (n)-reducible…then F^n is satisfied by $\mathcal{Q}_1 \times \dots \times \mathcal{Q}_n$ iff each composing f_i is satisfied by $\mathcal{Q}_i$ [12].

Van Eijck's work ([14]) introduces the notion of (m, n)-reducibility, making it possible to say something about polyadic quantifiers of type $\langle m + n \rangle$ that are

[10] We will stick to van Benthem's properties in the rest of this chapter, but we present the history and reformulations of cross-product characterizations for completeness.

[11] We can not find a natural language example where this is useful, but it is needed to show that the property of a relation being symmetric, which is product-equivalent to no iteration, is not reducible. $A \times A$ and $B \times B$ are symmetric, but neither of $A \times B$ nor $B \times A$ is [12].

not fully $(m+n)$-reducible.

Definition 8.1.14. Q of type $\langle m, n\rangle$ is (m,n)-reducible if there are Q_1 and Q_2 of types $\langle m\rangle$ and $\langle n\rangle$ such that $\mathsf{Q} = \mathsf{Q}_1 \cdot \mathsf{Q}_2$.

Van Eijck also defines the corresponding notions of reducibility equivalence and invariance for sets in products. The striking consequence of generalizing reducibility is the existence of a diamond property and normal form for quantifiers, meaning reducibility is confluent: if a quantifier reduces to two different iterations, these reducts must have a common further decomposition. If Q of type $\langle m+n\rangle$ reduces both to $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ (of types $\langle m\rangle$ and $\langle n\rangle$) and to $\mathsf{Q}'_1 \cdot \mathsf{Q}'_2$ (of types $\langle m'\rangle$ and $\langle m+n-m'\rangle$), then there exists Q_3 (of type $\langle m'-m\rangle$) such that $\mathsf{Q} = \mathsf{Q}_1 \cdot \mathsf{Q}_3 \cdot \mathsf{Q}'_2$.

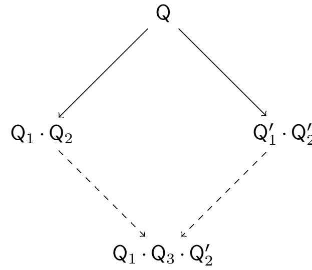


Figure 8.2: Van Eijck's Diamond Property

Example 8.1.15. Consider the sentence *Every teacher assigned different students different problems* analyzed as the type $\langle 3\rangle$ quantifier (every^T, *different*^{S},*different*^{P}) applied to the *assign* relation, and let $\mathsf{0}$ denote the unary quantifier that is false of every set. By Dekker's results we can see this is not fully 3-reducible, since it is equivalent to $\mathsf{every} \cdot \mathsf{0} \cdot \mathsf{0}$ on cross-products (i.e. it is true of no cross-product), but obviously is not generally equal to $\mathsf{every} \cdot \mathsf{0} \cdot \mathsf{0}$, since we can construct non-cross-product relations on which it *is* true. However, by van Eijck's results we can also state a positive result, that it is in fact $(1,2)$-reducible, equivalent to $\mathsf{every}\cdot$(*different*,*different*). Further, we know it cannot also be $(2,1)$-reducible to some type $\langle 2\rangle$ Q_1 and type $\langle 1\rangle$ Q_2, or else by the diamond property there would exist some type $\langle 1\rangle$ Q_3 making it 3-reducible to $\mathsf{every} \cdot \mathsf{Q}_3 \cdot \mathsf{Q}_2$, a contradiction.

To translate the characterizations based on cross-product behavior, define the languages $\mathsf{CROSS-PROD}_{\boxdot_n} = \{(\cdots((1^*\boxdot)^*\boxdot_2)^*\cdots\boxdot_n)^*\}$. For example, in the binary case we use $\mathsf{CROSS-PROD}_{\boxdot_1} = \{(1^*\boxdot_1)^*\}$ which is equivalent to $\mathsf{every} \cdot \mathsf{every}$. Then we can restate reducibility equivalence as:

Claim 8.1.16. Let $\mathcal{L}$ and $\mathcal{L}'$ be the languages of type $\langle 2\rangle$ quantifiers Q and Q', respectively, where it is known that $\mathsf{Q} = \mathsf{Q}_1 \cdot \mathsf{Q}_2$ is reducible. If $\mathcal{L} \cap \mathsf{CROSS-PROD}_{\boxdot_1} = \mathcal{L}' \cap \mathsf{CROSS-PROD}_{\boxdot_1}$, then if $\mathcal{L}' \neq \mathcal{L}$, Q' is not reducible. In general, for Q and Q' of type $\langle n\rangle$ and Q (n)-reducible with $\mathcal{L} \cap \mathsf{CROSS-PROD}_{\boxdot_n}$

$= \mathcal{L}' \cap \mathsf{CROSS-PROD}_{⧇_n}$, Q' is reducible if and only if $\mathcal{L}' = \mathcal{L}$.

8.2 Genuinely Polyadic Quantifier Languages are Not Context-Free

In trying to characterize the languages of non-Fregean quantifiers, it is almost immediately clear that, at minimum, they cannot be context-free. For the language of a non-Fregean quantifier to even *make sense*, two things must happen: (1) the subwords must all have the same length, and (2) the indices of the subwords must be related. (2) demands that (1) be the case, but (1) on its own is bad enough for context-freeness. A simple pumping lemma argument demonstrates that no language with an arbitrary number of equal-length subwords is context-free.

Fact 8.2.1. Let $\mathcal{L}$ be any language over $\Sigma = \{0, 1, ⧇\}$ with the requirement that for every $w \in \mathcal{L}$, w has the form $(w_i⧇)^*$ with $w_i \in \{0,1\}^*$ and $\mid w_i \mid = \mid w_j \mid$ for all i, j. $\mathcal{L}$ is not context-free.

Proof. Assume $\mathcal{L}$ is context-free. Then by the pumping lemma for context-free languages, there is a constant p such that for every $w \in \mathcal{L}$, w can be written as $uvxyz$ where:

(i) $\mid vxy \mid \leq p$

(ii) $\mid vy \mid \geq 1$

(iii) $uv^jxy^jz \in \mathcal{L}$ for all $j \geq 0$

Let $w = ((0+1)^{p_m}⧇)^n$ for any $n \geq 3$ and $p_m \geq p$ such that $w \in \mathcal{L}$ (in case $\mathcal{L}$ is empty for some choices of superscripts). Then $w = uvxyz$ where v and y can be removed or repeated any number of times to result in another string in $\mathcal{L}$.. Since $n \geq 3$, the result of pumping w can only be another string in $\mathcal{L}$ if each of v and y is ϵ or consists of exactly one or more entire subwords $w_i⧇$ (if n were 1, the longer or shorter subword might be in $\mathcal{L}$; if n were 2, x and y could be on either side of the first ⧇ and again potentially result in another string of $\mathcal{L}$). But by (ii) at least one of them is not ϵ, and by (i) neither is long enough to cover a whole subword. Thus pumping must result in at least one subword that differs in length from the rest, so we have a contradiction, and $\mathcal{L}$ is not context-free. □

However, it is unclear exactly how this fact can or should be used to decide the complexity of irreducible quantifier languages. We cannot otherwise make adequate sense of their meaning, but on the other hand, we only ever wish to evaluate strings having this property, for any kind of quantifier (that is, having equal-length subwords is a necessary by-product of being a model translation generated by τ_2). The languages of iterations allow subwords of varying lengths

because we can identify them with the languages of DFA and (D)PDA, which can impose no such restriction. If we require that strings both satisfy the constraint imposed by the quantifier *and are actually proper translations of models*, no polyadic quantifier is context-free. For the sake of investigation and perhaps out of fairness, we for the moment take this property as given and seek a different method to show non-context-freeness.

