Engineering a Runtime System for AQL

Thèse présentée par
Andrea Morciano

En vue de l'obtention du grade de
Docteur en XXX
(Département XXX, Faculté XXX)

de l'Université Libre de Bruxelles
Bruxelles, Belgique

2016
(DRAFT—NOT FOR DISTRIBUTION 19 : 42 March 5, 2016)

To my parents.

Acknowledgments

Thank you, my advisor!

"A great citation here"

X

Abstract

abstract of the thesis

Contents

Acknowledgments **iv**

Abstract **vi**

1 Introduction **1**
1.1 Problem . . . 1
1.2 Objective . . . 1
1.3 Outline . . . 1
1.4 Contributions . . . 1
1.5 Publications . . . 1
1.6 Notations . . . 1

2 Information Extraction **2**
2.1 Definition . . . 2
2.2 Example Applications . . . 2
2.3 Issues . . . 3
2.4 Entity Extraction . . . 4
2.5 Relationship Extraction . . . 7

3 SystemT and AQL **9**
3.1 SystemT . . . 9
3.2 AQL . . . 14
3.3 A Formalism for AQL's Core: VStack and VSet Automata . . . 15

4 Conclusions **21**
4.1 Discussion . . . 21
4.2 Future Works . . . 21
4.3 Promising Directions . . . 21

Bibliography **22**

A Do we need an Appendix? **23**

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Problem

citation example [Knuth, 1981]

1.2 Objective

index example

1.3 Outline

blabla

1.4 Contributions

blabla

1.5 Publications

blabla

1.6 Notations

blabla

Chapter 2

Information Extraction

2.1 Definition

Information Extraction (IE) is the discipline that addresses the task of extracting structured information from unstructured sources automatically. Sources usually take the form of text documents. The most explored aspect of IE is the extraction of *named entities.* Another main topic, that has become object of research only in recent years, is the extraction of *relationships* between entities. IE is a field with contributions from many different communities: Machine Learning, Information Retrieval, Database, Web, Document Analysis. The interest in towards it is motivated by the evergrowing amount of data that is generated by our society, that is at a point in which manual analysis is infeasible. Thus IE promises to bring value to many application domains, most notably the enterprise world and the Web. There are two main approaches to IE: *rule-based* and *statistical.*

2.2 Example Applications

News Tracking The activity of tracking events in news articles. It can result in a lot of useful services, like automatic creation of multimedia news, linking articles to information pages on the entities found, etc.

Data Cleaning Extracting structured forms from flat data strings (containing, e.g., addresses). It allows more effective decuplication of information, among the other things.

Citation Databases Articles, conferences sites, individual research sites and similar are explored to obain formatted citations of publications, later stored in publicly accessible databases. They are capable to forward references and may provide aggregate statistics and scoring information. See for instance Google Scholar (`https://scholar.google.it/`).

Relationship Web Search Relationship extraction would be a very useful feature to integrate into web search engines, as the keyword search that they offer at present time is only good for entity identification.

2.3 Issues

2.3.1 Accuracy

We measure the accuracy of an IE task with two quantities:

- *precision*: the percentage of correctly extracted entities among the extracted entities;
- *recall*: the percentage of entities extracted among all the existing entities in the input source.

The main difficulties to achieving a good level of accuracy are:

- the availability of a great set of clues that might be very different in nature (e. g. orghographic properties, part of speech, typical words, etc.) and that might be difficult to combine;
- the difficulty of finding out missed extractions;
- the fact that with the advancement of research the extracted data structures keep increasing in complexity (for instance it is becoming more difficult to identify the boundaries of an entity in a text).

While we are able to reach a very good level of accuracy for entity extraction (90%), relationship extraction is still quite unreliable ($50 - 70\%$ accuracy), mainly due to its intrinsic complexity, superior to that of the task of entity extraction.

