Certain Query Answering on Hyperstreams

Momar Sakho

Certain Query Answering on Hyperstreams

PhD Thesis

to obtain the title of PhD of Science of the doctoral school Sciences pour l'Ingénieur at Université de Lille

Specialty : Computer Science

defended on July 24, 2020 by

Momar Ndiouga Sakho

Thesis advisors: Joachim Niehren and Iovka Boneva

Committee :

Reviewers :	Sebastian Maneth	-	Universität Bremen
	Sylvain Schmitz	-	Université de Paris
Advisor :	Joachim Niehren	-	Inria Lille
Co-advisor :	Iovka Boneva	-	Université de Lille
President :	Mathieu Giraud	-	CNRS, Université de Lille
Examiner :	Olivier Gauwin	-	Université de Bordeaux

Requêtes Logiques sur les Hyperflux

Thèse

pour l'obtention du grade de Docteur de l'École Doctorale Sciences pour l'Ingénieur à l'Université de Lille

Spécialité : Informatique

soutenue le 24 juillet 2020 par

Momar Ndiouga Sakho

Direction de thèse: Joachim Niehren et
Iovka Boneva

Jury :

Rapporteurs :	Sebastian Maneth	-	Universität Bremen
	Sylvain Schmitz	-	Université de Paris
Directeur de thèse :	Joachim Niehren	-	Inria Lille
Co-encadrante :	Iovka Boneva	-	Université de Lille
Président :	Mathieu Giraud	-	CNRS, Université de Lille
Examinateur :	Olivier Gauwin	-	Université de Bordeaux

Abstract: Hyperstreams are collections of streams with references. The hope is that structured data on a hyperstream can be monitored with lower latency than when communicated on a stream, since data on hyperstreams may be received in parallel rather than in a purely sequential manner. In order to show this, however, it is necessary to develop algorithms for answering logical queries on hyperstreams, given that the existing algorithms are restricted to streams. Therefore, we study the question of whether certain query answering (CQA) on hyperstreams is feasible in theory and practice.

We study the complexity of CQA on hyperstreams. We first show that CQA is closely related to the problems of regular pattern matching and inclusion, and thus to the problem of transition inhabitation for automata for words and trees. This permits us to classify the complexity of CQA for various classes of automata and hyperstreams. We obtain polynomial time results for linear hyperstreams without compression and queries defined by deterministic stepwise hedge automata. For the general case, the complexity goes up to ExpTime.

We then develop an efficient approximation algorithm for CQA on hyperstreams. This algorithm has large coverage in that it applies to arbitrary hyperstreams and classes of query automata, and runs in polynomial time. However, it may not always detect the certain query answers with lowest latency.

The third contribution is an algorithm for CQA on streams that runs in combined linear time in the case of boolean queries. This algorithm is efficient in practice also in the monadic case, in contrast to all previous proposals. We show this experimentally by applying the algorithm to the navigational queries of the usual XPath benchmark, on which all previous approximation-free approaches to CQA failed. Deterministic stepwise hedge automata enable this algorithm.

Keywords: Certain Query Answering, Hyperstreaming, Automata, Complexity, Pattern Matching

Résumé: Les hyperflux sont des collections de flux de données avec références. Sachant qu'ils permettent la réception de données en parallèle plutôt que de manière purement séquentielle, l'on peut espérer que des données structurées transmises par leur biais puissent être traitées avec une latence moindre qu'avec un flux. Nous proposons ainsi de développer des algorithmes d'évaluation de requêtes logiques sur les hyperflux de données. Pour ce faire, nous nous intéressons à la faisabilité théorique et pratique d'un algorithme de décision du problème de certitude d'une réponse (CQA) sur les hyperflux.

Nous étudions la complexité du CQA sur les hyperflux. Nous montrons d'abord que le CQA est intimement lié à des problèmes de correspondance de motifs dans les langages réguliers, et donc aux problèmes d'habitation des transitions d'automates de mots et d'arbres. Cela nous permet d'établir une classification de la complexité du CQA pour des classes d'automates et d'hyperflux variées. Nous obtenons des résultats en temps polynomial pour les hyperflux linéaires sans compression et les requêtes définies par des automates déterministes pour forêts. Pour le cas général, la complexité atteint ExpTime.

Nous développons ensuite un algorithme efficace pour l'approximation du CQA sur les hyperflux. Cet algorithme a une grande couverture, du fait qu'il s'applique à des hyperflux et des classes d'automates arbitraires, et s'exécute en temps polynomial. Cependant, il ne détecte pas toujours les réponses certaines avec une latence minimale.

La troisième contribution est un algorithme pour le CQA sur les flux, qui s'exécute en temps polynomial dans le cas de requêtes booléennes. Cet algorithme est aussi efficace en pratique pour le cas monadique, à l'inverse de toutes les précédentes propositions. Nous le montrons expérimentalement en appliquant l'algorithme aux requêtes navigationnelles XPath habituellement prises pour référence, avec lesquelles toutes les approches précédentes de CQA sans approximation ont échoué. L'utilisation d'automates spéciaux pour les forêts a permis la mise en place dudit algorithme.

Mots-clés: Certain Query Answering, Hyperstreaming, Automates, Complexité, Correspondance de motifs

Acknowledgements

The writing and defense of this thesis happened at a time when the Covid-19 pandemic was affecting the life of almost everybody in the world. Sebastian Maneth and Sylvain Schmitz accepted to review this document in conditions that were far from being ideal, and for this reason I would like to thank them from the bottom of my heart. I am also grateful to Mathieu Giraud for having accepted to be part of the jury committee, and to Olivier Gauwin, whose work on stream processing paved the way for me.