One may object at this point that it makes perfect sense to relax the equal-length assumption in defining right-orientation, which after all only depends on the number of 1's in subwords, and not where they occur; thus, "spill-over" strings that don't belong to any particular $\mathcal{L}_{n,m}$ but satisfy the binary relation defining the quantifier are still meaningful in iterated languages, and right-orientation implies logicality, so we should take the equal-length requirement on irreducible languages at face-value and rest satisfied. There are two responses to this objection. First, the problem with relaxing the length assumption is that we lose the tidy correspondence with models. It is implicit that R and S are defined on the same sets when we write $| R_x |=| S_x |$ in the definition of right-orientation of a quantifier. There is no possibility of reverse-engineering a model from a string with unequal subwords. Second, and related to the first response, sometimes these "spill-over" strings are meaningful when intuitively, they should not be. Perhaps it is fine to have "extra" 0's in subwords of strings in $\mathcal{L}_{\mathsf{every \cdot some}}$, but *every* symbol counts in subwords of strings in $\mathcal{L}_{\mathsf{every \cdot half}}$–so what do we make of those strings?

Moving onward, can we make anything of the requirement that the symbols of subwords at the same indices can be taken to indicate the same elements of B? How can we formalize the sense in which the indices must "match"? As we saw in the previous section, a language's depending on the actual indices of the 1's rather than their numerosity amounts to having logicality closure while lacking right-orientation closure. Context-free languages can only enforce dependencies between two parts of a string, and these dependencies must be non-crossing–that is, they limit *cross-serial* dependency and allow unbounded dependency only if *nested*. That is why, for example, $\{a^n b^n c^m d^m : n, m > 0\}$ and $\{a^n b^m c^m d^n : n, m > 0\}$ are CFLs but $\{a^n b^n c^n : n > 0\}$ and $\{a^n b^m c^n d^m : n, m > 0\}$ are not. Any kind of relation between indices of subwords requires an arbitrary number of dependencies across arbitrarily many parts of the string, which are moreover necessarily crossing. But do we really want to require that all the symbols at index 1 are related, and all the symbols at index 2, and so forth, for an unbounded number of symbols at an unbounded number of indices? Simply put, that seems *too* hard. Moreover, reducible languages are *also* logical! Their possessing the stronger property of right-orientation removes the need to enforce logicality, and we don't know that there is not some similarly stronger property mitigating the requirement of logicality for irreducible languages. It is too simple to say that the property of logicality results in such a severe restriction.

8.3 Approaching a General Theorem for a Lower Limit

As we saw in Section 8.2, it is *obvious* that languages of non-Fregean quantifiers are not context-free, but that this requires some assumptions which may be unfair to make only of irreducible languages. We seek a theorem for a Chomsky hierarchy characterization that exhaustively covers all irreducible quantifier languages without overstating those assumptions. Note that a case-by-case analysis is all the more daunting, since there is great diversity in the kinds of constructions leading to irreducible quantifiers (see e.g. Keenan's [28] and [29] for examples of the many variations).

Let's observe just how difficult it is to state some of these irreducible quantifier languages consisting of strings generated by τ_2. In the following, let it be assumed that all w are of the form $(w_i \boxdot)^*$ with $w_i \in \{0,1\}^*$ and $|\, w_i \,|=|\, w_j \,|$, define $I_w = \{1, \ldots, |\, w_i \,|\}$ (the set of indices of w's subwords) and $I_{w_j} = \{i \in I_w : w_j[i] = 1\}$ (the set of indices at which w_j has a 1).

- Cumulation: A string is in the language of the cumulation of Q_1 and Q_2 if it is in the language of $\mathsf{Q}_1 \cdot \mathsf{some}$ and there are Q_2-many 1's in *distinct* indices among the subwords.

$$\mathcal{L}_{(\mathsf{Q}_1,\mathsf{Q}_2)^{\mathsf{cl}}} = \{w : w \in \mathcal{L}_{\mathsf{Q}_1 \cdot \mathsf{some}} \text{ and } (|\, I_w - \bigcup I_{w_i} \,|, |\, \bigcup I_{w_i} \,|) \in \mathsf{Q}_2^c\}$$

- A *same* construction: for simplicity, with the meaning of *only the same one*. A string is in the language of $(\mathsf{Q}, \mathit{same})$ if there are Q-many subwords that all have a single 1 and at the same index.

$$\mathcal{L}_{(\mathsf{Q},\mathit{same})} = \{w : \exists i \in I_w \text{ s.t. } (|\, \{w_j : I_{w_j} = i\} \,|, |\, \{w_j : I_{w_j} \neq i\} \,|) \in \mathsf{Q}^c\}$$

- Strong reciprocal: A string is in the language of $\mathsf{Ram_S}(\mathsf{Q})$ if there is a Q-large subset of indices such that every subword corresponding to an index in the set has 1's at all the other indices in the set.

$$\mathcal{L}_{\mathsf{Ram_S}(\mathsf{Q})} = \{w : \exists I_{\mathsf{Q}} \subseteq I_w \text{ s.t. } \forall i \in I_{\mathsf{Q}},\, w_i[I_{\mathsf{Q}} - i] = 1 \text{ and } (|\, I_w - I_{\mathsf{Q}} \,|, |\, I_{\mathsf{Q}} \,|) \in \mathsf{Q}^c\}$$

The preceding definitions depend on the indices of the 1's in each subword, and thus on the identities of the $b \in B$ they represent (in a relative sense: conditions on indices in multiple subwords means that there are comparisons between $b \in R_a$ and $b \in R_{a'}$ for distinct a and a').[12] It's clear that none of these languages are PDA-computable: even if a PDA could use its stack to track the relevant index information of one subword, it would have to pop its stack to compare against indices of the next subword, thus "forgetting" the important information after two subwords.

[12]Don't be fooled by the language of cumulation: the $\bigcup$ operation obscures the fact that it is the cardinality of the set of *distinct* indices that determines language membership, and not cardinality simpliciter.

Now we state the conclusion of Fact 8.2.1–*languages of irreducible quantifiers are not context-free*–in terms of the terminology introduced in Section 8.1 and investigate whether this lends insight into the possibility of a formal proof that takes equal-length subwords for granted and demonstrates non-context-freeness by other means, based on the lack of anonymity of symbols of subwords (but without outright requiring an arbitrary number of cross-serial dependencies).

To state this using the terminology created for van Benthem's reducibility characterization, we have to weaken the statement to apply to right complexes instead of iterations. That is, if a (logical) type $\langle 1, 1, 2\rangle$ quantifier Q is not a Boolean combination of iterations, then it is not context-free. Equivalently, if Q *is* context-free, then it is right-oriented, hence a right complex (possibly an iteration). In terms of languages:

If $\mathcal{L}_{\mathsf{Q}}$ is closed under $\overset{\smile}{\smile}$ and context-free, then $\mathcal{L}_{\mathsf{Q}}$ is also closed under $\smile$.

Toward proving this, take an arbitrary irreducible quantifier language $\mathcal{L}$ over $\{0, 1, \boxdot\}$. $\mathcal{L}$ is closed under $\overset{\smile}{\smile}$ by virtue of being a quantifier language. Suppose there are $n, m \in \mathbb{N} \cup \{0\}$ and w, w' such that $w \smile w'$ (and $w \not\overset{\smile}{\smile} w'$) but $w \in \mathcal{L}_{n,m}$ while $w' \notin \mathcal{L}_{n,m}$. So we have:

1. *every* v s.t. $w \overset{\smile}{\smile} v$ is in $\mathcal{L}_{n,m}$ (thus, in $\mathcal{L}$)
2. *every* v' s.t. $w' \overset{\smile}{\smile} v'$ is *not* in $\mathcal{L}_{n,m}$ (thus, not in $\mathcal{L}$), or else w' would be
3. *and yet*, $w \smile w'$

Can we show $\mathcal{L}$ is not context-free, based on these facts alone? We think it is possible to prove in this manner: intuitively, there is no way to enforce the invariance property of logicality in a context-free language unless it is in fact just a side-effect of right-orientation. This is indeed the case for iterated languages: we may infer logicality from right-orientation. It's not at all clear how to enforce logicality as a stand-alone property of a language. Perhaps there is also some simpler way of inferring logicality of irreducible languages–another general invariance property similar to right-orientation for iterated languages, or a more informative or otherwise more clever translation function.

