Computational Implementation of Non-Concatenative Morphology

Yael Cohen-Sygal

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE MASTER DEGREE

University of Haifa

Faculty of Social Science

Department of Computer Science

March, 2004

Computational Implementation of Non-Concatenative Morphology

By: Yael Cohen-Sygal

Supervised By: Dr. Shuly Wintner

THESIS SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR THE MASTER DEGREE

University of Haifa

Faculty of Social Science

Department of Computer Science

March, 2004

Approved by: _____ Date:_____

(supervisor)

Approved by: _____ Date:_____

(Chairman of M.A Committee)

Acknowledgment

The research was done under the supervision of Dr. Shuly Wintner in the department of Computer Science.

I would like to thank Shuly for his excellent guidance, assistance and support throughout this work. I highly regard his support both academic and socially throughout my graduate studies.

I would like to give my special thanks to my husband Marcelo for his endless love and support and for just being the great person he is.

Contents

Computational Implementation of Non-Concatenative Morphology

Yael Cohen-Sygal

Abstract

We introduce *finite state registered machines*, a new computational device within the framework of finite state technology that accounts for non-concatenative morphological processes such as word formation in Semitic languages. It extends and augments existing finite-state techniques, which are presently not sufficiently suitable for describing this kind of phenomena. We define the new model, prove it is indeed finite-state, show how it maintains the closure properties of regular languages and relations and use it to describe some non-concatenative phenomena of natural languages, including circumfixation, interdigitation and limited reduplication.

List of Figures

Chapter 1

Introduction

1.1 Non-concatenative morphology

Morphology is the area of linguistics which studies the structure of words. *Derivational morphology* describes processes of word formation, where words are constructed from roots (or stems) and derivational affixes. For example, the stem '*nation*' can be concatenated with the derivational affix '*al*' to form the word '*national*'.This word is a new stem, which can now be prefixed by '*inter*' to yield *international*. This word can then be suffixed by '*ize*' to yield the word *internationalize*, which can again be suffixed with '*ation*' to yield the word *internationalization*. Another example is the Hebrew word *makteba* (desk) that is formed by inserting the root k.t.b (meaning a notion of writing) into the pattern ma□□e□a (denoting tools), where the □ slots indicate where the root letters should be inserted. *Inflectional morphology* describes processes in which inflected forms are constructed from base forms and inflectional affixes. As an example, consider the Hebrew verb *$amar* (guard). The addition of the suffix *ti* to Hebrew verbs indicates past tense, first person singular. Therefore suffixing the verb *$amar* with the suffix *ti* results in the inflected form *$amarti* (I guarded).

Morphological analysis is the computational process which associates a given surface word with information about its structure. The information can consist of morphological and morpho-phonological features such as stem, tense, gender, number etc. Exactly what information is produced depends on the language and its morphological processes and on the application. An example is the morphological analysis of the Hebrew verb *ti$mru* (you will guard). The analysis could yield the following

information: *Part of Speech=verb, Root=$.m.r, Pattern=□a□a□, Tense=future, Person=2nd, Number=plural, Gender=masculine/feminine.*

While much of the inflectional morphology of Semitic languages can be rather straightforwardly described using concatenation as the primary operation, the main word formation process in such languages is inherently non-concatenative. The standard account describes words in Semitic languages as combinations of two morphemes: a root and a pattern (see Shimron (2003) for a survey). The root consists of consonants only, by default three (although four, five and even six consonantal roots are known, and some words are easier to describe using biconsonantal roots). The pattern is a combination of vowels and, possibly, consonants too, with "slots" into which the root consonants can be inserted. The process in which words are created by the insertion of roots into patterns is called *interdigitation*. Both the root and the pattern have meanings, although sometimes the meanings are rather general (as is usual in derivational morphology, combinations of roots and patterns are not always possible, and the meaning of a combination cannot be fully determined by the meanings of the two morphemes). As an example, consider the Hebrew roots g.d.l (roughly meaning growth) and z.r.x (shine). Seven of the patterns in Hebrew are verbal, including the pattern □a□a□, which is used for active verbs. When the above roots combine with this pattern the resulting lexemes are *gadal* and *zarax*, respectively. The first consonant of the root is inserted into the first consonantal slot of the pattern, the second root consonant fills the second slot and the third fills the last slot. Some patterns have four slots, usually indicating gemination of one of the root's consonants. For example, the verbal pattern □i□□e□ (usually indicating causation, or a certain reinforcement aspect of the active form) combines with the example roots to form the lexemes *giddel* and *zirrex*. The latter lexeme is not realized in Hebrew, but its form is well-defined and some notion of its meaning is clear to every native speaker. Now consider a nominal pattern, mi□□a□, usually denoting places. When the example roots combine with this pattern, the obtained lexemes are *migdal* (tower) and *mizrax* (east), respectively.

After the root combines with the pattern, some morpho-phonological alternations take place, which may be non-trivial. The verbal pattern hit□a□□e□, usually denoting reflexivity, can combine with the roots g.d.l and z.r.x to form the lexemes *hitgaddel* and *hizdarrex*, respectively. Note

that in the latter case the ‘t’ in the pattern becomes a vocalized ‘d’, and is transposed with the first consonant of the root. This metathesis happens in roots whose first consonant is an alveolar fricative. Another process has to do with roots whose first consonant is ‘y’: following an ‘i’ in a pattern, the sequence ‘iy’ can change to ‘o’. For example, when the root y.r.d (go down) combines with the pattern mi□□a□ (place), the result is *morad* (descent).

Another non-concatenative morphological process is *reduplication*, in which a morpheme or part of it is duplicated, thus creating a new word with a new related meaning. Reduplication can be *full* (the full word is duplicated) or *partial* (just part of the word is duplicated). For example, full reduplication can be found in Malay and Indonesian, as a pluralization process: the plural form of the Malay word *bagi* (bag, suitcase) is *bagibagi.* Examples for partial reduplication can be found in Chamorro, the native language of Guam, as a measure for intensivity. For example, the word *dakolo* means ‘big’. By duplicating its affix *lo*, the word *dakololo* is created, meaning, ‘very big’. Partial reduplication can also be found in Hebrew as a diminutive formation of nouns and adjectives as demonstrated in the following examples:

kelb	*klablab*	*xatul*	*xtaltul*	*$apan*	*$panpan*	*zaqan*	*zqanqan*
dog	puppy	cat	kitten	rabbit	bunny	beard	goatee

$amen	*$manman*	*$axor*	*$xarxar*	*qatan*	*qtantan*
fat	chubby	black	dark	little	tiny

1.2 Finite state technology

It has long been claimed that the morphology of many languages lies within the expressiveness of a class of formal languages known as regular languages, and computational morphologists have taken up this claim.

Koskenniemi (1983) presents the Two Level model which describes natural language morphology as a relation between two levels – surface and lexical. The model consists of two components, a linked lexicons component and a two-level rules component. The linked lexicon component contains the lexical representation of the words, separated into different lexicons, each representing a different attribute, common to all the words it contains. Thus, the lexical component is constructed of concatenated entries forming together the lexical representation. The two-level rules component consists of

the morphological rules which describe the relations between the surface and the lexical forms. The rules act in parallel to form the surface-lexical relations. The two-level model can be viewed as a relation between strings, but sometimes it is more convenient to view it as a procedure which, when given a lexical string, returns its surface string and vise versa.

Kaplan and Kay (1994) [1] present context-sensitive rewrite rules of the form $\phi \rightarrow \psi / \lambda \_ \rho$ to describe the morphological and phonological alternations in a natural language. Such a rule states that the string ϕ is replaced by the string ψ whenever it is preceded by the string λ and followed by the string ρ. Either one of the strings can be the empty string. Kaplan and Kay (1994) show that rewrite rules of this form define regular relations since they are not allowed to apply to their own output. Furthermore, they show how these rules are compiled into finite state transducers. Thus, a single transducer representing the whole set of rules can be constructed from transducers which correspond to different phenomena by using the composition and union operators.

Since Koskenniemi (1983) presented his two level approaches for morphological and phonological phenomena analysis and Kaplan and Kay (1994) showed how manipulating regular languages and relations can provide a solid basis for computational phonology, most of the accounts of such phenomena have been relying on finite state technology. Finite state analysis of morphological and phonological phenomena was presented for a variety of languages, including some with complex morphology such as Finnish and Turkish. Most of the morphological processes exhibited by natural languages, inflectional and derivational, are based on concatenation which can be easily modeled by regular languages (and therefore automata). Morphological alternations, even of the kind exemplified above, are within the scope of regular relations, and can be implemented through finite state transducers.

There exist several toolboxes (software packages) that provide extended regular expression description languages and compilers of the expressions to finite state devices, automata and transducers (Karttunen et al., 1996; Beesley and Karttunen, 2003; Mohri, 1996; van Noord and Gerdemann, 2001a; van Noord and Gerdemann, 2001b). Such toolboxes include efficient implementations of several standard algorithms on finite state machines, such as union, intersection, minimization, deter-

[1] This paper describes work that was done, but never published, in the early 1980's.

minization etc. More importantly, they also implement special operators that are useful for linguistic description, such as replacement (Kaplan and Kay, 1994; Mohri and Sproat, 1996; Karttunen, 1997; Gerdemann and van Noord, 1999) or predicates over alphabet symbols (van Noord and Gerdemann, 2001a; van Noord and Gerdemann, 2001b), and even operators for particular linguistic theories such as Optimality Theory (Karttunen, 1998; Gerdemann and van Noord, 2000).

Finite state technology has some important advantages, making it most appealing for implementing natural language morphology. One can find it very hard, almost impossible, to build the full automaton or transducer describing some morphological phenomenon. This difficulty arises from the fact that there are a great number of morpho-phonological processes combining together to create the full language. However, it is usually very easy to build a finite state machine to describe a specific morphological phenomenon. The class of regular languages is closed under union, concatenation, intersection and complementation. The class of regular relations is closed under union, concatenation and composition (but not under intersection and therefore complementation). These properties make it most convenient to implement each phenomenon independently and combine them together using the closure operations. Moreover, finite state techniques have the advantage of being efficient in their time and space complexity, as the membership problem is solvable in time linear in the length of the input. There are also known algorithms for minimizing and determinizing automata and some restricted kinds of transducers. All of the above indicate that finite state techniques are a vital tool in the implementation of natural language morphology.

1.3 Finite state techniques and non-concatenative morphology

While finite state approaches for natural language processing have generally been very successful, it is widely recognized that they are less suitable for nonconcatenative phenomena; in particular, finite state techniques are assumed not to be able to efficiently account for the nonconcatenative word formation processes that Semitic languages exhibit. The major problem that we will tackle in this work is medium-distance dependencies, whereby some elements that are related to each other in some deep-level representation (e.g., the consonants of the root) are separated on the surface. While these phenomena do not lie outside the descriptive power of finite state systems, a naïve implementation

of them in some finite state calculus is either impossible or, at best, results in huge networks that are very inefficient to process.

Example 1.1. *We begin with a simplified problem, namely accounting for circumfixes. Consider three Hebrew patterns: ha□□a□a, which is the deverbal noun pattern derived from Binyan hip&il, hit□a□a□ut, which is the deverbal noun derived from Binyan hitpa&el, both denoting actions, and the nominal pattern mi□□a□, denoting places. Notice that in Hebrew the patterns are written h□□□a, ht□□□ut and m□□□, respectively,*[2] *i.e., the consonants are inserted into the □ slots as one unit. An automaton that accepts all the possible combinations of roots and these three patterns is illustrated in Figure 1.1*[3]*. The number of its states is* $(number\_of\_roots \times 2 + 2) \times number\_of\_patterns + 2$ *(in some cases where an affix appears in several patterns or roots the number will be somewhat smaller). Thus, the number of states is* $O(number\_of\_roots \times number\_of\_patterns)$*, i.e., increases linearly with the number of roots and patterns. The number of arcs in this automaton is* $(number\_of\_roots \times 3) \times number\_of\_patterns + number\_of\_patterns \times 2$*, i.e, also* $O(number\_of\_roots \times number\_of\_patterns)$*. Evidently, the three basic different paths that result from the three patterns have the same body, which encodes the roots. A naïve attempt to avoid the reduplication of paths results in the automaton in Figure 1.2, which accepts the language that is denoted by the regular expression* $(ht + h + m)(root)(ut + a + \epsilon)$*. The number of states in this automaton is* $2 \times number\_of\_roots + 4$ *(in this case also, there can be some cases where the number will be smaller). Thus, the number of states in this automaton is* $O(number\_of\_roots)$*, i.e., it is independent of the number of patterns, and therefore is more efficient in its space compared with the naïve automaton. The number of arcs in this automaton is* $(number\_of\_roots \times 3 + number\_of\_patterns \times 2)$*, that is,* $O(number\_of\_roots + number\_of\_patterns)$*, and thus, the complexity of the number of arcs is also reduced. The problem of such an automaton is that it accepts also invalid words such as the pattern m□□□ut. In other words, it ignores the dependencies which hold between prefixes and suffixes of the same pattern. Since finite state devices have no memory, save for the states, there is no way to account for these dependencies, which hold between the prefix of a pattern and its suffix. In this work we create a new model, which facilitates*

[2] Many of the vowels are not explicitly depicted in the Hebrew script.

[3] This is an over-simplified example; in practice, the process of combining roots with patterns is highly idiosyncratic, like other derivational morphological processes.

the expression of such dependencies.

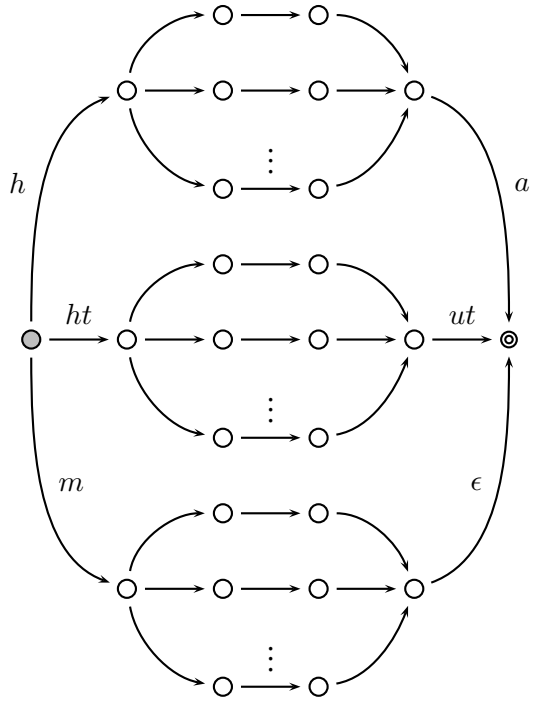


Figure 1.1: Naïve FSA with duplicated paths

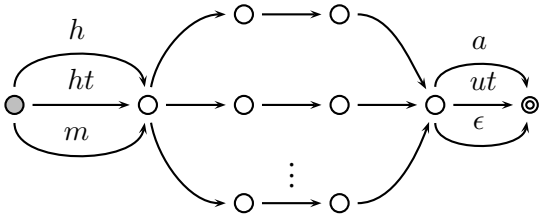


Figure 1.2: Over-generating FSA

Example 1.2. *Consider now a representation of Hebrew where all vowels are explicit, e.g., the pattern hit□a□e□. Consider also the roots r.g.z, b.$.l and g.b.r. The consonants of a given root are inserted into the □ slots to obtain bases such as hitragez, hitba$el and hitgaber. The finite state automaton defined by the diagram in Figure 1.3 is the minimized automaton accepting the language. It has fifteen states. If the number of three letters roots is r, then a general automaton accepting the combinations*

of the roots with this pattern will have $4r + 3$ states and $5r + 1$ arcs (in some cases where an affix appears in several roots the number will be smaller). Notice the reduplication in the arcs which stem from the pattern in the different paths.

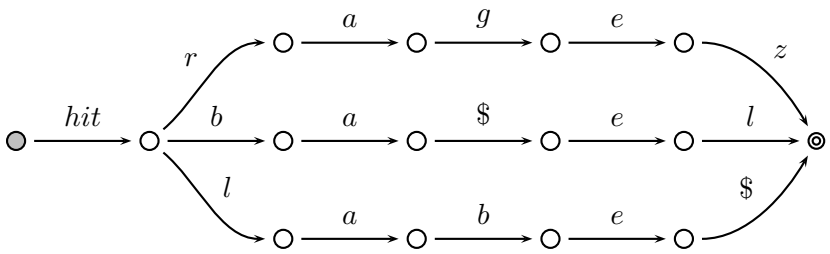


Figure 1.3: FSA for the pattern hit□a□e□

As another example of non-concatenative morphology consider the problem of reduplication. Let Σ be a finite alphabet. The language $L = \{ww \mid w \in \Sigma^*\}$ is known to be trans-regular, therefore no finite state automaton accepts it. However, the language $L_n = \{ww \mid w \in \Sigma^* \ , \ |w| = n\}$ for some constant n is regular. Recognizing L_n is a finite approximation of the general problem of recognizing L. The length of the words in natural languages can in most cases be bounded by some $n \in \mathbb{N}$, hence the amount of reduplication in natural languages is practically limited. Therefore, the descriptive power of L_n is sufficient for the amount of reduplication in natural languages (by constructing L_n for a small number of different n's). An automaton that accepts L_n can be constructed by listing a path for each accepted string. Since Σ and n are finite, the number of words in L_n is finite and therefore it is possible to build such an automaton. The main drawback of such an automaton is the growth in its number of states and arcs as $|\Sigma|$ and n increase: the number of strings in L_n is $|\Sigma|^n$. Thus, finite state techniques can describe reduplication, but they do so inefficiently regarding their space complexity.

1.4 Research goals

The goal of our work is to create a new computational model within the framework of finite state technology that will account for non-concatenative word formation processes in Semitic languages.

It will extend and augment existing finite state techniques, which are presently not sufficiently suitable for describing this kind of phenomena. Our purpose is to create such a new model, prove it is indeed finite state, show how it maintains the closure properties of regular relations and use it to describe the non-concatenative phenomena of Semitic languages.

It seems that any solution in the framework of pure finite automata and transducers technology will not be able to maintain reasonable size networks and will jeopardise the efficiency of the process. To keep the network minimal a creative solution within the finite state technology must be sought. The solution will be based on augmenting finite state networks with limited memory, or templates, such that they facilitate better generalizations while maintaining their desirable closure properties and computational efficiency. Such a solution will also enable us to keep the size of the networks reasonable since it will let us describe complex processes in a compact way by using limited memory, rather than increase the size of the network.

The closure properties are essential for such a model. Most of the morphological phenomena can be described using regular expressions and relations, and word formation processes can be described by a combination of these expression-rules using union, intersection, composition and concatenation operators. Since we want our solution to be a part of an existing model and working system we must preserve the closure properties in our solution. Moreover, most of the morpho-phonological processes in Semitic languages combine non-concatenative processes with those that can easily be described by existing finite state tools (and most of them have already been implemented), and it will be desirable to create a model that will benefit from existing knowledge and implementations.

Chapter 2

Literature survey

2.1 Finite state approaches to the morphology of Semitic languages

2.1.1 Two-level morphology

Kataja and Koskenniemi (1988) use two level technology to create a rule system for phonological and morphophonological alternations in Akkadian word inflection and regular verbal derivation. Being a Semitic language, the main word formation process in Akkadian is interdigitation. Kataja and Koskenniemi (1988) use two separate lexicons, one for the roots and the other for prefixes, flexional elements and suffixes. Entries for roots leave the flexional elements unspecified and vice versa. The intersection of these two lexicons defines the lexical representations of word-forms. The phonological and morphophonological variants are then described using standard finite state two-level morphological rules. As this solution effectively defines lexical representations of word-forms, its main disadvantage is that the final network is the naïve one, suffering from the space complexity problems discussed above.

Lavie et al. (1988) examine the applicability of Two Level Morphology to the description of Hebrew Morphology, and in particular to verb inflection. Their lexicon consists of three parts: verb primary bases (the past tense, third person, singular, masculine), verb prefixes and verb suffixes. The features attached to the entries are designed to generate only existing words. They attempt to describe the process of Hebrew verb inflection as prefix+base+suffix when the base is as above.

However, in most of the verbs the primary base changes when the verb is not in the past tense, and the change is systematic across all verbs in the same pattern and tense. As an example, consider the Hebrew verb *ektob* (Part of Speech=verb, Root=k.t.b, Pattern=□a□a□, Tense=future, Person=1st, Number=singular, Gender=masculine/feminine). The desired description of this verb by the above approach is *e+ktob*, with the primary base *katab* changing into the secondary base *ktob*. The change is highly productive, as one gets *$amar* → *$mor, badaq* → *bdoq* etc.

The goal of Lavie et al. (1988) is to present the verb inflection as a concatenative process, implementable by the Two Level model. They conclude that "The Two Level rules are not the natural way to describe ... verb inflection process. The only alternative choice ... is to keep all bases ... it seems wasteful to save all the secondary bases of verbs of the same pattern." This strengthens our claim that naïve finite state techniques are not suitable for describing non-concatenative phenomena in an appropriate way.

2.1.2 Multilevel automata

Kiraz (2000) expands the traditional two-level model of Koskenniemi (1983) into n-tape automata. He suggests that non-concatenative morphological processes in Semitic languages require more than two levels of expression. The surface level employs only one representation as before, therefore only one level is needed for it, but the lexical form employs multiple representations and therefore should be divided into different levels, one for each representation. A tuple of strings represents a surface-to-lexical mapping, where the first element represents the surface form and the others represent lexical forms.

For example, in Syriac, the lexical level may consist of three representations – a *CV pattern*, a *root*, and *vocalism*, therefore morphological analysis of Syriac will require 4-tape automata with the form

$$\langle \textit{surface, CV pattern,root,vocalism} \rangle.$$

For example, the tuple of strings that map the word *katab* into its lexical analysis is $\langle katab, cvcvc, ktb, a \rangle$. The model consists of three components. The first is a lexicon component which contains multiple sublexica, each consisting of all the possibilities for this sublexicon. For example, the roots

sublexicon contains all the possible roots in the Syriac language. Thus an n-tape automaton will contain n-1 sublexica. The second component consists of rewrite rules which map the multiple lexical representations into a surface representation and vice versa. Each rule is of the form:

LLC - *LEXICAL* - *RLC* $\{\Rightarrow, \Leftrightarrow\}$
LSC - SURFACE - RSC

Such a rule states that the n-1 tuples of strings denoted by LEXICAL are replaced by or matched against the string denoted by SURFACE whenever the following holds:

1. LEXICAL is preceded by the n-1 tuples denoted by LLC (Left Lexical Context).

2. LEXICAL is followed by the n-1 tuples denoted by RLC (Right Lexical Context).

3. SURFACE is preceded by the string denoted by LSC (Left Surface Context).

4. SURFACE is followed by the string denoted by RSC (Right Surface Context).

The lexical contexts will be n-1 tuples. The context denoted by '*' represents anything (Σ^*). The rules are divided into two groups: optional ones, which are denoted by $\Rightarrow$, and obligatory ones, which are denoted by $\Leftrightarrow$. A lexical string is mapped into a surface string iff they can be partitioned into pairs of lexical-surface subsequences where (i) each pair is licensed by a $\Rightarrow$ rule, and (ii) no sequence of zero or more adjacent pairs violates a $\Leftrightarrow$ rule. As an example, consider the rules in Figure 2.1. These rules derive the Syriac word *ktab* (underlying *katab*). Rules 1 and 2 insert the vowel and consonants into the C and V slots respectively. Rule 3 deletes the first vowel 'a'. The 4-tuple $\langle ktab, cvcvc, ktb, aa\rangle$ is admitted since it can be partitioned in a valid way as illustrated in Figure 2.2. The third component consists of morphotactic constraints which deal with non-templatic morphological phenomena using existing algorithms. The final step is compiling all the three units together. During this stage 0's are planted everywhere to create same-length n-tuples in order to enable intersection.[1]

Kiraz (2000) does not discuss the space complexity of the resulting machine compared to the naïve one, but it seems that the number of states in the resulting network still increases linearly with the number of roots and patterns, thus there is no real compactness compared to the naïve machine which just lists all the possible relations. Furthermore, Rules 1 and 2 seem to miss a linguistic

[1] n-way relations are not closed under intersection except for some cases, among them same-length n-way relations.

R_1 : * - $\langle c, X, \epsilon \rangle$ - * $\Rightarrow$

* - X - *

Where X is a consonant

R_2 : * - $\langle v, \epsilon, X \rangle$ - * $\Rightarrow$

* - X - *

Where X is a vowel

R_3 : * - $\langle v, \epsilon, X \rangle$ - $\langle cv, *, * \rangle$ $\Leftrightarrow$

* - ϵ - *

Figure 2.1: N-tape derivation rules

	a		a		vocalism
k		t		b	root
C	V	C	V	C	pattern
1	3	1	2	1	
k		t	a	b	surface

Figure 2.2: Lexical-Surface analysis

generalization. These rules are responsible for the insertion of the consonants and vowels into the C and V slots respectively. They are independent of the language and therefore expected to appear not in the rules creating the language but in the structure of the model. Moreover, the n-tape model requires a specification of the dependencies between symbols in different levels, which may be non-trivial. Therefore, the task of writing the appropriate rules that will form together a language may become complicated.

2.1.3 One-level morphology

Walther (2000a,2000b) suggests a solution for describing natural language reduplication using finite state methods. The idea is to enrich finite state automata with three new operations: *repeat*, *skip* and *self loops*. *Repeat* arcs allow moving backwards within a string and thus repeat a part of it (to model

reduplication). *Skip* arcs allow moving forwards in a string while suppressing the spell out of some of its letters (as in some cases of reduplication, only part of the word is reduplicated). *Self loop* arcs model infixation. In Walther (2000b) the above technique is used to describe Temiar reduplication, but there is no complexity analysis of the model. Moreover, this technique does not seem to be able to describe interdigitation.

2.1.4 The compile-replace algorithm

Beesley and Karttunen (2000) propose a solution to the problem of modeling non-concatenative morphological processes by finite state techniques. They describe a technique, called *compile-replace*, for constructing finite state transducers, that involves reapplying the regular-expression compiler to its own output. The main idea in this approach is to define networks using regular expressions but define the strings of an intermediate language so that they contain appropriate substrings that are themselves in the format of regular expressions. The compile-replace algorithm then reapplies the regular expression compiler to its own output, compiling the regular-expression substrings in the intermediate network and replacing them with the result of the compilation. The process is done by introducing new language symbols, "`^[`" and "`^]`", which indicate the appearance of a regular relation within the transducer language. The compile-replace algorithm finds delimited substrings of the form `^[ string ^]`, where `string` is just a string of symbols, which happens to have the format of a regular expression. Then the string is compiled as a regular expression, and the result replaces the delimited substring. This approach was used for describing Arabic stem interdigitation as a part of a finite state morphological analyzer for Arabic and for describing Malay full-stem reduplication (Beesley and Karttunen, 2000).

The compile-replace algorithm allows one to define non-concatenative morphological processes by a compact definition letting the compiler create the full machine. However, its result is the same result of the naïve automata, i.e., inefficient in its space complexity, and this is the main disadvantage of this technique. Furthermore, this is a compile-time mechanism rather than a theoretical, mathematically founded solution, as we desire.

2.1.5 Flag diacritics

Beesley (1998) suggests a way to constrain dependencies between separated morphemes in words. The method, called *flag diacritics*, adds features to symbols in regular expressions to enforce dependencies between separated parts of a string. The dependencies are forced by different kinds of unification actions. In this way, a small amount of finite memory is added, thus keeping the total size of the network relatively small. The main disadvantage of this method is that it is not formally defined, nor are its mathematical and computational properties proved. Furthermore, flag diacritics are manipulated at the level of the extended regular expressions, although it is clear that they are compiled into additional memory and operators in the networks themselves. The presentation of Beesley (1998) and Beesley and Karttunen (2003) does not explicate the implementation of such operators and does not provide an analysis of their complexity. Our approach is similar in spirit. However, we provide a complete mathematical and computational analysis of such extended networks, including a construction of the main closure properties. We present dedicated regular expression operations for non-concatenative processes and show how they are compiled into extended networks. We also prove that our model is indeed regular.

2.2 Extending the finite state model

Existing research concerned with automata with limited memory, albeit for other purposes, might provide insight into the desired solution.

2.2.1 Register Vector Grammar

Blank (Blank, 1985; Blank, 1989) presents a new model, called Register Vector Grammar (RVG), extending the finite state automata to be sensitive to context. The states and transitions of these automata are represented by ternary-valued vectors. The possible values are '+' (on), '-' (off) and '?' (don't care). The automaton uses two operators, *match* and *change*. *Match* is a function, taking as arguments two vectors and returning true if they match everywhere except for '?' values. *Change* is an asymmetric function, taking as arguments two vectors and returning a third vector, with definite

values '+' and '-' wherever they are located in the second argument, and the values of the first vector elsewhere. Each state is associated with two vectors: a condition vector restricting the transition between states and a change vector passing information forward. A special vector called *SynState* is responsible for manipulating the automaton. The SynState is a ternary-valued vector which the transition function matches and updates. In order to enter a state the match operation between SynState and the state condition vector should return true. After entering the state the change operator updates SynState according to the arc change vector. Thus, the '+' and '-' force some constraints and the '?' let some information pass through. In this way some restricted memory capabilities are given to the automata in the way that the condition vectors can convey multiple constrains and can produce multiple effects.

The main problem with this model is that although its time complexity is claimed to be linear, no formal proof of equivalence to regular languages is provided. We conjecture that the model indeed deviates from the class of regular languages. Furthermore, it is not clear how, given a set of morphological rules, the appropriate vectors and automata can be created.

2.2.2 Vectorized finite state automata

Kornai (1996) presents vectorized finite state automata, a new formalism within the finite state calculus, based on vectors and capable of modeling natural language problems. In this new model, both the states and the transitions are represented by vectors of elements of a partially ordered set. Two kinds of operations over vectors are defined: *unification* and *overwriting*. The *unification* of two symbols is defined in the usual way as the smallest partial binding that extends both. *Overwriting* of a symbol a over a symbol b is also defined in the usual way, and this operation (in contrast to unification) cannot fail. An arc from state u to state v (where both u and v are vectors) is a pair $(p; a)$ where both p and a are vectors, with the following meaning: the arc can be traversed only if the unification of u and p exists and v is the overwriting of a over u. The vectors need not be fully determined, as some of the elements can be unknown (free). In this way information can be moved through the transitions by the overwriting operation and traversing these transitions can be sanctioned through the unification operation. The free symbols are also the source of the efficiency of this new model, where a vector

with k free symbols actually represents t^k vectors, t being the number of different symbols that can be stored in the free places. Thus, this new model allows linear recognition time with efficient encoding of the network.