2.3.2 Running Time

In its initial phase IE was interesing only to research communities. With the enormous amount of data they need to process, companies are now concerned with IE too, and *scalability* has become a central issue, while many IE systems don't address the problem of efficiently carrying out extraction tasks with sufficient attention. Efficiency is influenced most notably by the efficiency of the following tasks:

- filtering documents in order to actually examine only the ones that have good chances to contain the desired information;
- focusing on the parts of a document that contain relevant information;

- processing steps, like database querying, that are typically very expensive and that might have to be performed on the selected pieces of input.

Recently many solutions have been proposed to target the scalability issue; one of these is indeed SystemT.

2.4 Entity Extraction

Named entities are elements of interest in a text. Example of entities are person names, street addresses, institutions, countries, and so on. In this section and in the next one, I assume that the output of an extraction task is a series of labels inserted into an input text document (that becomes *annotated*), although there are other possibilities.

2.4.1 Rule-based Methods

Rule-based methods employ sets of predicates that are "fired" independently. When a portion of input text satisfies a predicate, the predicate's associated action is executed. A predicate can be represented in the following generic form:

$$\text{"Contextual Pattern} \longrightarrow \text{Action"} \tag{2.1}$$

Contextual patterns are a way to identify entitities by exploiting their properties or the context in which they are usually met. The most common formalism used to express them is that of *regular expressions* over tokens of the input. Actions mark the entities that have been identified, and might consist in storing them in a database or adding delimiters directly in the text.
Most systems in this category present a cascaded structure: an input document goes through a series of processing phases, and the output of a phase is the input of the next one. A famous example is that of cascading grammars: formal grammars are evaluated in sequence on the input.

As I said, a contextual pattern seeks for (groups of) tokens that have certain features. In the case of entity extraction, features can be classified in the following categories:

- string representation;
- orthography (e.g. small case, mixed case, number, etc.);
- part of speech;
- list of dictionaries they're contained into;

- annotation obtained in previous phases.

Rules can be hand-coded by experts or learned through learning algorithms. In the second case the goal is to cover each of the entities of interest in the training set with at least one rule. The obtained rules should have good recall and precision on new input. Moreover, when learning a set of rules, we would like to achieve a good level of *generalizability*, that is: we would like to find the minimum set of rules accounting for the maximum portion of training data, with high precision. There are two main strategies for rule learning: *bottom-up* (start from very specific rules and make them more and more general) and *top-down* (start with rules covering all existing instances, then specialize them).

2.4.1.1 Example Rules

For the examples I use an idealized syntax, as in REF.

Example 2.1: *Consider the task of identifying all mentions ISO standards in a text. A rule for this purpose could be:*

$$(\{String = "ISO"\}\,\{String = "/IEC"\}\,\{?\}\,\{String = "/ASTM"\}\,\{?\}$$
$$\{Orthography\ type = number\}\,\{\{String = ":"\}\,\{Orthography\ type = number\}\}\,\{?\})$$
$$\longrightarrow ISO\ Standards \quad (2.2)$$

a

Example 2.2: *Multiple entities can be matched at once. Imagine we need to find mentions of (simple) street addresses in a text consisting of a street name and a street number. A rule that matches the name and number separatedly could be:*

$$(\{Orthography\ type = mixed\ case\ word\}\,\{*\}) : Name\,(\{String = ","\})$$
$$(\{Orthography\ type = number\}) : Number$$
$$\longrightarrow Street\ Name =: Name,\ Street\ Number =: Number \quad (2.3)$$

2.4.2 Statistical Methods

Statistical methods aim to decompose the source, assigning a label to each element in the decomposition. We distinguish between three types of statistical models:

- *Token-level models*: they assign a label to each token of the source. Since entities are usually comprised of multiple adjacent tokens, the tags used are of the forms "entity_begin", "entity_continue", "entity_end";
- *Segment-level models*: they try to find the best segmentation of the source text;
- *Grammar-based models:* they use formal grammars, outputting parse trees. All the valid parses are considered for an input document, assigning a score to each. The parse with the highest score is retained.

I now give a brief description of Token-level Models and Segment-level Models.