During my thesis, I was very lucky and had the great honor to be supervised by Iovka Boneva and Joachim Niehren. Thank you Iovka for your kindness, patience, rigor and the help you gave me. You always kept your good mood, even after having read my proofs :) Thank you Joachim, for trusting me and giving me the opportunity to work with you. You have been a real mentor and friend to me during this adventure, and none of this would have been accomplished without your availability and support.

Many thanks to all the members of the Links team, with whom I spent great moments and also learned a lot. Special thanks to Sylvain and his family who kindly made their garden available right after my defense. I'll highlight my fellow PhD students Tom, Lily, Paul, Jose, Nicolas and Antonio, with whom I had lots of laughters. Antonio has also been very helpful for the experiments.

Last, but not least, I would like to thank all the people that are very special to me. All my friends that I met in Lille, Lyon, Dakar, Louga and everywhere else. My two fathers, for their advice and support. My mother, who has made so many sacrifices that allowed me to be the person I am today. My brothers and sisters, for simply being who they are. My wife Borso, who lived all this adventure with me, and whose presence and love help me greatly to overcome the difficulties of life.

Funding Acknowledgements

Special acknowledgements to the ANR Agreg project and the Region Hauts-de-France that funded my Phd project. I am also grateful to the Inria research center of Lille, where I had an excellent working environment.

Contents

Some Notations

Σ	Alphabet
Σ^*	Set of words over alphabet Σ
w	Nested word
$\widetilde{w}$	Position-annotated nested word
$nWords_\Sigma$	Set of nested words over Σ
$nPatterns_\Sigma$	Set of nested patterns over Σ
$nLinPatterns_\Sigma$	Set of linear nested patterns over Σ
s	Stream
$\widetilde{s}$	Position-annotated stream
G	Hyperstream
$nStreams_\Sigma$	Set of streams of nested words over Σ
$\mathcal{T}_\Sigma$	Set of ranked trees over some ranked alphabet Σ
$\mathcal{H}_\Sigma$	Set of hedges over some alphabet Σ
$\mathcal{U}_\Sigma$	Set of unranked trees over Σ
Hyp_Σ	Set of nested hyperstreams over Σ
$LinHyp_\Sigma$	Set of linear nested hyperstreams over Σ
ε	Empty word
$\mathbf{Q}$	Query
$\mathcal{V}$	Set of pattern variables
$\mathcal{W}$	Set of query variables
Q	Set of states of automata
I	Set of initial states
Q_h	Set of hedge states
Q_t	Set of tree states
F	Set of final states
Δ	Transition relation
α	Candidate
$[]$	Empty candidate
$\mathcal{C}_{\mathcal{W}'}(\rho)$	Set of candidates of the pattern ρ with query variables in $\mathcal{W}'$
μ	Assignment of pattern variables

Chapter 1

Introduction

Contents

1.1 Communication over Streams

Interaction is essential for humans, computers, and intelligent systems. A typical form of interaction is the continuous communication of data and knowledge over streams. In social networks, for instance, human users produce and send data and knowledge over Twitter streams. Or else, mediator systems for newspapers may filter and send articles in topics of interest to a stream in a navigator. We are particularly interested in communication of semi-structured data. Stream processing systems must then detect complex events on the streams efficiently and react to them with low latency. Furthermore, they may read the events on the stream only once, and

$$(1, MSFT, 1024, 1111)(1, APPL, 562, 666)(2, GOGL, 1234, 1244)\ldots$$

Figure 1.1: Stream of stock quotes

```
<measure><timestamp>42.42</timestamp>39</measure>
<measure><timestamp>42.51</timestamp>40.5</measure>
<measure><timestamp>42.63</timestamp>41</measure>
...
```

Figure 1.2: Stream of semi-structured data

may buffer only a fraction of them at any time point, depending on the size of the available memory of the machine.

1.1.1 Complex Event Processing

Complex event processing (CEP) [Mozafari *et al.* 2012, Grez *et al.* 2019] requires to monitor semi-structured data streams for *complex events* that can be defined by some logical query. Most typically, the data on the stream may be produced by sensors measuring physical values such as temperature or humidity.

We consider semi-structured data streams as *incomplete semi-structured databases* with an open end, that may be instantiated incrementally. In this manner, logical queries as for semi-structured databases can be used to define complex events. An incomplete semi-structured database may be simply an open list of text tuples, to which new text tuples may be added at any time. The example in Figure 1.1 which is adapted from [Hallé 2017] shows a semi-structured data stream originating from stock markets. The elements of the tuples are respectively – from the first to the last – a timestamp, stock symbol, minimum price and closing price of a stock quote.

Semi-structured data can also be formatted in more standard formats like XML or JSON. For instance, a CEP system may be asked to detect overheating in a room using the XML stream of Figure 1.2, in which are written the temperatures measured in that room. Overheating is here a complex event, which could be defined in this case as a sequence of measures where the temperature is higher than $40° C$, during at least 10 seconds.