As one of the examples of the advantages of this new model Kornai (1996) shows it can efficiently solve the problem of 32-bit binary incrementor. The goal of this example is to construct a 32-bit incrementor, a transducer over $\Sigma = \{0, 1\}$ whose input is a number in 32 bit binary representation and whose output is the result of adding 1 to the input number. Constructing a transducer that performs addition by 1 on binary numbers results in a simple transducer with only 5 states and 12 arcs[2], but this transducer is neither sequential nor sequentiable. The problem is that since the input is scanned left to right but the carry moves right to left, the output of the first bit has to be delayed, possibly even until after the last input bit is scanned. Thus, for an n-bit binary incrementor, 2^n disjunctions have to be considered, and therefore a minimized transducer has to assign a separate state to each combination of bits, resulting in 2^n states and a similar number of transitions. Using vectorized finite state automata, a 32-bit incrementor is constructed where first, using overwriting, the input is scanned and stored by the vectors, and then, using unification, the result is calculated where the carry can be computed from right to left. We return to this example in example 5.3 (page 78), where we show how our new model can solve it efficiently too.

The new formalism presented by Kornai (1996) allows a significant reduction in the network size, but its main disadvantage lies in the fact that this is a completely new formalism, not based on the existing and commonly used finite state networks. Moreover, it is unclear how for a given problem, the corresponding network should be constructed: programming with vectorized automata seems to be unnatural, and no regular expression language is provided for them.

2.2.3 Automata over infinite alphabets

Kaminski and Francez (1994) present a computational model, which extends the familiar finite state automata model to the case of infinite alphabets. This new model is limited to recognizing only regular languages over infinite alphabets while maintaining closure under Kleene star and boolean operations,

[2] A complete explanation of the construction can be found in http://www.xrce.xerox.com/competencies/content-analysis/fsCompiler/fsexamples.html#Add1.

with the exception of closure under complementation. The familiar automaton is augmented with registers, whose number is fixed for each automaton and can vary from one automaton to another. A special symbol, '#', assumed not to belong to Σ, represents an empty register. Each state and arc are associated with registers. The state registers are responsible for dealing with new input, with the following meaning: If the automaton is located in a state, and the input symbol does not appear in any of the registers, then it will be written to the designated register of that state. The arc registers are part of the transition function, with the following meaning: the automaton is allowed to move from state s to state t while reading an input symbol σ, if σ is the content of the register associated with the arc connecting the two states. If a new input symbol is not yet the content of any of the registers, it will be written first to the register associated with the current state, and then the next move will be examined. A word belongs to the language accepted by this kind of automata if it has an accepting path, starting in the initial state and ending in a final state. This is the deterministic version of the automaton, which can be extended into a non-deterministic automaton by turning each value (arc or state) into a set of possible values. The main difference between the ordinary automaton and this new model is that while in the former the information is located on the arcs, in this new model the symbols are read and manipulated only through the registers and all the information in the automaton itself (i.e., arcs and states) act as pointers to the registers. Moreover, in this new model the states are not just temporary locations but also include vital data of the language.

The main advantage of this model is that it is well defined and all its closure properties are well established, making it a solid base for further use. However, it is not directly suitable for our purposes for the following reasons:

- It is designed to deal with infinite alphabets, and therefore it cannot distinguish between different symbols. It can identify different patterns but cannot distinguish between different symbols in the pattern as is often needed in natural languages.

- The transition function is divided into two functions, one writing into the registers (associated with the states) and another which is responsible for moving from one state to another according to the information in the registers (associated with the arcs). It seems beneficial to unify the two into one transition function associated with the arcs as in finite state automata.

- The register alphabet and the language alphabet are the same. We desire to separate the two, allowing the information stored in the registers to be more meaningful.

2.3 Predicates over transitions

van Noord and Gerdemann (2001b) extend finite state automata and transducers by replacing the atomic symbols over the transitions with predicates. Each predicate π is a total function such that for every $\sigma \in \Sigma$, $\pi(\sigma)$ is either true or false. The predicate groups together symbols over different transitions between the same two states, thus reducing the total number of transitions. Evidently, predicate augmented finite state machines are equivalent to ordinary finite state machines. The closure properties are dealt with as well. Operations on automata such as intersection, determinization, complementation and minimization and operations on transducers such as composition and determinization are done efficiently. The naïve approach for such operations will expand the predicate machine into the full machine, then perform the operation and in the end create the resulted predicate machine. It is clear though, that this approach looses all the benefits of such a model. van Noord and Gerdemann (2001b) show how such operations can be done directly, without expanding into the ordinary finite state machine. The goal of reducing the number of transitions is achieved in most of the transducers but in the worst case the number of transitions will remain the same.

Chapter 3

Finite state registered automata

We define a new model, FSRA, aimed at facilitating the expression of various non-concatenative morphological phenomena in an efficient way. The new model is reminiscent of Kaminski and Francez (1994) in the sense that it augments finite state automata with finite memory (registers) in a restricted way that saves space but does not add expressivity. The number of registers is finite, usually small, and eliminates the need to duplicate paths as it enables the automaton to 'remember' a finite number of symbols. In addition to being associated with an alphabet symbol, each arc is also associated with an action on the registers. There are two kinds of actions, *read* and *write*. The read action, denoted R, allows traversing an arc only if a designated register contains a specific symbol. The write action, denoted W, allows traversing an arc while writing a specific symbol into a designated register.

3.1 Definitions and examples

In this section we define FSRAs and give some examples of their capabilities.

Definition 3.1. *A finite state registered automaton (FSRA) is a tuple* $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F \rangle$, *where:*

- Q *is a finite set of states.*
- $q_0 \in Q$ *is the initial state.*
- Σ *is a finite alphabet (the language alphabet).*

- Γ *is a finite alphabet including the symbol '$\#$' (the registers alphabet).*
- $n \in \mathbb{N}$ *(indicating the number of registers).*
- $\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma \times Q$ *is the transition relation. The intuitive meaning of* δ *is as follows:*
  - $(s, \sigma, R, i, \gamma, t) \in \delta$ *where* $i > 0$ *implies that if* A *is in state* s*, the input symbol is* σ*, and the content of the* i*-th register is* γ*, then* A *may enter state* t*.*
  - $(s, \sigma, W, i, \gamma, t) \in \delta$ *where* $i > 0$ *implies that if* A *is in state* s *and the input symbol is* σ *then the content of the* i*-th register is changed into* γ *(overwriting whatever was there before) and* A *may enter state* t*.*
  - $(s, \sigma, R, 0, \#, t)$ *implies that if* A *is in state* s *and the input symbol is* σ *then* A *may enter state* t*. Notice that the content of register number 0 is always #. We use the shorthand notation* (s, σ, t) *for such transitions.*
- $F \subseteq Q$ *is the set of final states.*
- *The initial content of the registers is* $\#^n$*, meaning that the initial value of all the registers is 'empty'.*

We use meta variables u_i, v_i to range over Γ and u, v to range over Γ^n. A *configuration* of A is a pair (q, u), where $q \in Q$ and $u \in \Gamma^n$ (q is the current state and u represents the registers content). The set of all configurations of A is denoted by Q^c. The pair $q_0^c = (q_0, \#^n)$ is called the *initial configuration,* and configurations with the first component in F are called *final configurations*. The set of final configurations is denoted by F^c.

Let $u = u_0 u_1 \ldots u_{n-1}$ and $v = v_0 v_1 \ldots v_{n-1}$. Given a symbol $\alpha \in \Sigma \cup \{\epsilon\}$ and an FSRA A, we say that a configuration (s, u) *produces* a configuration (t, v), denoted $(s, u) \vdash_{\alpha, A} (t, v)$, iff either one of the following holds:

- There exists i, $0 \leq i \leq n-1$, and there exists $\gamma \in \Gamma$, such that the following hold:
  - $(s, \alpha, R, i, \gamma, t) \in \delta$.

- $u = v$.
- $u_i = v_i = \gamma$.

- There exists i, $0 \leq i \leq n-1$, and there exists $\gamma \in \Gamma$, such that the following hold:
  - $(s, \alpha, W, i, \gamma, t) \in \delta$.
  - For all k, $k \in \{0, 1, ..., n-1\}$, such that $k \neq i$, $u_k = v_k$.
  - $v_i = \gamma$.

The meaning of the product relation is as follows: a configuration c_1 produces a configuration c_2 iff the automaton can move from c_1 to c_2 when scanning the input α or without any input (when $\alpha = \epsilon$) in one step. If the register operation is R then the contents of the registers in the two configurations must be equal and in particular the contents of the designated register in the two configurations should be the expected symbol (γ). If the register operation is W then the contents of the registers in the two configurations is equal except for the designated register, whose contents in the produced configuration should be the expected symbol (γ).

A *run* of A on w is a sequence of configurations $c_0, ..., c_r$ such that $c_0 = q_0^c$, $c_r \in F^c$ and for every k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A} c_k$ and $w = \alpha_1 ... \alpha_r$. An FSRA A *accepts* a word w if there exists a run of A on w. Notice that $|w|$ might be less than r since some of the α_i might be ϵ. The *language* recognized by an FSRA A, denoted by $L(A)$, is the set of words over Σ^* accepted by A.

Example 3.1. *Consider again example 1.1 (page 6). By using the model defined above we can construct an efficient FSRA accepting all the possible combinations of roots and patterns and only them. If the number of roots is r, we define an FSRA*

$$A = \langle \{q_0, q_1, \ldots, q_{2r+2}, q_f\}, q_0, \{a, b, c, \ldots, z, ht, ut\}, \{ht\square\square\square ut, h\square\square\square a, m\square\square\square, \#\}, 2, \delta, \{q_f\} \rangle$$

where

$$\delta = \{(q_0, ht, W, 1, ht\square\square\square ut, q_1), (q_0, h, W, 1, h\square\square\square a, q_1), (q_0, m, W, 1, m\square\square\square, q_1),$$

$$(q_{2r+2}, ut, R, 1, ht\square\square\square ut, q_f), (q_{2r+2}, a, R, 1, h\square\square\square a, q_f), (q_{2r+2}, \epsilon, R, 1, m\square\square\square, q_f)\}$$

$$\bigcup$$

$\{(q_1, \alpha_1, q_i), (q_i, \alpha_2, q_{i+1}), (q_{i+1}, \alpha_3, q_{2r+2}) | \ 2 \leq i \leq 2r$ *and* $\alpha_1\alpha_2\alpha_3$ *is the* i*'th root*$\}$.

This automaton has the graph representation shown in Figure 3.1. The number of its states is $2r + 4$ *(just like the FSA in Figure 1.2), that is,* $O(r)$*, and in particular independent of the number of patterns. The number of arcs is also reduced from* $O(r \times p)$*, where p indicates the number of patterns, to* $O(r + p)$.

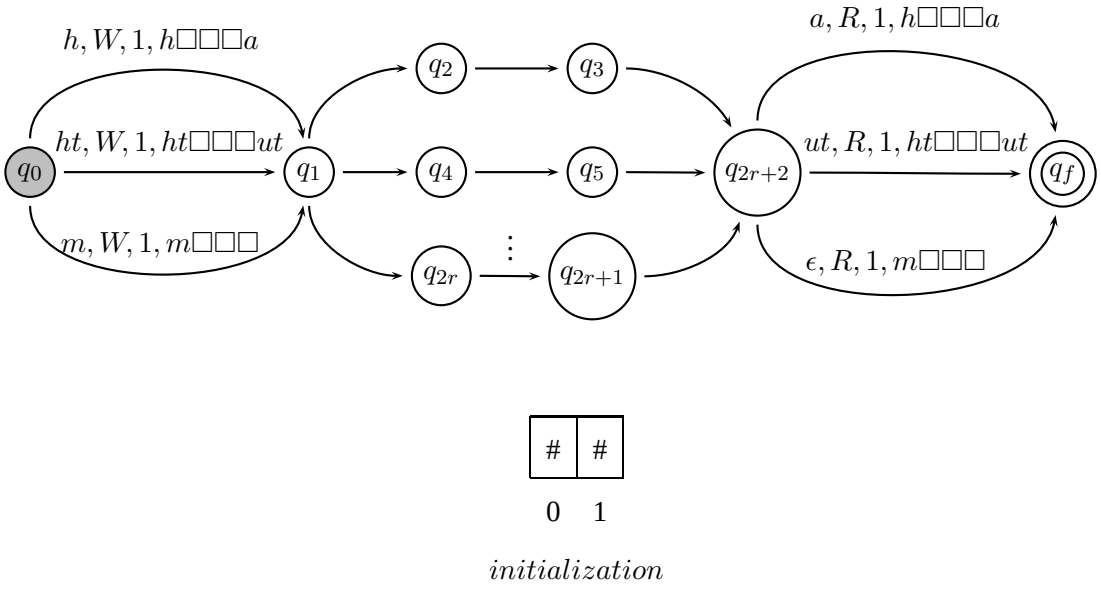


Figure 3.1: FSRA

Example 3.2. *Consider again example 1.2 (page 7). The FSRA defined by the diagram in Figure 3.2 also accepts the same language. This automaton has seven states and will have seven states for any number of roots. The number of arcs is also reduced to* $3r + 3$.

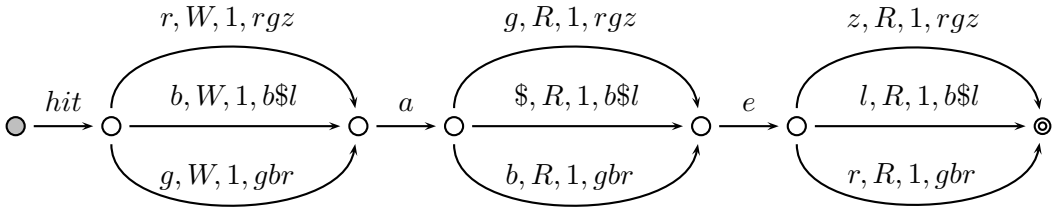


Figure 3.2: FSRA for the pattern hit□a□e□

3.2 Equivalence to regular languages

Next we show that finite state registered automata and finite state automata recognize the same class of languages.

Proposition 3.1. *Every finite state automaton has an equivalent FSRA.*

Proof. Given an FSA A, add to A one register to construct an equivalent FSRA A'. Since every transition (s, σ, t) in an FSRA is a shorthand notation for $(s, \sigma, R, 0, \#, t)$, every transition in A is also a legal transition in A'. Trivially $L(A') = L(A)$. □

Proposition 3.2. *Every FSRA has an equivalent finite state automaton.*

We prove by constructing an equivalent FSA to a given FSRA. The construction is based on the fact that in FSRAs n is a finite number and Γ and Q are finite sets, hence the number of configurations is finite. The FSA's states are the configurations of the FSRA and the transition function simulates the product relation. Notice that product between configurations in an FSRA is dependent on Σ only, similarly to the transition function in an FSA. The constructed FSA is a non-deterministic one with possible ϵ-moves.

Proof. Let $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$ be an FSRA. Construct a finite state automaton $A' = \langle Q', q_0', \Sigma, \delta', F'\rangle$, where:

- $Q' = Q^c$. Every state in A' is a configuration in A. Notice that n is a finite number and Γ and Q are finite sets, therefore A' has finitely many states.
- $q_0' = q_0^c$ where $q_0^c = (q_0, \#^n)$. The initial state of A' is the initial configuration of A.
- $F' = F^c$.
- $\delta' = \{(s', \alpha, t') |\ \alpha \in \Sigma \cup \{\epsilon\}\ ,\ s', t' \in Q' \ and\ s' \vdash_{\alpha,A} t'\}$. An arc connects states s' and t' in A' if A can move from the configuration s' to the configuration t' in one step when reading α. Notice that since the original FSRA can include ϵ-moves, so does the constructed FSA.

We now show that $L(A) = L(A')$:

Assume that $w \in L(A)$. Then there exists a series of configurations $c_0, \ldots, c_r$ such that $c_0 = q_0^c$,

$c_r \in F^c$ and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A} c_k$ and $w = \alpha_1 \ldots \alpha_r$. By the definition of Q' and δ' it follows that $c_0, \ldots, c_r \in Q'$, $c_0 = q'_0$, $c_r \in F'$ and for all k, $1 \leq k \leq r$, $(c_{k-1}, \alpha_k, c_k) \in \delta'$; therefore A' accepts $w = \alpha_1 \ldots \alpha_r$ and $w \in L(A')$. Hence $L(A) \subseteq L(A')$.

Assume that $w \in L(A')$. Then there exists a series of states $q'_0, \ldots, q'_n \in Q'$ where $q'_n \in F'$ and for all i, $0 \leq i \leq n-1$, $(q_i, w_{i+1}, q_{i+1}) \in \delta'$ and $w = w_1 \ldots w_n$. By the definition of A' each q_i (for some i, $0 \leq i \leq n$) is a configuration of A and q'_0 is the initial configuration of A and $q'_n \in F^c$. By the definition of δ', it follows that for all i, $0 \leq i \leq n-1$, $q'_i \vdash_{w_{i+1}, A} q'_{i+1}$, thus $q'_0, \ldots, q'_n$ is a run of A on $w = w_1, \ldots, w_n$. Therefore A accepts w and $w \in L(A)$. Hence $L(A') \subseteq L(A)$.

□

Notice that the number of configurations in A is $|Q| \times |\Gamma|^n$, therefore the growth in the number of states when constructing A' from A might be in the worst case exponential in the number of registers. In other words, the move from FSAs to FSRAs can yield an exponential reduction in the size of the network. In the next section it will be shown that the reduction in the number of states can be even more dramatic.

3.3 An extension of FSRAs: multiple register actions

The FSRA model defined above allows only one register operation on each transition. We extend the existing model to allow up to k register operations on each transition, where k is determined for each automaton separately. The register operations are defined as a sequence (rather than a set), in order to allow more than one operation on the same register over one transition. FSRA-k allow further reduction of the network size for some automata as well as other advantages that will be discussed later.

Definition 3.2. *An order-k finite state registered automaton (FSRA-k) is a tuple* $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$*, where:*

- $Q, q_0, \Sigma, \Gamma, n, F$ *and the initial content of the registers are as before.*
- $k \in \mathbb{N}$ *(indicating the maximum number of register operations allowed on each arc).*

- *Let* $Actions_n^\Gamma = \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma$*, then*

$$\delta \subseteq Q \times \Sigma \cup \{\epsilon\} \times \left(\bigcup_{j=1}^{k} \{ \langle a_1, ..., a_j \rangle \mid \textit{for all } \; i, \;\; 1 \leq i \leq j, \; a_i \in Actions_n^\Gamma \} \right) \times Q$$

is the transition relation. δ *is extended to allow each transition to be associated with a series of up to k operations on the registers. Each operation has the same meaning as before.*

The register operations are done in the order in which they are specified. Thus, $(s, \sigma, \langle a_1, ..., a_i \rangle, t) \in \delta$ where $i \leq k$ implies that if A is in state s, the input symbol is σ and all the register operations $a_1, ..., a_i$ are executed successfully, then A may enter state t. More formally, given $a \in Actions_n^\Gamma$ we define a relation over Γ^n, denoted $u \Vdash_a v$ for $u, v \in \Gamma^n$. We define $u \Vdash_a v$ where $u = u_0 \ldots u_{n-1}$ and $v = v_0 \ldots v_{n-1}$ iff the following holds:

- if $a = (R, i, \gamma)$ for some i, $0 \leq i \leq n-1$ and for some $\gamma \in \Gamma$ then $u = v$ and $u_i = v_i = \gamma$.
- if $a = (W, i, \gamma)$ for some i, $0 \leq i \leq n-1$ and for some $\gamma \in \Gamma$ then for all $k \in \{0, 1, \ldots, n-1\}$ such that $k \neq i$, $u_k = v_k$ and $v_i = \gamma$.

The above relation is expanded for series over $Actions_n^\Gamma$. Given a series $\langle a_1, ..., a_p \rangle \in (Actions_n^\Gamma)^p$ where $p \in \mathbb{N}$, we define a relation over Γ^n denoted $u \Vdash_{\langle a_1, ..., a_p \rangle} v$ for $u, v \in \Gamma^n$. We define $u \Vdash_{\langle a_1, ..., a_p \rangle} v$ iff the following holds:

- if $p = 1$ then $u \Vdash_{a_1} v$.
- if $p > 1$ then there exists $w \in \Gamma^n$ such that $u \Vdash_{a_1} w$ and $w \Vdash_{\langle a_2, ..., a_p \rangle} v$.

Let $u, v \in \Gamma^n$. Given a symbol $\alpha \in \Sigma \cup \{\epsilon\}$ and an FSRA-k A, we say that a configuration (s, u) *produces* a configuration (t, v), denoted $(s, u) \vdash_{\alpha, A} (t, v)$, iff there exist $\langle a_1, \ldots, a_p \rangle \in (Actions_n^\Gamma)^p$ for some $p \in \mathbb{N}$ such that $(s, \alpha, \langle a_1, \ldots, a_p \rangle, t) \in \delta$ and $u \Vdash_{\langle a_1, ..., a_p \rangle} v$.

A *run* of A on w is a sequence of configurations $c_0, ..., c_r$ such that $c_0 = q_0^c$, $c_r \in F^c$ and for every l, $1 \leq l \leq r$, $c_{l-1} \vdash_{\alpha_l, A} c_l$ and $w = \alpha_1 ... \alpha_r$. An FSRA-k A *accepts* a word w if there exists a run of A on w. The language recognized by an FSRA-k A, denoted by $L(A)$, is the set of words over Σ^* accepted by A.

Example 3.3. *Consider the Arabic nouns qamar (moon), kitaab (book), $ams (sun) and daftar (notebook). The definite article in Arabic is the prefix ''al', which is realized as ''al' when preceding most consonants; however, the 'l' of the prefix assimilates to the first consonant of the noun when the latter is 'd', '$', etc. Furthermore, Arabic distinguishes between definite and indefinite case markers. For example, nominative case is realized as the suffix 'u' on definite nouns, 'un' on indefinite nouns. Examples of the different forms of Arabic nouns are:*

word	*nominative definite*	*nominative indefinite*
qamar	*'alqamaru*	*qamarun*
kitaab	*'alkitaabu*	*kitaabun*
$ams	*'a$$amsu*	*$amsun*
daftar	*'addaftaru*	*daftarun*

The FSRA-2 of Figure 3.3 accepts all the nominative definite and indefinite forms of the above nouns. In order to account for the assimilation, register 2 stores information about the actual form of the definite article. Furthermore, to ensure that definite nouns occur with the correct case ending, register 1 stores information of whether or not a definite article was seen.

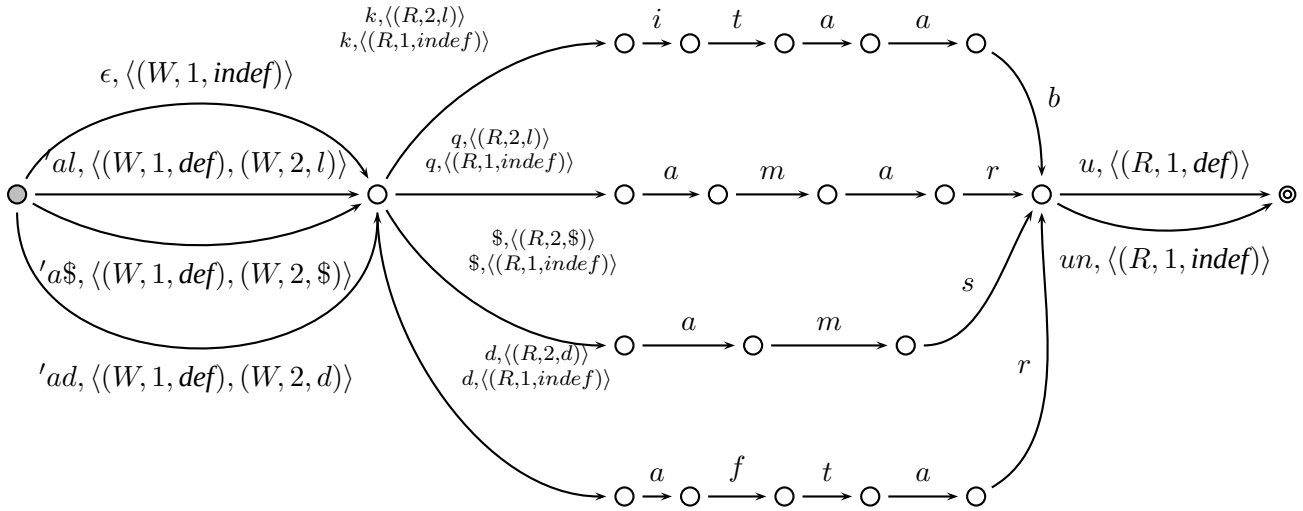


Figure 3.3: FSRA-2 for Arabic nominative definite and indefinite nouns

Next we show that FSRA-k and FSRAs recognize the same class of languages.

Proposition 3.3. *Every FSRA has an equivalent FSRA-k.*

Proof. Every FSRA is an FSRA-k for $k = 1$, thus evidently the proposition holds. □

For the reverse direction we show how to construct an equivalent FSRA (or FSRA-1) A' given an FSRA-k A. Each transition in A is replaced by a series of transitions in A', each of which performs one operation on the registers. The first transition in the series deals with the new input symbol and the rest are ϵ-transitions. This construction requires additional states to enable the addition of transitions. Each transition in A that is replaced requires addition of as many states as the number of register operations performed on this transition minus one.

Proposition 3.4. *Every FSRA-k has an equivalent FSRA.*

Proof. Let $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$ be an FSRA-k. For every $d \in \delta$ we define the *rank* of d, denoted $r(d)$, as the number of register operations specified for this transition. For all $d \in \delta$, $1 \leq r(d) \leq k$. Construct an FSRA $A' = \langle Q', q_0', \Sigma', \Gamma', n', \delta', F'\rangle$, where:

- $Q' = Q \cup \bigcup_{d\in\delta} \left\{t_1^d, t_2^d, ..., t_{r(d)-1}^d\right\}$ where each t_i^d is a fresh node $\notin Q$ and $t_i^d \neq t_j^{d'}$ if $i \neq j$ or $d \neq d'$.
- $q_0' = q_0$
- $\Sigma' = \Sigma$
- $\Gamma' = \Gamma$
- $n' = n$
- $F' = F$
- $\delta' = \{(s, \sigma, a_1, t_1^d), (t_1^d, \epsilon, a_2, t_2^d), ..., (t_{r(d)-2}^d, \epsilon, a_{r(d)-1}, t_{r(d)-1}^d), (t_{r(d)-1}^d, \epsilon, a_{r(d)}, t) \mid d = (s, \sigma, \langle a_1, ..., a_{r(d)}\rangle, t) \in \delta \ \wedge \ r(d) > 1\} \ \cup \ \{d \mid d \in \delta \ \wedge \ r(d) = 1\}$. δ is extended into δ' in the following way: for every $d \in \delta$ if $r(d) > 1$ then it is replaced by a series of arcs in δ', each performing only one register operation. The needed new states are added to Q as well. Otherwise, if $r(d) = 1$ then $d \in \delta'$. This process is illustrated in Figure 3.4.

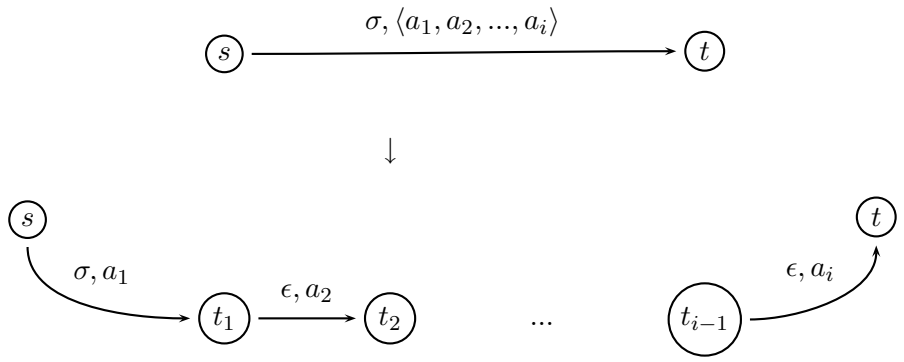


Figure 3.4: Arcs replacement for FSRA-k

Evidently, $L(A) = L(A')$. □

From now on, the term FSRA will be used to denote FSRA-k. Simple FRSA will be referred to as FSRA-1. For the sake of emphasis, however, the term FSRA-k will still be used in some cases.

FSRA is very space-efficient finite state device. The next proposition shows how ordinary finite state automata can be encoded efficiently by the FSRA-2 model. Given a finite state automaton A, an equivalent FSRA-2 A' is constructed. A' has three states and two registers (actually only one register is used since register number 0 is never addressed). One state functions as a representative for the final states in A, another one functions as a representative for the non-final states in A and the third as an initial state. The registers alphabet consists of the states of A and the symbol '$\#$'. Each arc in A has an equivalent arc in A' with two register operations. The first reads the current state of A from the register and the second writes the new state into the register. If the source state of a transition in A is a final state then the source state of the corresponding transition in A' will be the final states representative; if the source state of a transition in A is a non-final state then the source state of the corresponding transition in A' will be the non-final states representative. The same holds also for the target states. The purpose of the initial state is to write the start state in A into the register. In this way A' simulates the behavior of A. Notice that the number of arcs in A' equals the number of arcs in A plus one (as many implementations of finite state devices use space that is a function of the number

of arcs, there may not be a clear practical advantage to the following proposition).

Proposition 3.5. *Every finite state automaton has an equivalent FSRA-2 with three states and two registers.*

Proof. Let $A = \langle Q, q_0, \Sigma, \delta, F\rangle$ be an FSA and let $f : Q \to \left\{q_f, q_{nf}\right\}$ be a total function defined by

$$f(q) = \begin{cases} q_f & q \in F \\ q_{nf} & q \notin F \end{cases}$$

Construct an FSRA-2 $A' = \langle Q', q_0', \Sigma', \Gamma', 2, 2, \delta', F'\rangle$, where:

- $Q' = \{q_0', q_{nf}, q_f\}$. q_0' is the initial state, q_f is the final states representative and q_{nf} is the non-final states representative
- $\Sigma' = \Sigma$
- $\Gamma = Q \cup \{\#\}$
- $F' = \{q_f\}$
- $\delta' = \{(f(s), \sigma, \langle (R, 1, s), (W, 1, t)\rangle, f(t)) \mid (s, \sigma, t) \in \delta\} \cup \{(q_0', \epsilon, \langle (W, 1, q_0)\rangle, f(q_0))\}$.