2.4.2.1 Token-Level Models

Features Clues are of the form

$$f : (x, y, i) \longmapsto \mathbb{R} \tag{2.4}$$

where x is a sequence of tokens, i is a position in x and y is a candidate label for the token at i. We distinguish between these types of features:

- *word features*;
- *orthographic features;*
- *dictionary lookup features.*

Models for Labeling Token The best models are the ones that take into account dependencies between tokens, among which we may find:

- *ordered classifiers*;
- *Hidden Markov models;*
- *Maximum Entropy Taggers;*
- *Conditional Markov Models;*
- *Conditional Random Fields* (the state of the art).

2.4.2.2 Segment-Level Models

Features In these models the label for a segment depends on the properties of its token and on the previous segment. We can describe a segment s_j as:

$$s_j = (y_i, l_j, u_j) \tag{2.5}$$

where y_i is the proposed label for s_j and l_j, u_j are the start and end positions of s_j. Therefore, a feature is of the form:

$$f\left(y_i, y_{i-1}, x, l_j, u_j\right) \tag{2.6}$$

where x is the input sequence of tokens, y_{i-1} is the proposed label for the previous segment and the other symbols are defined as for Equation 2.5. Besides token-level features (see subsection 2.4.2.1), we can exploit the following feature types:

- *Similarity to an entity in a database;*
- *Length of the segment.*

There are also global segmentation models, that try to find the best segmentation of a token sequence by maximizing a target function.

2.5 Relationship Extraction

When exctracting relationship between entities, we might face three types of specific tasks:

- given a pair of entities, find the relationship between them;
- given an entity e and a relationship r, find all the other entities e' such that $(e, e') \in r$;
- given a big and open-ended input and a relationship r find all pairs of entities e', e'' such that $(e', e'') \in r$.

2.5.1 Predicting the Relationship between a Entity Pair

For the first task, we can exploit the following resources:

- *surface tokens*: tokens that are usually placed between entities. They are strong clues;
- *part of speech tags* (the most important being verbs);

- *syntactic parse tree*: allows grouping words in phrase types, e.g. noun phrases, propositional phrases, and so on;
- *dependency graph*: it is a less expensive structure to compute than the parse tree and it links a word to those that depend on it.

The main methods available to carry out the task are:

- *Feature-based methods*, that simply transform the clues mentioned above for usage by conventional classifier models;
- *Kernel-based methods*, that use kernel functions to encode the similarity between two graphs;
- *Rule-based methods*, creating rules over structures around pairs of entities.

The second task is a special case of the third one so I don't treat it directly.

2.5.2 Finding All Possible Entity Pairs Belonging to a Relationship

The third task can be met especially when dealing with the Web. Usually we can exploit the following resources to fulfill it:

- the *types of arguments* of r (that might need specific recognition patterns);
- a *seed database* of pairs of entities belonging to r;
- *manually coded patterns*.

The generic procedure that is used in this case can be described with these steps:

1. Use the seed database to learn the relevant extraction patterns;
2. Use the obtained patterns to define candidate triples of the form (e', e'', r);
3. Retain a subset of the candidate triples, using a statistical test.

There exist also rule-based methods for the task.

Chapter 3

SystemT and AQL

In this chapter, I give a description of *SystemT* and its extraction rule language *AQL* (Annotation Query Language), and of a new way of evaluating AQL queries, that uses a formalism derived from *Finite State Automata*. Producing and analysing a concrete runtime system for AQL queries that employs this new method is the object of my thesis. Since SystemT is a rule-based system, I now focus only on this category.

3.1 SystemT

As I mentioned in the previous chapter, scalability has now become a central issue to IE. Companies now rely on IE for many tasks; a prominent example is *Business Intelligence*. Unfortunately, many systems developed in the past don't address this concern correctly. Traditionally, most rule-based system rely on cascading grammars: formal grammars are executed in sequence on the input, each grammar representing a stage that takes as input the ouput of the previous one. Rules in such grammars are matched using regular expressions: if a part of the input text satisfies the regular expression associated with a rule, that rule is activated (i.e. "fires") and the corresponding action is executed. Evaluating a grammar on a document tends to be costly, because simply evaluating a single rule might require scanning the whole document. On large datasets, the running time becomes enormous. What's more, these kind of systems are not able to fulfill the expressivity requirements of the complex extraction tasks.
SystemT was developed at IBM to address these issues. This system is based on a new approach to extraction rules: the *algebraic approach*. Here, data manipulations are viewed as operations in a (relational) algebra. Extraction tasks are defined using *annotators* (see Section 2.4) whose rules are viewed as *queries* on input documents, that act as virtual databases. Complex annotators are obtained by combining simple ones by means of *relational operators*. By doing so, all the optimization techniques used in regular databases