A task for which CEP systems are commonly used for is monitoring streams, where the input stream is queried and eventually transformed. Monitoring streams of semi-structured data requires to filter complex events on the streams, and

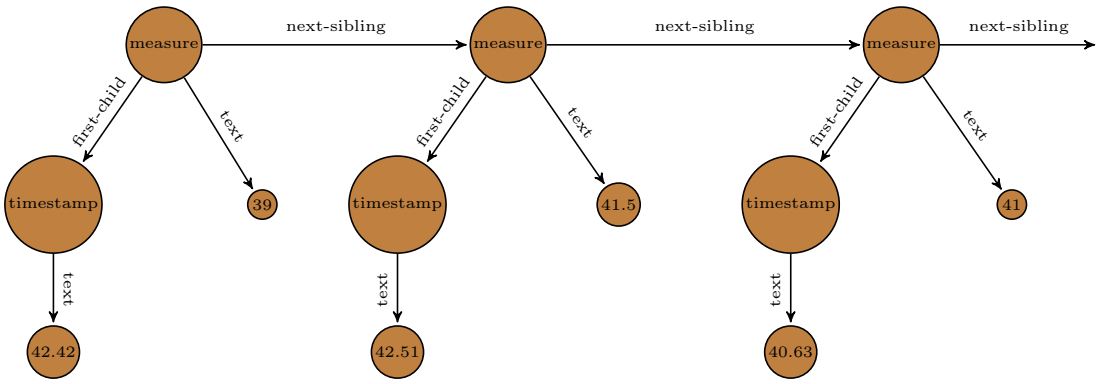


Figure 1.3: Representation of the XML stream of Figure 1.2 as a data graph

to produce output from them, which may be sent over complex event streams to external applications. The complex events are output after the filtering and processing of input events. For instance, a network monitoring system could search for complex events representing intrusion attempts, and then send messages with the details of the intrusion attempts to the network administrator whenever it detects them. The conversion from the input to the output streams can be defined by query-based rewrite rules, for instance in a query-based transformation language for XML *stream processing*, such as XSLT [Kay 2004], CDUCE [Benzaken *et al.* 2003, Castagna *et al.* 2015], or XFUN [Labath & Niehren 2015].

1.1.2 Query Languages

We argued that semi-structured data streams extend on sequences of text tuples. More generally, we will consider streams as sequences of data trees which are also called *data hedges*. A data hedge in turn can be seen as an edge-labeled data graph, as illustrated in Figure 1.3. Therefore, classical logical languages for querying data graphs can be used for defining complex events on streams. This is the approach that we follow in this thesis. Alternatively, various special purpose CEP systems were proposed, many of which support *(sliding) window* techniques for defining complex events [Carney *et al.* 2002, Chandrasekaran *et al.* 2003, Abadi *et al.* 2005, Suhothayan *et al.* 2011, Hallé 2017].

As it was noticed in [Mozafari *et al.* 2012], a query language for CEP should be powerful enough to express the Kleene-star and capture complex events involving an unbounded number of data elements. The same observation was remade more recently in [Grez *et al.* 2019]. Instead of defining an ad hoc language as done there, *nested regular path queries* [Martens & Trautner 2018] are a natural candidate for

this purpose. Regular path queries are regular expressions built from edge labels, that allow to navigate in graphs having labeled edges. In addition, one needs also to be able to filter data trees for which there exists some particular paths. The addition of such filters with the usual Boolean connectives, i.e. conjunction, disjunction, negation, leads to the language of *nested regular path queries* [Libkin *et al.* 2013]. Indeed this language was already introduced in the seventies [Fischer & Ladner 1979] under the name of *propositional dynamic logic* (PDL). More concretely, nested regular path queries are the programs of PDL while the filters are the formulas of PDL.

A data hedge can be turned into a data tree by adding an artificial root node on top. We note that this requires the admission of unranked data trees, given that the length of the stream cannot be bounded. Therefore, it is natural to use nested regular path queries but interpreted on unranked data trees for defining complex events. In other words, PDL on unranked data trees is the query language of our choice. This query language is essentially the same as the navigational core of XPath 3.0, that was introduced and standardized by the W3C for querying Xml documents.

The navigational core of XPath 1.0 was formalized under the name CoreXPath in [Gottlob *et al.* 2003]. In contrast to nested regular path queries, CoreXPath does not admit the Kleene star P^* for any path query P. The navigational core of XPath 3.0, however, should permit the Kleene star given that XPath 3.0 can express it, even though still quite indirectly by using recursive functions.

The basic steps of CoreXPath are *self*, *child*, *following-sibling*, *descendant* $=$ *child*$^+$, *preceding-sibling* $=$ *following-sibling*$^{-1}$, *parent* $=$ *child*$^{-1}$, and *ancestor* $=$ *parent*$^+$. These steps are graphically illustrated in Figure 1.4. From these steps, one can define XPath's *following* axis by the regular path query:

$$\mathit{ancestor}/\mathit{following\text{-}sibling}/(\mathit{descendant} \cup \mathit{self}).$$

On our example streams in Figure 1.2, the following navigational XPath query selects all the *timestamp* elements below *measure* nodes having temperature values greater than or equal to 40° C.

$$\mathit{descendant::measure[number(text())} >= \mathit{40]/child::timestamp}.$$

When it comes to stream processing, one often restricts the nested regular path queries to forwards steps only [Olteanu 2007b, Olteanu 2007a, Sebastian 2016]. In the case of CoreXPath, only the following steps are allowed to build the path