One such potential general property relates to compositionality. Every polyadic quantifier language seems to involve a "normal" quantifier that restricts the first set in the argument.[13] We witness this in the sample definitions we gave above for cumulation, same, and strong reciprocal languages: the first quantifier in the lift places some requirement on the number of $\boxdot$-ended subwords that satisfy *some* property. For iterations, that property is being in (or not in) the language

[13] Says Keenan, "One observation which may lead to a stronger constraint on the expressible quantifiers of type $\langle 1, 1, 2\rangle$ is the following: So far at least all the expressions (d, d') which induced quantifiers of that type are ones in which d is itself at least homophonous with a standard quantifier. We may in these cases then think of d' as denoting a function mapping standard quantifiers (of type $\langle 1, 1\rangle$) to ones of type $\langle 1, 1, 2\rangle$. Moreover the functions we need here (so far at least) appear fairly highly constrained" [28].

of the second quantifier, making the subwords independent of each other, and allowing us to write the iterated language as a substitution.

Thus in a lift resulting in a type $\langle 1,1,2\rangle$ quantifier, which we will write very generally as (Q, F) with language $\mathcal{L}$, $\mathcal{L}_{\mathsf{Q}}$ seems to contribute the same meaning (requiring a certain number of witnessing subwords, and in no particular order) to $\mathcal{L}$ regardless of whether $\mathcal{L}$ is reducible or irreducible. $\mathcal{L}_{\mathsf{Q}}$ supplies that meaning to an iterated language by being substituted into. Can $\mathcal{L}_{\mathsf{Q}}$ supply that same meaning to an irreducible language *without* the result being a substitution? If we knew that every polyadic quantifier language is essentially a substitution, we might try to show non-context-freeness of non-Fregean languages by proving that any logical context-free quantifier language over $\{0, 1, \boxdot\}$ is a substitution if and only if it's an iteration. In any case, the uniformity of the constraints placed by $\mathcal{L}_{\mathsf{Q}}$ should be further explored as additional information from which to derive a general theorem about the Chomsky complexity of irreducible languages.

Summary and Open Questions

In this chapter we reviewed how various characterizations of the Frege boundary have built upon each other over the years, and how we can link the boundary with the semantic automata paradigm by translating those results into statements about languages. We argued that irreducible languages are at least non-context-free, but that it is difficult to formally prove this claim without making assumptions that might rightly be made of *any* polyadic quantifier language. This chapter opens up many interesting questions to explore in future research.

Question 8.3.1. The Frege boundary is above context-freeness in the Chomsky hierarchy, but where exactly is it?

Question 8.3.2. Reducibility to unary components and irreducibility are binary (yes or no) notions, but irreducible languages probably come in different flavors of difficulty. How can we further stratify languages of irreducible polyadic quantifiers in terms of the Chomsky hierarchy?

Maybe Chomsky hierarchy characterizations could be more easily obtained by finding suitable automata models, as this is often somehow more intuitive, especially given that it's not obvious how to go about finding non-iterated grammars for polyadic quantifier languages. For example, a *queue automaton* (an automaton with the functionality of a PDA, but with the first-in-last-out queue data structure in lieu of the first-in-first-out stack data structure) seems an appropriate way to handle the cross-serial dependencies we encountered. But that already yields far too much computing power: a queue automaton is Turing universal![14]

[14] That equivalence is a common introductory theory of computation exercise; see for example Exercise 3.14 in [44].

Question 8.3.3. The representation of a problem plays a crucial role in its difficulty. How does representation, i.e. the translation function for models, affect difficulty here?

For example, we saw in Chapter 7 that the cumulation of regular quantifiers is DFA-computable given a strengthening of τ_2 that allows translation of R^{-1}. Conversely, in this chapter we saw that restricting model representations with τ_2 makes the cumulation of two regular quantifiers not even context-free.[15] Of course, there is no mandate that we must use τ_2. It makes sense that τ_2 perfectly captures the information relevant to recognizing an iteration–it is the natural extension of τ, and iteration is the natural extension of monadic quantification. Are there ways of representing models that are more appropriate to recognizing irreducible quantifiers, corresponding to the use of R^{-1}, R^*, or some entirely different possibility? How would the languages of specific quantifiers be affected by such extensions, and how would the Frege boundary move up or down the Chomsky hierarchy as a result? We further consider the importance of these questions in Section 9.2.2, where we suggest the centrality of representations to continuing to extend the semantic automata model to other types of polyadic quantification.

[15]That is, any fully general operation on DFA for regular quantifiers resulting in an automaton for their cumulation cannot yield a PDA; the introduction to this chapter shows that some non-trivial but "easy" cumulations are in fact iterations.

Chapter 9

Practical Relevance and Future Work

9.1 Applications

9.1.1 Model Checking

Verification tasks have long been used in cognitive science in an attempt to study the neural and psychological representations of generalized quantifiers. Such tasks are basically model-checking: presented with a visual scene and a quantified sentence describing it, people are asked to judge whether the sentence is true of the scene. Controlling for features of the scene, scientists can study how particular semantic features affect processing by measuring response time, accuracy, and activation in particular brain regions among other things.

Verification Studies Using Monadic Semantic Automata

McMillan et al. [34] were the first to investigate the neural bases of generalized quantifier comprehension by observing patterns of neuroanatomical recruitment using BOLD fMRI while people assessed the truth-value of a quantified sentence paired with a pictorial scene. McMillan et al. concluded that higher-order quantifiers such as *even* and *most* recruit the prefrontal cortex, including executive resources like working memory, while first-order quantifiers like *some* and *at least three* do not, but that both recruit the right inferior parietal cortex (indicating a numerosity component). They further claimed that this maps onto the distinction between DFA and PDA, the former being memoryless while the latter possesses memory in the form of a stack. Their subsequent study [35] further

supported that automata-based properties of quantifiers are related to the actual neural underpinnings of quantifier comprehension by studying people with particular neurological diseases. They found that patients with FTD (frontotemporal demential) and AD (Alzheimer's disease), which involve working memory limitations, had a harder time understanding higher-order quantifiers, while patients with CBD (corticobasal degeneration), which involves number knowledge impairment, had more trouble than the other groups, for both kinds of quantifiers.

Szymanik [48] points out that their interpretation of those results is not entirely correct since, as we saw in Section 3.2, divisibility or parity quantifiers, while not definable in first-order logic, *are* computable by (looping) finite automaton. Thus parity and proportional quantifiers cannot be lumped together. Szymanik and Zajenkowski [52] created studies to test whether there are interesting correspondences between computational models and logical definability when all the relevant distinctions are made. They compared the following three basic types of quantifiers:

- FO-definable (Aristotelian and counting), computable by acyclic finite automata
- Parity (computable by finite automata with loops)
- Proportional (computable by PDA)[1]

Additionally, they chose a counting quantifier of "high-rank" (requiring counting to at least seven or eight) based on the hypothesis that the number of states of the automaton has a greater impact on resource recruitment than the existence of loops. Fascinatingly, all the predictions based on structural dissimilarities in the automata were actually attested in the response times of the verification task. Proportional quantifiers required the longest time, followed by high-rank cardinals, then parity, and finally Aristotelian. A subsequent study [53] also demonstrated that proportional quantifiers place a higher demand on working memory than parity quantifiers.