become available, but new techniques are possible too, due to the characteristics of text documents. Another advantage of SystemT is that it is capable to handle *overlapping rule matchings*, due to rulels being fired independently, which are difficult to resolve in cascading grammars, where they tipically require disambiguation policies that only partially solve the problem.
I now give an overview of SystemT's high-level structure. Then, I talk about its data model, execution model and algebra.

3.1.1 Architecture

These are the main components of SystemT:

Development Environment: allows constructing annotators for extraction tasks. Rules are expressed in AQL, and compiled into an algebra. It supports an iterative development process.

Optimizer: seeks for the best query plan for an extraction task, evaluating the most convenient optimization techniques in a cost-based manner.

Runtime Environment: given a query plan, it instantiates the physical operators corresponding to the logical ones in the plan, then it proceeds evaluating it on input documents.

Once the development of an annotator is complete, it is *published* to the optimizer and runtime. After optimization, the runtime evaluates it on a continuous stream of documents.

3.1.2 Data Model

SystemT uses an *object-relational* data model for annotations on a text document, that allows applying logical operators over it, that can be composed. There is an important assumption to mention before continuing: an extraction task over a set of documents is performed one document at a time. This means that any relationships between entities in different documents are disregarded. This assumption is crucial to some optimization techniques that SystemT uses. While there exist tasks where considering these relationships would be useful (think of the Web), still a large number of relevant tasks can be carried out this way. In the following, I define the basic data types of SystemT's data model.
I consider a finite alphabet Σ of symbols (characters). $\sum^*$ denotes the set of all strings of finite length over $\sum^*$. A document is modeled as one such string.

Definition 3.1: *A document is a string* $s \in \sum^*$.

The most basic data type is the *span*.

Algorithm 3.1 Annotating all local databases in a global database (taken from REF).

$E \longleftarrow$ {algebra expression}
for localDB in globalDB do
begin

1. {Read localDB into main memory}
2. $R \longleftarrow E$ (localDB)
3. {Add R to localDB}
4. {Write modified localDB to disk}

Definition 3.2: *Given a string $s = \sigma_1 ... \sigma_n$ where $\forall i$, $\sigma_i \in \Sigma$, with length$|s| = n$ and whose characters are indexed in the natural way, a span is a substring $[i, j\rangle$, where i, j are indices of s satisfying $1 \leq i \leq j \leq n + 1$. A span of s is also denoted as $s_{[i,j\rangle}$.*

Spans can be aggregated in *tuples*, which are finite sequences of spans. Let us denote the set of all possible spans of a string s by Spans(s) and by SVars an infinite set of span variables, which can be assigned spans. A $(V, s) -$ tuple is defined as follows:

Definition 3.3: *Given a set $V \subseteq$ SVars, V being finite, and a string $s \in \sum^*$, a $(V, s) -$ tuple is a mapping $\mu : V \longmapsto$ Spans(s). When V is clear from the context, we might call μ simply a $s -$ tuple.*

A set of tuples is called a *relation*. Here, I focus on *span relations*, which I formally define as $(V, s) -$ relations.

Definition 3.4: *A $(V, s) -$ relation is a set of $(V, s) -$ tuples. As before, we can speak of $s -$ relation when V is clear.*

Each operator in the algebra takes one or more span relations as input and outputs a single span relation.

3.1.3 Execution Model

A single document is conceived as a *local annotation database*, to which annotators are applied in order to build *views*. In general, a local database fits into main memory. Local databases are contained in a *global annotation database*. The runtime of SystemT takes a global database and it annotates its local databases. The procedure is described by Algorithm 3.1.
In this procedure, step 2 in the for loop is the most expensive.