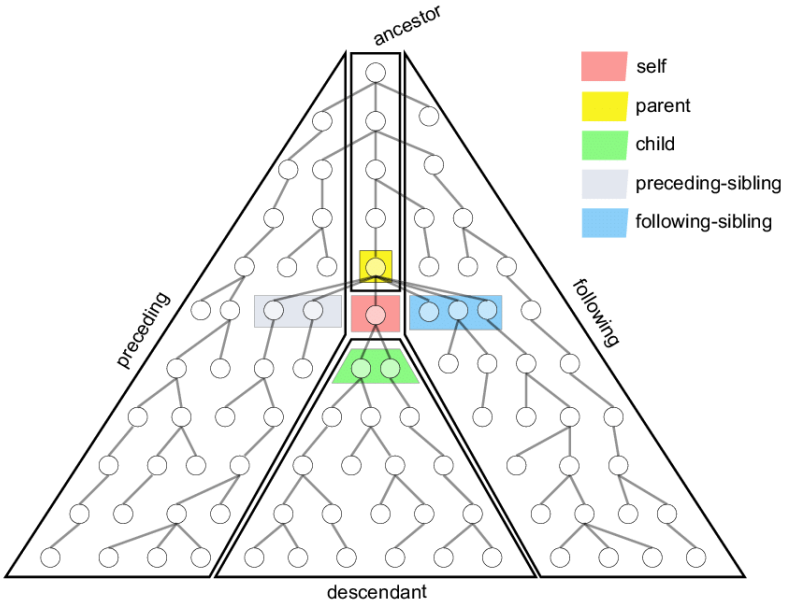


Figure 1.4: Graphic illustration of XPath axes, taken from [Genevès & Layaïda 2006]

queries: *self*, *child*, *descendant*, *following-sibling*. What is ruled out is backwards steps P^{-1}. It should be noticed that the XPath axis *following* uses the backwards step $ancestor = (child^{-1})^+$ in its definition, so it is not purely forward.

Besides nested regular path queries, XPath supports further expressiveness relevant to CEP. In particular, there are non navigational filters with *data joins* $[P/text() = P'/text()]$ by which to compare the data values of nodes reached over the path queries P and P' respectively. However, such queries are no more regular, so they can no more be defined by nested regular path queries.

Furthermore, aggregation is supported by XPath to count the number of query answers or to compute some statistics.

1.1.3 Certain Query Answering (CQA)

Streams can be seen as incomplete databases in which the open end can be instantiated continuously. The standard notion of answers to a logical query, however, is defined for logical structures, that is for complete databases. For incomplete databases [Libkin 2015], the notion of *certain query answers* was widely studied instead. These are query answers that are valid for all completions of an incomplete database.

For instance, consider the nested regular path query below, that selects all *a*-

events on the streams that are followed by some b-event, but not necessarily immediately:

$$\mathit{following\text{-}sibling} :: a[\mathit{following\text{-}sibling} :: b]$$

On the following stream with open end X, all a-events but the last five are certain answers of the above query:

$$aabaabaaaaaX$$

The last five a-events, however, are not certain, since they are not selected on the completion of the stream where $X = \varepsilon$ so that no further event can be added. The b-events, in contrast are certain non-answers for this query. The last five a-events are called *alive*, since they are neither certain query answers nor certain query nonanswers. The alive answer candidates must be buffered by any algorithm computing the certain query answers on a stream.

Certain query answering (CQA) on streams is the problem of selecting the certain query answers on a stream as early as possible. It was introduced in [Gauwin *et al.* 2009, Gauwin 2009] under the name of earliest query answering. In order to solve the online version of CQA, one must be able to decide the decision version of CQA, i.e., given a stream, an event of the stream, and a query, whether the event is a certain query answer of the query on the stream.

It turned out that the decision version of CQA is a computationally hard problem even for tiny fragments of CoreXPath [Gauwin & Niehren 2011] and also not feasible from the perspective of online verification [Benedikt *et al.* 2008, Kupferman & Vardi 2001]. This is basically a universality problem, since it implies reasoning about all completion of the stream.

On the positive side, CQA can be done in polynomial time for queries on streams defined by deterministic nested word automata (NWA) [Gauwin *et al.* 2009]. In theory, every nested regular path query can be compiled to an NWA, which can then be determinized. In practice, however, the determinization takes hours, and the results are huge, even for simple XPath queries such as $\mathit{child} :: a/\mathit{child} :: b$ [Debarbieux *et al.* 2015]. Furthermore, the streaming algorithm for deterministic NWAs requires quadratic time per step depending on the size of the automata, which is by far too slow given that the automata are huge.

As a consequence, practical approaches to answering path queries on streams cannot compute the CQAs as early as possible for general nested regular path queries. Two solutions to this difficulty were proposed.

1. In CEP systems [Mozafari *et al.* 2012, Grez *et al.* 2019] the query languages are restricted such that the certainty of an answer candidate depends only on

the past of the stream but not on the future. For instance,

$$\mathit{following\text{-}sibling} :: a[\mathit{descendant} :: b]/\mathit{following\text{-}sibling} :: c$$

is such a query. Whether a c-event is selected only depends on whether there exists a preceding-sibling a-event that has a b-descendant, also seen before the c-event.

2. In XML stream processing, the CQAs of a XPATH query are approximated as long as the stream is open. Examples for tools based on early CQA algorithms that are not earliest as Olteanu's SPEX [Olteanu 2007b] and Sebastian's QuiXPath [Debarbieux *et al.* 2015]. SPEX compiles the queries from Forward CoreXPath to networks of nondeterministic transducers, while Sebastian compiles them to nondeterministic NWAs. Determinization is avoided by both approaches, so that CQA remains hard for queries defined by the class of finite state machines used there.

It is also interesting to note that sliding window approaches to CEP can detect complex events with the delay bounded by the window's size. Furthermore, for queries defined by deterministic NWAs, it can be decided in PTIME whether they have k-bounded delay [Gauwin *et al.* 2011], so whether they can be answered with some algorithm with a sliding window of size k. Finally, it is possible to define the sliding window queries as part of the nested regular path queries, if one wants to bound the delay artificially.