The qualitative difference between proportional and other quantifiers is further corroborated by [57], finding that schizophrenic patients perform on par with healthy subjects in verification tasks with the exception of proportional quantifiers. This suggests the semantic automata model gives a partial explanation of the combined working memory and language deficits observed in those with schizophrenia. See also [58] and [59] for further research connecting the processing of proportional quantifiers and working memory.

Next we move on to verification tasks involving multi-quantifier sentences and whether semantic automata for polyadic quantifiers have similar predictive power. But first, we list a few of the interesting open questions remaining concerning the processing of simple quantifiers:

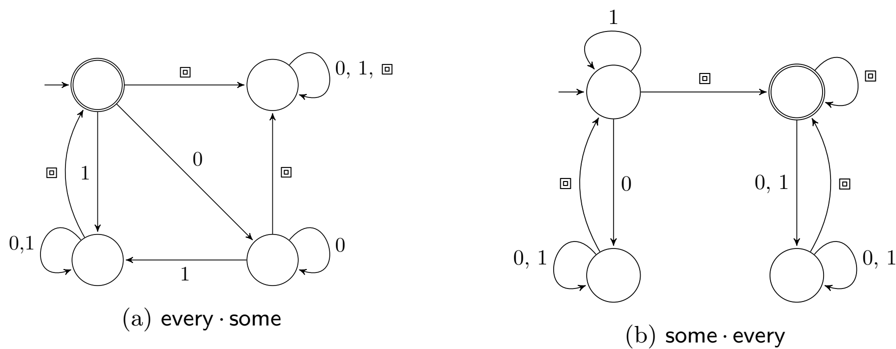
[1] In fact, every quantifier tested so far in this way has been DPDA-computable.

Question 9.1.1. [52] found that, while high-rank cardinals were more difficult than parity quantifiers, these two had the pairwise smallest difference among the types of quantifiers tested. Could further studies decide conclusively whether the number of states or the existence of loops has a greater effect on difficulty? For example, it might be informative to study processing times for cardinal quantifiers of even higher number, and parity quantifiers other than *even* and *odd*.

Question 9.1.2. In light of Kanazawa's recent characterization of nondeterministic PDA (see Section 3.3), it would be interesting to compare processing times for DPDA and NPDA-computable quantifiers (for instance, *more than 1/3* versus *more than 1/3 and less than 2/3*). As it stands, it is unclear how to reconcile the existence of semantic automata positing non-deterministic algorithms with the deterministic nature of human cognition.

Predictions for Multi-quantifier Sentences Using Iterating Semantic Automata

After the publishing of [46] proposing stack iteration automata (presented in Chapter 4)–but before a general mechanism to generate iteration DFA was known–Szymanik, Steinert-Threlkeld, Zajenkowski, and Icard III [51] took the first steps toward answering whether iterated quantifiers actually utilize memory as the stack construction predicts. In particular, they compared stack versions of every · some and some · every with their minimal DFA versions, shown in Figure 9.1 (the only difference is that our depiction of every · some has a complete transition function, so there is an "extra" dead state looping on every symbol).



(a) every · some

(b) some · every

Figure 9.1: Iteration DFA used in multi-quantifier verification study

They predicted that true instances of every · some would be harder than true instances of some · every, since in the former case one must run through the entire model to verify a sentence, and in the latter one need only find a single witness. They further predicted that the opposite relationship would hold for

false instances, since the negations of these iterations become $\mathsf{some} \cdot \mathsf{none}$ and $\mathsf{every} \cdot$ $\mathsf{not\ every}$, respectively.

Interestingly, their results were best explained by positing the stack model for $\mathsf{some} \cdot \mathsf{every}$ and the DFA model for $\mathsf{every} \cdot \mathsf{some}$. True instances of the latter were indeed more difficult, but the expected relation for false instances was not observed. Subjects took longer response time and were less accurate for *every...some* sentences, while the study showed that only *some...every* sentences engaged working memory. If their explanation is correct, it explains the memory engagement observed for *some...every* sentences and suggests a strategy resembling the DFA is less reliable than strategy resembling the PDA, which is plausible.

It remains to be seen *why* these particular results were observed. Now that a completely general method is known for constructing iteration DFA, more empirical predictions comparing the stack and DFA versions may be made by judiciously choosing from the whole gamut of regular iterations. Since differences were already observed when the iterated quantifiers were as simple as some and every, we think there are likely interesting phenomena to observe by studying (1) differences between combinations of different types of quantifiers and (2) for a given type of combination, the difference between the two orderings.

Our initial suspicion was that the DFA model would be appropriate for most simple applications, while a stack-like algorithm would be invoked for more difficult sentences, under various possible interpretations of "difficulty." We might think that difficulty correlates with the number of states of the semantic automata. Recalling that for $\mathsf{Q}_1 \cdot \mathsf{Q}_2$ where $| \mathcal{Q}_1 |= m$ and $| \mathcal{Q}_2 |= n$ the stack version has precisely $m + n + 1$ states while the DFA version has on the order of $m \cdot n$ states, the state-space of the latter becomes unwieldy when one or more of the quantifiers is a high-rank cardinal. However, we also noted previously that a computation of the stack version always takes more steps than its DFA counterpart since it at least reads the entire input before processing its stack contents. For this reason, we should expect a mechanism utilizing memory like the stack version to take longer. This does not mesh with the conclusion of [51]; however, studies may now be done with quantifiers with a wider range of complexity (loops and state size) to see how response time and memory recruitment vary. As Steinert-Threlkeld said in [46], "Indeed, a general notion of complexity for automata in the context of language processing would be useful in this context."[2]

One potential issue with teasing apart the explanatory contributions of the *kind* of iteration automata presented in this thesis and stack automata is that, once iterations involve simple quantifiers that are already themselves DPDA-computable, it is no longer a question of whether the proper model has a stack or not (because even the version defined here must utilize a stack). There is

[2]Note that, for example, state-space is probably not a good metric for the complexity of $(\mathsf{Q}_1 \cdot \mathsf{Q}_2)_{\mathsf{stack}}$, since there is no unique "minimal" PDA for a language.

still a qualitative difference between DFA/DPDA iteration automata and stack automata, which is more apparent if one considers the generalizations in Sections 5.2 and 5.3.

In [46], Steinert-Threlkeld comments on the finding that parity quantifiers, which have both DFA and PDA representations,[3] recruit working memory, suggesting "This provides *prima facie* reason to believe that working memory will be recruited when processing sentences with multiple quantifiers each computable by a DFA. This would show that the PDA [stack] representation more closely resembles the actual processing mechanism." We note this is not necessarily the case. It could be that a good model is given by using the DPDA representation of even in one of our definitions from Section 6.2 instead of using either the DFA or DPDA representation of even in the stack model. Perhaps eye-tracking studies could provide evidence for one or the other method when memory-recruitment cannot be the distinguishing factor.

9.1.2 Formal Learning Theory

Formal learning theory studies, among other things, how people can come to associate meanings with words. In this domain quantifiers strongly contrast with say, common nouns. We can imagine a young child learning the meaning of *dog* through repeated association of actual dogs with utterances of *dog* ("semantic bootstrapping"). It is unclear how the diverse situations across which a particular quantifier may be used in a sentence can contribute to learning its meaning. The idea that quantifier comprehension involves knowing an algorithm to check its proper application makes immense sense in this context.