3.1.4 Algebra of Operators

The operators of SystemT's algebra can be classified in three groups:

- *relational operators;*
- *span extraction operators;*
- *span aggregation operators.*

In addition, there exist some *span selection predicates* that are used for span selection.

3.1.4.1 Relational Operators

Relational operators are the usual operators of relational algebra that appear in classical database query plans. Important examples are:

- *selection* (σ);
- *projection* (π);
- *Cartesian product* ($\times$);
- *Union* ($\cup$);
- *Intersection* ($\cap$).

Span Extraction Operators

Loosely speaking, span extraction operators take a pattern and a document as input and output the maximal set of spans that match that pattern. There are two main span extraction operators:

standard regular expression matcher (ε_{re}): this operator takes a regular expression r as input and it identifies all non-overlapping matchings of r in the current document, from left to right.

dictionary matcher (ε_d): this operator outputs all the spans that match some entry in a given dictionary.

Although the dictionary matcher might seem useless since there is a regular expression matcher, it has some advantages over it, as the fact that it can find overlapping matchings or that it enforces the semantics of word boundaries.

Span Aggregation Operators

Span aggregation operators are used to aggregate spans in a meaningful way. They are of two types:

- *Consolidation*: they are used to coalesce overlapping spans that were matched using patterns for the same concept. They are:

  containment consolidation (Ω_c): it discards a span if it is fully contained into another one.

  overlap consolidation (Ω_o): it merges spans that overlap repeatedly.

- *Block (β)*: it matches a series of spans, each at a distance from its neighbor span(s) that is not superior to a threshold. It is thought to identify regularity. It is tuned by two parameters: a *distance constraint* to control regularity and a *count constraint* that establishes the minimum number of spans in a block.

3.1.4.2 Span Selection Predicates

Consider two spans s_1, s_2. The main span predicates that may be used for selection are the following:

$s_1 \preceq_d s_2$ when s_1, s_2 do not overlap, s_1 precedes s_2 and there are less than d characters between them;

$s_1 \simeq s_2$ when the two spans overlap;

$s_1 \subset s_2$ when s_1 is strictly contained in s_2;

$s_1 = s_2$ when s_1 equals s_2.

3.1.5 Optimization Techniques

As I mentioned in the introduction of this chapter, SystemT can make use of the optimization techniques from traditional database systems, but there exist some peculiar aspects of SystemT and span extraction tasks that enable new optimization methods.

Remark 3.1: *The effect of document-at-a-time processing is that the span relations produced and consumed for a single document by operators are very small in size and often empty.*

Remark 3.2: *Evaluating an annotator on a large set of document is a* CPU-intensive process. *This is because the running time is by far dominated by the execution of the operators ε_{re} and ε_d, that are applied to each document in an input set.*

Remark 3.3: *Spans are nothing but* intervals*, so we can exploit* interval algebra*.*

With Remark 3.1 and Remark 3.3 in mind, running time can be reduced in a number of ways. I present them briefly.

Regular Expression Strength Reduction Some classes of regular expressions, as defined in the POSIX standard, are executed more efficiently by using *specialized engines* that are able to improve performance. For example, an expression that looks for a finite number of strings in a text is evaluated more efficiently by a string-matching engine.

Shared Dictionary Matching (SDM) Dictionary lookups are usually very expensive, as we need to consult the dictionary thousand of times in a typical setting. Instead of evaluating each ε_d operator in an extraction task independently for each input document, it is evaluated once and for all at the beginning of the process, and the obtained matches are shared among the single documents.

Conditional Evaluation (CE) As we know, an annotator is evaluated indipendently on each input document. Thus, by employing some heuristic, it can be guessed if a (sub)query will give any matches in a document without loss of generality, and in case it does not, it is not evaluated on it.

Restricted Span Extraction (RSE) It consists in executing expensive operations on selected regions of a document. It is used for *join* operators involving dictionary matching operators and/or a regular expression matching operators in their argments. One input operator to a join is evaluated on the whole document, while the other operators are modified to consider only some neighborhood of the results from the first one. These neighborhoods are established by some ad hoc heuristic.