In the present thesis we do not want to admit so hard restrictions of class of complex events as adopted by CEP. Therefore, we are ready to accept approximations of certain query answering, for general nested regular path queries with forward steps only. As concrete query languages we therefore adopt forward navigational XPATH 3.0.

1.1.4 Quality Criteria

We next discuss the main quality criteria for algorithms querying streams. These are online algorithms that receive the input stream incrementally. Therefore, the usual complexity measures for offline algorithms and problems are no more appropriate. Instead we are interested in criteria such as low latency beside of high-time efficiency, low memory consumption. Of course, which quality can be reached depends on the query language that is chosen. The higher its coverage the better.

Low latency. The latency of the selection of a certain query answer is the number of events that passed since the answer became certain and its selection. The QuiXPath

[Debarbieux *et al.* 2015] tool has a very low latency in practice, but cannot always select with zero latency. Similarly, the SPEX tool also has low latency in practice, but only for queries without negation.

Large coverage. The QuiXPath tool covers a large part of XPath 3.0, including the non-navigational and functional programming aspect of the language. In that aspect QuiXPath outperforms all other tools, whose coverage remains quite limited.

High time efficiency. The *per-event time* spent by the query answering algorithm on the stream is an important quality criterion. Experience has shown that the per-event time complexity for queries represented by automata should be at most linear in the size of the nondeterministic automaton. In particular, a quadratic per-event time complexity depending on the size of the deterministic NWA is too slow, as obtained by the approximation-free CQA algorithm from [Gauwin *et al.* 2009]. Another important aspect is the usage of projection, so that only relevant events that may change the state are to feed to the automaton evaluator [Sebastian & Niehren 2016, Benzaken *et al.* 2013]. Furthermore, the time for parsing the stream should not be included in the measurement [Debarbieux *et al.* 2015]. This is particularly important if many queries are to be answered on a same stream.

Low memory consumption. Streaming query answering algorithms must memorize the *alive answer candidates* at every event, but may also need to buffer some more candidates if the certainty of some answers or nonanswers could not be detected as soon as possible. So the lower the latency of selection and rejection, the lower is the memory consumption.

1.2 Problem: CQA on Hyperstreams

A frequently blocking aspect for query answering on streams is whether the set of certain query answers may be produced in any arbitrary order, or whether one needs to return it as a list in a fixed order. We argue that this problem can be solved by outputting hyperstreams, which however then requires to develop CQA algorithms for hyperstreams.

1.2.1 Hyperstreams

Returning the set of certain query answers in a fixed order has the advantage that the set is presented in a unique manner. However, fixing an order may quickly spoil the latency of the whole algorithm, since a single certain query answer with a large delay may delay all other certain query answers, that can be output only after it.

In the context of XPath, this problem is well known. The answers sets can be

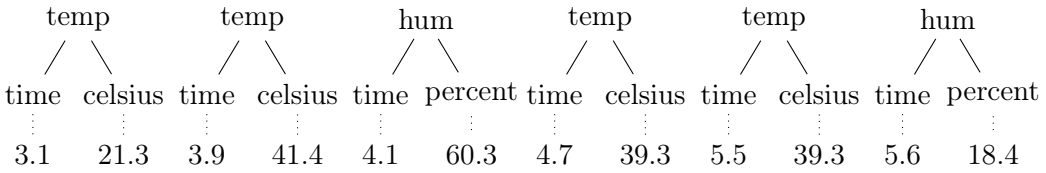


Figure 1.5: A data hedge produced by two sensors.

presented in any order when XPath is used within XQuery transformations, while it must be presented in the order of the input document when within XPath is used within Xslt transformations. So even though standardized, XPath has two different semantics.

In the context of CEP, this problem is equally relevant. To see this, we consider an example where two streams originating from some sensors need to be merged in some order. We argue that this order should not be fixed, if the merged stream is to be produced with low latency. Let's consider the example with two sensors sending timestamped information from [Grez *et al.* 2019]. The first sensor measures the temperature and the second the humidity in some room. Once merged into a single stream it is easier to raise fire alarms based on both informations.

The complete sequence of events that are eventually sent by the sensors is given in increasing temporal order in Figure 1.5. Each event is a data tree whose data values are strings over UTF-8. Each node of a data tree carries two data values, its label and its value. The root node of the first event has label *temp* and the empty value. Its leftmost child has the label *time* and the value 3.1. Its next sibling in turn has label *celsius* and value 21.3. Now suppose that the events on the temperature stream were delayed for some technical reason, so that the last available event has timestamp 3.9, while all humidity events are available. Furthermore, we assume that the events on both streams always arrive in the order of increasing timestamps. How could we then merge the two streams into a single one? If the merger wants to produce a merged stream in which the order of the timestamps is increasing, then it cannot output the already available humidity events with timestamps 4.1 and 5.6. So it will either have to buffer them, which increases the latency and the memory consumption, or discard these events at the cost of losing data.