Van Benthem already suggested the potential utility of the semantic automata framework and Tree of Numbers for quantifier learning in [5]. In 1996, Clark [7] demonstrated the learnability of FO-definable determiners due to the algorithm $L*$ proposed by Angluin in 1987, which allows for the learnability of *any* regular set. The algorithm gives both positive and negative information, meaning the procedure is controlled by a "minimally adequate teacher:" basically, the learner is provided with counterexamples. The learner observes strings and builds an "observation table" that can be transformed into the minimal finite automaton accepting the language. $L*$ is polynomially bounded in the number of states of the minimal DFA and the size of the counterexamples provided by the teacher.

Noting that the set of FO quantifiers is indeed a "very proper" subset of the regular languages, in [9] and [10] Clark gives a new proof of their learnability that relies on the nice properties of these quantifiers. Input is in the form of a sequence of pairs $\mathsf{Det}(A, B)$ and pairs (m, n) (points in the Tree of Numbers) such that $m = | A - B |$ and $n = | A \cap B |$, e.g. $\langle \mathsf{every}(\text{DOG}, \text{ANIMAL}); (0, 3) \rangle$.

[3] Of course, every DFA language is recognized by a DPDA that does not make use of its stack; even can be recognized by a DPDA that tracks parity with a non-trivial use of its stack, but only ever needs to store a single symbol.

The output of the algorithm is the automata recognizing the language of Det: "The learner converges (successfully learns) a quantifier denotation if she posits an automaton that correctly accepts all and only the strings associated with the quantifier's set of points in the tree of numbers." Since these quantifiers are uniquely determined by the upper triangle above the Fraïssé threshold (see Section 3.2), only a finite amount of input is required to distinguish Det from the rest of the possible quantifiers. In fact, there are at most:

$$\sum_{j=0}^{n} \sum_{i=0}^{j} \binom{j}{i}, \text{ where } n \text{ is the Fraïssé level}$$

strings represented by the top triangle (we can simply enumerate them all); since the positive instances are themselves distinguishing, there are even fewer strings needed; if the learner knows the information is permutation-invariant, there are fewer still. Moreover, there is an upper bound on the number of potential automata[4], and these may also be effectively enumerated (see Section 9.2.1) for remarks on this construction). Thus after a finite number of examples, the learner will find the correct automaton.

Learnability of higher-order quantifiers is trickier, since there is not obviously any finite part of the Tree of Numbers which is sufficient to tell them apart from one another. Clark [10] does give the following formulation as a first step: the set of higher-order determiners is learnable if they have finite *Vapnik-Chervonenkis* dimension, meaning that there is a finite set of points in the Tree of Numbers containing a data point for each determiner in the set that distinguishes it from all the others ("shatters" the set of points).

Gierasimczuk [18] gives two criticisms of utilizing semantic automata in this domain. One we have already mentioned in Section 1.2.3: these methods do not distinguish between the capacity for comprehension and the capacity for production. The former corresponds to model-checking: given $\mathcal{M}$ and ϕ, decide whether $\mathcal{M} \vDash \phi$; the latter corresponds to giving an adquate description of a model: given $\mathcal{M}$, produce ϕ such that $\mathcal{M} \vDash \phi$ [18]. These are equivalent with respect to the descriptive power of the formalisms of grammars and automata, but research shows that verification is easier for people than generation. The second has to do with *inferential meaning*, another prong of semantic competence. Unlike referential meaning, which involves a procedure to decide or *directly verify* the value of φ in any situation (like, e.g., a semantic automata), inferential meaning draws on logical relationships among formula to determine their truth value. If $\psi \Rightarrow \varphi$ and we know ψ, we may conclude φ. The above methods seem to lend no insight in this direction. Szymanik [49] discusses this distinction with respect to complexity concerns: it may be that a sentence is not directly verifiable because it is computationally intractable, yet we can infer its truth indirectly from one or more tractable sentences.[5]

[4]The threshold is related to the pumping length of the language, which is the number of states in the minimal automaton.

[5]Consider his example: If we know (1) *most villagers are communists*, (2) *most townsmen are capitalists*, and (3) *all communists and all capitalists hate each other* (all of which are

Some remaining open questions are:

Question 9.1.3. Can the learnability of higher-order, context-free quantifiers be demonstrated? Is the problem easier if restricted to *deterministic* context-free quantifiers?

Question 9.1.4. The learnability of regular iterated quantifiers follows from $L*$ as in [7], but is there also a Fraïssé-style argument demonstrating this?

Recall that that iterated languages take the form $(w_i \boxdot)^*$. Using uniform strings (with all the w_i of the same length) is sufficient to tell them apart, so they could be enumerated following a "zig-zag" pattern in the $x - y$ plane where the point (x, y) corresponds to strings of the language where $\#_{\boxdot}(w) = x$ and $| w_i |= y$. And as this thesis demonstrates, there is an effective procedure to build the minimal DFA for an iterated quantifier. Is there a point (x, y) in this enumeration, related to the Fraïssé thresholds for Q_1 and Q_2, such that once the learner has seen that much (finite) input, she can identify the correct automaton?

9.2 Directions for Future Research

9.2.1 Generating FO Semantic Automata by Construction

In [9], Clark shows that the class of finite automata recognizing FO-definable $\langle 1, 1 \rangle$ quantifiers can be constructed from a minimal set of acyclic finite automata. First he shows that the set of automata computing at least n for every n is constructible. Let the "assembly" function[6] @ take the minimal DFA $\mathsf{M}_{\geq i}$ and $\mathsf{M}_{\geq j}$ computing at least i and at least j respectively, and return $\mathsf{M}_{\geq i}@\mathsf{M}_{\geq j} = \mathsf{M}_{\geq i+j}$ computing at least i+j by (more-or-less) replacing the final state of the former with the start state of the latter.

Then M_{card}, the "basic cardinal automata" is the smallest set such that:

1. $\mathsf{M}_{\geq 1} \in \mathsf{M}_{\text{card}}$
2. if $\mathsf{M}_{\geq i}, \mathsf{M}_{\geq j} \in \mathsf{M}_{\text{card}}$, then $\mathsf{M}_{\geq i}@\mathsf{M}_{\geq j} \in \mathsf{M}_{\text{card}}$

Closing M_{card} under complement yields automata computing at most n ($\neg$at least n+1) for every n. At this point, the constructions have only produced minimal DFA. Quantifiers exactly n and the other combinations of simple quantifiers (i.e. Q_1 and Q_2, Q_1 or Q_2) are obtained by closing the set under union and intersection. Finally, define the set $\text{Basic}_{\mathsf{M}} = \{\langle \{\mathsf{M}_{\geq 1}, @\rangle\} \cup \{\mathsf{M}_{\text{all}}\}$. Then M_{FO},

directly verifiable in polynomial time), we may infer that *most villagers and most townsmen hate each other*, though directly verifying this conclusion, on its branching interpretation, is an NP-complete problem. He proposes that NPTIME captures the indirect verifiability of everyday language: given (or, non-deterministically guessing) some proofs or certificates of the truth of a sentence (like (1)-(3) above), we can verify an intractable sentence in polynomial time.

[6] We create a different symbol so as not to confuse notations.

the set of automata computing FO-definable $\langle 1,1\rangle$ quantifiers, is $\text{Basic}_{\mathsf{M}}$ closed under finite[7] applications of $@, \neg, \cup$, and $\cap$.