The techniques described allow the use of a *cost-based plan optimizer*: in a query plan, the subgraphs suitable for optimization are identified, and all the possible optimized plans for them are formulated. In the end, the best ones are retained. Results of sample experiments on a large dataset are decribed in REF and REF.

3.2 AQL

AQL is the concrete language used by SystemT to express annotators, designed to support the execution model I described in Subsection 3.1.3. It is a declarative language with a syntax very similar to that of SQL. It supports all the operators that I presented in Subsection 3.1.4. When coding an extraction task in AQL, we are able to build a series of

views of a document, that are of progressively higher level, and can be based on lower-level views. The content of a view corresponds with annotations in the text. An input document is modeled as a view too, which is provided by default. The main advantages of AQL are:

- it allows formulating *complex low-level* patterns in a declarative fashion;
- it enables *modularization* and *reuse* of the queries, making development and maintance of *complex high-level structures* easier.

3.3 A Formalism for AQL's Core: VStack and VSet Automata

AUTHORS describe in REF a formal model that captures the core functionality of AQL. They show a way to represent annotators expressed in AQL by means of modified finite state automata, namely *VStack* and *VSet automata*. In my work, I define and implement a runtime system based on this new formalism. I now describe the model and discuss the relative expressive power of its main elements. First of all, we need to be more precise on some basic concepts.

3.3.1 Basic Definitions

I consider Σ and Σ^*, defined as in Subsection 3.1.2. Let us start with the definition of a language.

Definition 3.5: *A language L is a subset of Σ^*.*

Now we can state formally define regular expressions, by giving their language.

Definition 3.6: *Regular expressions over Σ are those that belong to the language γ defined as:*

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \tag{3.1}$$

where:

- *$\emptyset$ is the empty language;*
- *ϵ is the empty string;*
- *$\sigma \in \Sigma$;*
- *$\vee$ is the ordinary disjunction operator;*
- *$\cdot$ is the concatenation operator;*

- $*$ *is the Kleene Closure.*

Additionally, we might use γ^+ as a shortcut for $\gamma \cdot \gamma^*$, and Σ as an abbreviation of $\sigma_1 \vee ... \vee \sigma_n$ (with a slight abuse of notation). We also need the definition of a *string relation.*

Definition 3.7: *A n-ary string relation is a subset of* $(\Sigma^*)^n$.

I have already defined span relations (Definition 3.4). They allow us to give a precise definition of annotators, by introducing the concept of *document spanner.*

Definition 3.8: *Given a string* s*, a document spanner* P *is a function that maps* s *to a* $(V, s) -$ *relation* r *, where* $V := Svars(P)$*. We have* $r = P(s)$*. We say* P *is n-ary if* $|V| = n$.

I now present two types of spanners that will be useful later on. The first one is that of *Hierarchical Spanners.* First, let us see when a $s -$ tuple is hierarchical.

Definition 3.9: *Given a string* s*, a document spanner* P *and an* $s -$ *tuple* $\mu \in P(s)$*,* μ*is hiearchical if, for every* $x, y \in SVars(P)$*, one of the following conditions holds:*

- $\mu(x) \supseteq \mu(y)$;
- $\mu(x) \subseteq \mu(y)$;
- $\mu(x)$ *is disjoint from* $\mu(y)$.

The definition of *hierarchical spanner* follows.

Definition 3.10: *Given a document spanner* P*,* P *is hierarchical if* $\forall s \in \Sigma^*$*,* $\forall \mu \in P(s)$*,* μ *is hierarchical.*

The class of hierarchical spanners is denoted by **HS**. As it emerges from Definition 3.9 and Definition 3.10, a hierarchical spanner outputs only $s -$ tuples whose spans don't overlap. The second type of spanners I introduce is that of *Universal Spanners.* Some accessory definitions are needed.

Definition 3.11: *Given a string* s *and a document spanner* P*,* P *is total on* s *if* $P(s)$ *consists of all the possible* $s -$ *tuples on* $SVars(P)$.