We subscribe to a third solution which is to produce hyperstreams as output as proposed by [Labath & Niehren 2013]. These are multiple streams with references, of which another variant was introduced in [Maneth *et al.* 2015]. In the example, we would like to output the hyperstream in Figure 1.6 as the result of the merging process. This hyperstream, named U, has three hedge variables: X stands for all temperature events with timestamps in the interval $]3.9, 4.1]$, variable Y for all

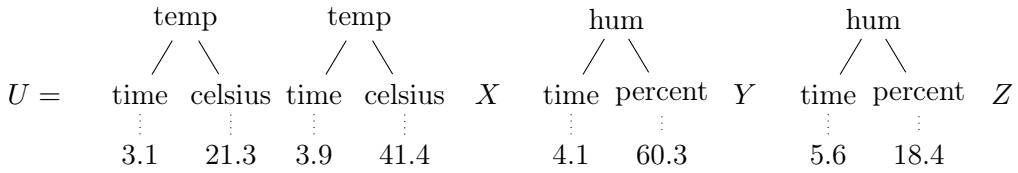


Figure 1.6: A hyperstream for the merged stream with temperature events until time 3.9 and humidity events until time 5.6.

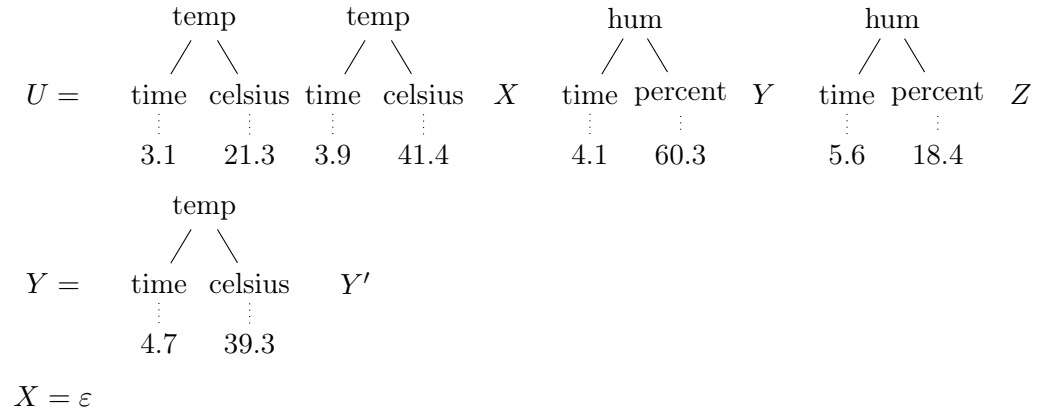


Figure 1.7: Closing X and instantiating Y

temperature events in the interval $]4.1, 5.6]$, and Z for all temperature events with timestamps in the interval $]5.6, \infty[$. These hedge variables are actually references to streams containing data trees.

The merger will know later that there is no temperature event in $]3.9, 4.1[$, and that the first temperature event in $]4.1, 5.6]$ is the one with timestamp 4.7 in Figure 1.5. This is achieved by binding the values of the variables as in Figure 1.7. The symbol ε is used to denote the empty data hedge and the variable Y' refer to all the eventual temperature events with timestamp in $]4.7, 5.6]$. The variables U, X and Y are called *bound* since they have patterns associated to them. The variables Y' and Z are called *free* since they are not bound. This hyperstream is *compression-free*, since each of the bound variable appears at most once in the right-hand sides of the equations. It is *linear* in that each of its free variables appear at most once in the right-hand sides of the equations.

1.2.2 Finding Certain Query Answers

A natural question is: given a query and a hyperstream, how to find its certain query answers on the hyperstream? For instance, one may be interested into detecting fire alarms on the hyperstream obtained from the merge of the temperature and

humidity streams. This could be formulated into a query that selects all nodes x of the data hedge described by the hyperstream, for which there is a node y such that x is a temperature event with more than 40° C and y a humidity event with less than 20 percent, such that y follows x or vice-versa. Considering an XML stream that represents the data hedge in Figure 1.6, the following XPATH query expresses the complex event:

$$\begin{aligned}
\mathbf{Q} &= \mathbf{Q}_1 \; \mathit{union} \; \mathbf{Q}_2 \\
\mathbf{Q}_1 &= /\mathit{child} :: \mathit{temp}[F_1][\mathit{following\text{-}sibling} :: \mathit{hum}[F_2]] \\
\mathbf{Q}_2 &= /\mathit{child} :: \mathit{hum}[F_2]/\mathit{following\text{-}sibling} :: \mathit{temp}[F_1] \\
F_1 &= \mathit{child} :: \mathit{celsius}[\mathit{number}(\mathit{text}()) > 40] \\
F_2 &= \mathit{child} :: \mathit{percent}[\mathit{number}(\mathit{text}()) < 20].
\end{aligned}$$

Fire alarms should be detected independently of how the free variables Y' and Z will be instantiated. Thus the temperature event with timestamp 3.9 is a certain answer of query $\mathbf{Q}$ on the hyperstream in Figure 1.6, since it has 41.4° C and is followed by the humidity event with timestamp 5.6 having a percentage of 18.4.

1.3 Contributions

We now expose the main contributions of this dissertation. While searching for efficient algorithms for answering regular path queries on hyperstreams, we found some new results improving the state of the art on CQA on streams to our own surprise.

1.3.1 Small Deterministic Automata for Regular Path Queries

The only previous algorithms for CQA on streams without approximation [Gauwin *et al.* 2009] applies to queries defined by deterministic NWAs. The first reason why this algorithm could not be applied in practice is related to the determinization algorithm for NWAs from [Alur & Madhusudan 2009, Debarbieux *et al.* 2015], see also Section 2.4.3. The NWAs obtained from the navigational XPATH queries of the XPathMark benchmark with the compiler from [Debarbieux *et al.* 2015, Sebastian 2016] are often not deterministic and can not be determinized with this algorithm neither. For simple queries of the benchmark, the determinization algorithm produces huge automata, while for others its termination could not be observed after few hours.