He takes some pains to define the @ function instead of just taking the concatenation of $\mathsf{M}_{\geq i}$ and $\mathsf{M}_{\geq j}$ (which would be equivalent, but result in an NFA by the standard sequential composition operation). In an earlier manuscript [8], he gives similarly clever procedures for the intersection and union of two simple automata M_1 and M_2, using the cross-product of their states. The procedure for the intersection is the product construction, standard in the proof of intersection closure, requiring that the run of the input end in a state $\langle q,p\rangle$ where $q \in F_1$ and $p \in F_2$. The procedure for union requires that the run end in a state $\langle q,p\rangle$ where either $q \in F_1$ or $p \in F_2$ (or both), rather than the construction often cited in proofs of union closure resulting in an NFA.[8]

Of course, these still do not uniformly result in *minimal* DFA. The automatons at least 3 and less than 4 have three and five states respectively, so their conjunction and disjunction each have fifteen states using the product construction. But at least 3 and less than 4 is equivalent to exactly 3, which can be computed by a five state DFA, and at least 3 or less than 4 can be computed by the trivial one state DFA that accepts every word over $\{0,1\}$. In the spirit of Clark's use of @ for constructing M_{card}, and in the spirit of this thesis, we think it would be interesting to try to find similar methods for directly constructing minimal DFA for boolean combinations of minimal DFA computing simple monadic quantifiers. Moreover, success in that endeavour could more generally be seen as finding the state complexity of operations on permutation-invariant acyclic finite state automata.

Also in [8], Clark describes his query system SAM (Semantic Automata + Models), which is basically a Common Lisp implementation of model-checking for first-order logic with DFA-computable quantifiers. The user can define a model and then ask queries of the form, e.g., (`(query '((Q) A B))`. The system would then look at the extensions of `A` and `B`, compute a string of 0's and 1's using a function `bits` (i.e. τ), construct Q (from the basic elements M_{all} and $\mathsf{M}_{\geq 1}$), and run Q on the string.[9]

SAM can also handle embedded queries like (a), whose truth conditions are (b):

a. `(scope (at-least 4) students read (at-least 1) books)`

b. at least 4 $(\{x : x \text{ is a student}\},$

$$\{y : \text{at least } 1(\{z : z \text{ is a book}\}, \{w : y \text{ read } w\})\})$$

Of course, such embedded queries are examples of iteration. The system first

[7]He only allows finite applications so that the results are actually finite automata; an automata with infinite states, or a formula with infinitely many clauses, could define higher-order quantifiers like most that have no Fraissé threshhold.

[8]See the proof of Theorem 2.2.4 for those constructions.

[9]The system also answers *wh-* questions, but we focus on queries of this familiar verification form as the example.

computes a set Y by running at least 1 on the string generated by comparing the sets books and R_x for every x, and then runs at least 4 on the string generated by comparing the sets students and Y. Since the quantifiers work in isolation, this is very much like using a stack iteration automaton. SAM could be extended to implement actual semantic iteration automata with a function to encode the model all at once and a function to compose automata according to our definitions in Chapter 5.[10] SAM could also be extended with semantic automata for parity quantifiers by adding even to the set of basic elements and creating a procedure similar to @ to generate divisible by n+1 from divisible by n (from Figure 3.6 in Section 3.2, we can see this is as simple as inserting another state in a single cycle). Implementing minimal constructions and iteration automata may even afford SAM's computations some speed-up as queries grow more and more complex, requiring more and more operations to generate the desired automata from $\text{Basic}_{\mathsf{M}}$.

9.2.2 Further Extensions of Semantic Automata and the Role of Representations

In this thesis we have given constructions for iteration and cumulation semantic automata–based on a specific method of translating models with relations to strings–but many types of polyadic quantification remain unaccounted for by the semantic automata model. Resumption, reciprocals, branching, "same" constructions, etc. could be investigated in the future. For quantifiers whose evaluation requires more knowledge about the model, it could be helpful to consider other possibilities for the translation function–perhaps yielding a kind of parameterized Chomsky designation for quantifier languages.

For example, the language of a cumulation at least initially appears very difficult to recognize if strings can only encode indirect information about the inverse relation, but we saw that given some extra allowances, cumulations are DFA-computable. Van Benthem addresses this concern, saying "an important issue has been left implicit, *viz.* the *representation* of the data fed to the procedure associated with a linguistic expression. There is a matter of 'division of labour' here, which can affect judgments of complexity" [5] and "final judgments of complexity will then depend on a balance between these two components" [4].

How can we find and formulate new representations? The vanilla version of τ_2 with only R directly encodes all the information relevant to an iteration, while τ_2^{cl} with both R and R^{-1} directly encodes all the information relevant to a cumulation. The links between those two lifts and their appropriate τ's has to do with their invariance properties: iteration is right oriented, while cumulation is left-and-right oriented. Are there other such correspondences? For example,

[10] There is currently much work being done to integrate GQ theory into *controlled languages* (unambiguous fragments of natural language that facilitate reasoning about and interacting with databases)–though not explicitly using semantic automata like SAM. See Thorne [1] and Badia [54] for work in this direction.

the resumption of Q to k-tuples, $Res^k(\mathsf{Q})$, is invariant under permutations of k-tuples–can we capture that property with some extension of τ_2? Are there other permutation-invariance properties of certain classes of irreducible quantifiers that we can program into τ_2 to more adequately represent models? What other properties of non-Fregean quantifiers are there, in general, that we can exploit to make the information encoded by a language more accessible to simpler automata?

Using a new translation function we successfully reined in cumulation: regular and context-free languages are closed under the cumulation operation we defined in Chapter 7.[11] However, we think it is unlikely that finding similarly suitable translation functions for other kinds of irreducible quantification will necessarily have the same effect.[12] Cumulation languages do not *essentially* rely on the relations between symbols in separate subwords, unlike many other lifts, so even with upgraded representations, recognizers of irreducible languages may still need some means of counting indices. It seems like such a process is not aided at all by a pushdown stack–intuitively some kind of back-and-forth functionality is required. Anticipating this, we want to suggest a few potentially fruitful automata models for future consideration: perhaps two-way automata, multitape and multihead automata, and automata with erasure, would give ways of "having a finger on" more than one symbol at a time. For reciprocals and branching quantifiers in particular it may be useful to look into graph automata, as their interpretations are captured by graph properties.

[12]When a change of τ_2 does bestow some automata operation representing a polyadic lift with nice closure properties, we might say it was "on" the "Chomsky-Frege" boundary, similarly to how cumulation is "on" the Frege boundary.

Bibliography

[1] Antonio Badia. *Quantifiers in Action: Generalized Quantification in Query, Logical and Natural Languages*. Springer, 2009.

[2] Y. Bar-Hillel, M. Perles, and E. Shamir. On formal properties of simple phrase-structure grammars. In *Z.Phonetik. Sprachwiss. Kommunikationsforch.*, volume 14, pages 143–172. 1961.

[3] Jon Barwise and Robin Cooper. Generalized quantifiers and natural language. *Linguistics and Philosophy*, 4(2):159–219, 1981.

[4] Johan van Benthem. *Essays in Logical Semantics*. D. Reidel Publishing Company, 1986.

[5] Johan van Benthem. Towards a computational semantics. In P. Gardenförs, editor, *Generalized Quantifiers*, pages 31–71. D. Reidel Publishing Company, 1987.

[6] Johan van Benthem. Polyadic quantifiers. *Linguistics and Philosophy*, 12(4):437–464, 1989.

[7] Robin Clark. Learning first order quantifier denotations: An essay in semantic learnability. *IRCS Technical Reports Series*, 1996.

[8] Robin Clark. Programming semantic automata. `ftp://babel.ling.upenn.edu/facpapers/robin_clark/sam.pdf`, 2001. (unpublished manuscript).

[9] Robin Clark. Some computational properties of generalized quantifiers. `ftp://babel.ling.upenn.edu/facpapers/robin_clark/compq2.pdf`, 2010. (unpublished manuscript).

[10] Robin Clark. On the learnability of quantifiers. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, chapter 20, pages 911–923. Elsevier, 2nd edition, 2011.

[11] Mary Dalrymple, Makoto Kanazawa, Yookyung Kim, Sam Mchombo, and Stanley Peters. Reciprocal expressions and the concept of reciprocity. *Linguistics and Philosophy*, 21(2):159–210, 1998.