We now show that $L(A) = L(A')$:

Assume that $w \in L(A)$. Then there exists a series of states $q_0, \ldots, q_n \in Q$ such that $q_n \in F$ and for all i, $0 \leq i \leq n-1$, $(q_i, w_{i+1}, q_{i+1}) \in \delta$ and $w = w_1 \ldots w_n$. By the definition of δ', for all i, $0 \leq i \leq n-1$, $(f(q_i), w_{i+1}, \langle (R, 1, q_i), (W, 1, q_{i+1})\rangle, f(q_{i+1})) \in \delta'$ and in addition $\langle \#, q_i\rangle \Vdash_{\langle (R,1,q_i),(W,1,q_{i+1})\rangle} \langle \#, q_{i+1}\rangle$ (Notice that $\langle \#, q_i\rangle$ denotes the content of the registers and means that register 1 contains the symbol '$\#$' and register 2 contains the symbol 'q_i'; by the definition of Γ, all the states in A are symbols in Γ). Therefore for all i, $0 \leq i \leq n-1$, $(f(q_i), \langle \#, q_i\rangle) \vdash_{w_{i+1}, A'} (f(q_{i+1}), \langle \#, q_{i+1}\rangle)$. Since $q_n \in F$, $f(q_n) = q_f$ and therefore $(f(q_n), \langle \#, q_n\rangle) \in F'^c$. In addition, by the definition of δ', $(q_0', \epsilon, \langle (W, 1, q_0)\rangle, f(q_0)) \in \delta'$ and $\langle \#, \#\rangle \Vdash_{\langle (W,1,q_0)\rangle} \langle \#, q_0\rangle$, therefore $(q_0', \langle \#, \#\rangle) \vdash_{\epsilon, A'} (f(q_o), \langle \#, q_0\rangle)$ where $(q_0', \langle \#, \#\rangle)$ is the initial configuration of A'. In sum, we have a series of configurations $(q_0', \langle \#, \#\rangle), (f(q_0), \langle \#, q_0\rangle), \ldots, (f(q_i), \langle \#, q_i\rangle), \ldots, (f(q_n), \langle \#, q_n\rangle)$ such that $(q_o', \langle \#, \#\rangle) \vdash_{\epsilon, A'} (f(q_0), \langle \#, q_0\rangle)$ and for all i, $0 \leq i \leq n-1$, $(f(q_i), \langle \#, q_i\rangle)$

$\vdash_{w_{i+1},A'} (f(q_{i+1}), \langle \#, q_{i+1} \rangle)$ and $(q'_0, \langle \#, \# \rangle)$ is the initial configuration of A' and $(f(q_n), \langle \#, q_n \rangle)$ is a final configuration of A', hence $w \in L(A')$. Hence $L(A) \subseteq L(A')$.

Assume that $w \in L(A')$. Then there exists a series of configurations $c_0, \ldots, c_r$ such that $c_0 = {q'_0}^c = (q'_0, \langle \#, \# \rangle)$, $c_r \in {F'}^c$ and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A'} c_k$ and $w = \alpha_1 \ldots \alpha_r$. $c_0 = {q'_0}^c = (q'_0, \langle \#, \# \rangle)$ and since by the definition of δ', the only arc that goes out from q'_0 is $(q'_0, \epsilon, \langle (W, 1, q_0) \rangle, f(q_0))$ (notice that for all $q \in Q$, $f(q) \neq q'_0$) and since $c_0 \vdash_{\alpha_1, A'} c_1$ we can conclude that $\alpha_1 = \epsilon$ and $c_1 = (f(q_0), \langle \#, q_0 \rangle)$.

Lemma 3.1. *For all n, $2 \leq n \leq r$, there exist $q_{n-1}, q_{n-2} \in Q$ such that $(f(q_{n-2}), \alpha_n, \langle (R, 1, q_{n-2}), (W, 1, q_{n-1}) \rangle, f(q_{n-1})) \in \delta'$ and $c_n = (f(q_{n-1}), \langle \#, q_{n-1} \rangle)$.*

Proof. Notice first that for all $s \in Q$, by the definition of f, $f(s) \neq q'_0$ and therefore by the definition of δ', if there is an arc that goes out of $f(s)$ it must be of the form $(f(s), \sigma, \langle (R, 1, s), (W, 1, t) \rangle, f(t))$ for some $t \in Q$ and some $\sigma \in \Sigma$. We denote this observation by $\star$. We prove the lemma by induction on n.

For $n = 2$, we have seen that $c_1 = (f(q_0), \langle \#, q_0 \rangle))$ and we also know that $c_1 \vdash_{\alpha_2, A'} c_2$ and therefore from $\star$ there is $q_1 \in Q$ such that $(f(q_0), \alpha_2, \langle (R, 1, q_0), (W, 1, q_1) \rangle, f(q_1)) \in \delta'$ and $\langle \#, q_0 \rangle \Vdash_{\langle (R,1,q_0),(W,1,q_1) \rangle} \langle \#, q_1 \rangle$ and therefore $c_2 = (f(q_1), \langle \#, q_1 \rangle)$. Assume the correctness of the lemma for $n - 1$. By the induction hypothesis, $c_{n-1} = (f(q_{n-2}), \langle \#, q_{n-2} \rangle)$. We know that $c_{n-1} \vdash_{\alpha_n, A'} c_n$ and therefore from $\star$ we can conclude that there exists $q_{n-1} \in Q$ such that $(f(q_{n-2}), \alpha_n, \langle (R, 1, q_{n-2}), (W, 1, q_{n-1}) \rangle, f(q_{n-1})) \in \delta'$ and $\langle \#, q_{n-2} \rangle \Vdash_{\langle (R,1,q_{n-2}),(W,1,q_{n-1}) \rangle} \langle \#, q_{n-1} \rangle$ and hence $c_n = (f(q_{n-1}), \langle \#, q_{n-1} \rangle)$. □

Let us return to the main proof. We obtain that for all n, $2 \leq n \leq r$, there exist q_{n-1}, q_{n-2} such that $(f(q_{n-2}), \alpha_n, \langle (R, 1, q_{n-2}), (W, 1, q_{n-1}) \rangle, f(q_{n-1})) \in \delta'$ and for all n, $1 \leq n \leq r$, $c_n = (f(q_{n-1}), \langle \#, q_{n-1} \rangle)$. By the definition of δ' it follows that for all n, $2 \leq n \leq r$, there exist $q_{n-2}, q_{n-1} \in Q$ such that $(q_{n-2}, \alpha_n, q_{n-1}) \in \delta$. We obtain a series of transitions in A: (q_0, α_2, q_1), $(q_1, \alpha_3, q_2), \ldots, (q_{r-2}, \alpha_r, q_{r-1})$. Since $c_r = (f(q_{r-1}, \langle \#, q_{r-1} \rangle) \in {F'}^c$ it follows that $f(q_{r-1}) =$

q_f and therefore by the definition of f, $q_{r-1} \in F$. Therefore $\alpha_2 \dots \alpha_r \in L(A)$. Since $\alpha_1 = \epsilon$, $w = \alpha_1 \dots \alpha_r = \alpha_2 \dots \alpha_r \in L(A)$, hence $L(A') \subseteq L(A)$. □

Example 3.4. *Consider L, the language over $\Sigma = \{a, b\}$ consisting of all the words in which the number of a's is a multiple of 6. The minimized finite state automaton that recognizes L is illustrated in Figure 3.5. This automaton can be compiled into an FSRA-2 as described above. The resulting automaton is illustrated in Figure 3.6.*

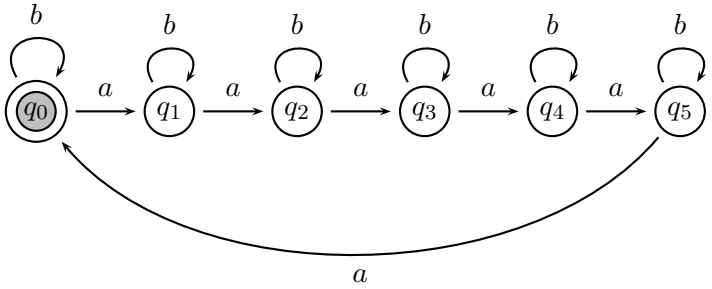


Figure 3.5: Fsa - number of a's instances divides by 6

3.4 Closure properties

The equivalence shown in Section 3.2 between the class of languages recognized by finite state automata and finite state registered automata immediately implies that finite state registered automata maintain the closure properties of regular languages. Thus, performing the regular operations on finite state registered automata can be easily done by converting them first into finite state automata. However, as shown above, such a conversion may result in an exponential increase in the size of the automaton, invalidating the advantages of this new model. Therefore, we show how these operations can be done directly on finite state registered automata. The constructions are mostly based on the standard constructions for FSAs with some essential modifications.

In what follows, let $A_1 = \langle Q_1, q_0^1, \Sigma_1, \Gamma_1, n_1, k_1, \delta_1, F_1 \rangle$ and $A_2 = \langle Q_2, q_0^2, \Sigma_2, \Gamma_2, n_2, k_2, \delta_2, F_2 \rangle$ be finite state registered automata. The variable op is used as a meta-variable over $\{R, W\}$, the variables a, b are used as meta-variables over $Actions_n^\Gamma$ and the variables $\vec{a}$, $\vec{b}$, $\vec{c}$ are used as meta-variables over $\left(Actions_n^\Gamma\right)^+$ (i.e., $\vec{a}$, $\vec{b}$, $\vec{c}$ represent vectors of register operations).
Define a total function $\mathit{shift}_{n_1} : \left(Actions_{n_2}^{\Gamma_2}\right)^+ \to \left(Actions_{n_2+n_1}^{\Gamma_2} \setminus Actions_{n_1}^{\Gamma_2}\right)^+$ by:

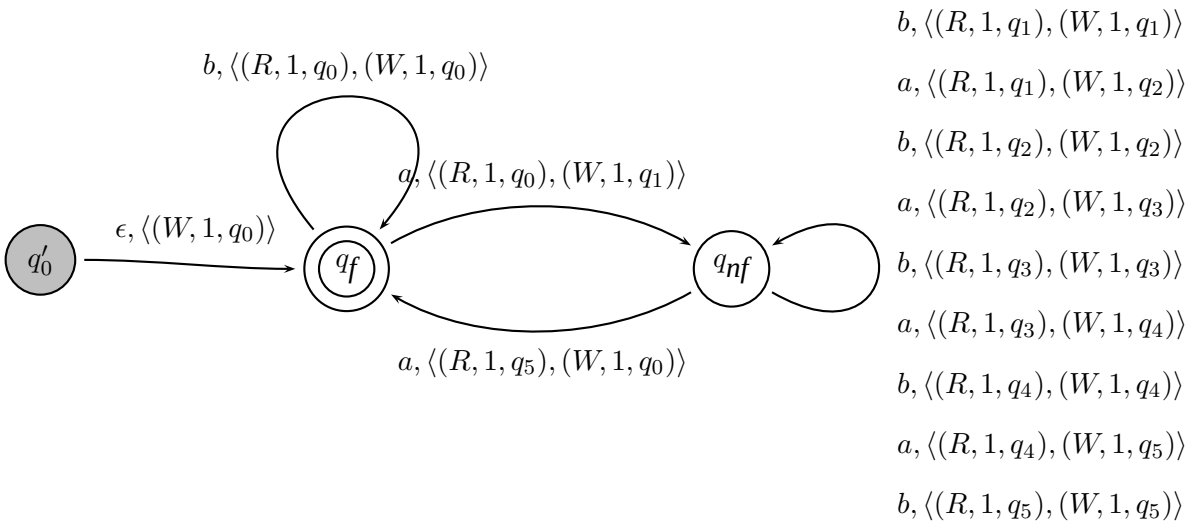


Figure 3.6: FSRA-2 for the FSA of example 3.4

$$\textit{shift}_{n_1}\left((op_1,i_1,\gamma_1),\ldots,(op_j,i_j,\gamma_j)\right) = \left((op_1,i_1+n_1,\gamma_1),\ldots,(op_j,i_j+n_1,\gamma_j)\right)$$

This function takes a series of register operations and shifts the registers on which the different operations perform by n_1 registers. Notice that $\textit{shift}_{n_1}$ is one to one and onto function, therefore it has an inverse function, which will be used in the sequel.

3.4.1 Union

Let $q_0 \notin Q_1 \cup Q_2$ and assume that Q_1 and Q_2 are disjoint sets. We construct an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$ to recognize $L(A_1) \cup L(A_2)$, where:

- $Q = \{q_0\} \cup Q_1 \cup Q_2$.
- $\Sigma = \Sigma_1 \cup \Sigma_2$.

- $\Gamma = \Gamma_1 \cup \Gamma_2$.
- $n = max\{n_1, n_2\}$.
- $k = max\{k_1, k_2\}$.
- $F = F_1 \cup F_2$.
- $\delta = \delta_1 \cup \delta_2 \cup \{(q_0, \epsilon, q_0^1), (q_0, \epsilon, q_0^2)\}$.

Notice that in any specific run of A, the computation goes through just one of the original automata, therefore there is no need to take the sum of the registers of the two automata since the set of registers can be used for strings of $L(A_1)$ or $L(A_2)$ as needed. The formal proof that $L(A) = L(A_1) \cup L(A_2)$ is suppressed.

3.4.2 Concatenation

We show two different constructions of an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ to recognize $L(A_1) \cdot L(A_2)$. Concatenation in finite state automata is done simply by leaving only the accepting states of the second automaton as accepting states and adding an ϵ-arc from every accepting state of the first automaton to the initial state of the second automaton. Doing just this in FSRA is not enough because using the same registers might cause undesired effects: the result might be effected by the content left in the registers after dealing with a substring from $L(A_1)$. Thus, this basic construction is used with some modifications to solve this problem. In the first alternative we employ more registers in the FSRA. In this way when dealing with a substring from $L(A_1)$ the first n_1 registers are used, and when moving to deal with a substring from $L(A_2)$ the next n_2 registers are used. The second alternative uses additional register operations. These operations clear the content of the registers before handling the next substring from $L(A_2)$. This solution may be less intuitive, but it will be useful for Kleene closure, shown in section 3.4.3.

Alternative 1

Construct an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$, where:

- $Q = Q_1 \cup Q_2$.
- $q_0 = q_0^1$.
- $\Sigma = \Sigma_1 \cup \Sigma_2$.
- $\Gamma = \Gamma_1 \cup \Gamma_2$.
- $n = n_1 + n_2$. The number of registers is the sum of the registers of the two automata. Actually only $n = n_1 + n_2 - 1$ registers are needed since register 0 in the two automata can be unified, but for the sake of convenience we leave them apart.
- $k = max\{k_1, k_2\}$.
- $F = F_2$. The final states are the final states of A_2.
- $\delta = \delta_1 \ \cup \ \{(f, \epsilon, q_0^2) \mid f \in F_1\} \cup \{(s, \sigma, \langle \mathit{shift}_{n_1}(\vec{a})\rangle, t) \mid (s, \sigma, \langle \vec{a}\rangle, t) \in \delta_2\}$. Notice that $\vec{a}$ is used as a meta-variable over $\left(Actions_{n_2}^{\Gamma_2}\right)^+$.

Alternative 2

Construct an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$, where:

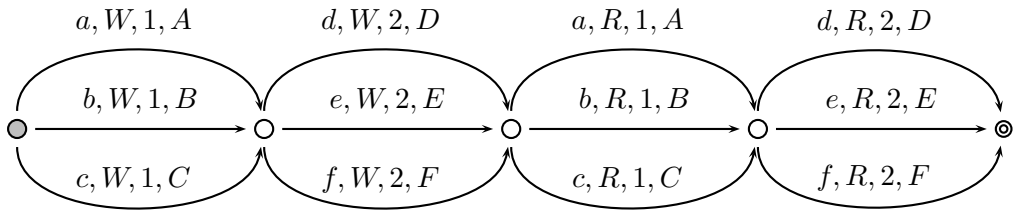
- $Q = Q_1 \ \cup \ Q_2$.
- $q_0 = q_0^1$.
- $\Sigma = \Sigma_1 \cup \Sigma_2$.
- $\Gamma = \Gamma_1 \cup \Gamma_2$.
- $n = max\{n_1, n_2\}$. The number of registers is the maximum number needed in A_1 or A_2.
- $k = max\{k_1, k_2, n_2 - 1\}$. The value of k is the maximal value of registers operations that can be performed on an arc. Notice that $n_2 - 1$ is the number of register operations performed when traversing an arc from a final state in A_1 to the initial state in A_2.
- $F = F_2$. The final states are the final states of A_2.

- $\delta = \delta_1 \cup \{(f, \epsilon, \langle (W, 1, \#), \ldots, (W, n_2 - 1, \#) \rangle, q_0^2) \mid f \in F_1\} \cup \delta_2$. The intuitive meaning of δ is that δ_1 handles the first substring from $L(A_1)$, then the added arcs delete the contents of the registers, leaving them empty and ready to handle the second substring from $L(A_2)$. Only the contents of the first $n_2 - 1$ registers after register 0 have to be deleted because they are the only ones to be used from that point on.

Again, we suppress the proof that $L(A) = L(A_1) \cdot L(A_2)$.

Example 3.5. *Consider A_1 and A_2, the FSRAs defined in Figure 3.7. Observe that $L(A_1) = \{\sigma_1\sigma_2\sigma_1\sigma_2 \mid \sigma_1 \in \{a, b, c\}$, $\sigma_2 \in \{d, e, f\}\}$ and $L(A_2) = \{\sigma_1\sigma_2\sigma_2\sigma_1 \mid \sigma_1, \sigma_2 \in \{g, h, k\}\}$. The FSRA's defined in Figure 3.8 and Figure 3.9 both accept the language $L(A_1) \cdot L(A_2)$, using extra registers (alternative 1) and extra register operations (alternative 2), respectively.*

A_1 for the concatenation example:



A_2 for the concatenation example:

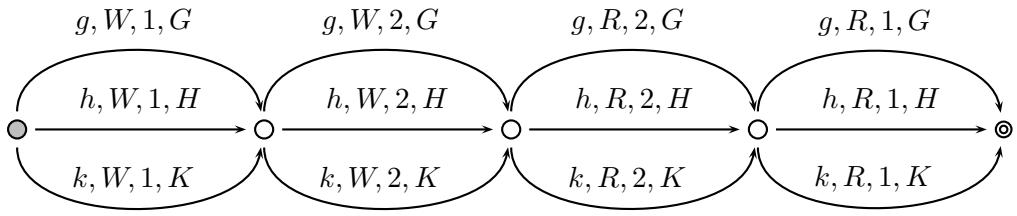


Figure 3.7: FSRAs for the concatenation example

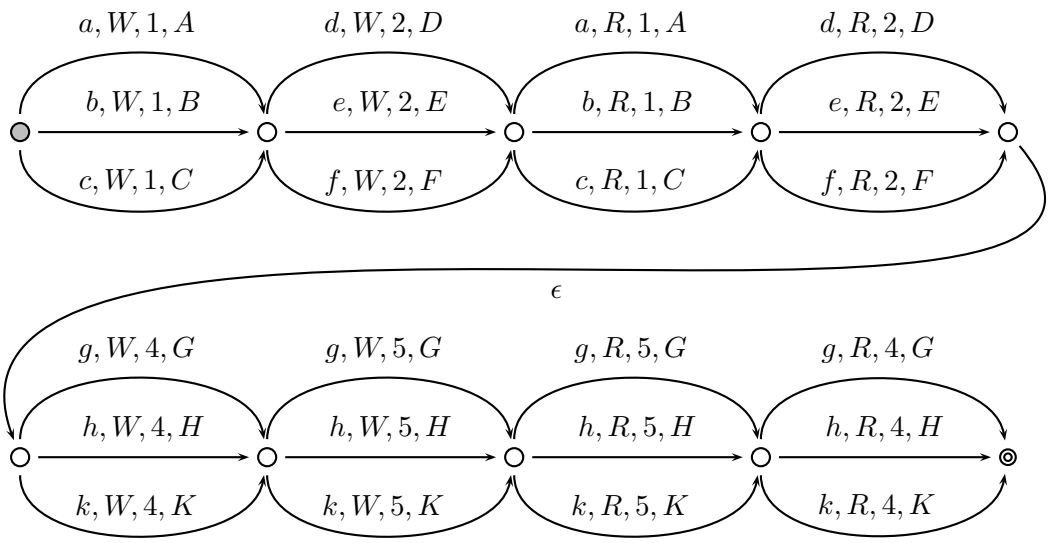


Figure 3.8: Concatenation example – adding registers

3.4.3 Kleene closure

The following construction is based on the concatenation construction. Notice that it cannot be based on the first alternative (adding registers) due to the fact that the number of iterations in Kleene star is not limited, and therefore the number of registers needed cannot be bounded. Thus, we use the second alternative (adding register operations to delete registers content). We construct an FSRA-k $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ to recognize $L(A_1)^*$, where:

- $Q = Q_1$
- $q_0 = q_0^1$.
- $\Sigma = \Sigma_1$.
- $\Gamma = \Gamma_1$.
- $n = n_1$.
- $k = max\{k_1, n - 1\}$. Notice that if $n = 1$ then $k = k_1$ and the resulting FSRA-k is a legal one.

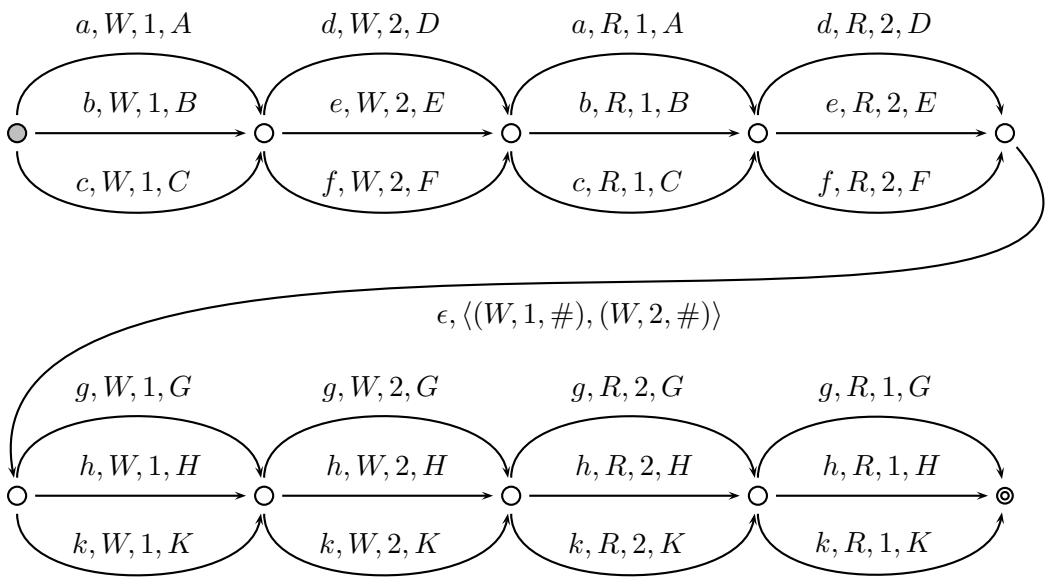


Figure 3.9: Concatenation example – adding register operations

- $F = F_1 \cup \{q_0\}$. The final states are the final states of A_1 with the initial state (to allow the empty string).

- $\delta = \delta_1 \cup \{(f, \epsilon, \langle (W, 1, \#), \ldots, (W, n-1, \#) \rangle, q_0) \mid f \in F\}$. The intuitive meaning of δ is that δ_1 handles a substring from $L(A_1)$, then the added register operations delete the contents of the registers, leaving them ready to handle the next substring from $L(A_1)$.

Again, we suppress the proof that $L(A) = L(A_1)^*$.

3.4.4 Intersection

For the intersection construction, assume that A_1 and A_2 are ϵ-free (we show an algorithm for removing ϵ-arcs in section 3.5). The following construction simulates the runs of A_1 and A_2 simultaneously. It is based on the basic construction for intersection of finite state automata, augmented by a simulation of the registers and their behavior. Thus, each transition is associated with two sequences of operations on the registers, one for each automaton. The number of the registers is the sum of the registers in the two automata. Thus, in the intersection automaton the first n_1 registers will be designated to simulate the behavior of the registers of the first automaton and the next n_2 registers will simulate

the behavior of the registers of the second automaton. Here, too, only n_1+n_2-1 registers are needed since register number 0 of the two automata can be unified into only one, but for the sake of simplicity we leave the two. In this way a word can be accepted by the intersection automaton iff it can be accepted by each one of the automata separately. We construct an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$ to recognize $L(A_1) \cap L(A_2)$, where:

- $Q = Q_1 \times Q_2$
- $q_0 = (q_0^1, q_0^2)$.
- $\Sigma = \Sigma_1 \cap \Sigma_2$.
- $\Gamma = \Gamma_1 \cup \Gamma_2$.
- $n = n_1 + n_2$. The first n_1 registers will simulate A_1 and the next n_2 will simulate A_2. Register number n_1 is register number 0 of the second automaton.
- $k = k_1 + k_2$. In some cases k can be smaller, but this is its value in the worst case.
- $F = F_1 \times F_2$.
- $\delta = \{((s_1, s_2), \sigma, \langle \vec{a}, \mathit{shift}_{n_1}(\vec{b})\rangle, (t_1, t_2)) \mid (s_1, \sigma, \langle \vec{a}\rangle, t_1) \in \delta_1$ and $(s_2, \sigma, \langle \vec{b}\rangle, t_2) \in \delta_2\}$ where $\vec{a}$ and $\vec{b}$ are meta-variables over $\left(Action_{n_1}^{\Gamma_1}\right)^+$ and $\left(Action_{n_2}^{\Gamma_2}\right)^+$ respectively. No special treatment is needed for the case in which a register operation is performed over register 0 in δ_2 since registers number 0 of the two automata were kept apart.

Notice that register operations from δ_1 and δ_2 cannot be associated with the same register. This guarantees that no information is lost during the simulation of the two intersected automata.

Proposition 3.6. $L(A) = L(A_1) \cap L(A_2)$.

Proof. Assume that $w \in L(A_1) \cap L(A_2)$. Then $w \in L(A_1)$, therefore there exists a series of configurations $c_0^1, c_1^1, \ldots, c_r^1$ such that $c_0^1 = (q_0^1)^c = (q_0^1, \#^{n_1})$, $c_r^1 \in F_1^c$ and for all k, $1 \leq k \leq r$, $c_{k-1}^1 \vdash_{\alpha_k, A_1} c_k^1$ and $w = \alpha_1, \ldots, \alpha_r$. Define for all i, $0 \leq i \leq r$, $c_i^1 = (q_i^1, u_i^1)$. Similarly, $w \in L(A_2)$, therefore there exists a series of configurations $c_0^2, c_1^2, \ldots, c_r^2$ such that $c_0^2 = (q_0^2)^c =$

$(q_0^2, \#^{n_2})$, $c_r^2 \in F_2^c$ and for all k, $1 \leq k \leq r$, $c_{k-1}^2 \vdash_{\alpha_k, A_1} c_k^2$ and $w = \alpha_1, \ldots, \alpha_r$. Define for all i, $0 \leq i \leq r$, $c_i^2 = (q_i^2, u_i^2)$. Note that the same r is used for both A_1 and A_2 since both are assumed to be ϵ-free, hence no $\alpha_i = \epsilon$. By the definition of the product relation it follows that for all k, $1 \leq k \leq r$, there exist $\vec{a_k}$ and $\vec{b_k}$ such that:

(1) $(q_{k-1}^1, \alpha_k, \langle \vec{a_k} \rangle, q_k^1) \in \delta_1$.

(2) $(q_{k-1}^2, \alpha_k, \langle \vec{b_k} \rangle, q_k^2) \in \delta_2$.

(3) $u_{k-1}^1 \Vdash_{\langle \vec{a_k} \rangle} u_k^1$.

(4) $u_{k-1}^2 \Vdash_{\langle \vec{b_k} \rangle} u_k^2$.

From (1) and (2) and by the definition of δ it follows that:

(5) For all k, $1 \leq k \leq r$, $\left((q_{k-1}^1, q_{k-1}^2), \alpha_k, \langle \vec{a_k}, \textit{shift}_{n_1}(\vec{b_k}) \rangle, (q_k^1, q_k^2) \right) \in \delta$.

Notice that for all k, $1 \leq k \leq r$, $\vec{a_k}$ operates only on the first n_1 registers in A, and $\textit{shift}_{n_1}(\vec{b_k})$ operates only on the next n_2 registers. In other words, $\vec{a_k}$ and $\textit{shift}_{n_1}(\vec{b_k})$ cannot operate on the same register. Hence from (3) and (4) it follows that:

(6) For all k, $1 \leq k \leq r$, $\langle u_{k-1}^1, u_{k-1}^2 \rangle \Vdash_{\langle \vec{a_k}, \textit{shift}_{n_1}(\vec{b_k}) \rangle} \langle u_k^1, u_k^2 \rangle$.

From (5) and (6), by defining for all k, $0 \leq k \leq r$, $c_k = ((q_k^1, q_k^2), \langle u_k^1, u_k^2 \rangle)$, it follows that $c_0 = ((q_0^1, q_0^2), \#^{n_1+n_2}) = {q_0}^c$, $c_r \in F^c$ (because $(q_r^1, q_r^2) \in F_1 \times F_2$) and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A} c_k$, therefore $w = \alpha_1 \ldots \alpha_k \in L(A)$. Hence $L(A_1) \cap L(A_2) \subseteq L(A)$.

Assume that $w \in L(A)$. Therefore, there exists a series of configurations $c_0, \ldots, c_r$ such that $c_0 = {q_0}^c$, $c_r \in F^c$ and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A} c_k$ and $w = \alpha_1 \ldots \alpha_r$. Notice that since both A_1 and A_2 are ϵ-free, it follows by the definition of A that A is ϵ-free too, thus for all k, $1 \leq k \leq r$, $\alpha_k \neq \epsilon$. Define for all k, $0 \leq k \leq r$, $c_k = \left((q_k^1, q_k^2), u_k \right)$ (and indeed $c_0 = \left((q_0^1, q_0^2), \#^{n_1+n_2} \right)$). By the definitions of the product relation and δ, it follows that for all k, $1 \leq k \leq r$, there exist $\vec{a_k} \in Actions_{n_1}^{\Gamma_1}$, $\vec{b_k} \in Actions_{n_2}^{\Gamma_2}$ such that $\left((q_{k-1}^1, q_{k-1}^2), \alpha_k, \langle \vec{a_k}, \textit{shift}_{n_1}(\vec{b_k}) \rangle, (q_k^1, q_k^2) \right) \in \delta$ and such that:

(7) $(q_{k-1}^1, \alpha_k, \langle \vec{a_k} \rangle, q_k^1) \in \delta_1$.