Our first contribution that we published in [Boneva *et al.* 2020] shows that all navigational forward XPATH queries from the XPathMark benchmark can be

compiled to small deterministic NWAs nevertheless. The trick is to use *stepwise hedge automata* (SHAs) as intermediates. These are a variant of stepwise tree automata [Carme *et al.* 2004] that we introduce. SHAs are a mixture of standard tree automata and finite state automata on words (NFAs), avoiding the trouble with the notion of determinism of the quite different notion hedge automata from [Comon *et al.* 2007, Murata 2000, Thatcher 1967] pointed out in [Martens & Niehren 2007] (see also Section 3.3.3). SHAs have a natural notion of bottom-up and left-to-right determinism, mixing the notions of bottom-up determinism of tree automata with the notion of left-to-right determinism for NFAs. In contrast to NWAs, however, they do not support any form of top-down determinism.

It turns out that the determinization algorithm for NWAs from [Alur & Madhusudan 2009, Debarbieux *et al.* 2015] behaves very badly for NWAs that do nontrivial work in a top-down manner. What this means can be formalized syntactically by not having the single-entry property. In contrast, all NWAs obtained by compilation from SHAs have the single-entry property (up to minor details). A different compiler to that from [Debarbieux *et al.* 2015] for mapping path queries to NWAs can be obtained by compiling them in a first step to SHAs as intermediates, and then compiling the SHAs obtained to NWAs. The NWAs obtained this way have the single-entry property. Using this different compiler indeed solves the problem, since the the NWAs obtained thereby can be determinized in practice by the algorithm [Alur & Madhusudan 2009, Debarbieux *et al.* 2015].

An alternative solution can be obtained by determinizing the SHA obtained from the path queries directly, and then compiling them to NWAs while preserving the determinism. This alternative solution has the advantage that it permits to apply unique minimization to the deterministic SHAs, while unique minimization is not available for deterministic NWAs [Alur *et al.* 2005].

A further difficulty that we needed to overcome here is worth mentioning. The NWAs used by [Debarbieux *et al.* 2015] are symbolic, using descriptors for the complex labels of XML nodes. These descriptors needed to eliminated before determinization, increasing the size of the NWAs considerably. We show in this thesis that we can compile path queries to NWAs with else rules, by adapting the previous compiler. We then lift the determinization algorithm of [Alur & Madhusudan 2009, Debarbieux *et al.* 2015] to NWAs with else rules, thus avoiding any size increase all over. See Section 2.4.3 for further details.

1.3.2 Efficient CQA Algorithm on Streams

The second problem with the approximation-free CQA algorithm on XML streams from [Gauwin *et al.* 2009] is its high polynomial time complexity. Given a query defined by a deterministic NWA this CQA algorithm requires quadratic time per event in the size of the NWA.

Now that we have deterministic NWAs of small size for the queries of interest, one can hope to make this algorithm run in practice. The quadratic running time, however, is still too big. To see this we note that the sizes of the deterministic automata for the queries A1-A8 for the XPathMark benchmark is roughly between 400 and 2500. Therefore quadratic factor per event will be between $400^2 = 160.000$ and 62.500.000.000, which clearly is too big.

We contribute in Chapter 4 a new algorithm that we did not yet submit to a conference. This algorithm can find the certain query answers of a monadic query defined by a deterministic SHA in combined linear time in the size of the automaton and the stream, plus a polynomial time depending of the number certain query answers. The per-event time of our streaming algorithm is linear in the number of states of the SHA, if the polynomial time for the actual creation of the certain queries answers is taken apart. The number of states for the automata obtained for the XPathMark benchmark is between 38 and 124, so the factor per event is reasonably small for our new algorithm.

The shift of the model of deterministic automata from NWAs to SHAs is one of the keys that led us to the finding of the new algorithm for CQA, the first efficient algorithm without approximation. It makes us believe that CQA may be feasible in practice, in contrast to what we believed before. This conjecture still needs to be proven experimentally. We implemented a prototype of our algorithm, but it needs more work to become competitive with the best existing streaming algorithm for XPATH queries from [Sebastian 2016]. In particular, we need to develop and integrate a projection technique for SHAs for replacing those for NWAs [Sebastian & Niehren 2016].

1.3.3 Complexity of CQA on Hyperstreams

As the third contribution, we determine the complexity of the decision version of CQA problem for hyperstreams [Boneva *et al.* 2019]. We show that CQA is EXP-complete, if queries are represented by nondeterministic SHAs, and PSPACE-complete in the case of SHAs. We also obtain a positive result for linear hyperstreams without compression, where CQA is in PTIME.

Establishing the lower bounds is not very difficult, when knowing the complexity

classes of standard automata problems, in particular of the universality problem and of tree automata and of finite state automata. Also the nonemptiness problems for a number of automata is relevant here. For the upper bounds some more work is needed. A hyperstream can be identified with a compressed pattern P for data hedges, whose variables do also denote data hedges. Let $Inst(P)$ be the set of all completions of hyperstream P, that is the set of ground instances of the pattern.

We first reduce CQA for general queries to CQA for Boolean queries. For the latter, CQA can be identified with the problem of *regular pattern inclusion*, i.e. whether $Inst(P) \subseteq L(A)$ for a given compressed pattern for unranked trees P and an SHA A with language $L(A)$. Via determinization regular pattern matching can be reduced to the problem of *regular pattern matching*, i.e. whether $Inst(P) \cap L(A) \neq \emptyset$.