[12] Paul Dekker. Meanwhile, within the Frege boundary. *Linguistics and Philosophy*, 26(5):547–556, 2003.

[13] J. Dotlačil, J. Szymanik, and M. Zajenkowski. Probabilistic semantic automata in the verification of quantified statements. In *Proceedings of the 36th Annual Conference of the Cognitive Science Society*, 2014.

[14] Jan van Eijck. Normal forms for characteristic functions on n-ary relations. *Journal of Logic and Computation*, 15(2):85–98, 2005.

[15] J. Evey. Application of pushdown store machines. In *Proceedings of Fall Joint Computer Conference*, pages 215–227. AFIPS Press, 1963.

[16] Gottlob Frege. Über sinn und bedeutung. *Zeitschrift für Philosophie und Philosophische Kritik*, 100:25–50, 1892.

[17] Yuan Gao, Nelma Moreira, Rogério Reis, and Sheng Yu. A review on state complexity of individual operations. Technical report, Universidade do Porto, Technical Report Series DCC-2011-08, Version 1.1 (September 2012), `http://www.dcc.fc.up.pt/Pubs`, 2012.

[18] Nina Gierasimcauk. The problem of learning the semantics of quantifiers. *Lecture Notes in Computer Science*, 4363:117–126.

[19] Nina Gierasimczuk and Jakub Szymanik. Branching quantification v. two-way quantification. *Journal of Semantics*, 26(4):367–392, 2009.

[20] Seymour Ginsburg. *The Mathematical Theory of Context Free Languages*. McGraw-Hill Book Company, 1966.

[21] Seymour Ginsburg and Sheila Greibach. Deterministic context free languages. *Information and Control*, 9(6):620–648, 1966.

[22] Oded Goldreich. *Computational Complexity: A Conceptual Perspective*. Cambridge University Press, 2008.

[23] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation*. Addison-Wesley, 2nd edition, 2001.

[24] Neil Immerman. *Descriptive Complexity*. Graduate Texts in Computer Science. Springer, 1999.

[25] Theo Janssen. Compositionality. In J. van Benthem and A. ter Meulen, editors, *Handbook of Logic and Language*, chapter 10, pages 495–553. Elsevier, 2nd edition, 2011.

[26] Makoto Kanazawa. Monadic quantifiers recognized by deterministic puhsdown automata. `http://research.nii.ac.jp/~kanazawa/talks/ac2013talk.pdf`, 2013. slides presented at 19th Amsterdam Colloquium.

[27] Makoto Kanazawa. Monadic quantifiers recognized by deterministic pushdown automata. In *Proceedings of the 19th Amsterdam Colloquium*, pages 139–146, 2013.

[28] Edward L Keenan. Unreducible n-ary quantifiers in natural language. In Peter Gärdenfors, editor, *Generalized Quantifiers: Linguistic and Logical Approaches*, pages 109–150. D. Reidel Publishing Company, 1987.

[29] Edward L Keenan. Beyond the Frege boundary. *Linguistics and Philosophy*, 15(2):199–221, 1992.

[30] S.C. Kleene. Representation of events in nerve nets and finite automata. In *Automata Studies*, pages 3–42. Princeton University Press, 1956.

[31] Donald E Knuth. On the translation of languages from left to right. *Information and Control*, 8(6):607–639, 1965.

[32] Per Lindström. First order predicate logic with generalized quantifiers. *Theoria*, 32(3):186–195, 1966.

[33] David Marr. Vision: A computational investigation into the human representation and processing of visual information. *WH San Francisco: Freeman and Company*, 1982.

[34] Corey T McMillan, Robin Clark, Peachie Moore, Christian Devita, and Murray Grossman. Neural basis for generalized quantifier comprehension. *Neuropsychologia*, 43(12):1729–1737, 2005.

[35] Corey T McMillan, Robin Clark, Peachie Moore, and Murray Grossman. Quantifier comprehension in corticobasal degeneration. *Brain and Cognition*, 62(3):250–260, 2006.

[36] Alexander Meduna. *Automata and Languages: Theory and Applications*. Springer, 2000.

[37] Andrzej Mostowski. On a generalization of quantifiers. *Fundamenta Mathematicae*, 44:12–36, 1957.

[38] Marcin Mostowski. Computational semantics for monadic quantifiers. *Journal of Applied Non-Classical Logics*, 8(1-2):107–121, 1998.

[39] Peter Pagin. Communication and the complexity of semantics. In M. Werning, W. Hinzer, and E. Mackery, editors, *The Oxford Handbook of Compositionality*. Oxford University Press, 2012.

[40] Rohit J. Parikh. On context-free languages. *Journal of the Assocation for Computing Machinery*, 13(4):570–581, 1966.

[41] Stanely Peters and Dag Westerståhl. *Quantifiers in Language and Logic*. Clarendon Press, 2006.

[42] Iris van Rooij. The tractable cognition thesis. *Cognitive Science*, 32(6):939–984, 2008.

[43] Fabian Schlotterbeck and Oliver Bott. Easy solutions for a hard problem? The computational complexity of reciprocals with quantificational antecedents. *Journal of Logic, Language and Information*, pages 1–28, 2013.

[44] Michael Sipser. *Introduction to the Theory of Computation.* Cengage Learning, 2006.

[45] Shane Steinert-Threlkeld. On the decidability of iterated languages. In *Proceedings of Philosophy, Mathematics, Linguistics: Aspects of Interaction*, pages 215–224, 2014.

[46] Shane Steinert-Threlkeld and Thomas F Icard III. Iterating semantic automata. *Linguistics and Philosophy*, 36(2):151–173, 2013.

[47] Patrick Suppes. Variable-free semantics with remark on procedural extensions. In H. Bunt, I. Sluis, and R. Morante, editors, *Language, Mind and Brain*, pages 21–34.

[48] Jakub Szymanik. A comment on a neuroimaging study of natural language quantifier comprehension. *Neuropsychologia*, 45(9):2158–2160, 2007.

[49] Jakub Szymanik. *Quantifiers in TIME and SPACE. Computational complexity of generalized quantifiers in natural language.* PhD thesis, ILLC, Universiteit van Amsterdam, 2009.

[50] Jakub Szymanik. Computational complexity of polyadic lifts of generalized quantifiers in natural language. *Linguistics and Philosophy*, 33(3):215–250, 2010.

[51] Jakub Szymanik, Shane Steinert-Threlkeld, Marcin Zajenkowski, and Thomas Icard. Automata and complexity in multiple-quantifier sentence verification. In *Proceedings of the 12th International Conference on Cognitive Modeling.* Carleton University, 2013.

[52] Jakub Szymanik and Marcin Zajenkowski. Comprehension of simple quantifiers: Empirical evaluation of a computational model. *Cognitive Science*, 34(3):521–532, 2010.

[53] Jakub Szymanik and Marcin Zajenkowski. Contribution of working memory in parity and proportional judgments. *Belgian Journal of Linguistics*, 25(1):176–194, 2011.

[54] Camilo Thorne. *Query answering over ontologies using controlled natural languages.* PhD thesis, KRDB Dissertation Series, Faculty of Computer Science, Free University of Bozen-Bolzano, 2010.

[55] Pavel Tichỳ. Intension in terms of Turing machines. *Studia Logica*, 24(1):7–21, 1969.

[56] Dag Westerståhl. Iterated quantifiers. In M. Kanazawa and Ch. Pinon, editors, *Dynamics, Polarity, and Quantification*, pages 173–209, 1994.

[57] Marcin Zajenkowski, Rafał Styła, and Jakub Szymanik. A computational approach to quantifiers as an explanation for some language impairments in schizophrenia. *Journal of Communication Disorders*, 44(6):595–600, 2011.