(8) $(q_{k-1}^2, \alpha_k, \langle \vec{b_k} \rangle, q_k^2) \in \delta_2$.

(9) $\langle u_{k-1}^1, u_{k-1}^2 \rangle \Vdash_{\langle \vec{a_k}, \textit{shift}_{n_1}(\vec{b_k}) \rangle} \langle u_k^1, u_k^2 \rangle$ where for all i, $0 \leq i \leq r$, $u_i = \langle u_i^1, u_i^2 \rangle$ and $u_i^1 \in \Gamma_1$ is the content of the first n_1 registers and $u_i^2 \in \Gamma_2$ is the content of the next n_2 registers.

Notice that as before, for all k, $1 \leq k \leq r$, $\vec{a_k}$ operates only on the first n_1 registers in A, and $\mathit{shift}_{n_1}(\vec{b_k})$ operates only on the next n_2 registers. Hence, from (9) it follows that:

(10) $u^1_{k-1} \Vdash_{\langle \vec{a_k} \rangle} u^1_k$.

(11) $u^2_{k-1} \Vdash_{\langle \vec{b_k} \rangle} u^2_k$.

From (7) and (10) by defining for all k, $0 \leq k \leq r$, $c^1_k = (q^1_k, u^1_k)$ it follows that $c^1_0 = (q^1_0, u^1_0) = (q^1_0, \#^{n_1}) = (q^1_0)^c$, $c^1_r \in F^c_1$ (because $q^1_r \in F_1$ since $c_r \in F^c$) and for all k, $1 \leq k \leq r$, $c^1_{k-1} \vdash_{\alpha_k, A_1} c^1_k$ and $w = \alpha_1 \ldots \alpha_r$. Therefore $w \in L(A_1)$. From (8) and (11) by defining for all k, $0 \leq k \leq r$, $c^2_i = (q^2_i, u^2_i)$ it follows that $c^2_0 = (q^2_0, u^2_0) = (q^2_0, \#^{n_2}) = (q^2_0)^c$, $c^2_r \in F^c_2$ (because $q^2_r \in F_2$ since $c_r \in F^c$) and for all k, $1 \leq k \leq r$, $c^2_{k-1} \vdash_{\alpha_k, A_2} c^2_k$ and $w = \alpha_1 \ldots \alpha_r$. Therefore $w \in L(A_2)$. Hence $w \in L(A_1) \cap L(A_2)$ and therefore $L(A) \subseteq L(A_1) \cap L(A_2)$. □

3.4.5 Complementation

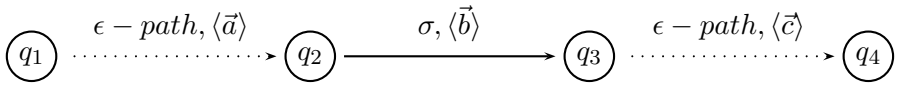
Ordinary FSAs are trivially closed under complementation. However, given an FSA A whose language is $L(A)$, the minimal FSA recognizing the complement of $L(A)$ can be exponentially large. More precisely, for any integer $n > 2$, there exists an n-state NFA A, such that any NFA that accepts the complement of $L(A)$ needs at least 2^{n-2} states (Holzer and Kutrib, 2002). We have no reason to believe that FSRAs will demonstrate a different behavior; therefore, we maintain that in the worst case, the best approach for complementing an FSRA would be to convert it into FSA and complement the latter. We therefore do not provide a dedicated construction for this operator.

3.5 ϵ-removal

An ϵ-arc in an FSRA is an arc of the form $(s, \epsilon, \langle \vec{a} \rangle, t)$. Notice that this kind of arcs might occur in an FSRA by its definition. Given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ that might contain ϵ-arcs, an equivalent FSRA without ϵ-arcs can be constructed. The construction is based on the algorithm for ϵ removal in finite state automata, but the register operations that are associated with the ϵ-arc have to be dealt with, and this requires some care. The resulting FSRA has one more state than the original, and some additional arcs may be added, too. The basic idea behind the construction is as follows: If there is an ϵ-path form q_1 to q_2 with the register operations $\vec{a}$ over its arcs, and if there is

an arc $(q_2, \sigma, \langle \vec{b} \rangle, q_3)$ where $\sigma \neq \epsilon$ and if there is an ϵ-path form q_3 to q_4 with the register operations $\vec{c}$ over its arcs, then the equivalent ϵ-free network will contain the arcs $(q_2, \sigma, \langle \vec{b} \rangle, q_3)$, $(q_1, \sigma, \langle \vec{a}, \vec{b} \rangle, q_3)$, $(q_2, \sigma, \langle \vec{b}, \vec{c} \rangle, q_4)$ and $(q_1, \sigma, \langle \vec{a}, \vec{b}, \vec{c} \rangle, q_3)$, with all the ϵ-arcs removed. This is illustrated in Figure 3.10. Notice that if q_1 and q_2 are the same state, then states q_2 and q_3 will be connected by two parallel arcs differing in their associated register operations and the same holds for states q_2 and q_4. Similarly, when q_3 and q_4 are the same state.

Original:



ϵ-free:

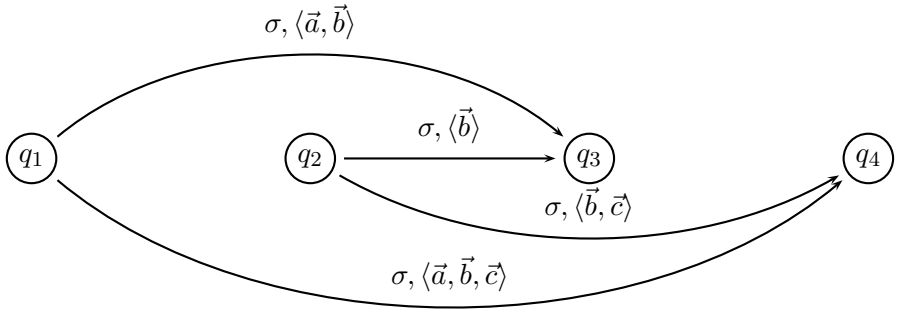


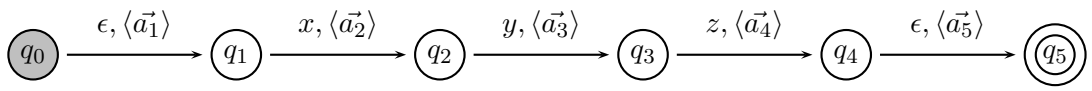
Figure 3.10: ϵ removal paradigm

Example 3.6. *The FSRAs in Figure 3.11 both accept only the word xyz (assuming that the register operations on the accepting path can be satisfied) and both contain ϵ-transitions. The corresponding ϵ-free FSRAs created by the algorithm are shown in Figure 3.12. Notice that the corresponding ϵ-free FSRAs contain redundant arcs and states. In section 3.6 we show how they can be removed.*

In addition to the above changes, special care is needed for the case in which the empty word is accepted by the original automaton. We will refer to this issue later.

We now formally define, for every FSRA, an equivalent ϵ-free FSRA, and prove the correctness of the construction. We begin by defining, for every state q in an FSRA, the set $\varepsilon(q)$. This is the set of all the states q' such that an ϵ-path leads from q to q'; and each state q' in $\varepsilon(q)$ is paired with

A_1:



A_2:

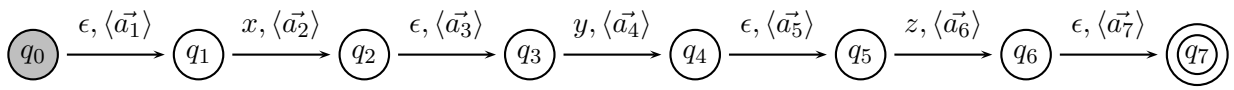


Figure 3.11: FSRAs containing ϵ-transitions

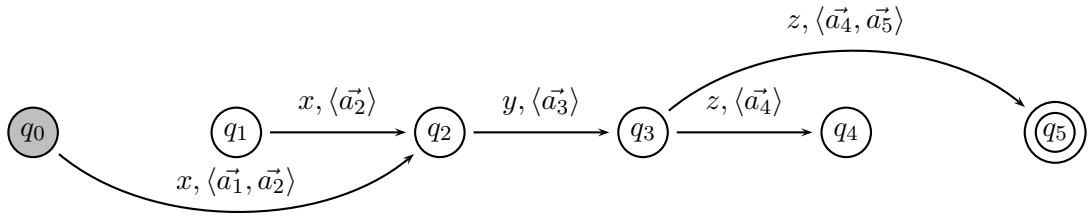
the series of register operations along the ϵ-path that leads from q to q'. Formally, given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ and $q \in Q$, define $\varepsilon(q) = \{(q', \langle \vec{a_1}, \vec{a_2}, \ldots, \vec{a_j} \rangle) \mid$ there exists $j \geq 2$ and there exist $q_1, \ldots, q_{j-1} \in Q$ such that $(q, \epsilon, \langle \vec{a_1} \rangle, q_1), (q_1, \epsilon, \langle \vec{a_2} \rangle, q_2), \ldots, (q_{j-1}, \epsilon, \langle \vec{a_j} \rangle, q') \in \delta\} \cup \{(q', \langle \vec{a} \rangle) \mid (q, \epsilon, \langle \vec{a} \rangle, q') \in \delta\}$. Notice that since the network might contain ϵ loops, $\varepsilon(q)$ might be infinite. However, as we presently show, if the set of such pairs is divided into different subsets, where each subset has the same effect on the registers content, only finite number of such subsets exist. We define a relation over $\left(Actions_n^{\Gamma}\right)^+$, denoted $\equiv_\Gamma$: $\vec{a} \equiv_\Gamma \vec{b}$ iff for all $u, v \in \Gamma^n$, $u \Vdash_{\langle \vec{a} \rangle} v$ iff $u \Vdash_{\langle \vec{b} \rangle} v$. Next, we define a relation over $\varepsilon(q)$ denoted '$\approx_q$': $(q_1, \langle \vec{a} \rangle) \approx_q (q_2, \langle \vec{b} \rangle)$ iff $q_1 = q_2$ and $\vec{a} \equiv_\Gamma \vec{b}$. Intuitively, two elements of $\varepsilon(q)$ are equivalent iff they contain the same state, and their two series of register operations have the same effect on the state of the FSRA.

Proposition 3.7. *Given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ and $q \in Q$, $\approx_q$ is an equivalence relation over $\varepsilon(q)$.*

Proof. As $\approx_q$ is defined in terms of identity and logical equivalence, it is clearly an equivalence relation. □

The $\approx_q$ relation partitions the elements of $\varepsilon(q)$ to equivalence classes such that $(q, \langle \vec{a} \rangle)$ and $(q, \langle \vec{b} \rangle)$ are in the same equivalence class if and only if the two register operations series have the same effect on the registers content.

A_1 equivalent ϵ-free FSRA:



A_2 equivalent ϵ-free FSRA:

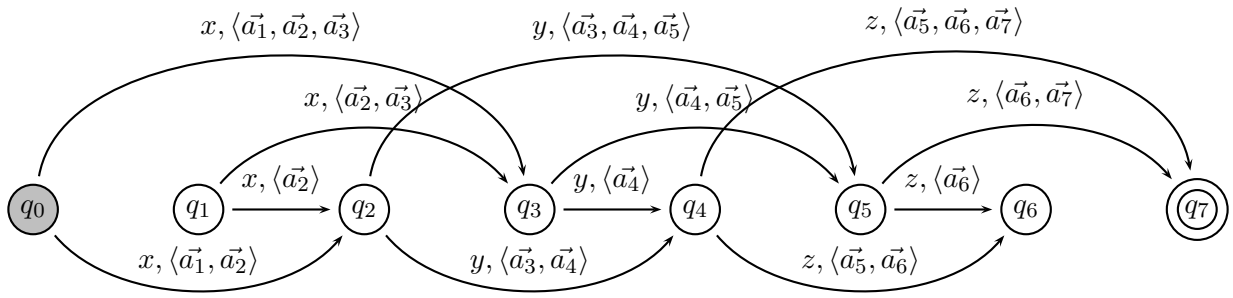


Figure 3.12: ϵ-free FSRAs

Proposition 3.8. *Given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ and $q \in Q$, '$\approx_q$' has a finite index.*

Proof. Let $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ and $q \in Q$. Assume toward a contradiction that $\approx_q$ partitions the elements of $\varepsilon(q)$ into an infinite number of equivalence classes. Since in each equivalence class all the first coordinates of the pairs (which are nodes in Q) are the same node, there exists $q' \in Q$ such that for an infinite number of equivalence classes, the first coordinate of the pairs is q'. For each equivalence class $[(q', \vec{a})]_{\approx_q}$ for $(q', \vec{a}) \in \varepsilon(q)$ we define a total function $f_{\vec{a}} : \Gamma^n \to \Gamma^n$ by:

$$\text{for all } \ u \in \Gamma^n \quad f_{\vec{a}}(u) = \begin{cases} v & \text{if there exists } v \in \Gamma^n \ \text{ such that } \ u \Vdash_{\vec{a}} v \\ u & \text{otherwise} \end{cases}$$

Notice that by the definition of the relation $\Vdash$, if there exists $v \in \Gamma^n$ such that $u \Vdash_{\vec{a}} v$, then it is unique and hence $f_{\vec{a}}$ is indeed well defined. Moreover, the definition of $f_{\vec{a}}$ does not depend on the equivalence class representative, $\vec{a}$, since all the elements in its equivalence class effect the registers content in the same way (by the definitions of the relations $\approx_q$ and $\equiv_\Gamma$). For $(q', \vec{a_1}), (q', \vec{a_2}) \in \varepsilon(q)$,

if $(q', \vec{a_1}) \not\approx (q', \vec{a_2})$ then there is $u \in \Gamma^n$ such that $f_{\vec{a_1}}(u) \neq f_{\vec{a_2}}(u)$, thus $f_{\vec{a_1}} \neq f_{\vec{a_2}}$. In sum, each equivalence class defines a different function, and therefore by the assumption there is an infinite number of such functions, but since Γ and n are finite, there is only a finite number of such functions (there is a finite number of possible total functions from a finite set to itself) – a contradiction. □

For every $q \in Q$, let $\zeta(q)$ be a set containing exactly one representative of every equivalence class $[(q', \vec{a})]_{\approx_q}$. Which representative is chosen is irrelevant. Thus $\zeta(q)$ is a finite subset of $\varepsilon(q)$ whose size is equal to the index of $\approx_q$.

We are now ready to define the equivalent ϵ-free FSRA for a given FSRA that might contain ϵ-arcs. We first consider the case where the given FSRA does not accept the empty string. The ϵ-removal paradigm is as described in Figure 3.10, where the ϵ-paths that are accounted for by this paradigm are only the ones in $\zeta(q)$; the other ϵ-paths are simply ignored (i.e., removed without any effect) as they are represented in $\zeta(q)$ by other ϵ-paths, equivalent to them in their effect on the state of the FSRA. $\zeta(q)$ is a finite set (unlike $\varepsilon(q)$ which may be infinite) and thus ensures that δ' is finite too.

Definition 3.3. *Given an FSRA* $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F \rangle$ *where* $\epsilon \notin L(A)$, *EpsFree*$(A) = \langle Q', q_0', \Sigma', \Gamma', n', k', \delta', F' \rangle$ *is the FSRA defined by:*

- $Q' = Q$.
- $q_0' = q_0$.
- $\Sigma' = \Sigma$.
- $\Gamma' = \Gamma$.
- $n' = n$.
- $F' = F$.
- $\delta' = \{(q_1, \sigma, \langle \vec{a} \rangle, q_2) \in \delta \mid \sigma \neq \epsilon\}$

  $\cup$

  $\{(q, \sigma, \langle \vec{a}, \vec{b}, \vec{c} \rangle, q') \mid$ *there exist* $q_1, q_2 \in Q$ *such that* $(q_1, \langle \vec{a} \rangle) \in \zeta(q)$, $(q_1, \sigma, \langle \vec{b} \rangle, q_2) \in \delta$,

$(q', \langle \vec{c} \rangle) \in \zeta(q_2)$ *and* $\sigma \neq \epsilon\}$

$\cup$

$\{(q, \sigma, \langle \vec{a}, \vec{b} \rangle, q') \mid$ *there exist* $q_1 \in Q$ *such that* $(q_1, \langle \vec{a} \rangle) \in \zeta(q)$ *,* $(q_1, \sigma, \langle \vec{b} \rangle, q') \in \delta$ *, and* $\sigma \neq \epsilon\}$

$\cup$

$\{(q, \sigma, \langle \vec{b}, \vec{c} \rangle, q') \mid$ *there exists* $q_1 \in Q$ *such that* $(q, \sigma, \langle \vec{b} \rangle, q_1) \in \delta$ *,* $(q', \langle \vec{c} \rangle) \in \zeta(q_1)$ *and* $\sigma \neq \epsilon\}$.

Notice that since Q*,* δ *and* $\zeta(q)$ *for all* $q \in Q$ *are finite sets,* δ' *is finite too.*

- $k' = max\{p \mid$ *there exist* $q_1, q_2 \in Q$ *and there exists* $\sigma \in \Sigma$ *and there exist* $a_1, \ldots, a_p \in Actions_{n'}^{\Gamma'}$ *such that* $(q_1, \sigma, \langle a_1, \ldots, a_p \rangle, q_2) \in \delta'\}$.

Proposition 3.9. $L(A) = L\,(\textit{EpsFree}(A))$.

Proof. Assume that $w \in L(A)$. The idea for showing that $w \in L(\textit{EpsFree}(A))$ is to construct the equivalent accepting path for w in *EpsFree*(A). Notice that the states in the path accepting w in A are also states in *EpsFree*(A), and the idea is to use them to construct an accepting path for w in *EpsFree*(A). This path is constructed as follows: a non-ϵ arc with all its consecutive predecessor ϵ-arcs is replaced by a single non-ϵ transition. If the arc has no ϵ-arcs predecessors, then it remains as is. Special care is needed for the last transition: the last non-ϵ arc with all its consecutive predecessor ϵ-arcs and all its consecutive successor ϵ-arcs (if such exist) is replaced by a single non-ϵ transition.

By the assumption, $w \neq \epsilon$ and therefore, there exists a series of configurations $c_0, \ldots, c_r$ such that $c_0 = q_0^c = (q_0, \#^n)$, $c_r \in F^c$ and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, A} c_k$ and $w = \alpha_1 \ldots \alpha_r$. Define for all k, $0 \leq k \leq r$, $c_i = (q_i, u_i)$ (and indeed $c_0 = (q_0, u_0)$). Then for all k, $1 \leq k \leq r$, there exists $\vec{a} \in \left(Actions_n^{\Gamma}\right)^+$ such that $u_{k-1} \Vdash_{\langle \vec{a_k} \rangle} u_k$ and $(q_{k-1}, \alpha_k, \langle \vec{a_k} \rangle, q_k) \in \delta$. Define $w' = \alpha_{i_1} \ldots \alpha_{i_j}$ as the substring of w consisting of all the non ϵ-symbols of w and only them (for example, if $w = a\epsilon b$ and thus $\alpha_1 = a$, $\alpha_2 = \epsilon$, $\alpha_3 = b$ then $w' = \alpha_1\alpha_3$). This substring is used to mark all the non-ϵ transitions in the accepting path for w in A (the non-ϵ transitions are those connecting $q_{i_k - 1}$ and q_{i_k} marked by the symbol α_{i_k}, for $1 \leq k \leq j$).

The first arc in *EpsFree*(A) corresponds to the first non-ϵ transition in A (connecting $q_{i_1 - 1}$ with q_{i_1}). The two possible cases and their corresponding changes are illustrated in Figure 3.13. Formally, for i_1 the following holds:

If $i_1 = 1$ (first case), then $(q_0, \alpha_{i_1}, \langle \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta$ and therefore $(q_0, \alpha_{i_1}, \langle \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta'$. In addition, $u_0 \Vdash_{\langle \vec{a}_{i_1} \rangle} u_{i_1}$. Notice that $(q'_0, u_0) = (q_0, u_0)$ is the initial configuration of *EpsFree*(A).

Otherwise, if $i_1 \neq 1$ (second case), then $(q_{i_1-1}, \langle \vec{a}_1, \ldots, \vec{a}_{i_1-1} \rangle) \in \varepsilon(q_0)$ and therefore there exists $\vec{b}_{i_1} \in \left(Actions_n^{\Gamma}\right)^+$ such that $(q_{i_1-1}, \langle \vec{b}_{i_1} \rangle \in \zeta(q_0)$. In addition, $(q_{i_1-1}, \alpha_{i_1}, \langle \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta$ and $\alpha_{i_1} \neq \epsilon$, and hence $(q_0, \alpha_{i_1}, \langle \vec{b}_{i_1}, \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta'$. In addition, by the definitions of the relation $\Vdash$ and the equivalence relation $\approx_{q_0}$, it follows that $u_0 \Vdash_{\langle \vec{b}_{i_1}, \vec{a}_{i_1} \rangle} u_{i_1}$.

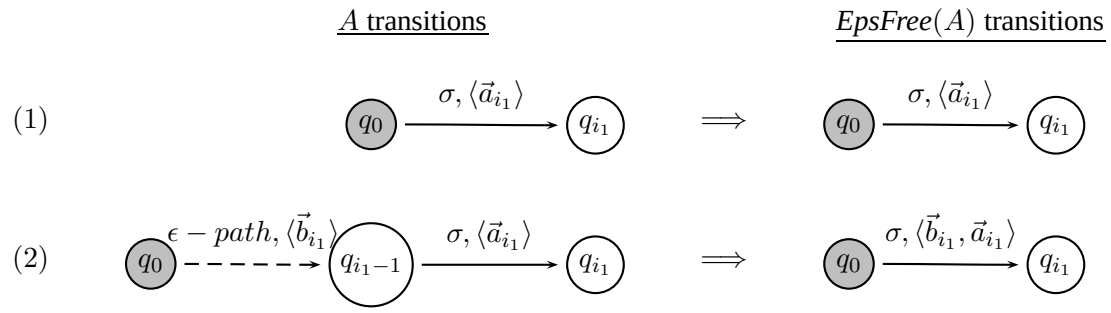


Figure 3.13: First arc construction in the ϵ-free FSRA

The central transitions in *EpsFree*(A) correspond to the central non-ϵ transitions in A (connecting q_{i_k-1} with q_{i_k}, for $2 \leq k \leq j$). The two possible cases and their corresponding changes are illustrated in Figure 3.14. Formally, for all k, $2 \leq k \leq j-1$, the following holds:

If $i_k = i_{(k-1)} + 1$ (first case – there are no preceding ϵ-transitions), then since $\alpha_{i_k} \neq \epsilon$ and by the definition of δ' it follows that $(q_{i_{(k-1)}}, \alpha_{i_k}, \langle \vec{a}_{i_k} \rangle, q_{i_k}) \in \delta'$ and $u_{i_{(k-1)}} \vdash_{\langle \vec{a}_{i_k} \rangle} u_{i_k}$.

Otherwise (second case), $(q_{i_k-1}, \langle \vec{a}_{i_{(k-1)}+1}, \ldots, \vec{a}_{i_k-1} \rangle) \in \varepsilon(q_{i_{(k-1)}})$ and therefore there exists $\vec{b}_{i_k} \in \left(Actions_n^{\Gamma}\right)^+$ such that $(q_{i_k-1}, \langle \vec{b}_{i_k} \rangle) \in \zeta(q_{i_{(k-1)}})$. In addition, $(q_{i_k-1}, \alpha_{i_k}, \langle \vec{a}_{i_k} \rangle, q_{i_k}) \in \delta$ and $\alpha_{i_k} \neq \epsilon$ and therefore $(q_{i_{(k-1)}}, \alpha_{i_k}, \langle \vec{b}_{i_k}, \vec{a}_{i_k} \rangle, q_{i_k}) \in \delta'$. In addition, by the definitions of the relation $\Vdash$ and the equivalence relation $\approx_{q_{i_{(k-1)}}}$, it follows that $u_{i_{(k-1)}} \Vdash_{\langle \vec{b}_{i_k}, \vec{a}_{i_k} \rangle} u_{i_k}$.

The last transition in *EpsFree*(A) corresponds to the last non-ϵ transition in A (connecting q_{i_j-1} with q_{i_j}). The four possible cases and their corresponding changes are illustrated in Figure 3.15 (notice that in the first two cases $q_{i_j} = q_r$ and that the two extra cases are due to possible ϵ-transitions at the end of the accepting path for w in A). Formally, for i_j the following holds:

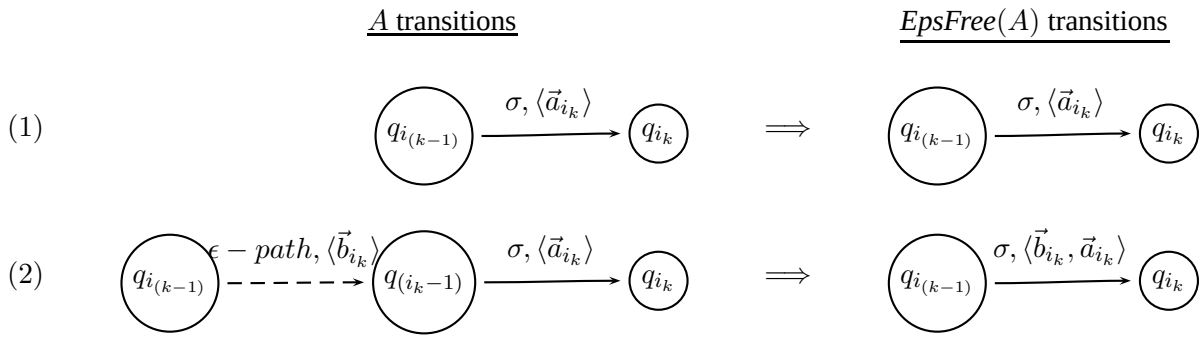


Figure 3.14: Central arc construction in the ϵ-free FSRA

If $i_j = r$ (first and second cases – there are no successive ϵ-transitions), then simply extend the above from $2 \leq k \leq j-1$ to $2 \leq k \leq j$.

Otherwise, if $i_j \neq r$, then $(q_r, \langle \vec{a}_{i_j+1}, \ldots, \vec{a}_r \rangle) \in \varepsilon(q_{i_j})$, and therefore there exists $\vec{c}_{i_j} \in \left(Actions_n^{\Gamma}\right)^+$ such that $(q_r, \langle \vec{c}_{i_j} \rangle) \in \zeta(q_{i_j})$. In this case the following holds:

If $i_j = i_{(j-1)} + 1$ (third case – there are no preceding ϵ-transitions) then $(q_{i_{(j-1)}}, \alpha_{i_j}, \langle \vec{a}_{i_j} \rangle, q_{i_j}) \in \delta$ and $\alpha_{i_j} \neq \epsilon$, and therefore by the definition of δ' it follows that $(q_{i_{(j-1)}}, \alpha_{i_j}, \langle \vec{a}_{i_j}, \vec{c}_{i_j} \rangle, q_r) \in \delta'$. In addition, by the definitions of the relation $\Vdash$ and the equivalence relation $\approx_{q_{i_j}}$, it follows that $u_{i_{(j-1)}} \Vdash_{\langle \vec{a}_{i_j}, \vec{c}_{i_j} \rangle} u_r$.

Otherwise, if $i_j \neq i_{(j-1)}+1$ (fourth case), then $(q_{i_{(j-1)}}, \langle \vec{a}_{i_{(j-1)}+1}, \ldots, \vec{a}_{i_j-1} \rangle) \in \varepsilon(q_{i_{(j-1)}})$ and therefore there exists $\vec{b}_{i_j} \in \left(Actions_n^{\Gamma}\right)^+$ such that $(q_{i_j-1}, \langle \vec{b}_{i_j} \rangle) \in \zeta(q_{i_{(j-1)}})$. In addition, $(q_{i_j-1}, \alpha_{i_j}, \langle \vec{a}_{i_j} \rangle, q_{i_j}) \in \delta$ and $\alpha_{i_j} \neq \epsilon$ and therefore $(q_{i_{(j-1)}}, \alpha_{i_j}, \langle \vec{b}_{i_j}, \vec{a}_{i_j}, \vec{c}_{i_j} \rangle, q_r) \in \delta'$. In addition, by the definitions of the relation $\Vdash$ and the equivalence relations $\approx_{q_{i_{(j-1)}}}$ and $\approx_{q_{i_j}}$, it follows that $u_{i_{(j-1)}} \Vdash_{\langle \vec{b}_{i_j}, \vec{a}_{i_j}, \vec{c}_{i_j} \rangle} u_{i_j}$.

In sum, there is a series of configurations of *EpsFree*(A), $g_0, g_{i_1}, \ldots, g_{i_j}$, where $g_0 = (q_0', \#^n) = (q_0')^c$, $g_{i_j} \in (F')^c$ and for all k, $1 \leq k \leq j$, $g_{i_k} = (q_{i_k}, u_{i_k})$ and $g_0 \vdash_{\alpha_{i_1},\, \textit{EpsFree}(A)} g_{i_1}$ and for all k, $2 \leq k \leq j$, $g_{i_{(k-1)}} \vdash_{\alpha_{i_k}, \textit{EpsFree}(A)} g_{i_k}$ and $w = \alpha_{i_1} \ldots \alpha_{i_j}$. Hence, $w \in L(\textit{EpsFree}(A))$ and therefore $L(A) \subseteq L(\textit{EpsFree}(A))$.