Definition 3.12: *Given a string* s *and a document spanner* P*,* P *is hierarchically total on* s *if* $P(s)$ *consists of all the possible hierarchical* $s -$ *tuples on* $SVars(P)$.

There are two kinds of universal spanners: the *universal spanner* and the *universal hierarchical spanner.*

Definition 3.13: *Given a set of span variables $Y \subseteq SVars$, the universal spanner Υ_Y over Y is the unique document spanner P such that $SVars(P) = Y$ and P is total on every string $s \in \Sigma^*$.*

Definition 3.14: *Given a set of span variables $Y \subseteq SVars$, the universal hierarchical spanner Υ_Y^H over Y is the unique document spanner P such that $SVars(P) = Y$ and P is hierarchically total on every string $s \in \Sigma^*$.*

I now present the formal spanner represantations described in REF.

3.3.2 Spanner Representations

We saw in Subsection 3.1.4 that the operations used by SystemT and AQL to extract spans relations are regular expression matching and dictionary matching. Here I focus on regular expressions. AQL's regular expressions can be seen as usual regular expressions enriched with *capture variables*, that are nothing but the span variables that constitute the span relations I defined. In what follows. In REF, three formalisms are described that are able to model this kind of modified regular expressions, namely:

- *Regex Formulas;*
- *Variable Stack Automata;*
- *Variable Set Automata.*

3.3.2.1 Regex Formulas

To define regex formula, I first introduce the concept of *variable regex.*

Definition 3.15: *A variable regex is a regular expression with capture variables that extends usual regular expressions in the following way:*

$$\gamma := \emptyset \mid \epsilon \mid \sigma \mid \gamma \vee \gamma \mid \gamma \cdot \gamma \mid \gamma^* \mid x\{\gamma\} \tag{3.2}$$

where:

- $x \in SVars$*;*
- $x\{\gamma\}$ *means that we assign the result of the evaluation of* γ *to* x*.*

Evaluating a variable regex on a string produces a parse tree over the alphabet $\Lambda = \Sigma \cup \text{SVars} \cup \{\in, \vee, \cdot, *\}$. A valid parse tree for a variable regexγ is called a $\gamma -$ parse. We accept only those variable regex expressions whose parses have only one occurence

of each of their variables, otherwise the variable assignment would remain unclear. Such expressions are referred to as *functional*.

Definition 3.16: *A variable regex γ is functional if $\forall s \in \Sigma^*$, $\forall \gamma - $ parse t for s, $\forall x \in SVars(\gamma)$, x occurs exactly one time in t.*

Definition 3.17: *A regex formula is a functional variable regex.*

The spanner represented by a regex formula γ may be denoted as $[\![\gamma]\!]$. We have that $\mathrm{SVars}([\![\gamma]\!]) = \mathrm{SVars}(\gamma)$, and the span relation $[\![\gamma]\!](s)$ is the set $\{\mu^p \mid p \text{ is a } \gamma - \text{parse for } s\}$, where μ^p is a tuple defined by a parse p.

3.3.2.2 Variable Stack Automata

Variable stack automata (VStack Automata or vstk-automata for short) are representations of document spanners by means of modified finite state automata (FAs). Basically, a FA is augmented with a *stack of span variables*. A variable is pushed on the stack when its corresponding span is opened, and popped when it is closed. The formal definition of a vstk-automaton follows.

Definition 3.18: *A Variable Stack Automaton is a tuple (Q, q_0, q_f, δ), where:*

- *Q is a finite set of states;*
- *$q_0 \in Q$ is the initial state;*
- *$q_f \in Q$ is the accepting state;*
- *δ is a finite transition relation, containing triples of the forms (q, σ, q'), (q, ϵ, q'), $(q, x\vdash, q')$, $(q, \dashv, q')$, where:*
  - *$q, q' \in Q$;*
  - *$\sigma \in \Sigma$;*
  - *$x \in SVars$;*
  - *$\vdash$ is the push symbol;*
  - *$\dashv$ is the pop symbol.*

Notice that we don't need to specify which variable we want to pop, as it is naturally the last that was pushed on the stack. Given a vstk-automaton A, the set of variables that appear in its transitions is denoted as $\mathrm{SVars}(A)$. Next I give the definitions of a *configuration* and of a *run* of a vstk-automaton.