[58] Marcin Zajenkowski and Jakub Szymanik. Most intelligent people are accurate and some fast people are intelligent: Intelligence, working memory, and semantic processing of quantifiers from a computational perspective. *Intelligence*, 41(5):456–466, 2013.

[59] Marcin Zajenkowski, Jakub Szymanik, and Maria Garraffa. Working memory mechanism in proportional quantifier verification. *Journal of Psycholinguistic Research*, pages 1–15, 2013.

[60] Richard Zuber. A note on the monotonicity of reducible quantifiers. *Journal of Logic, Language and Information*, 19(1):123–128, 2010.

Appendix A

Implementing Semantic Automata in Haskell with HaLeX

Since iteration automata are roughly multiplicative in the state-size of all the component automata, they easily become large and unwieldy. And while the general idea of their construction is simple and highly intuitive, certainty in the details is inversely correlated with the arity of the iteration, i.e. size of the automaton alphabet, i.e. number of transitions. Luckily there exists a Haskell library called HaLeX for the manipulation of regular expressions and finite automata which allowed checking for correctness when coming up with the automata definitions in this thesis. Documentation and the downloadable files are available at `<http://www3.di.uminho.pt/~jas/Research/HaLeX/HaLeX.html>`. We briefly introduce the code and then show how it was utilized to build iteration automata.

HaLeX defines a new data type `Dfa` with a constructor of the same name, clearly mimicking the standard definition of finite automata as five-tuples, parameterized with the type of the states (`st`) and the the type of the elements of the alphabet (`sy`). This of course is sufficient to determine the type of every component. We supply the types `Int` and `Char` for tese parameter values throughout.

```
data Dfa st sy = Dfa [sy]
                     [st]
                     st
                     [st]
                     (st -> sy -> st)
```

Representing a DFA in HaLeX is then just a matter of specifying all the compo-

nents. For example, recall that the finite automaton some is a two-state machine that enters its final state after reading 1. We can easily read this behavior off of the definition below.

```
some :: Dfa Int Char
some = Dfa sigma states s final delta
  where
    sigma       = "01"
    states      = [1,2]
    s           = 1
    final       = [2]
    delta 1 '0' = 1
    delta 1 '1' = 2
    delta 2  _  = 2
```

To test whether some string is in the language that an automaton accepts, we supply the function `dfaaccept'` with the automaton and the string. This function calls `dfawalk`, which recursively processes the string symbol by symbol, updating the state of the automaton. If after processing the entire string the automaton is in a final state, `dfaaccept'` is true.

```
dfaaccept' :: Eq st
              => Dfa st sy
              -> [sy]
              -> Bool
dfaaccept' (Dfa v q s z delta) simb =
        (dfawalk delta s simb) `elem` z

dfawalk :: (st -> sy -> st)
           -> st
           -> [sy]
           -> st
dfawalk delta s [] = s
dfawalk delta s (x:xs) = dfawalk delta (delta s x) xs
```

HaLeX also lets us take advantage of the equivalency of regular expressions and finite automata with the functionality to convert between the two via standard algorithms.[1] Composing a few given HaLeX functions yields a minimal DFA with a readable transition function given a regular expression:

```
regExp2MinDfa expression =
  beautifulDfaWithSyncSt $ minimizeDfa $ regExp2Dfa expression
```

[1]There appears to be an issue in a section of the given HaLeX code to deal with the conversion from automata to regular expressions; this is inconvenient since we must define expressions for complements by hand, but otherwise not a problem.

To generate iteration automata from regular expressions, we need to substitute expressions for symbols. Defining iterated expressions by hand is inefficient, so we instead create functions that are pseudo-expressions, taking as input two other expressions. These arguments could be the literals '0' and '1', yielding an expression for a type $\langle 1, 1\rangle$ quantifier, or expressions for a type $\langle 1, 1, \ldots, n\rangle$ quantifier and its complemenent, yielding an expression for a type $\langle 1, 1, \ldots, n+1\rangle$ quantifier. For shorthand, we define a type synonym `OpenRegExp` for these functions.

```
type OpenRegExp = RegExp Char -> RegExp Char -> RegExp Char

some' :: OpenRegExp
some' = \ a b -> Then (Star a) (Then b (Star (Or a b)))

no' :: OpenRegExp
no' = \ a b -> (Star a)

every' :: OpenRegExp
every' = \ a b -> (Star b)

notEvery' :: OpenRegExp
notEvery' = \ a b -> Then (Star b) (Then a (Star (Or a b)))

exactlyOne' :: OpenRegExp
exactlyOne' = \ a b -> Then (Star a) (Then b (Star a))

exactlyTwo' :: OpenRegExp
exactlyTwo' = \ a b -> Then (exactlyOne' a b) (exactlyOne' a b)

exactlyThree' :: OpenRegExp
exactlyThree' = \ a b -> Then (exactlyTwo' a b) (exactlyOne' a b)

two' :: OpenRegExp
two' = \ a b -> Then (some' a b) (some' a b)

three' :: OpenRegExp
three' = \ a b -> Then (two' a b) (some' a b)

four' :: OpenRegExp
four' = \ a b -> Then (three' a b) (some' a b)

atMostOne' :: OpenRegExp
atMostOne' = \ a b -> Or (no' a b) (exactlyOne' a b)

atMostTwo' :: OpenRegExp
```

```
atMostTwo' = \ a b -> Or (atMostOne' a b) (exactlyTwo' a b)
```

The following functions implement the iteration of n `OpenRegExp`'s to form a `RegExp` generating the language of a type $\langle 1, 1, \ldots, n \rangle$ quantifier. We assume the arguments to an initial call of one of these functions do not already contain any literals, since the separator symbols must be distinct for each level (the empty string may be in a sublanguage, meaning simply stacking the same separator symbol is ambiguous). Implementing these iterations fully generally and recursively would require taking care to make sure that a different separator symbol is used in every recursive step. Since we are here concerned with iterations occurring in natural language, hardcoding the first few levels is sufficient for our purposes.[2]

```
iter1 :: OpenRegExp -> RegExp Char
iter1 = \ q -> q (Literal '0') (Literal '1')

iter2 :: OpenRegExp -> OpenRegExp -> OpenRegExp -> RegExp Char
iter2 = \q1 q2c q2 -> q1 (Then (iter1 q2c) (Literal 'a'))
                         (Then (iter1 q2) (Literal 'a'))

iter3 :: OpenRegExp -> OpenRegExp -> OpenRegExp -> OpenRegExp
        -> OpenRegExp -> RegExp Char
iter3 = \q1 q2c q2 q3c q3 ->
             q1 (Then (iter2 q2c q3c q3) (Literal 'b'))
                (Then (iter2 q2 q3c q3) (Literal 'b'))

iter4 :: OpenRegExp -> OpenRegExp -> OpenRegExp -> OpenRegExp
        -> OpenRegExp -> OpenRegExp -> OpenRegExp -> RegExp Char
iter4 = \q1 q2c q2 q3c q3 q4c q4 ->
           q1 (Then (iter3 q2c q3c q3 q4c q4) (Literal 'e'))
              (Then (iter3 q2 q3c q3 q4c q4) (Literal 'e'))
```

Then, for example, to create the iterated expression generating the language of two · every·exactly three·every, we supply `iter4` with the expressions for two, every and its complement, exactly three and its complement, and again every and its complement.

```
twoEveryExactlyThreeEvery' =
        iter4 two' notEvery' every' exactlyThreeC' exactlyThree'
            notEvery' every'
```

[2]But we could, for example, have a general function that takes a list of expressions and a large-enough list of distinct characters. However, we only use these expressions to build the automata, and already at five levels of iteration the subset construction takes prohibitively long.