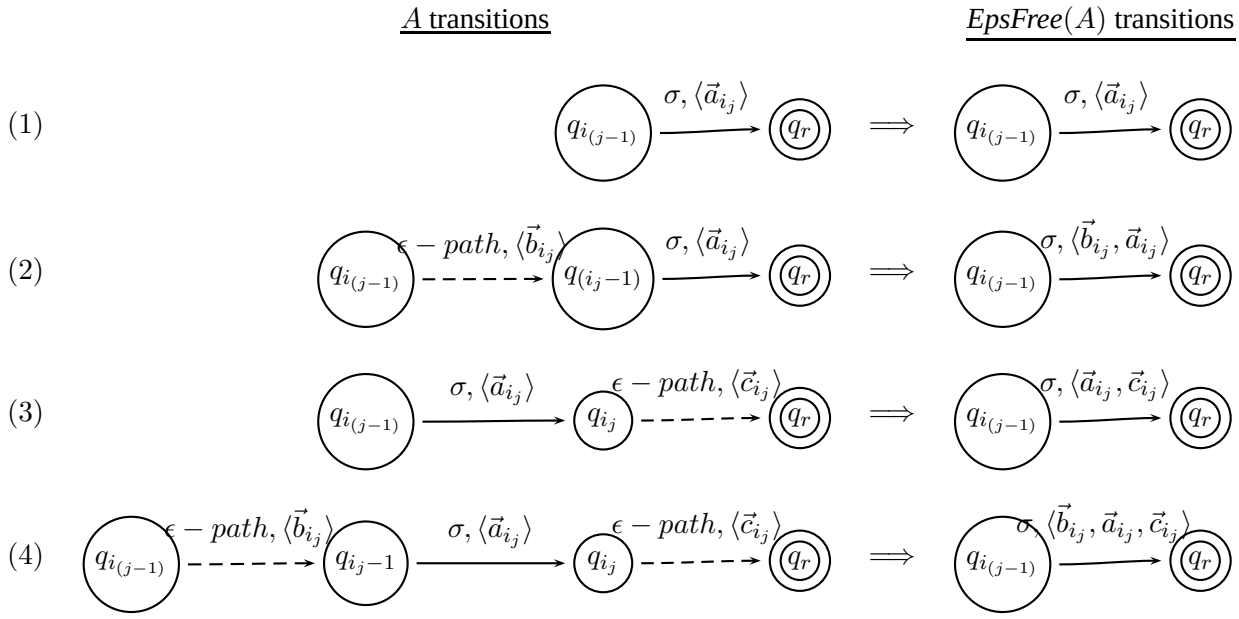


Figure 3.15: Final arc construction in the ϵ-free FSRA

Assume that $w \in L(\textit{EpsFree}(A))$. If $w = \epsilon$, then since *EpsFree*(A) is ϵ-free, it follows that $q'_0 \in F'$. Hence, since $q'_0 = q_0$ and $F' = F$ it follows that $q_0 \in F$ and therefore $\epsilon \in L(A)$ – a contradiction to the assumption that $\epsilon \notin L(A)$. Therefore, $\epsilon \notin L(\textit{EpsFree}(A))$ and we assume then that $w \neq \epsilon$.

The main idea for showing that $w \in L(A)$ is to construct the equivalent accepting path for w in A. Notice that again, the states in the path accepting w in *EpsFree*(A) are also states in A, and the idea is to use them to construct an accepting path for w in A. The accepting path for w in A is constructed by replacing each transition in *EpsFree*(A) by its counterparts in A. For a given transition there are four possible ways in which it was created and they are illustrated in Figure 3.16 (notice that each of the ϵ-paths can be either one ϵ-transition or a series of consecutive ϵ-transitions).

Since $w \in L(\textit{EpsFree}(A))$, there is a series of configurations of *EpsFree*(A) $c_0, \ldots, c_r$ such that $c_0 = (q'_0, \#^n)$, $c_r \in (F')^c$ and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{\alpha_k, \textit{EpsFree}(A)} c_k$ and $w = \alpha_1, \ldots, \alpha_r$

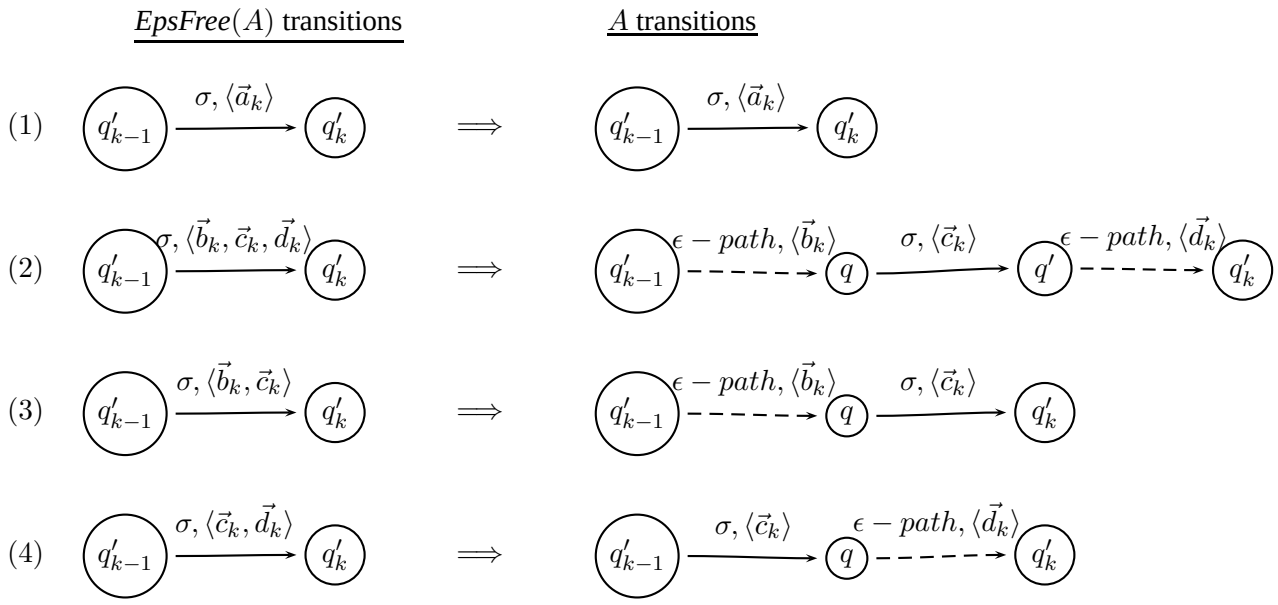


Figure 3.16: Arcs construction in the FSRA containing ϵ-arcs

(notice that since *EpsFree*(A) is ϵ free, $|w| = r$). Define for all k, $0 \leq k \leq r$, $c_k = (q'_k, u_k)$ (and indeed $c_0 = (q'_0, u_0)$). Therefore, by the definition of the product relation, for all k, $1 \leq k \leq r$, there exists $\vec{a}_k \in \left(Actions_n^{\Gamma}\right)^+$ such that $(q'_{k-1}, \alpha_k, \langle \vec{a}_k \rangle, q'_k) \in \delta'$ and $u_{k-1} \Vdash_{\langle \vec{a}_k \rangle} u_k$. δ' is a union of four sets, and therefore $(q'_{k-1}, \alpha_k, \langle \vec{a}_k \rangle, q'_k)$ must belong to one of this sets. By going over all the four possibilities we get the following (each possibility corresponds to the same possibility number in the diagram of Figure 3.16):

Possibility 1: $(q'_{k-1}, \alpha_k, \langle \vec{a}_k \rangle, q'_k) \in \delta$. Define then $c'_{k-1} = c_{k-1}$ and $c'_k = c_k$ and thus $c'_{k-1} \vdash_{\alpha_k, A} c'_k$.

Possibility 2: There exist $q, q' \in Q$ and there exist $\vec{b}_k, \vec{c}_k, \vec{d}_k \in \left(Actions_n^{\Gamma}\right)^+$ such that $\langle \vec{a}_k \rangle = \langle \vec{b}_k, \vec{c}_k, \vec{d}_k \rangle$ and such that:

(1) $(q, \langle \vec{b}_k \rangle) \in \zeta(q'_{k-1})$.

(2) $(q, \alpha_k, \langle \vec{c}_k \rangle, q') \in \delta$.

(3) $(q'_k, \langle \vec{d}_k \rangle) \in \zeta(q')$.

In addition, by the definition of the $\Vdash$ relation and from the fact that $u_{k-1} \Vdash_{\langle \vec{a}_k \rangle} u_k$, it follows that there exist $v_k, y_k \in \Gamma^n$ such that $u_{k-1} \Vdash_{\langle \vec{b}_k \rangle} v_k$, $v_k \Vdash_{\langle \vec{c}_k \rangle} y_k$ and $y_k \Vdash_{\langle \vec{d}_k \rangle} u_k$.

From (1) and from the fact that $\zeta(q'_{k-1})$ is a subset of $\varepsilon(q'_{k-1})$ it follows that either one of the following holds (as $\varepsilon(q'_{k-1})$ is defined as a union of two sets, each of its elements must belong to one of the sets):

(a) $(q'_{k-1}, \epsilon, \langle \vec{b}_k \rangle, q) \in \delta$ (a single ϵ-arc). Define then $c'_{k-1} = c_{k-1}$, $g_{k-1} = (q, v_k)$ and thus $c'_{k-1} \vdash_{\epsilon,A} g_{k-1}$.

(b) There exists $j \geq 2$ and there exist $q_k^1, \ldots, q_k^j \in Q$ and there exist $\vec{b}_k^1, \ldots, \vec{b}_k^{j+1} \in \left(Actions_n^\Gamma\right)^+$ such that $(q'_{k-1}, \epsilon, \langle \vec{b}_k^1 \rangle, q_k^1)$, $(q_k^1, \epsilon, \langle \vec{b}_k^2 \rangle, q_k^2)$, $\ldots$, $(q_k^j, \epsilon, \langle \vec{b}_k^{j+1} \rangle, q) \in \delta$ and $\langle \vec{b}_k \rangle = \langle \vec{b}_k^1, \vec{b}_k^2, \ldots, \vec{b}_k^{j+1} \rangle$ (a series of ϵ-arcs). In addition, since $u_{k-1} \Vdash_{\langle \vec{b}_k \rangle} v_k$ and by the definition of the relation $\Vdash$, there exist $v_k^1, \ldots, v_k^j \in \Gamma^n$ such that $u_{k-1} \Vdash_{\langle \vec{b}_k^1 \rangle} v_k^1$, $v_k^1 \Vdash_{\langle \vec{b}_k^2 \rangle} v_k^2$, $\ldots$, $v_k^{j-1} \Vdash_{\langle \vec{b}_k^j \rangle} v_k^j$, $v_k^j \Vdash_{\langle \vec{b}_k^{j+1} \rangle} v_k$. Define then $c'_{k-1} = c_{k-1}$, $G_{k-1}^i = (q_k^i, v_k^i)$ for all i, $1 \leq i \leq j$, and $g_{k-1} = (q, v_k)$. Thus, $c'_{k-1} \vdash_{\epsilon,A} G_{k-1}^1$, $G_{k-1}^{i-1} \vdash_{\epsilon,A} G_{k-1}^i$ for all i, $2 \leq i \leq j$, and $G_{k-1}^j \vdash_{\epsilon,A} g_{k-1}$.

From (2), by defining $h_{k-1} = (q', y_k)$, it follows that $g_{k-1} \vdash_{\alpha_k,a} h_{k-1}$.

From (3) and from the fact that $\zeta(q')$ is a subset of $\varepsilon(q')$ it follows that either one of the following holds (as $\varepsilon(q')$ is defined as a union of two sets, each of its elements must belong to one of the sets):

(c) $(q', \epsilon, \langle \vec{d}_k \rangle, q'_k) \in \delta$ (a single ϵ-arc). Define then $c'_k = c_k$ and thus $h_{k-1} \vdash_{\epsilon,A} c'_k$.

(d) There exist $m \geq 2$ and there exist $x_k^1, \ldots, x_k^m \in Q$ and there exist $\vec{d}_k^1, \ldots, \vec{d}_k^{m+1} \in \left(Actions_n^\Gamma\right)^+$ such that $(q', \epsilon, \langle \vec{d}_k^1 \rangle, x_k^1)$, $(x_k^1, \epsilon, \langle \vec{d}_k^2 \rangle, x_k^2)$, $\ldots$, $(x_k^m, \epsilon, \langle \vec{d}_k^{m+1} \rangle, q'_k) \in \delta$ and $\langle \vec{d}_k \rangle = \langle \vec{d}_k^1, \vec{d}_k^2, \ldots, \vec{d}_k^{m+1} \rangle$ (a series of ϵ-arcs). In addition, since $y_k \Vdash_{\langle \vec{d}_k \rangle} u_k$ and by the definition of the relation $\Vdash$, there exist $y_k^1, \ldots, y_k^m \in \Gamma^n$ such that $y_k \Vdash_{\langle \vec{d}_k^1 \rangle} y_k^1$, $y_k^1 \Vdash_{\langle \vec{d}_k^2 \rangle} y_k^2$, $\ldots$, $y_k^{m-1} \Vdash_{\langle \vec{d}_k^m \rangle} y_k^m$, $y_k^m \Vdash_{\langle \vec{d}_k^{m+1} \rangle} u_k$. Define then $H_{k-1}^i = (x_k^i, y_k^i)$ for all i, $1 \leq i \leq m$, and $c'_k = c_k$. Thus, $h_{k-1} \vdash_{\epsilon,A} H_{k-1}^1$, $H_{k-1}^{i-1} \vdash_{\epsilon,A} H_{k-1}^i$ for all i, $2 \leq i \leq m$, and $H_{k-1}^m \vdash_{\epsilon,A} c'_k$.

Possibility 3 and 4 are sub-cases of possibility 2 (where one of the ϵ-sides is missing). The construction is done in the same way as in possibility 2.

$c'_r = c_r = (q'_r, u_i)$ is the last configuration in the series. Since $c_r \in (F')^c$, $q'_r \in F'$ and hence, $q'_r \in F$. Therefore, c'_r is a final configuration of A.

In sum, we obtain a series of configurations in A for which the product relation holds, where the

first is the initial configuration of A and the last is a final configuration of A. Hence $w \in L(A)$ and $L(\textit{EpsFree}(A)) \subseteq L(A)$. □

As was mentioned before, special care is needed for the case in which the empty word is accepted by the automaton. In finite state automata this case is dealt with by simply making the initial state an accepting state. This solution in finite state register automata might cause over generation, as the following example demonstrates.

Example 3.7. *Consider the FSRA in Figure 3.17. This automaton accepts the language denoted by $\epsilon \mid a^+b$. Removing ϵ-arcs in the naïve way, and turning the initial state into an accepting one, results in the network in Figure 3.18. This network accepts also strings of the form a^+, and thus over generates.*

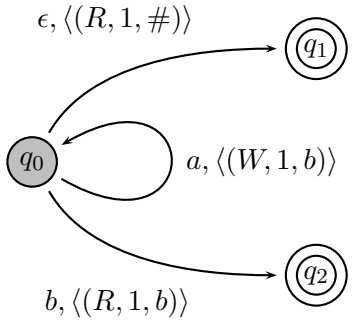


Figure 3.17: FSRA with ϵ-arcs

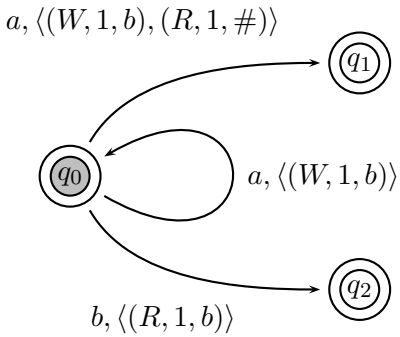


Figure 3.18: Over generating ϵ-free FSRA

The reason for this over generation is that in contrast to finite state automata, in finite state registered automata not every ϵ-path can be traversed, since the possibility to traverse an ϵ path depends on its associated register operations. The problem occurs only in cases where the empty string is accepted and the initial state is not an accepting state. The solution in such cases is to add a new state which will be the new initial state and will duplicate in a restricted way the former initial state (which no longer will be the initial state). The new initial state is turned into a final state and will have all the outgoing arcs of the former initial state but will have no incoming arcs. The equivalent ϵ-free resulting network for the FSRA in Example 3.7 is illustrated in Figure 3.19. Note that paths labeled by strings of 'a's indeed connect the initial state with the accepting state q_1, but they cannot be used for recognizing such strings because of contradicting register operations.

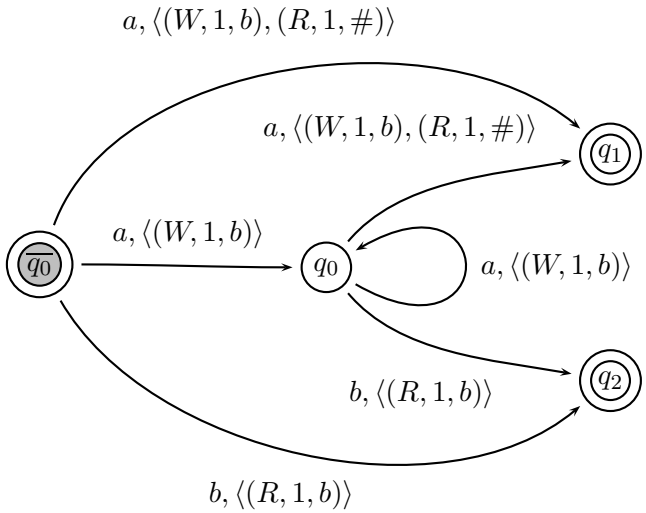


Figure 3.19: Correct ϵ-free FSRA

Definition 3.4. *Given an FSRA* $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$ *where* $\epsilon \in L(A)$, *EPSfree*$(A) = \langle Q', q_0', \Sigma', \Gamma', n', k', \delta', F'\rangle$ *is the FSRA defined by:*

- *Let* $\overline{q_0} \notin Q$. *Define* $Q' = Q \cup \{\overline{q_0}\}$.
- $q_0' = \overline{q_0}$.
- $\Sigma' = \Sigma$.
- $\Gamma' = \Gamma$.

- $n' = n$.
- $F' = F \cup \{q'_0\}$.
- *Define* $\delta^* = \{(q_1, \sigma, \langle \vec{a} \rangle, q_2) \in \delta \mid \sigma \neq \epsilon\}$

  $\cup$

  $\{(q, \sigma, \langle \vec{a}, \vec{b}, \vec{c} \rangle, q') \mid$ *there exist* $q_1, q_2 \in Q$ *such that* $(q_1, \langle \vec{a} \rangle) \in \zeta(q)$, $(q_1, \sigma, \langle \vec{b} \rangle, q_2) \in \delta$, $(q', \langle \vec{c} \rangle) \in \zeta(q_2)$ *and* $\sigma \neq \epsilon\}$

  $\cup$

  $\{(q, \sigma, \langle \vec{a}, \vec{b} \rangle, q') \mid$ *there exist* $q_1 \in Q$ *such that* $(q_1, \langle \vec{a} \rangle) \in \zeta(q)$, $(q_1, \sigma, \langle \vec{b} \rangle, q') \in \delta$, *and* $\sigma \neq \epsilon\}$

  $\cup$

  $\{(q, \sigma, \langle \vec{b}, \vec{c} \rangle, q') \mid$ *there exists* $q_1 \in Q$ *such that* $(q, \sigma, \langle \vec{b} \rangle, q_1) \in \delta$, $(q', \langle \vec{c} \rangle) \in \zeta(q_1)$ *and* $\sigma \neq \epsilon\}$. *Define then* $\delta' = \delta^* \cup \{(\overline{q_0}, \sigma, \langle \vec{a} \rangle, q) \mid (q_0, \sigma, \langle \vec{a} \rangle, q) \in \delta^*\}$. *Notice that since* Q, δ *and* $\zeta(q)$ *for all* $q \in Q$ *are finite sets,* δ^* *is finite, and hence* δ' *is finite too.*
- $k' = max\{p \mid$ *there exist* $q_1, q_2 \in Q$ *and there exists* $\sigma \in \Sigma$ *and there exist* $a_1, \ldots, a_p \in Actions_{n'}^{\Gamma'}$ *such that* $(q_1, \sigma, \langle a_1, \ldots, a_p \rangle, q_2) \in \delta'\}$.

Proposition 3.10. $L(A) = L(\textit{EPSfree}(A))$.

Proof. Assume that $w \in L(A)$. If $w = \epsilon$, then $w \in L(\textit{EPSfree}(A))$ since $q'_0 \in F'$. Assume then that $w \neq \epsilon$. The proof that $w \in L(\textit{EPSfree}(A))$ is the same as the corresponding proof for the case where $\epsilon \notin L(A)$, with a modification for the construction of the first transition in the accepting path for w in *EPSfree*(A). As in the previous case, there exist an arc $(q_0, \alpha_{i_1}, \langle \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta^*$ (if $i_1 = 1$) or an arc $(q_0, \alpha_{i_1}, \langle \vec{b}_{i_1}, \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta^*$ (if $i_1 \neq 1$), then, by the definition of δ', $(\overline{q_0}, \alpha_{i_1}, \langle \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta'$ or $(\overline{q_0}, \alpha_{i_1}, \langle \vec{b}_{i_1}, \vec{a}_{i_1} \rangle, q_{i_1}) \in \delta'$ respectively. In addition, as before, $u_0 \Vdash_{\langle \vec{a}_{i_1} \rangle} u_{i_1}$ or $u_0 \Vdash_{\langle \vec{b}_{i_1}, \vec{a}_{i_1} \rangle} u_{i_1}$ respectively (and in addition $(q'_0, u_0) = (\overline{q_0}, u_0)$ is the initial configuration of *EPSfree*(A)). The rest of the proof that $w \in L(\textit{EPSfree}(A))$ is the same as in the previous proof.

Assume that $w \in L(\textit{EPSfree}(A))$. If $w = \epsilon$ then clearly $w \in L(A)$ by the assumption. Assume then that $w \neq \epsilon$. Again, the proof is the same as the proof for the case where $\epsilon \notin L(A)$, with a modification for the first arc in the accepting path of w in *EPSfree*(A). As before, $(q'_0, \alpha_1, \langle \vec{a}_1 \rangle, q'_1) \in$

δ' is the first transition in the accepting path of w in *EPSfree*(A). By the definition of δ' and since $q'_0 = \overline{q_0}$, it follows that $(q_0, \alpha_1, \langle \vec{a}_1 \rangle, q'_1) \in \delta^*$. Then, simply replace c_0 by $c_0 = (q_0, u_0)$ and still $c_0 \vdash_{\alpha_1, \textit{EPSfree}(A)} c_1$ and c_0 is the initial configuration of A. Notice that for all k, $2 \leq k \leq r$, $(q'_{k-1}, \alpha_k, \langle \vec{a}_k \rangle, q'_k) \in \delta^*$ (since there are no arcs in δ' that are entering $\overline{q_0}$, no arcs except for the first arc can include this state and the first arc was dealt with). Then, as before, each of the arcs must belong to one of the four sets that δ^* was created as a union of. The rest of the proof is the same as in the previous proof.

□

3.6 FSRA optimizations

In FSRAs, traversing an arc depends not only on the input symbol but also on satisfying the series of register operations. Sometimes, a given series of register operations can never be satisfied, and thus the arc to which it is attached cannot be traversed. For example, the series of register operations $\langle (W, 1, a), (R, 1, b) \rangle$ can never be satisfied, thus an arc of the form $(q_1, \sigma, \langle (W, 1, a), (R, 1, b) \rangle, q_2)$ is redundant. In addition, the constructions shown in sections 3.4 and 3.5 might result in redundant states and arcs that can never be reached or can never lead to a final state. Moreover, in many cases a series of register operations can be minimized into a shorter series with the same effect. For example, the series of register operations $\langle (W, 1, a), (R, 1, a), (W, 1, b) \rangle$ is equal in its effect to the series $\langle (W, 1, b) \rangle$. Therefore, we show an algorithm for optimizing a given FSRA, i.e., minimizing the series of register operations over its arcs and removing the redundant arcs and states.

3.6.1 Register operations optimization

For a given FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F \rangle$, we construct an equivalent FSRA $A' = \langle Q, q_0, \Sigma, \Gamma, n, \delta', F \rangle = Opt(A)$ such that δ' is created from δ by removing redundant arcs and by optimizing all the series of register operations.

We begin by defining $\left(Actions_n^{\Gamma}\right)^+_{|i}$ as the subset of $\left(Actions_n^{\Gamma}\right)^+$ that consist only of operations

over the i-th register. Define a total function $sat_i : \left(Actions_n^{\Gamma}\right)^+_{|i} \longrightarrow \{\text{true,false}\}$ by:

$$sat_i(\vec{a}) = \begin{cases} true & \text{if there exist } u, v \in \Gamma^n \text{ such that } u \Vdash_{\vec{a}} v \\ \textit{false} & \text{otherwise} \end{cases}$$

$sat_i(\vec{a}) = true$ iff the series of register operations $\vec{a}$ is satisfiable, i.e., there exists a configuration of registers content for which all the operations in the series can be executed successfully. Determining whether $sat_i(\vec{a}) = true$ by exhaustively checking all the vectors in Γ^n may not be efficient. Therefore, we show a necessary and sufficient condition for determining whether $sat_i(\vec{a}) = true$ for some $\vec{a} \in \left(Actions_n^{\Gamma}\right)^+_{|i}$ which can be checked efficiently. In addition, this condition will be useful in optimizing the series of register operations as will be shown later. A series of register operations over the i-th register is *not satisfiable* if either one of the following holds:

- A write operation is followed by a read operation expecting a different value.
- A read operation is immediately followed by a read operation expecting a different value.

Proposition 3.11. *For all $\vec{a} = \langle (op_1, i, \gamma_1), (op_2, i, \gamma_2), \ldots, (op_s, i, \gamma_s) \rangle \in \left(Actions_n^{\Gamma}\right)^+_{|i}$, $sat_i(\vec{a}) =$ false if and only if either one of the following holds:*

1. *There exists k, $1 \leq k < s$, such that $op_k = W$ and there exists m, $k < m \leq s$, such that $op_m = R$, $\gamma_k \neq \gamma_m$ and for all j, $k < j < m$, $op_j = R$.*
2. *There exists k, $1 \leq k < s$, such that $op_k = op_{k+1} = R$ and $\gamma_k \neq \gamma_{k+1}$.*

Notice that if $i = 0$, then by the definition of FSRAs, all the register operations in the series are the same operation which is $(R, 0, \#)$ and this operation can never fail. In addition, if all the operations in the series are write operations, then again, by the definition of FSRAs, these operations can never fail. If none of the two conditions of the proposition holds, then the series of register operations is satisfiable.

We now show how to optimize a series of operations over a given register. An optimized series is defined only over satisfiable series of register operations in the following way:

- If all the operations are write operations, then leave only the last one (since it will overwrite all its predecessors).
- If all the operations are read operations, then by proposition 3.11, they are all the same operation, and in this case just leave one of them.
- If there are both read and write operations, then distinguish between two cases:
  - If the first operation is a write operation, leave only the last write operation in the series.
  - If the first operation is a read operation, leave the first operation (which is read) and the last write operation in the series. If the last write operation writes into the register the same symbol that the read operation required, then the write is redundant; leave only the read operation.

Definition 3.5. *Define a function* $min_i : \left(Actions_n^\Gamma\right)^+{}_{|i} \longrightarrow \left(Actions_n^\Gamma\right)^+{}_{|i}$. *Let* $\vec{a} = \langle (op_1, i, \gamma_1), \ldots, (op_s, i, \gamma_s)\rangle$. *If* $sat_i(\vec{a}) = true$ *then:*

- *If for all* k, $1 \leq k \leq s$, $op_k = W$, *define* $min_i(\vec{a}) = \langle (W, i, \gamma_s)\rangle$.
- *If for all* k, $1 \leq k \leq s$, $op_k = R$ *then define* $min_i(\vec{a}) = \langle (R, i, \gamma_s)\rangle$.
- *If there exists* m, $1 \leq m \leq s$ *such that* $op_m = W$ *and if there exists* t, $1 \leq t \leq s$, *such that* $op_t = R$ *then:*

$$min_i(\vec{a}) = \begin{cases} \langle (W, i, \gamma_j)\rangle & \text{if } op_1 = W \text{ and for all } k,\ j < k \leq s,\ op_k = R \\ \langle (R, i, \gamma_1), (W, i, \gamma_j)\rangle & \text{if } op_1 = R \text{ and for all } k,\ j < k \leq s,\ op_k = R \text{ and } \gamma_1 \neq \gamma_j \\ \langle (R, i, \gamma_1)\rangle & \text{if } op_1 = R \text{ and if there exists } j,\ 1 \leq j \leq s, \text{ such that for all } k,\ j < k \leq s,\ op_k = R \text{ and } \gamma_1 = \gamma_j \end{cases}$$

The formal proof that $min_i(\vec{a})$ is the minimal equivalent series of register operations of $\vec{a}$ is suppressed.

We now show how to optimize a series of register operations. Define a function $min : \left(Actions_n^\Gamma\right)^+ \longrightarrow \left(Actions_n^\Gamma\right)^+ \cup \{null\}$. For all $\vec{a} \in \left(Actions_n^\Gamma\right)^+$ define $min(\vec{a}) = \vec{b}$ where $\vec{b}$ is obtained from $\vec{a}$ by:

- Each subseries $\vec{a}_i$ of $\vec{a}$, consisting of all the register operations on the i-th register, is checked for satisfaction. If $sat_i(\vec{a}_i) = \textit{false}$ then the arc cannot be traversed $min(\vec{a}) = \vec{b} = null$. If $sat_i(\vec{a}_i) = true$ then $\vec{a}_i$ is replaced in $\vec{a}$ by $min(\vec{a}_i)$. Notice that the order of the the minimized subseries in the complete series is unimportant as they operate on different registers.

- If there exists $i \neq 0$, such that $\vec{a_i}$ is not empty, then the subseries $\vec{a}_0$ consisting only of operations of the form $(R, 0, \#)$, is deleted from $\vec{a}$.

Finally, given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, \delta, F\rangle$, construct an equivalent FSRA $A' = \langle Q, q_0, \Sigma, \Gamma, n, \delta', F\rangle = Opt(A)$ where $\delta' = \{(q_1, \sigma, \langle min(\vec{a})\rangle, q_2) \mid (q_1, \sigma, \langle \vec{a}\rangle, q_2) \in \delta$ and $min(\vec{a}) \neq null\}$. $Opt(A)$ is optimized with respect to register operations.

3.6.2 Redundant States and arcs removal

Like FSAs, FSRAs might contain states that can never be reached or can never lead to a final state. These states (and their connected arcs) can be removed in the same way they are removed in FSAs. In sum, FSRA optimization is done in two stages:

1. Minimizing the series of register operations over the FSRA transitions.

2. Removing redundant states and arcs.

Notice that stage 1 must be performed before stage 2 as it can result in further reduction in the size of the network when performing the second stage. For a given FSRA A, define $OPT(A)$ as the FSRA obtained from $Opt(A)$ by removing all the redundant states and transitions. An FSRA A is *optimized* if $OPT(A) = A$ (notice that $OPT(A)$ is unique, i.e., if $B = OPT(A)$ and $C = OPT(A)$ then $B = C$).