Definition 3.19: *Given a string s with length $|s| = n$ and a vstk-automaton A , a configuration of A is a tuple $c = (q, \overrightarrow{v}, Y, i)$, where:*

- $q \in Q$ *is the current state;*
- $\overrightarrow{v}$ *is the current variable stack;*
- $Y \in Vars(A)$ *is the set of available variables (those not already pushed on the stack);*
- $i \in \{1, ..., n+1\}$ *is the position of the next character to be read in s.*

Here, as for regex formulas, we want the variable assignment to be clear, so once a variable is pushed on the stack, it is removed from the set of available variables, thus it can be pushed only once. For the rest, a run of a vstk-automaton is similar to those of ordinary FAs.

Definition 3.20: *Given a string s and a vstk-automaton A, a run ρ of A on s is a sequence of configurations $c_0, ..., c_m$ such that:*

- $c_0 = (q_0, \epsilon, SVars(A), 1)$;
- $\forall j \in \{0, ..., m-1\}$, *for* $c_j = (q_j, \overrightarrow{v_j}, Y_j, i_j)$, $c_{j+1} = (q_{j+1}, \overrightarrow{v_{j+1}}, Y_{j+1}, i_{j+1})$ *one of the following holds:*
  - $\overrightarrow{v_{j+1}} = \overrightarrow{v_j}$, $Y_{j+1} = Y_j$ *and either:*
    - $i_{j+1} = i_j + 1$, $(q_j, s_{i_j}, q_{j+1}) \in \delta$;
    - $i_{j+1} = i_j$, $(q_j, \in, q_{j+1}) \in \delta$.
  - $i_{j+1} = i_j$ *and for some* $x \in SVars(A)$ *either:*
    - $\overrightarrow{v_{j+1}} = \overrightarrow{v_j} \cdot x$, $x \in Y_j$, $Y_{j+1} = Y_j \setminus \{x\}$, $(q_j, x\vdash, q_{j+1}) \in \delta$ *(x is pushed on the stack);*
    - $\overrightarrow{v_j} = \overrightarrow{v_{j+1}} \cdot x$, $x \in Y_j$, $Y_{j+1} = Y_j$, $(q_j, \dashv, q_{j+1}) \in \delta$ *(a variable is popped from the stack).*

Definition 3.21: *Given a string with length $|s| = n$ and a vstk-automaton A, a run $\rho = c_0, ..., c_m$ of A on s is accepting if $c_m = (q_f, \epsilon, \emptyset, n+1)$.*

The set of all possible accepting runs of a vstk-automaton A on a string s is denoted as $\mathrm{ARuns}(A, s)$. The spanner represented by A may be referred to as $[\![A]\!]$. We have that $\mathrm{SVars}([\![A]\!]) = \mathrm{SVars}(A)$, and the span relation $[\![A]\!](s)$ is the set $\{\mu^\rho \mid \rho \in \mathrm{ARuns}(A, s)\}$, where μ^ρ is a tuple defined by a run ρ. In particular, for every variable $x \in \mathrm{SVars}(A)$, $\mu^\rho(x)$ is the span $[i_b, i_e\rangle$, where:

- c_b=$(q_b, \overrightarrow{v_b}, Y_b, i_b)$ is the unique configuration of ρ where x appears in the stack for the fist time;
- c_e=$(q_e, \overrightarrow{v_e}, Y_e, i_e)$ is the unique configuration of ρ where x appears in the stack for the last time.

3.3.2.3 Variable Set Automata

Chapter 4

Conclusions

4.1 Discussion

blabla

4.2 Future Works

blabla

4.3 Promising Directions

blabla

Bibliography

[Knuth, 1981] Knuth, D. E. (1981). *Seminumerical Algorithms*, volume 2 of *The Art of Computer Programming*. Addison-Wesley, Reading, Massachusetts, second edition. This is a full BOOK entry.

Appendix A

Do we need an Appendix?

maybe...