3.7 FSRA minimization

FSRAs can be minimized along three different axes: states, arcs and registers. Reduction in the number of registers can always be achieved by converting an FSRA to an FSA (section 3.2, page 24),

eliminating registers altogether. Since FSRAs are inherently non-deterministic (see section 3.8 below), their minimization is related to the problem of NFA minimization. NFA minimization (in terms of both arcs and states) is known to be NP-hard.[1] However, while FSRA arc minimization is NP-hard as will be shown presently, FSRA state minimization is different. Recall that in proposition 3.5 (page 30) we have shown that any FSA has an equivalent FSRA-2 with 3 states and 2 registers. It thus follows that any FSRA has an equivalent FSRA-2 with 3 states (simply turn the FSRA into an FSA and then turn it into an FSRA-2 with 3 states). Notice that minimizing an FSRA in terms of states or registers can significantly increase the number of arcs. As many implementations of finite state devices use space that is a function of the number of arcs, the benefit that lies in such minimization is limited. Therefore, a different minimization function, involving all the three axes, is called for. However, we did not address this problem in this work.

Proposition 3.12. *FSRA arc minimization is NP-hard.*

Proof. Let ϕ be a CNF formula with m clauses and n variables. Construct an NFA that accepts exactly the assignments that do not satisfy ϕ:

Let $x_1, \ldots, x_n$ be the variables of ϕ. Each assignment can be thought of as an n-bit binary number where the bit in the i-th position represents x_i's assignment. 0 represents *false* and 1 represents *true*. An assignment v does not satisfy ϕ iff it does not satisfy one of ϕ's clauses. Construct an NFA with m paths such that an assignment that does not satisfy one of ϕ's clauses is accepted by the path representing this clause. The general NFA for a given formula ϕ is shown in Figure 3.20. The i-th path simulates the assignment on the i-th clause as following:

- For all j, $1 \leq j \leq n$, if x_j occurs in the i-th clause then the arc connecting state $q_{i(j-1)}$ and q_{ij} is labeled 0.
- For all j, $1 \leq j \leq n$, if $\overline{x_j}$ occurs in the i-th clause then the arc connecting state $q_{i(j-1)}$ and q_{ij} is labeled 1.
- If neither x_j nor $\overline{x_j}$ occurs in the i-th clause then there are two arcs connecting state $q_{i(j-1)}$ and

[1] While this proposition is a part of folklore, we were unable to find a formal proof. Therefore, we explicitly prove this proposition for FSRAs.

q_{ij}, labeled 1 and 0, respectively.

- For q_{i1} the above holds for the arc connecting q_0 and q_{i1}.

Evidently, if an assignment v does not satisfy ϕ, then it does not satisfy at least one of ϕ's clauses, say the i-th clause. Hence the binary number representing this assignment over ϕ's variables is accepted by the path $q_0, q_{i1}, q_{i2}, \ldots, q_{in}$, where q_{in} is a final state. Observe that the number of states and arcs in this NFA is $O(mn)$. The constructed NFA recognizes all the n-bit binary words that do not satisfy ϕ. Extend this NFA to recognize also all the binary words whose length is not n, by unioning it with the automata of Figure 3.21 that accept all the words whose length is less than n and greater than n, respectively. The obtained NFA, A, whose size is polynomial in n, recognizes $\{0,1\}^*$ iff ϕ is not satisfiable.

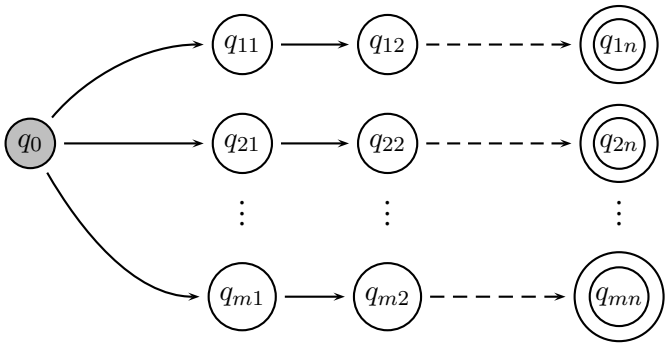
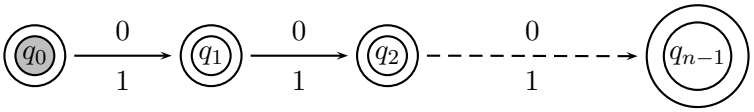


Figure 3.20: NFA for a cnf formula

The NFA A is also an FSRA with 1 register (register number 0). A has only one register, polynomial number of states and arcs and 0 register operations per arc. Now, minimize A with respect to arcs into an FSRA A', and assume that this can be done in a polynomial time. The size of A', in terms of states, arcs, registers and number of register operations per arc, is polynomial in n, since the minimization is done in polynomial time. Let $B = OPT(A')$. Notice that FSRA optimization can be done in time polynomial in the FSRA size (again, in terms of states, arcs, registers and number of

NFA for recognizing binary words with length less than n:



NFA for recognizing binary words with length greater than n:

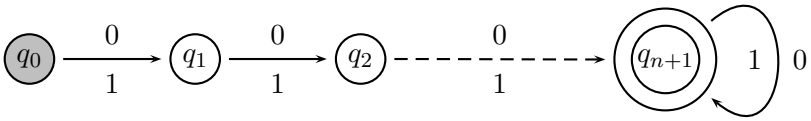


Figure 3.21: NFA for recognizing binary words with length not equal to n

register operations per arc) since minimizing the series of register operations over the transitions can be done in polynomial time and so is the redundant states and arcs removal. Thus, the size of B, in terms of states, arcs, registers and number of register operations per arc, is polynomial in n. Assume that $L(A) = L(B) = \{0,1\}^*$. Evidently, B must be the FSRA of Figure 3.22.

Lemma 3.2. *The subseries of $\vec{a}$ and $\vec{b}$ operating on the i-th register must be either one of the following:*

- *Both of the subseries of $\vec{a}$ and $\vec{b}$ have only one W instruction on register i.*
- *Each of the subseries is either empty or $\langle(W, i, \#)\rangle$ or $\langle(R, i, \#)\rangle$.*

Proof. B is optimized, hence each subseries of $\vec{a}$ and $\vec{b}$ operating on the i-th register must be either empty or $\langle(W, i, \gamma)\rangle$ or $\langle(R, i, \gamma)\rangle$ or $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$ (where $\gamma_1 \neq \gamma_2$). B accepts $\{0,1\}^*$, therefore the series of register operations $\vec{a}$ and $\vec{b}$ must be satisfied by any registers content, and hence it easy to see that each subseries of $\vec{a}$ and $\vec{b}$ operating on the i-th register must be either empty or

$\langle(W, i, \#)\rangle$ or $\langle(R, i, \#)\rangle$ (the symbol must be $\#$ as this is the initial content of the registers) or that both of the subseries of $\vec{a}$ and $\vec{b}$ have only W instruction on register i. □

Notice that verifying that $\vec{a}$ and $\vec{b}$ have this property can be done in polynomial time as their length is polynomial in n. Thus, it can be verified in polynomial time that $L(A) = L(B) = \{0, 1\}^*$. In sum, if FSRA arcs minimization could be done in polynomial time, we could determine the satisfiability of a given CNF formula ϕ in a polynomial time by constructing an FSRA accepting all the nonsatisfactory assignments of ϕ, minimizing and optimizing it and verifying that the resulting FSRA accepts $\{0, 1\}^*$.

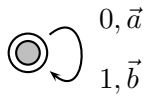


Figure 3.22: Minimal arcs FSRA for $\{0, 1\}^*$

□

3.8 FSRA Linearization

The main advantage of finite state devices is their linear recognition time. In finite state automata, linear recognition time is provided by determinizing the network: ensuring that the transition relation is a function. However, a functional transition relation in FSRA does not guarantee linear recognition time, since multiple possible transitions can exist for a given state and a given input symbol. For example, given an FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$, and some $q, q_1, q_2 \in Q$ and $\sigma \in \Sigma$, two arcs such as $(q, \sigma, \langle(W, 1, a)\rangle, q_1), (q, \sigma, \langle(W, 1, b)\rangle, q_2) \in \delta$ do not hamper the functionality of the FSRA transition relation (as δ can still be a function). However, they do imply that for the state q and for the same input symbol (σ), more than one possible arc can be traversed. Therefore, we use *deterministic*

to denote FSRAs in which the transition relation is a function, and a new term, *linearized*, is used to denote FSRAs for which linear recognition time is guaranteed.

Generally, an FSRA is linearized if it is optimized, ϵ-free, and given a current state and a new input symbol, at most one transition can be traversed. Thus, if the transition relation includes two arcs of the form $(q, \sigma, \langle\vec{a}\rangle, q_1), (q, \sigma, \langle\vec{b}\rangle, q_2)$, then $\vec{a}$ and $\vec{b}$ must be contradicting series of register operations. Two series of register operations are contradicting if at most one of them is satisfiable. Since the FSRA is optimized, each series of register operations is a concatenation of subseries, each operating on a different register, and the subseries operating on the i-th register must be either empty or $\langle(W, i, \gamma)\rangle$ or $\langle(R, i, \gamma)\rangle$ or $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$. $\langle(W, i, \gamma)\rangle$ contradicts neither $\langle(R, i, \gamma)\rangle$ nor $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$. $\langle(R, i, \gamma)\rangle$ and $\langle(R, i, \gamma_1), (W, i, \gamma_2)\rangle$ are contradicting only if $\gamma \neq \gamma_1$.

Definition 3.6. *An FSRA $A = \langle Q, q_0, \Sigma, \Gamma, n, k, \delta, F\rangle$, is linearized if it is optimized, ϵ-free and for all $(q, \sigma, \langle\vec{a}\rangle, q_1), (q, \sigma, \langle\vec{b}\rangle, q_2) \in \delta$ such that $\langle\vec{a}\rangle \neq \langle\vec{b}\rangle$, where $\langle\vec{a}\rangle = \langle(op_1^1, i_1^1, \gamma_1^1), \ldots, (op_k^1, i_k^1, \gamma_k^1)\rangle$ and $\langle\vec{b}\rangle = \langle(op_1^2, i_1^2, \gamma_1^2), \ldots, (op_m^2, i_m^2, \gamma_m^2)\rangle$, there exists j_1, $1 \leq j_1 \leq k$ and there exists j_2, $1 \leq j_2 \leq m$, such that $op_{j_1}^1 = op_{j_2}^2 = R$, $i_{j_1}^1 = i_{j_2}^2$ and $\gamma_{j_1}^1 \neq \gamma_{j_2}^2$.*

A naïve algorithm for converting a given FSRA into an equivalent linearized one is to turn it into an FSA and then determinize it. In the worst case, this results in an exponential increase in the network size. We were unable to find a better algorithm for turning a given FSRA into an equivalent linearized one. Moreover, we conjecture this is an NP-hard problem. This is an instance of the common tradeoff of time versus space: FSRA improve the network size which comes at the cost of slower analysis time. When using finite state devices for natural languages processing, often the generated networks are huge and their size is impractical for use. In such cases, using FSRAs can make network size manageable. Using the closure constructions one can build the desired network having reasonable size, and at the end decide whether to turn it into an ordinary FSA, if time performance is an issue.

Chapter 4

Linguistic applications

4.1 Circumfixes

Regular expressions are a formal way for defining regular languages. Regular language operations construct regular expressions in a convenient way. Several toolboxes (software packages) provide extended regular expression description languages and compilers of the expressions to finite state devices, automata and transducers (see section 1.2, page 4). We showed in example 3.1 that FSRAs can model phenomena of circumfixation in a more efficient way than finite state devices. We now introduce a new regular expression operator, accounting for circumfixes, and show how expressions using this operator are compiled into the appropriate FSRA. The new operator accepts a set of strings over Σ^*, representing a set of bases, and a list of circumfixes, each of which containing a prefix and a suffix (where one or both can be empty). It yields as a result an FSRA denoting the set containing all the strings created by prefixing and suffixing each of the bases with each of the circumfixes. For example, consider the Hebrew roots *r.$.m, n.h.l, p.q.d* and the Hebrew circumfixes *ht-ut,m-ϵ, h-h.* Given these two inputs, the new operator yields an FSRA accepting the set

$$\{htr\$mut, htnhlut, htpqdut, mr\$m, mnhl, mpqd, hr\$mh, hnhlh, hpqdh\}.$$

Definition 4.1. *Let Σ be a finite set such that $\Box, \{, \}, \langle, \rangle, \otimes \notin \Sigma$. We define the $\otimes$ operation to be of the form*

$$\{\langle\alpha_1\rangle, \langle\alpha_2\rangle, \ldots, \langle\alpha_n\rangle\}$$

$$\otimes$$

$$\{\langle\beta_1\Box\gamma_1\rangle, \langle\beta_2\Box\gamma_2\rangle, \ldots, \langle\beta_m\Box\gamma_m\rangle\}$$

where:

- $n \in \mathbb{N}$ *is the number of bases.*
- $m \in \mathbb{N}$ *is the number of circumfixes.*
- $\alpha_i \in \Sigma^*$ *for* $1 \leq i \leq n$ *is a base.*
- $\beta_i \in \Sigma^*$ *for* $1 \leq i \leq m$ *is the prefix of the* i*-th circumfix.*
- $\gamma_i \in \Sigma^*$ *for* $1 \leq i \leq m$ *is the suffix of the* i*-th circumfix.*

Consider first the case where $\alpha_i \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq n$ and $\beta_i, \gamma_i \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq m$. In this case the $\otimes$ operation yields as a result an FSRA-1 $A = \langle Q, q_0, \Sigma, \Gamma, 2, \delta, F\rangle$, such that $L(A) = \{\beta_i\alpha_j\gamma_i \mid 1 \leq i \leq m\ ,\ 1 \leq j \leq n\}$, where:

- $Q = \{q_0, q_1, q_2, q_3\}$
- $F = \{q_3\}$
- $\Sigma = (\{\alpha_i |\ 1 \leq i \leq n\}\ \cup\ \{\beta_i |\ 1 \leq i \leq m\}\ \cup\ \{\gamma_i |\ 1 \leq i \leq m\}) \setminus \{\epsilon\}$
- $\Gamma = \{\beta_i\Box\gamma_i |\ 1 \leq i \leq m\}\ \cup\ \{\#\}$.
- $\delta = \{(q_0, \beta_i, W, 1, \beta_i\Box\gamma_i, q_1) |\ 1 \leq i \leq m\}$
  $\cup$
  $\{(q_1, \alpha_i, q_2) |\ 1 \leq i \leq n\}$
  $\cup$
  $\{(q_2, \gamma_i, R, 1, \beta_i\Box\gamma_i, q_3) |\ 1 \leq i \leq m\}$

This FSRA is shown in Figure 4.1. It has 2 registers, where register 0 is the extra register to enable transitions of the form (s, σ, t) and register 1 'remembers' the circumfix. Notice that the FSRA will have 2 registers and 4 states for any number of roots and circumfixes. In other words, its number of

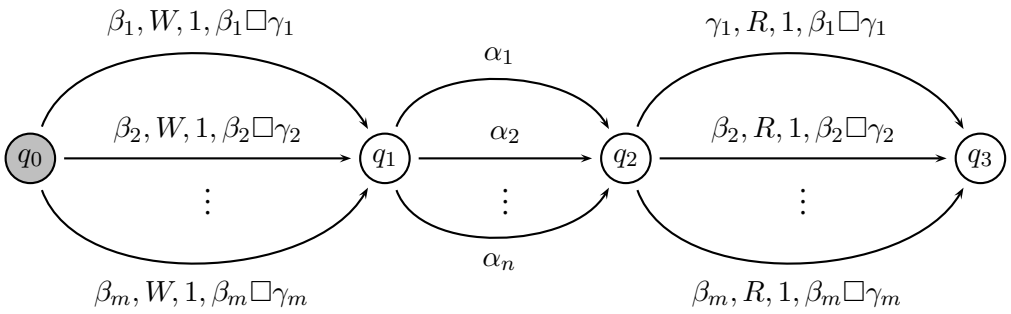


Figure 4.1: Circumfixes FSRA – general

states is independent of the number of roots and circumfixes. The number of arcs in this automaton is $2m + n$.

Consider now the general case where α_i, β_i and γ_i can be strings of letters. In this case, the $\otimes$ operation will be of the form

$$\{\langle\alpha_1^1, \alpha_1^2, \ldots, \alpha_1^{i_1}\rangle, \langle\alpha_2^1, \alpha_2^2, \ldots, \alpha_2^{i_1}\rangle, \ldots, \langle\alpha_n^1, \alpha_n^2, \ldots, \alpha_n^{i_n}\rangle\}$$

$$\otimes$$

$$\{\langle\beta_1^1, \beta_1^2, \ldots, \beta_1^{j_1}\Box\gamma_1^1, \gamma_1^2, \ldots, \gamma_1^{k_1}\rangle, \ldots, \langle\beta_m^1, \beta_m^2, \ldots, \beta_m^{j_m}\Box\gamma_m^1, \gamma_m^2, \ldots, \gamma_m^{k_m}\rangle\}$$

where:

- $n \in \mathbb{N}$ is the number roots.
- $m \in \mathbb{N}$ is the number of circumfixes.
- $\alpha_p^s \in \Sigma \cup \{\epsilon\}$ for $1 \leq p \leq n$ and $1 \leq s \leq i_p$.
- $\beta_p^s \in \Sigma \cup \{\epsilon\}$ for $1 \leq p \leq m$ and $1 \leq s \leq j_p$.
- $\gamma_p^s \in \Sigma \cup \{\epsilon\}$ for $1 \leq p \leq m$ and $1 \leq s \leq k_p$.

In this case arcs of the FSRA defined above are simply replaced by paths: where an arc in the above definition is labeled by the letter α_i it is replaced by a sequence of arcs labeled by the word α_i, and similarly for β_i and γ_i. The register operation, if there is one, is attached to the first arc in the series. For example, Figure 4.2 demonstrates the replacement of an arc of the form $(q_0, \beta_p, W, 1, \beta_p \Box \gamma_p, q_1)$ where $\beta_p = \beta_p^1 \beta_p^2 \dots \beta_p^j \in \Sigma^+$ and $j \geq 2$. Notice that $t_1, t_2, \dots, t_{j-1} \notin Q$.

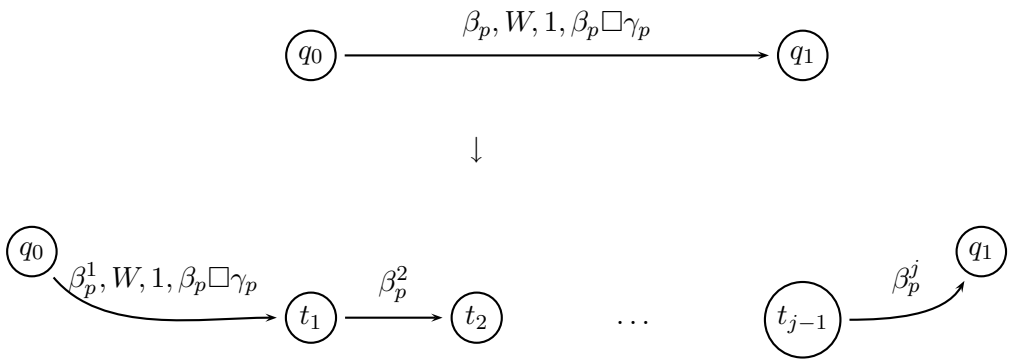


Figure 4.2: Arcs replacement

Example 4.1. *Consider once again the Hebrew roots r.$.m, n.h.l, p.q.d and the Hebrew circumfixes ht-ut, m-ϵ, h-h. The ⊗ operation*

$$\{\langle r, \$, m\rangle\langle n, h, l\rangle\langle p, q, d\rangle\} \otimes \{\langle h, t\Box u, t\rangle\langle m\Box\rangle\langle h\Box h\rangle\}$$

yields the FSRA of Figure 4.3.

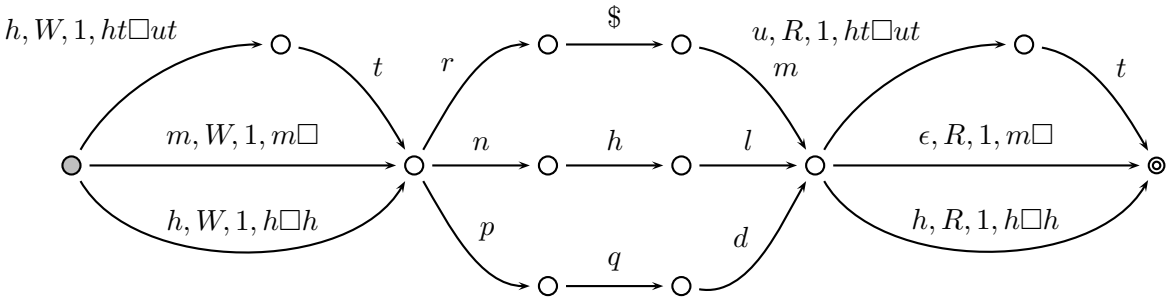


Figure 4.3: Circumfixes example

4.2 Interdigitation

We showed in example 3.2 that FSRAs can model the phenomenon of interdigitation in a more efficient way than finite state automata. We now introduce a new regular expression operator, defining the interdigitation process, and show how expressions using this operator are compiled into the appropriate FSRA.

The new operator accepts a set of strings of length n over Σ^*, representing a set of roots, and a list of patterns, each of which containing exactly n 'slots'. It yields as a result an FSRA denoting the set containing all the strings created by splicing the roots into the slots in the patterns. For example consider the Hebrew roots *r.$.m, p.&.l, p.q.d* and the Hebrew patterns *hit□a□e□,mi□□a□, ha□□a□a.* The roots are all trilateral, and the patterns have three slots each, represented by the symbol '□'. Given these two inputs, the new operator yields an FSRA denoting the set

$$\{hitra\$em, hitpa\&el, hitpaqed, mir\$am, mip\&al, mipqad, har\$ama, hap\&ala, hapqada\}.$$

Definition 4.2. *Let Σ be a finite set such that $\square, \{, \}, \langle, \rangle, \oplus \notin \Sigma$. We define the splice operation to be of the form*

$$\{\langle \alpha_{1\ 1}, \alpha_{1\ 2}, ..., \alpha_{1\ n}\rangle, \langle \alpha_{2\ 1}, \alpha_{2\ 2}, ..., \alpha_{2\ n}\rangle, ..., \langle \alpha_{m\ 1}, \alpha_{m\ 2}, ..., \alpha_{m\ n}\rangle\}$$

$$\oplus$$

$$\{\langle \beta_{1\ 1}\square\beta_{1\ 2}\square...\beta_{1\ n}\square\beta_{1\ n+1}\rangle, \langle \beta_{2\ 1}\square\beta_{2\ 2}\square...\beta_{2\ n}\square\beta_{2\ n+1}\rangle, ..., \langle \beta_{k\ 1}\square\beta_{k\ 2}\square...\beta_{k\ n}\square\beta_{k\ n+1}\rangle\}$$

where:

- $n \in \mathbb{N}$ *is the number of slots (represented by '□') in the patterns into which the roots letters should be inserted.*
- $m \in \mathbb{N}$ *is the number of roots to be inserted.*
- $k \in \mathbb{N}$ *is the number of patterns.*
- $\alpha_{ij} \in \Sigma^*$ *for* $1 \leq i \leq m$ *and* $1 \leq j \leq n$.
- $\beta_{ij} \in \Sigma^*$ *for* $1 \leq i \leq k$ *and* $1 \leq j \leq n+1$.

The left set is a set of roots to be inserted into the slots in the right set of patterns. The slots are represented by the symbol '$\Box$'. For the sake of brevity, β_i and α_i are used as shorthand notations for $\beta_{i1}\Box\beta_{i2}\Box...\Box\beta_{i(n+1)}$ and $\alpha_{i1}\alpha_{i2}...\alpha_{in}$, respectively. Consider first the case where $\alpha_{ij} \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$ and $\beta_{ij} \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq k$ and $1 \leq j \leq n+1$. In this case the splice operation yields as a result an FSRA-1 $A = \langle Q, q_0, \Sigma, \Gamma, 3, \delta, F\rangle$, such that $L(A) = \{\beta_{j1}\alpha_{i1}\beta_{j2}\alpha_{i2}...\beta_{jn}\alpha_{in}\beta_{j(n+1)} \mid 1 \leq i \leq m\ ,\ 1 \leq j \leq k\}$, where:

- $Q = \{q_0, q_1, ..., q_{2n+1}\}$
- $F = \{q_{2n+1}\}$
- $\Sigma = (\{\alpha_{ij}|\ 1 \leq i \leq m\ ,\ 1 \leq j \leq n\}\ \cup\ \{\beta_{ij}|\ 1 \leq i \leq k\ ,\ 1 \leq j \leq n+1\}) \setminus \{\epsilon\}$
- $\Gamma = \{\beta_i|\ 1 \leq i \leq k\}\ \cup\ \{\alpha_i|\ 1 \leq i \leq m\}\ \cup\ \{\#\}$.
- $\delta = \{(q_0, \beta_{i1}, W, 1, \beta_i, q_1)|\ 1 \leq i \leq k\}$

  $\cup$

  $\{(q_1, \alpha_{i1}, W, 2, \alpha_i, q_2)|\ 1 \leq i \leq m\}$

  $\cup$

  $\{(q_{2j-2}, \beta_{ij}, R, 1, \beta_i, q_{2j-1})|\ 1 \leq i \leq k\ ,\ 2 \leq j \leq n+1\}$

  $\cup$

  $\{(q_{2j-1}, \alpha_{ij}, R, 2, \alpha_i, q_{2j})|\ 1 \leq i \leq m\ ,\ 2 \leq j \leq n\}$

This FSRA is shown in Figure 4.4. It has 3 registers, where register 1 'remembers' the pattern and register 2 'remembers' the root. Notice that the FSRA will have 3 registers and $2n+2$ states for any number of roots and patterns. In other words, its number of states is independent of the number of roots and patterns. The number of arcs in this automaton is $k \times (n+1) + m \times n$. In the case of trilateral roots, for m roots and k patterns the resulting machine has a constant number of states and $O(k+m)$ arcs.

Consider now the general case where α_{ij} and β_{ij} can be strings of letters. In this case, arcs of the FSRA defined above are simply replaced by paths, similarly to the case of circumfixes (section 4.1).

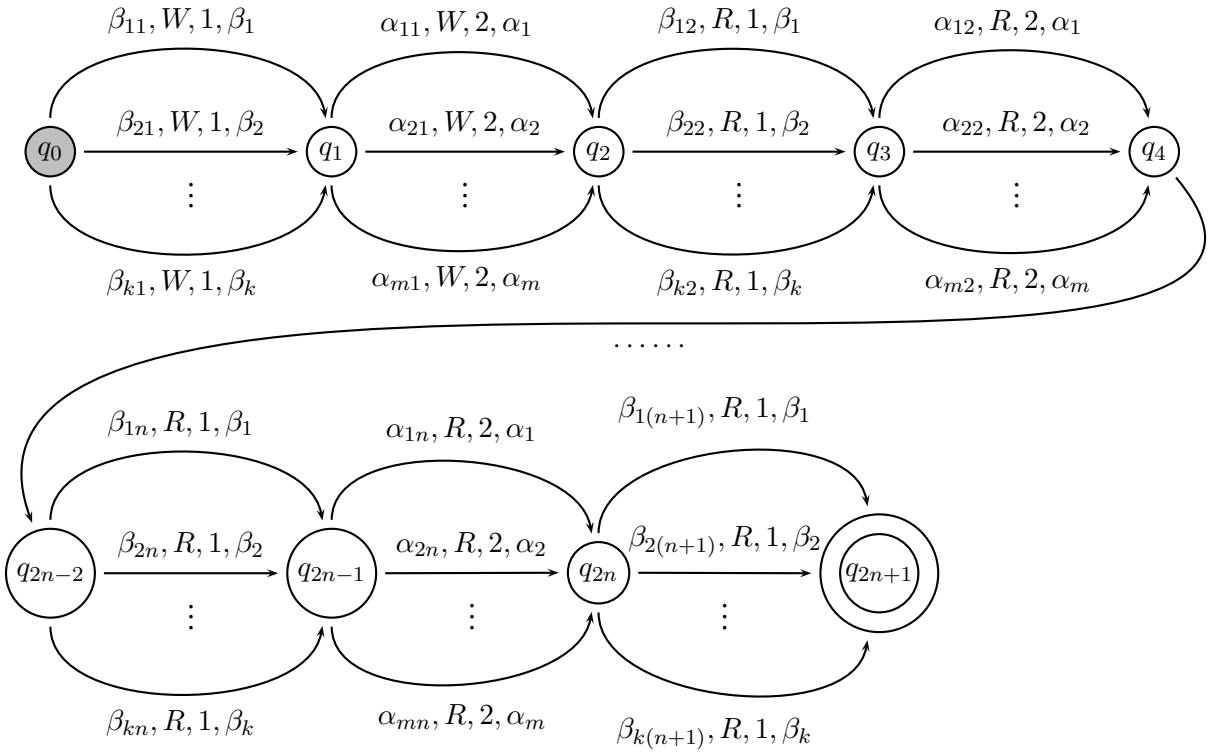


Figure 4.4: Interdigitation FSRA – general

Example 4.2. *Consider again the Hebrew roots r.$.m, p.&.l, p.q.d and the Hebrew patterns hit□a□e□, mi□□a□ and ha□□a□a. The splice operation*

$$\{\langle r, \$, m\rangle\langle p, \&, l\rangle\langle p, q, d\rangle\} \oplus \{\langle hit\Box a\Box e\Box\rangle\langle mi\Box\Box a\Box\rangle\langle ha\Box\Box a\Box a\rangle\}$$

yields the FSRA of Figure 4.5. The ϵ-arc was added only for the convenience of the drawing.

It should be noted that like other processes of derivational morphology, Hebrew word formation is highly idiosyncratic: not all roots combine with all patterns, and there is no systematic way to determine when such combinations will be realized in the language. Yet, this does not render our proposed operators useless: one can naturally characterize classes of roots and classes of patterns for which all the combinations exist. Furthermore, even in cases where such a characterization is difficult to come by, the splice operator can be used, in combination with other extended regular expression operators,

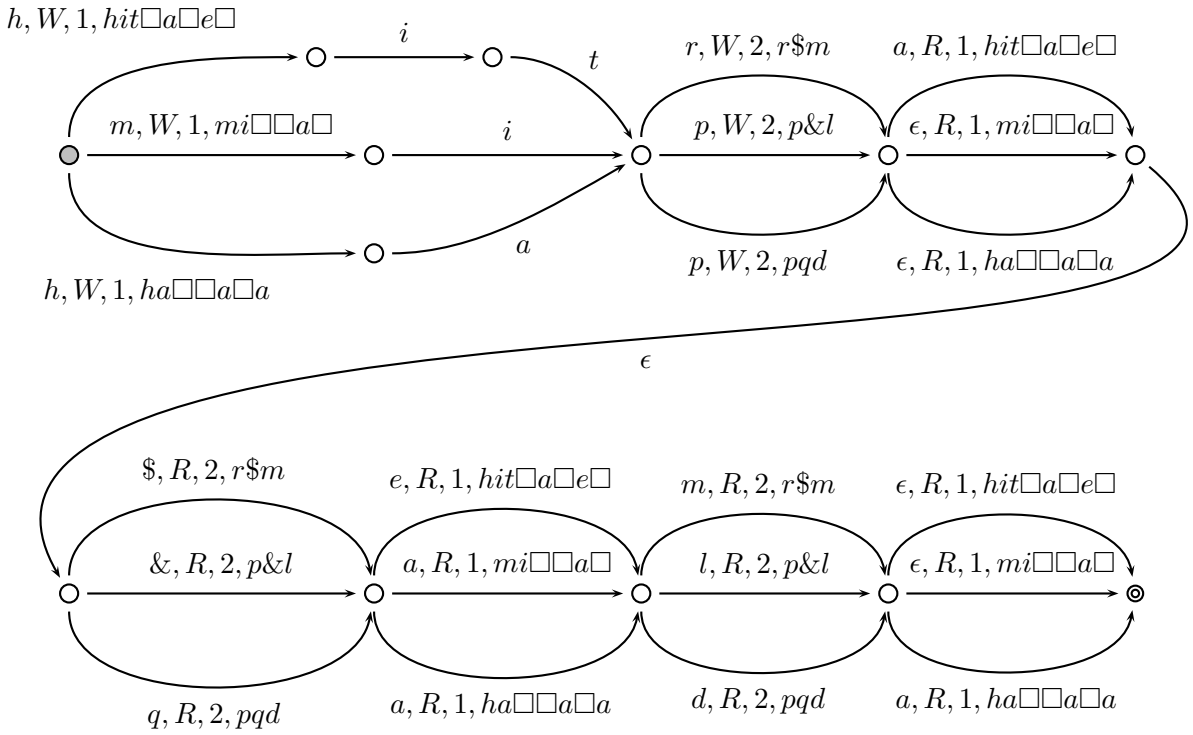


Figure 4.5: Interdigitation example

to define complex expressions for generating the required language. This is compatible with the general approach for using finite state techniques, implementing each phenomenon independently and combining them together using closure properties.

4.3 Reduplication

We now return to the reduplication problem as was presented in section 1.3 (page 8). We extend the finite state registered model to efficiently accept L_n, a finite instance of the general problem, which is practically sufficient for describing reduplication in natural languages. Using FSRAs as defined above does not improve space efficiency, because a separate path for each reduplication is still needed. Notice that the different symbols in L_n have no significance except the pattern they create. Therefore,

FSRAs have to be extended in order to be able to identify a pattern without actually distinguishing between different symbols in it. The extended model, FSRA*, is created from the FSRA-1 model by adding a new symbol, '*', assumed not to belong to Σ, and by forcing Γ to be equal to Σ. The '*' indicates equality between the input symbol and the designated register content, eliminating the need to duplicate paths for different symbols.

Definition 4.3. *Let $* \notin \Sigma$. An FSRA* is an FSRA-1 where $\Sigma = \Gamma$ (and thus includes '#') and the transition function is extended to be $\delta \subseteq Q \times \Sigma \cup \{\epsilon, *\} \times \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Sigma \cup \{*\} \times Q$. The extended meaning of δ is as follows:*

- $(s, \sigma, R, i, \gamma, t) \in \delta$, $(s, \sigma, W, i, \gamma, t) \in \delta$ *where $\sigma, \gamma \neq *$ imply the same as before.*
- $(s, \sigma, R, i, *, t) \in \delta$ *and* $(s, *, R, i, \sigma, t) \in \delta$ *for $\sigma \neq \epsilon$ imply that if the automaton is in state s, the input symbol is σ and the content of the i-th register is the same σ, then the automaton may enter state t.*
- $(s, \sigma, W, i, *, t) \in \delta$ *and* $(s, *, W, i, \sigma, t) \in \delta$ *for $\sigma \neq \epsilon$ imply that if the automaton is in state s and the input symbol is σ, then the content of the i-th register is changed to σ, and the automaton may enter state t.*
- $(s, *, R, i, *, t) \in \delta$ *implies that if the automaton is in state s, the input symbol is some $\sigma \in \Sigma$ and the content of the i-th register is the same σ, then the automaton may enter state t.*
- $(s, *, W, i, *, t) \in \delta$ *implies that if the automaton is in state s and the input symbol is some $\sigma \in \Sigma$, then the content of the i-th register is changed to the same σ, and the automaton may enter state t.*

With this extended model we can construct an efficient registered automaton for L_n: The number of registers is n+1. Register number 0 is the extra register to allow transitions as in the ordinary FSA. Registers $1, ..., n$ 'remember' the first n symbols to be duplicated. Figure 4.6 depicts an extended registered automaton that accepts L_n for $n = 4$. Notice that the number of states depends only on n and not on the size of Σ. Figure 4.7 schematically depicts an extended registered automaton that accepts L_n for some $n \in \mathbb{N}$. The language $\{ww \mid |w| \leq n\}$ for some $n \in \mathbb{N}$ can be generated by a

union of FSRA*, each one generating L_n for some $i \leq n$. Since n is usually small in natural language reduplication, the resulting automaton is manageable, and in any case, considerably smaller then the naïve automaton.

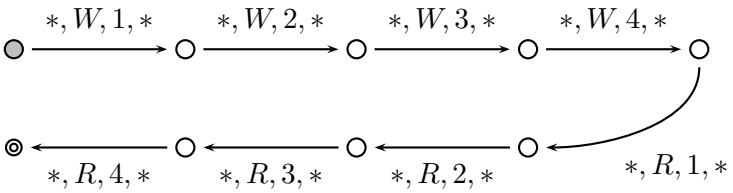


Figure 4.6: Reduplication for n=4

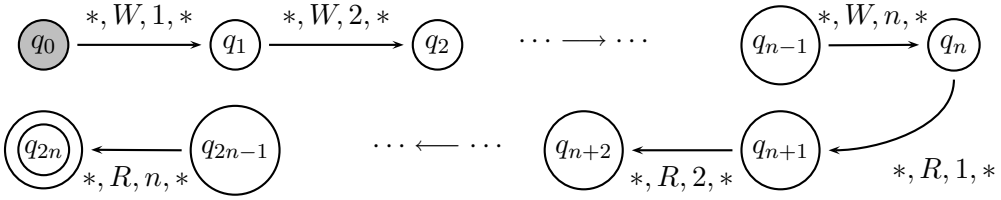


Figure 4.7: Reduplication – general case

4.4 Assimilation

In example 3.3 (page 27) FSRAs are used to model assimilation in nominative definite nouns in Arabic. Using the FSRA* model defined above, further reduction in the network size can be achieved. The FSRA* of Figure 4.8 accepts all the nominative definite forms of the Arabic nouns *kitaab*, *qamar* and *daftar*. In the same way it can be extended to accept any set of nominative definite Arabic nouns. Register 1 stores information about the actual form of the definite article, to ensure that assimilation occurs when needed and only then.

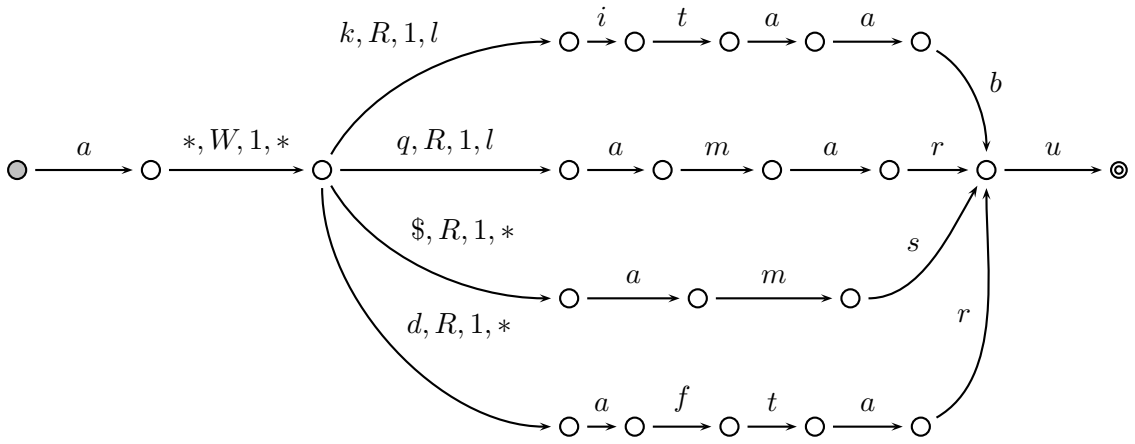


Figure 4.8: FSRA* for Arabic nominative definite nouns

Chapter 5

Finite state registered transducers

We extend the FSRA model to transducers, denoting relations over two finite alphabets. The extension is done by adding to each transition an output symbol.

5.1 Definitions and examples

An *finite state registered transducer (FSRT)* extends an FSRA in the same way that an FST extends FSA: the only difference between an FSRA and an FSRT is an additional alphabet in the latter.

Definition 5.1. *A finite state registered transducer (FSRT) is a tuple* $A = \langle Q, q_0, \Sigma_1, \Sigma_2, \Gamma, n, \delta, F \rangle$*, where:*

- *Q is a finite set of states.*
- $q_0 \in Q$ *is the initial state.*
- Σ_1 *is a finite alphabet (the upper language alphabet).*
- Σ_2 *is a finite alphabet (the lower language alphabet).*
- Γ *is a finite alphabet including '*$\#$*' (the registers alphabet).*
- $n \in \mathbb{N}$ *(indicates the number of registers).*
- $\delta \subseteq Q \times \Sigma_1 \cup \{\epsilon\} \times \Sigma_2 \cup \{\epsilon\} \times \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma \times Q$ *is the transition relation.*

- *$F \subseteq Q$ is the set of final states.*
- *The initial content of the registers is $\#^n$, meaning that the initial value of all the registers is 'empty'.*

The relation recognized by an FSRT A, denoted $R(A)$, is the set of pairs in $\Sigma_1^* \times \Sigma_2^*$ accepted by A. The formal definition is a naïve extension of the corresponding definition for FSRAs (simply replace all the occurrences of individual symbols by pairs of symbols).

Example 5.1. *Consider again example 3.1 (page 22). The automaton of example 3.1 accepts all the legal combinations of roots and patterns. This automaton can be extended to an FSRT, giving as output the correct analysis as pattern and root. This transducer is shown in Figure 5.1. Where in FSRAs the registers are used to control the transition function, in FSRTs they are used also to control the output. In this case, the relation defined by the transducer of Figure 5.1 is:* $\{hr\$ma : h\square\square\square a + r\$m, hnhla : h\square\square\square a + nhl, hpqda : h\square\square\square a + pqd, \ldots\} \cup \{htr\$mut : ht\square\square\square ut + r\$m, htnhlut : ht\square\square\square ut + nhl, htpqdut : ht\square\square\square ut + pqd, \ldots\} \cup \{mr\$m : m\square\square\square + r\$m, mnhl : m\square\square\square + nhl, mpqd : m\square\square\square + pqd, \ldots\}$.

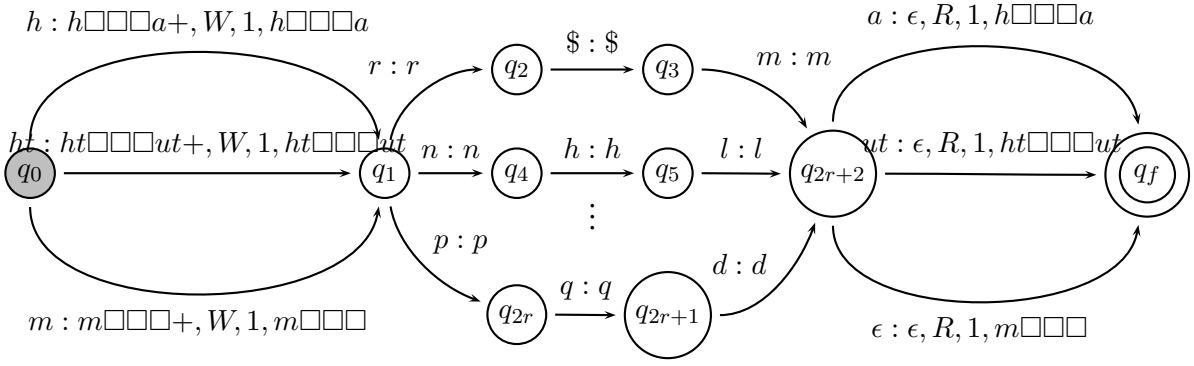


Figure 5.1: Transducer for roots and patterns

Example 5.2. *Consider again example 3.2. The automaton in example 3.2 accepts all the legal combinations of roots and the pattern. This automaton can also be extended to an FSRT, giving as output the correct analysis as pattern and root. This transducer is shown in Figure 5.2. Notice that $\square \in \Sigma_2$ and indicates the slots in the patterns. The relation defined by the transducer of Figure 5.2 is:* $\{hitragez : hit\square a\square e\square + rgz, hitba\$el : hit\square a\square e\square + b\$l, hitgaber : hit\square a\square e\square + gbr\}$.

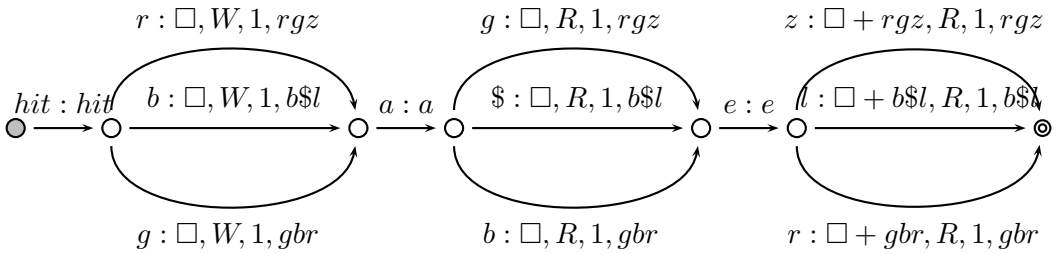


Figure 5.2: Transducer for roots and patterns

As in the FSRA model, we extend the FSRT model to allow multiple register operations on each transition.

Definition 5.2. *An order-k finite state registered transducer (FSRT-k) is a tuple* $A = \langle Q, q_0, \Sigma_1, \Sigma_2, \Gamma, n, k, \delta, F \rangle$*, where:*

- $Q, q_0, \Sigma_1, \Sigma_2, \Gamma, n, F$ *and the initial content of the registers are as before.*
- $k \in \mathbb{N}$ *(indicates the maximum number of register operations allowed on each arc).*
- *Let* $Actions_n^\Gamma = \{R, W\} \times \{0, 1, 2, \ldots, n-1\} \times \Gamma$*, then*

  $$\delta \subseteq Q \times \Sigma_1 \cup \{\epsilon\} \times \Sigma_2 \cup \{\epsilon\} \times \left(\bigcup_{j=1}^{k} \left\{ \langle a_1, ..., a_j \rangle \mid for\ \ all\ \ i,\ \ 1 \leq i \leq j,\ a_i \in Actions_n^\Gamma \right\} \right) \times Q$$

  is the transition relation. δ *is extended to allow each transition to be associated with a series of up to k operations on the registers. Each operation has the same meaning as before.*

The register operations are done in the order in which they are specified. Thus, $(s, \sigma_1, \sigma_2, \langle a_1, ..., a_i \rangle, t) \in \delta$ where $i \leq k$ implies that if A is in state s, the input symbol is σ_1 and all the register operations $a_1, ..., a_i$ are executed successfully, then A outputs σ_2 and may enter state t. The relation recognized by an FSRT-k A, denoted $R(A)$, is the set of pairs in $\Sigma_1^* \times \Sigma_2^*$ accepted by A. As in FSRAs, the term FSRT will be used for denoting FSRT-k; the ordinary FSRT will be referred to as FSRT-1.

Example 5.3. *Consider again the 32-bit incrementor example mentioned in section 2.2.2 (page 16). Recall that a sequential transducer for an n-bit binary incrementor would require 2^n states and a similar number of transitions. Using the FSRT model, a more efficient n-bit transducer can be constructed. A 4-bit FSRT incrementor is shown in Figure 5.3. The first four transitions copy the input string into the registers, then the input is scanned (using the registers) from right to left (as the carry moves), calculating the result, and the last four transitions output the result (in case the input is 1^n, an extra 1 is added in the beginning). Notice that this transducer guarantees linear recognition time, since from each state only one arc can be traversed in each step, even when there are ϵ-arcs. In the same way, an n-bit transducer can be constructed for all $n \in \mathbb{N}$. This transducer will have n registers, $3n + 1$ states and $6n$ arcs. The FSRT model solves the incrementor problem in much the same way it is solved by vectorized finite state automata, but the FSRT solution is more intuitive and is based on existing finite state techniques.*

5.2 Closure properties

As in FSRAs, implementing the closure properties directly on FSRTs is essential for benefiting from this new model. Union, concatenation and Kleene closure of FSRTs are done in the same way as for FSRAs. Therefore we only show how to perform a composition of two FSRTs.

5.2.1 Closure under composition

Given two transducers $A_1 = \langle Q_1, q_0^1, \Sigma, \delta_1, F_1 \rangle$ and $A_2 = \langle Q_2, q_0^2, \Sigma, \delta_2, F_2 \rangle$, the composition of $R(A_1)$ and $R(A_2)$, denoted $R(A_1) \circ R(A_2)$, is the relation consisting of all the pairs (x, y) such that there exists some intermediate string z, where $(x, z) \in R(A_1)$ and $(z, y) \in R(A_2)$. Kaplan and Kay (1994) suggest a construction for a transducer $A = \langle Q, q_0, \Sigma, \delta, F \rangle$ such that $R(A) = R(A_1) \circ R(A_2)$ by defining:

- $Q = Q_1 \times Q_2$.
- $q_0 = (q_0^1, q_0^2)$.
- $F = F_1 \times F_2$.

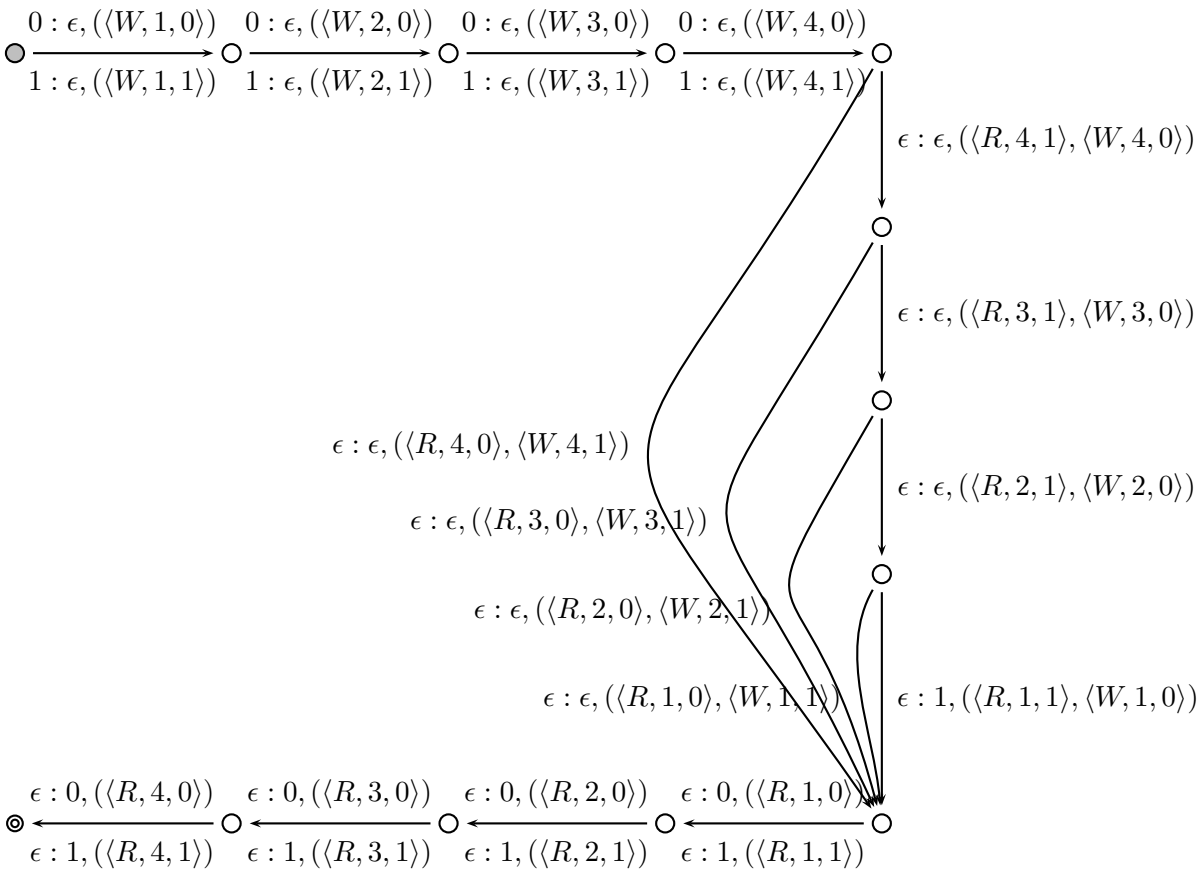


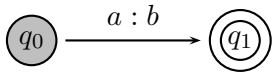
Figure 5.3: 4-bit incrementor using FSRT

- $\delta = \{((s_1, s_2), \sigma_1, \sigma_2, (t_1, t_2)) \mid$ there exists $\sigma \in \Sigma \cup \{\epsilon\}$ such that $(s_1, \sigma_1, \sigma, t_1) \in \delta_1$ and $(s_2, \sigma, \sigma_2, t_2) \in \delta_2\}$.

The problem with this construction is that it is not fully correct as it does not deal correctly with transducers containing ϵ-transitions, as the following example demonstrates.

Example 5.4. *Consider A and B, the transducers of Figure 5.4. $R(A) = \{a : b\}$ and $R(B) = \{b : cd\}$, hence $R(A) \circ R(B) = \{a : cd\}$. However, Kaplan and Kay (1994)'s construction results in the transducer of Figure 5.5 that accepts the empty language.*

A:



B:

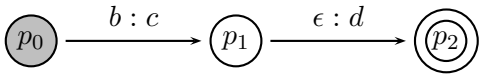


Figure 5.4: Transducers A and B

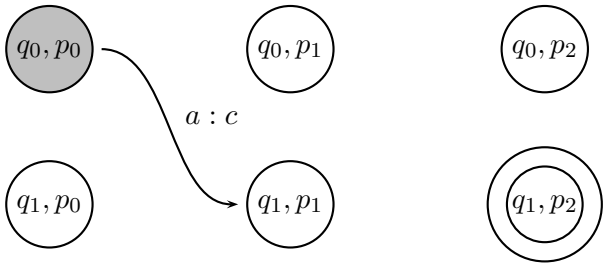


Figure 5.5: Kaplan and Kay construction for $A \circ B$

The solution for this problem (Mohri, Pereira, and Riley, 1996; Pereira and Riley, 1997; Riley, Pereira, and Mohri, 1997) is to extend the transducers transitions by adding implicit ϵ-self-loops in both of the transducers. After constructing the composition transducer, the remaining ϵ-self-loops are removed. Thus, the correct composition transducer for $R(A) \circ R(B)$ (for A, B, the transducers in Figure 5.4) is described in Figure 5.6. However, in some cases multiple accepting paths are obtained by using this extension, as the following example demonstrates.

Example 5.5. *Consider A and B, the transducers of Figure 5.7. $R(A) = \{ac : b\}$ and $R(B) = \{b : de\}$, hence $R(A) \circ R(B) = \{ac : de\}$. Indeed, the transducer of Figure 5.8, constructed by the above technique, accepts this set but it contains three accepting paths for the pair $ac : de$.*

To keep only one of those paths, Pereira and Riley (1997) show how a filter (which is a transducer itself) can be inserted between A and B to remove the redundant paths. The algorithm we show for

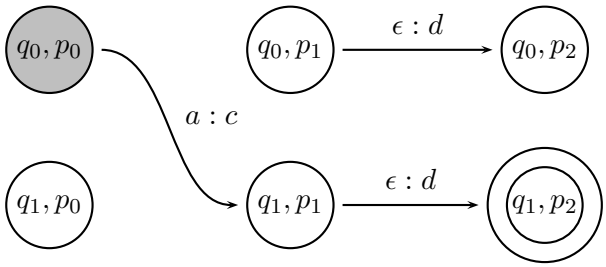
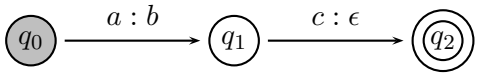


Figure 5.6: Correct construction for $A \circ B$

A:



B:

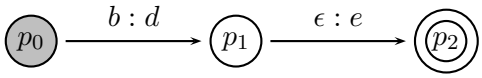


Figure 5.7: Transducers A and B

FSRT composition is based on the basic construction of Kaplan and Kay (1994) with the ϵ-self-loops extension.

In what follows, let $A_1 = \langle Q_1, q_0^1, \Psi_1, \Psi_2, \Gamma_1, n_1, k_1, \delta_1, F_1 \rangle$ and $A_2 = \langle Q_2, q_0^2, \Omega_1, \Omega_2, \Gamma_2, n_2, k_2, \delta_2, F_2 \rangle$ be finite state registered transducers. We use the function ***shift***$_{n_1}$, defined in section 3.4 (page 32). First, we extend A_1 and A_2 to contain all the implicit ϵ-self-loops. Formally, define $A_1' = \langle Q_1, q_0^1, \Psi_1, \Psi_2, \Gamma_1, n_1, k_1, \delta_1', F_1 \rangle$ where $\delta_1' = \delta_1 \cup \{(q, \epsilon, \epsilon, q) \mid q \in Q_1\}$ and define $A_2' = \langle Q_2, q_0^2, \Omega_1, \Omega_2, \Gamma_2, n_2, k_2, \delta_2' , F_2 \rangle$ where $\delta_2' = \delta_2 \cup \{(q, \epsilon, \epsilon, q) \mid q \in Q_2\}$. We construct an FSRT $A = \langle Q, q_0, \Sigma_1, \Sigma_2, \Gamma, n, k, \delta, F \rangle$, where:

- $Q = Q_1 \times Q_2$.
- $q_0 = (q_0^1, q_0^2)$.
- $F = F_1 \times F_2$.

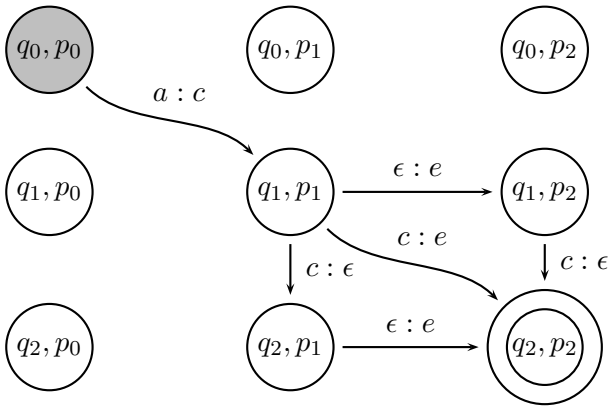


Figure 5.8: Multiple accepting paths in the correct construction for $A \circ B$

- $\Sigma_1 = \Psi_1$.
- $\Sigma_2 = \Omega_2$.
- $\Gamma = \Gamma_1 \cup \Gamma_2$.
- $n = n_1 + n_2$. Actually only $n = n_1 + n_2 - 1$ registers are needed as the two register 0 of A_1 and A_2 can be unified into one, but for the sake of simplicity they are left apart.
- $k = k_1 + k_2$. Actually k can be smaller, but this is its value in the worse case.
- Define $\delta' = \Big\{\Big((s_1, s_2), \sigma_1, \sigma_2, \langle \vec{a}, \mathit{shift}_{n_1}(\vec{b})\rangle, (t_1, t_2)\Big) \mid$ there exists $\sigma \in (\Psi_2 \cap \Omega_1) \cup \{\epsilon\}$ such that $(s_1, \sigma_1, \sigma, \langle \vec{a}\rangle, t_1) \in \delta'_1$ and $(s_2, \sigma, \sigma_2, \langle \vec{b}\rangle, t_2) \in \delta'_2\Big\}$.

  Define $\delta = \delta' \setminus \{(q, \epsilon, \epsilon, q) \mid q \in Q\}$. No special treatment is needed for the case in which a register operation is performed over register 0 in δ_2 since registers number 0 of the two transducers were kept apart. δ' is the composition relation after the implicit ϵ-self-loops extension, where δ is created from δ' by removing all the redundant ϵ-self-loops. Notice that redundant paths can occur in A (as was shown in example 5.5). They can be removed in the same way in which they are removed in finite state transducers.

Proposition 5.1. $R(A) = R(A_1) \circ R(A_2)$

Proof. Assume $(w,v) \in R(A)$. Then there exists a series of configurations $c_0,\ldots,c_r$ such that $c_0 = {q_0}^c$, $c_r \in F^c$ and for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{w_k:v_k,A} c_k$, $w = w_1 \ldots w_r$ and $v = v_1 \ldots v_r$. Define for all k, $0 \leq k \leq r$, $c_k = \left((q_k^1, q_k^2), u_k\right)$ (and indeed $c_0 = \left((q_0^1, q_0^2), \#^{n_1+n_2}\right) = q_0^c$. By the definition of the product relation it follows that for all k, $1 \leq k \leq r$, there exists $\vec{d_k} \in \left(Actions_{n_1+n_2}^{\Gamma_1 \cup \Gamma_2}\right)^+$ such that $\left((q_{k-1}^1, q_{k-1}^2), w_k, v_k, \langle \vec{d_k} \rangle, (q_k^1, q_k^2)\right) \in \delta$ and $u_{k-1} \Vdash_{\langle \vec{d_k} \rangle} u_k$.

Define for all k, $0 \leq k \leq r$, $u_k = \langle u_k^1, u_k^2 \rangle$ where $u_k^1 \in \Gamma_1^{n_1}$ is the content of the first n_1 registers and $u_k^2 \in \Gamma_2^{n_2}$ is the content of the next n_2 registers. Notice that for all $\vec{a} \in \left(Actions_{n_1}^{\Gamma_1}\right)^+$, $\vec{a}$ operates only on the first n_1 registers in A, and for all $\vec{b} \in \left(Actions_{n_2}^{\Gamma_2}\right)^+$, $\mathit{shift}_{n_1}(\vec{b})$ operates only on the next n_2 registers. Now, define for all k, $0 \leq k \leq r$, $c_k^1 = (q_k^1, u_k^1)$ and $c_k^2 = (q_k^2, u_k^2)$. Notice that c_0^1 and c_0^2 are the initial configurations of A_1' and A_2' respectively, and that c_r^1 and c_r^2 are final configurations of A_1' and A_2' respectively. By the definition of δ it follows that there exists $\alpha_k \in (\Psi_2 \cap \Omega_1) \cup \{\epsilon\}$ and there exist $\vec{a} \in \left(Actions_{n_1}^{\Gamma_1}\right)^+$, $\vec{b} \in \left(Actions_{n_2}^{\Gamma_2}\right)^+$ such that:

(1) $(q_{k-1}^1, w_k, \alpha_k, \langle \vec{a}_k \rangle, q_k^1) \in \delta_1'$.

(2) $(q_{k-1}^2, \alpha_k, v_k, \langle \vec{b}_k \rangle, q_k^2) \in \delta_2'$.

(3) $\langle \vec{d_k} \rangle = \langle \vec{a}_k, \mathit{shift}_{n_1}(\vec{a}_k) \rangle$.

(4) $u_{k-1}^1 \Vdash_{\langle \vec{a}_k \rangle} u_k^1$.

(5) $u_{k-1}^2 \Vdash_{\langle \vec{b}_k \rangle} u_k^2$.

From (1) and (4) it follows that $c_{k-1}^1 \vdash_{w_k:\alpha_k,A_1'} c_k^1$, and from (2) and (5) it follows that $c_{k-1}^2 \vdash_{\alpha_k:v_k,A_2'} c_k^2$. In sum, by defining $x = \alpha_1, \ldots, \alpha_r$ we obtain that there exists a series of configurations $c_0^1, \ldots, c_r^1$, such that for all k, $1 \leq k \leq r$, $c_{k-1}^1 \vdash_{w_k:\alpha_k,A_1'} c_k^1$. In addition, c_0^1 is the initial configuration of A_1' and c_r^1 is a final configuration of A_1'. Therefore, $(w_1 \ldots w_r, \alpha_1 \ldots \alpha_r) \in R(A_1')$, and hence $(w,x) \in R(A_1')$. Evidently, $R(A_1) = R(A_1')$ (as A_1' was created by adding A_1 the implicit ϵ-self-loops), and hence $(w,x) \in R(A_1)$. Moreover, there exists a series of configurations $c_0^2, \ldots, c_r^2$, such that for all k, $1 \leq k \leq r$, $c_{k-1}^2 \vdash_{\alpha_k:v_k,A_2'} c_k^2$. In addition, c_0^2 is the initial configuration of A_2' and c_r^2 is a final configuration of A_2'. Therefore, $(\alpha_1 \ldots \alpha_r, v_1 \ldots v_r) \in R(A_2')$, and hence $(x,v) \in R(A_2') = R(A_2)$. Hence $(w,v) \in R(A_1) \circ R(A_2)$ and $R(A) \subseteq R(A_1) \circ R(A_2)$.

Assume $(w,v) \in R(A_1) \circ R(A_2)$. Evidently, $R(A_1) = R(A_1')$ and $R(A_2) = R(A_2')$, therefore, $(w,v) \in R(A_1') \circ R(A_2')$. Then there exists x such that $(w,x) \in R(A_1')$ and $(x,v) \in R(A_2')$.

$(w,x) \in R(A'_1)$, therefore there exists a series of configurations $c^1_0,\ldots,c^1_r$ such that $c^1_0 = (q^1_0)^c$, $c^1_r \in F^c_1$ and for all k, $1 \leq k \leq r$, $c^1_{k-1} \vdash_{w_k:x_k,A'_1} c^1_k$ and $w = w_1 \ldots w_r$ and $x = x_1 \ldots x_r$. Define for all k, $0 \leq k \leq r$, $c^1_k = (q^1_k, u^1_k)$ (and indeed $c^1_0 = (q^1_0, u^1_0) = (q^1_0, \#^{n_1})$). By the definition of the product relation it follows that for all k, $1 \leq k \leq r$, there exist $\vec{a}_k \in \left(Actions^{\Gamma_1}_{n_1}\right)^+$ such that $(q^1_{k-1}, w_k, x_k, \langle\vec{a}_k\rangle, q^1_k) \in \delta'_1$.

$(x,v) \in R(A'_2)$, therefore there exists a series of configurations $c^2_0,\ldots,c^2_r$ such that $c^2_0 = (q^2_0)^c$, $c^2_r \in F^c_2$ and for all k, $1 \leq k \leq r$, $c^2_{k-1} \vdash_{x_k:v_k,A'_2} c^2_k$ and $x = x_1 \ldots x_r$ and $v = v_1 \ldots v_r$. Notice that it can be assumed that both of the accepting paths (for $w : x$ and $x : v$) have the same number of configurations, and moreover, it can be assumed that the word x is read in both of the accepting paths in the exact same way (as $x = x_1,\ldots,x_r$). These assumptions can be made as a result of the existence of the ϵ-self-loops in A'_1 and A'_2. Thus, if the two accepting paths are not of the same length, then simply pump the shorter one of the two with ϵ-self-loops to extend it to the length of the longer one. In the same way, by pumping the paths with ϵ-self-loops, it can be guaranteed that they both read the word x in the exact same way. In addition, we can assume that for all k, $1 \leq k \leq r$, if $x_k = \epsilon$ then either $w_k \neq \epsilon$ or $v_k \neq \epsilon$. By using this assumption, it is guaranteed that in the composition transducer there will be no ϵ-self-loops. Notice that the term ϵ-self-loops refers only to transition of the form (q,ϵ,ϵ,q) and not to transitions of the form $(q,\epsilon,\epsilon,\vec{a},q)$ (where $\vec{a}$ is a series of register operations that operate on registers other than 0). Define for all k, $0 \leq k \leq r$, $c^2_k = (q^2_k, u^2_k)$ (and indeed $c^2_0 = (q^2_0, u^2_0) = (q^2_0, \#^{n_2})$). By the definition of the product relation it follows that for all k, $1 \leq k \leq r$, there exists $\vec{b}_k \in \left(Actions^{\Gamma_2}_{n_2}\right)^+$ such that $(q^2_{k-1}, x_k, v_k, \langle\vec{b}_k\rangle, q^2_k) \in \delta_2$.

Define for all k, $0 \leq k \leq r$, $c_k = \left(\left(q^1_k, q^2_k\right), \langle u^1_k, u^2_k\rangle\right)$, a series of configurations of A where c_0 is the initial configuration of A and c_r is a final configuration of A. Since for all k, $1 \leq k \leq r$, $(q^1_{k-1}, w_k, x_k, \langle\vec{a}_k\rangle, q^1_k) \in \delta'_1$ and $(q^2_{k-1}, x_k, v_k, \langle\vec{b}_k\rangle, q^2_k) \in \delta'_2$, it follows from the definition of δ' that $\left(\left(q^1_{k-1}, q^2_{k-1}\right), w_k, v_k, \langle\vec{a}_k,\ \mathit{shift}_{n_1}(\vec{b}_k)\rangle, \left(q^1_k, q^2_k\right)\right) \in \delta'$ and by the above assumptions in δ. In addition, $\langle u^1_{k-1}, u^2_{k-1}\rangle \Vdash_{\langle\vec{a}_k, \mathit{shift}_{n_1}(\vec{b}_k)\rangle} \langle u^1_k, u^2_k\rangle$, and hence $c_{k-1} \vdash_{w_k:v_k,A} c_k$.

In sum, there exists a series of configuration of A, $c_0,\ldots,c_r$ such that for all k, $1 \leq k \leq r$, $c_{k-1} \vdash_{w_k:v_k,A} c_k$ and c_0 is the initial configuration of A and c_r is a final configuration of A. Hence $(w,v) \in R(A)$ and $R(A_1) \circ R(A_2) \subseteq R(A)$. □

5.3 Regular expressions denoting relations

In Section 4.2 the splice operation (generating languages) was introduced. We now extend it to a new regular relation operator, defining the interdigitation relation, and show how expressions using this operator are compiled into the appropriate transducers. The original splice operation was compiled into an automaton accepting all the legal combinations of the operands. The revised, relation-denoting splice operation is compiled into a transducer accepting all legal combinations and giving an analysis of each legal combination as pattern and root. For example, on the input $\{\langle r, \$, m\rangle\langle p, \&, l\rangle\langle p, q, d\rangle\}, \{\langle hit\Box a\Box e\Box\rangle\langle mi\Box\Box a\Box\rangle\langle ha\Box\Box a\Box a\rangle\}$ the operator yields an FSRT denoting the relation $\{hitra\$em : hit\Box a\Box e\Box + r\$m, hitpa\&el : hit\Box a\Box e\Box + p\&l, hitpaqed : hit\Box a\Box e\Box + pqd, mir\$am : mi\Box\Box a\Box + r\$m, mip\&al : mi\Box\Box a\Box + p\&l, mipqad : mi\Box\Box a\Box + pqd, har\$ama : ha\Box\Box a\Box a + r\$m, hap\&ala : ha\Box\Box a\Box a + p\&l, hapqada : ha\Box\Box a\Box a + pqd\}$.

Definition 5.3. *Let Σ be a finite set such that $\Box, \{, \}, \langle, \rangle, \oplus \notin \Sigma$. We define the splice relation operation to be of the form*

$$\{\langle \alpha_{1\ 1}, \alpha_{1\ 2}, ..., \alpha_{1\ n}\rangle\langle \alpha_{2\ 1}, \alpha_{2\ 2}, ..., \alpha_{2\ n}\rangle ... \langle \alpha_{m\ 1}, \alpha_{m\ 2}, ..., \alpha_{m\ n}\rangle\}$$

$$\oplus : \oplus$$

$$\{\langle \beta_{1\ 1}\Box\beta_{1\ 2}\Box ... \beta_{1\ n}\Box\beta_{1\ n+1}\rangle\langle \beta_{2\ 1}\Box\beta_{2\ 2}\Box ... \beta_{2\ n}\Box\beta_{2\ n+1}\rangle ... \langle \beta_{k\ 1}\Box\beta_{k\ 2}\Box ... \beta_{k\ n}\Box\beta_{k\ n+1}\rangle\}$$

where:

- $n \in \mathbb{N}$ *is the number of slots (represented by '$\Box$') in the patterns into which the roots letters should be inserted.*
- $m \in \mathbb{N}$ *is the number of roots to be inserted.*
- $k \in \mathbb{N}$ *is the number of patterns.*
- $\alpha_{ij} \in \Sigma^*$ *for* $1 \leq i \leq m$ *and* $1 \leq j \leq n$.
- $\beta_{ij} \in \Sigma^*$ *for* $1 \leq i \leq k$ *and* $1 \leq j \leq n + 1$.

The left set is a set of roots to be inserted into the slots in the right set of patterns. The slots are represented by the symbol '□'. Again, for the sake of brevity β_i and α_i are used as shorthand notations for $\beta_{i1}\Box\beta_{i2}\Box...\Box\beta_{i(n+1)}$ and $\alpha_{i1}\alpha_{i2}...\alpha_{in}$, respectively. As before, consider first the case where $\alpha_{ij} \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq m$ and $1 \leq j \leq n$ and $\beta_{ij} \in \Sigma \cup \{\epsilon\}$ for $1 \leq i \leq k$ and $1 \leq j \leq n+1$. In this case the splice relation operation gives as a result an FSRT-1 $A = \langle Q, q_0, \Sigma, \Sigma, \Gamma, 3, \delta, F\rangle$ such that $L(A) = \{\beta_{j1}\alpha_{i1}\beta_{j2}\alpha_{i2}...\beta_{jn}\alpha_{in}\beta_{j(n+1)} : \beta_j + \alpha_i \mid 1 \leq i \leq m\ ,\ 1 \leq j \leq k\}$, where:

- $Q = \{q_0, q_1, ..., q_{2n+1}\}$
- $F = \{q_{2n+1}\}$
- $\Sigma = (\{\alpha_{ij} |\ 1 \leq i \leq m\ ,\ 1 \leq j \leq n\}\ \cup\ \{\beta_{ij} |\ 1 \leq i \leq k\ ,\ 1 \leq j \leq n+1\}) \setminus \{\epsilon\}$
- $\Gamma = \{\beta_i |\ 1 \leq i \leq k\}\ \cup\ \{\alpha_i |\ 1 \leq i \leq m\}\ \cup\ \{\#\}$.
- $\delta = \{(q_0, \beta_{i1} : \beta_i+, W, 1, \beta_i, q_1) |\ 1 \leq i \leq k\}$

  $\cup$

  $\{(q_1, \alpha_{i1} : \alpha_{i1}, W, 2, \alpha_i, q_2) |\ 1 \leq i \leq m\}$

  $\cup$

  $\{(q_{2j-2}, \beta_{ij} : \epsilon, R, 1, \beta_i, q_{2j-1}) |\ 1 \leq i \leq k\ ,\ 2 \leq j \leq n+1\}$

  $\cup$

  $\{(q_{2j-1}, \alpha_{ij} : \alpha_{ij}, R, 2, \alpha_i, q_{2j}) |\ 1 \leq i \leq m\ ,\ 2 \leq j \leq n\}$

This transducer is depicted in Figure 5.9. It has 3 registers, where register 1 'remembers' the pattern and register 2 'remembers' the root. Notice that the transducer will have 3 registers and $2n + 2$ states for any number of roots, patterns and slots.

Consider now the general case where α_{ij} and β_{ij} can be strings of letters. In this case, arcs of the FSRT defined above are simply replaced by paths as was done for FSRAs.

Example 5.6. *Consider again the Hebrew roots r.$.m, p.&.l, p.q.d and the patterns hit□a□e□, mi□□a□, ha□□a□a. The operation*

$$\{\langle r, \$, m\rangle\langle p, \&, l\rangle\langle p, q, d\rangle\}\oplus : \oplus\{\langle hit\Box a\Box e\Box\rangle\langle mi\Box\Box a\Box\rangle\langle ha\Box\Box a\Box a\rangle\}$$

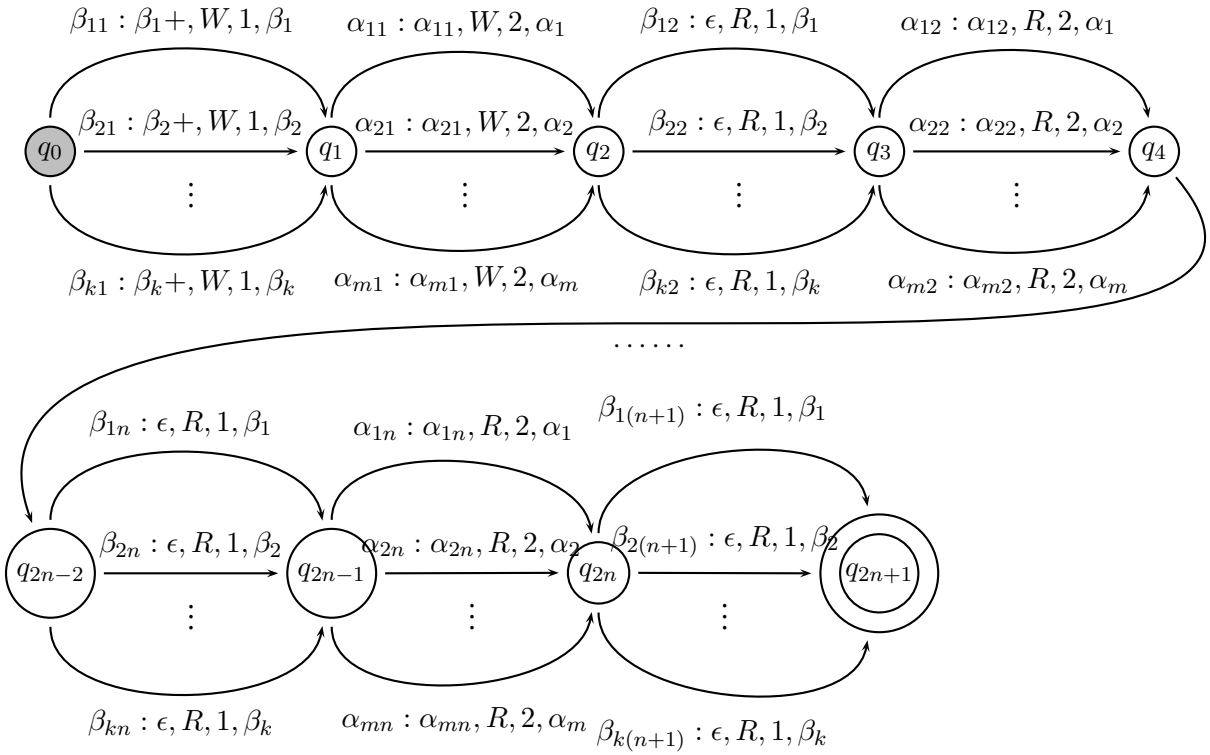


Figure 5.9: Interdigitation transducer – general

yields the FSRT depicted in Figure 5.10 (the ϵ-arc was only added for brevity). The relation denoted by the expression is $\{hitra\$em : hit\Box a\Box e\Box + r\$m, hitpa\&el : hit\Box a\Box e\Box + p\&l, hitpaqed : hit\Box a\Box e\Box + pqd, mir\$am : mi\Box\Box a\Box + r\$m, mip\&al : mi\Box\Box a\Box + p\&l, mipqad : mi\Box\Box a\Box + pqd, har\$ama : ha\Box\Box a\Box a + r\$m, hap\&ala : ha\Box\Box a\Box a + p\&l, hapqada : ha\Box\Box a\Box a + pqd\}$.

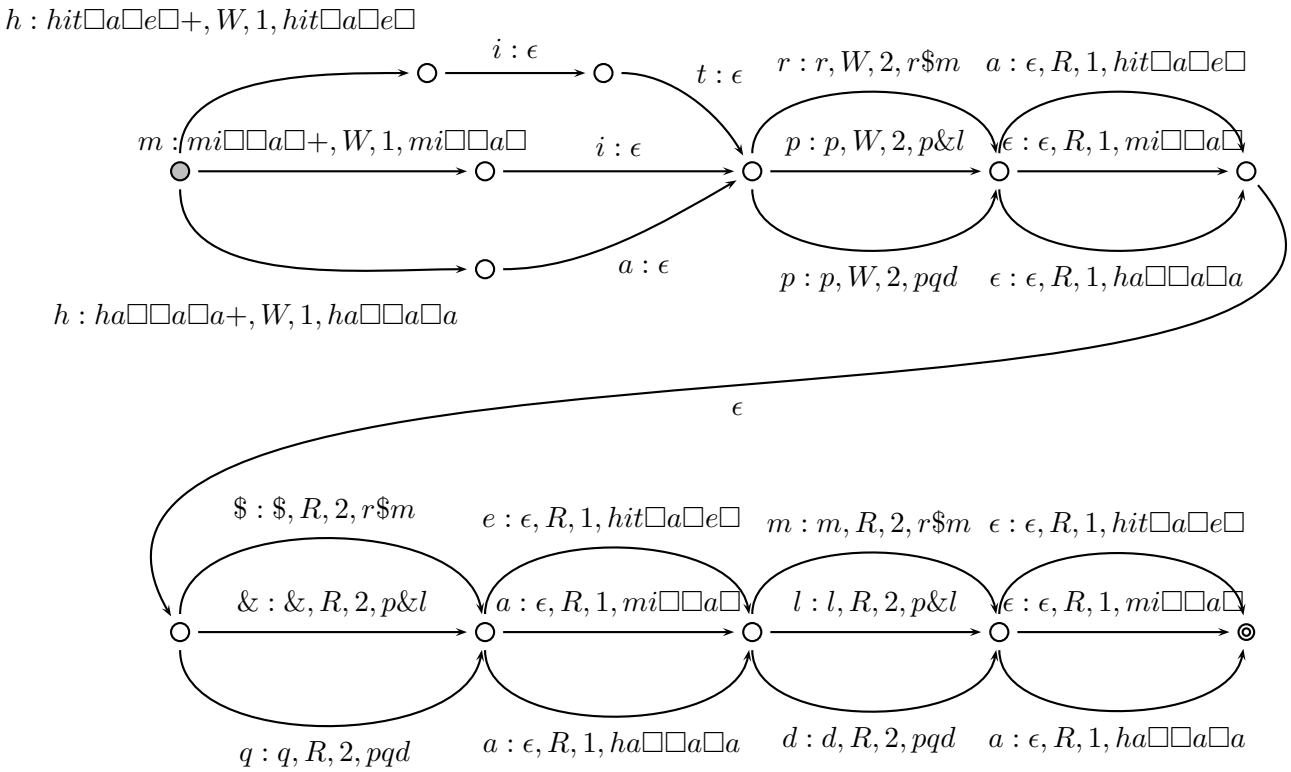


Figure 5.10: Interdigitation transducer – example

Chapter 6

Conclusions

In this work we introduce finite state registered networks (automata and transducers), an extension of finite state networks which adds a limited amount of memory, in the form of *registers*, to each transition. We show how FSRAs can be used to efficiently model several non-concatenative morphological phenomena, including circumfixation, root and pattern word formation in Semitic languages, limited reduplication etc.

The main advantage of finite state registered networks is their space efficiency. We show that every FSA can be simulated by an equivalent FSRA with three states and two registers. For the motivating linguistic examples, we show a significant decrease in the number of states *and* the number of transitions. For example, to account for all the possible combinations of r roots and p patterns, an ordinary FSA requires $O(r \times p)$ arcs whereas an FSRA requires only $O(r + p)$. As a non-linguistic example, we show a transducer which computes n-bit increment of binary numbers. While an ordinary (sequential) FST requires $O(2^n)$ states and arcs, an FSRT which guarantees linear recognition time requires only $O(n)$ states and arcs.

In spite of their efficiency, finite state registered networks are equivalent, in terms of their expressive power, to ordinary finite state networks. We provide an algorithm for converting FSRAs to FSAs and prove the equivalence of the models. Furthermore, we provide direct constructions of the main closure properties of FSAs for FSRAs, including concatenation, union, intersection and composition.

In order for finite state networks to be useful for linguistic processing, we provide a set of extended

regular expression operators which denote FSRAs and FSRTs. We show how such operators can be used to account for our motivating phenomena, including circumfixation and interdigitation. These dedicated operators can be used in conjunction with standard finite state calculii, thereby providing a complete set of tools for computational treatment of non-concatenative morphology.

This work opens a variety of directions for future research. An immediate question is the conversion of FSAs to FSRAs. While it is always possible to convert a given FSA to an FSRA (simply add one register which is never used), we believe that it is possible to automatically convert space inefficient FSAs to more compact FSRAs. A pre-requisite is a clear understanding of the parameters for minimization: these include the number of states, arcs and registers and the size of the register alphabet. For a given FSRA, the number of states can always be reduced to a constant (proposition 3.5); registers can be done with entirely (by converting the FSRA to an FSA, section 3.2) and similarly the size of the alphabet can be zero. In contrast, minimizing the number of arcs in an FSRA is NP-hard (section 3.7). A useful conversion of FSAs to FSRAs must minimize some combination of these parameters, and while it may be intractable in general, it can be practical in many special cases. In particular, the case of finite languages (acyclic FSAs) is both of practical importance and – we conjecture – can result in good compaction.

While FSRAs are equivalent to FSAs, there is no notion of *deterministic* FSRAs. Instead, we introduced the notion of *linearization*, which similarly guarantees linear recognition time. We cannot provide an efficient algorithm for converting a given FSRA into an equivalent linearized one, and we conjecture that this problem is NP-hard. More research is required in order to prove this conjecture.

More work is also needed in order to establish more properties of FSRTs. In particular, we did not address issues such as sequentiality or sequentiability for this model. Similarly, FSRA* can benefit from further research. All the closure constructions for FSRA*s can be done in a similar way to FSRAs, with the exception of intersection. For intersection, we believe that the use of predicates (van Noord and Gerdemann, 2001b) can be beneficial. In addition, the FSRA* model can be extended into transducers.

Finally, while this work is mainly theoretical, an implementation of FSRAs, along with the closure constructions suggested in this work, could lead to the inclusion of the extended regular expression

operators which we introduced in chapter 4 in an existing finite state calculus. This will result in a better calculus which could be used to computationally model morphological phenomena in all the world's languages, including those with non-concatenative processes.

Chapter 7

Bibliography

References

Beesley, Kenneth R. 1998. Constraining separated morphotactic dependencies in finite-state grammars. In *FSMNLP-98.*, pages 118–127, Bilkent, Turkey.

Beesley, Kenneth R. and Lauri Karttunen. 2000. Finite-state non-concatenative morphotactics. In *Proceedings of the fifth workshop of the ACL special interest group in computational phonology, SIGPHON-2000*, pages 1–12, Luxembourg, August.

Beesley, Kenneth R. and Lauri Karttunen. 2003. *Finite-State Morphology*. CSLI Publications.

Blank, Glenn D. 1985. A new kind of finite-state automaton: Register vector grammar. In *IJCAI-1985*, pages 749–755.

Blank, Glenn D. 1989. A finite and real-time processor for natural language. *Communications of the ACM*, 32(10):1174–1189.

Gerdemann, Dale and Gertjan van Noord. 1999. Transducers from rewrite rules with backreferences. In *Ninth Conference of the European Chapter of the Association for Computational Linguistics*, pages 126–133, Bergen Norway.

Gerdemann, Dale and Gertjan van Noord. 2000. Approximation and exactness in finite state optimality theory. In *Proceedings of Sigphon Workshop on Finite State Phonology (invited pa-*

per), pages 34–45, Luxembourg. http://xxx.lanl.gov/ps/cs.CL/0006038 or ROA-403-08100 at http://ruccs.rutgers.edu/roa.html.

Holzer, Markus and Martin Kutrib. 2002. State complexity of basic operations on nondeterministic finite automata. In *Implementation and Application of Automata (CIAA '02)*, pages 151–160.

Kaminski, Michael and Nissim Francez. 1994. Finite memory automata. *Theoretical Computer Science*, 134(2):329–364, November.

Kaplan, Ronald M. and Martin Kay. 1994. Regular models of phonological rule systems. *Computational Linguistics*, 20(3):331–378, September.

Karttunen, Lauri. 1997. The replace operator. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*, Language, Speech and Communication. MIT Press, Cambridge, MA, chapter 4, pages 117–147.

Karttunen, Lauri. 1998. The proper treatment of Optimality Theory in computational phonology. In *Finite-state methods in natural language processing*, pages 1–12, Ankara, June.

Karttunen, Lauri, Jean-Pierre Chanod, Gregory Grefenstette, and Anne Schiller. 1996. Regular expressions for language engineering. *Natural Language Engineering*, 2(4):305–328.

Kataja, Laura and Kimmo Koskenniemi. 1988. Finite-state description of Semitic morphology: A case study of Ancient Akkadian. In *COLING*, pages 313–315.

Kiraz, George Anton. 2000. Multitiered nonlinear morphology using multitape finite automata: a case study on Syriac and Arabic. *Computational Linguistics*, 26(1):77–105, March.

Kornai, András. 1996. Vectorized finite state automata. In *Proceedings of the workshop on extended finite state models of languages in the 12th European Conference on Artificial Intelligence*, pages 36–41, Budapest.

Koskenniemi, Kimmo. 1983. *Two-Level Morphology: a General Computational Model for Word-Form Recognition and Production*. The Department of General Linguistics, University of Helsinki.

Lavie, Alon, Alon Itai, Uzzi Ornan, and Mori Rimon. 1988. On the applicability of two-level morphology to the inflection of Hebrew verbs. In *Proceedings of the International Conference of the ALLC*, Jerusalem, Israel.

Mohri, Mehryar. 1996. On some applications of finite-state automata theory to natural language processing. *Natural Language Engineering*, 2(1):61–80.

Mohri, Mehryar, Fernando Pereira, and Michael Riley. 1996. Weighted automata in text and speech processing. In *Proceedings of the ECAI'96 Workshop on Extended Finite State Models of Language*, pages 46–50.

Mohri, Mehryar and Richard Sproat. 1996. An efficient compiler for weighted rewrite rules. In *Proceedings of the 34th Annual Meeting of the Association for Computational Linguistics*, pages 231–238, Santa Cruz.

Pereira, Fernando and Michael Riley. 1997. Speech recognition by composition of weighted finite automata. In Emmanuel Roche and Yves Schabes, editors, *Finite-State Language Processing*. MIT Press, Cambridge, pages 431–453.

Riley, Michael, Fernando Pereira, and Mehryar Mohri. 1997. Transducer composition for context-dependent network expansion. In *Proc. Eurospeech '97*, pages 1427–1430, Rhodes, Greece.

Shimron, Joseph. 2003. *Language processing and acquisition in languages of semitic, root-based, morphology*. John Benjamins Publishing, 2003, Philadelphia.

van Noord, Gertjan and Dale Gerdemann. 2001a. An extendible regular expression compiler for finite-state approaches in natural language processing. In O. Boldt and H. Jürgensen, editors, *Automata Implementation*, number 2214 in Lecture Notes in Computer Science. Springer.

van Noord, Gertjan and Dale Gerdemann. 2001b. Finite state transducers with predicates and identity. *Grammars*, 4(3):263–286.

Walther, Markus. 2000a. Finite-state reduplication in one-level prosodic morphology. In *Proceedings of the First Conference of the North American Chapter of the Association for Computational Linguistics, Seattle*, pages 296–302.

Walther, Markus. 2000b. Temiar reduplication in one-level prosodic morphology. In *Proceedings of SIGPHON, workshop on finite state phonology, August 2000, Luxembourg*, pages 13–21.