What is more tedious is to deal with the unrankedness of data hedges, and that pattern variables also match data hedges and not only unranked trees. The automata for the data hedges are then taken from the class of SHAs. The general idea to get rid of the unrankedness is to use a reduction to the case of ranked trees. But then variables for hedges have to be replaced by variables for contexts. Furthermore, one has to deal with the fact, that not all ranked trees are encodings of unranked trees. We do so by considering an generalized CQA problem, where one can express regular constraints on the values of the variables.

1.3.4 An Approximation Algorithm for CQA on Hyperstreams

The last contribution is the first algorithm that approximates CQA on hyperstreams for queries defined by SHAs. Our algorithm is in polynomial time for compression-free hyperstreams, and correct in that it computes only certain query answers at any event. It applies to the general case, where the CQA problem is EXP-complete. However when restricted to the simpler case where the hyperstreams are linear and compression-free and that SHAs are deterministic, it is complete in that it outputs all certain query answers at every event.

As a first approximation we make the hyperstreams linear by replacing all occurrences of free variables in the hyperstream by fresh variables. In the third approximation we make hyperstreams compression free, by replacing its bound variables by fresh bound variables. For the second approximation step, we introduce the notion of strong certainty by exploiting the accessibility relation of the SHAs S defining the query, mainly in the same manner than in the efficient streaming algorithm for SHA*s*. The idea is to distinguish sets of states q of S that are *safe for a hyperstream* in that S must reach some final state when starting with q for all completions of the hyperstream. For instance, consider the hyperstream $X = YP$ where Y is a free

variable and P a pattern. If Q is safe for P then the set of states that are safe for X is:

$$safe(Q) = \{q \mid \text{ not exists } q' \notin Q.\ acc(q, q')\}$$

where acc is the accessibility relation of the SHA S. The idea of safe states was first introduced in [Gauwin *et al.* 2009, Gauwin 2009] but there with respect to NWAs instead of SHAs. The set $safe(Q)$ can be computed in linear time if S is a SHA. In contrast if S was an NWA, then it requires quadratic time for each Q after a shared cubic time precomputation.

It remains to solve the problem of strong certainty for linear and compression-free hyperstreams. This is done by evaluating the hyperstream in the *nesting monoid* of the SHA, whose domain consists of the functions mapping sets of states to sets of states. Free variables such as Y are interpreted by the function *safe* in the nesting monoid. The concatenation operator of the hyperstreams is then interpreted as function composition in the nesting monoid. And finally, we define how a tree pattern $< P >$ that is constructed by the nesting operator can be interpreted in the nesting monoid.

What remains to be done is to turn the decision algorithm for CQA on hyperstreams into an online algorithm, while generalizing on the online algorithm that we developed in the case of simpler streams.

1.4 Further Related Work

Nested regular path queries and first-order logic. Nested regular path queries on unranked data trees can express all first-order queries built from atomic formulas $descendant(x, y)$, $following\text{-}sibling(x, y)$, and $text(x) = \texttt{'constant'}$. To see this, we note that the language of nested regular path queries for unranked data trees is basically the same as the navigational core of XPath 3.0 which is known to subsume the above first-order logic [Marx 2004]. This is in contrast to CoreXPath, where the Kleene star is not permitted. And the Kleene star is needed to express first-order queries such as $(child :: a)^*$ that are called conditional axes by Maarten Marx.

Conversely, nested regular path queries can express some queries which are not first-order definable, such as $(child :: a/child :: a)^*$. These queries are still definable in the monadic second-order logic, but not all of monadic second-order logic is captured by nested regular path queries [Bojańczyk *et al.* 2006, Samuelides 2007].

XPath. We also mention that XPath queries with data joins cannot be defined in the above first-order logic, but can be defined in the first-order logic in which comparison of data values $text(x) = text(y)$ are admitted as atomic formulas. When

doing so, no more all first-order queries can be defined by nested regular path queries. This lack of expressiveness is solved in XPath 3.0 by the addition of variables, existential quantifiers, and universal quantifiers, since XPath 2.0 [Filiot *et al.* 2007].

XPath fragments other than CoreXPath have been explored [Benedikt & Koch 2008]. Most of these fragments were only studied in theory, and the percentage of the real-world most used queries that they contain is not really clear. Recently [Baelde *et al.* 2019] designed a benchmark for the XPath most used queries in practice, extracted from projects with Xslt or XQuery components. They show that these fragments – including CoreXPath – cover only a small part of the XPath queries written in those projects. However, they proposed some extensions to these fragments that do not really affect their satisfiability but extends their coverage a lot.

Certain query answering on streams. In the context of stream processing, certain query answers were called answers that are safe for selection and certain query non-answers were called safe for rejection [Gauwin & Niehren 2011]. Certain query non-answers were studied for fast failure [Benedikt *et al.* 2008] and for reducing the memory consumption of streaming systems. As for CQA, certain query nonanswers on streams has been shown to be computationally hard even for queries defined in tiny fragments of first-order logic [Gauwin & Niehren 2011]. It was also shown to be hard in the context of online verification [Benedikt *et al.* 2008, Kupferman & Vardi 2001].

Streams with references. Similar objects to hyperstreams have been studied before. For instance, the idea of producing trees with references arose in the context of Active Xml [Abiteboul *et al.* 2008], but even much earlier in functional programming languages with futures [Halstead 1985, Niehren *et al.* 2006].

Hyperstreams with compression. Our notion of hyperstreams admits compression, when bound references are used multiply, similarly to singleton context-free tree grammars [Plandowski 1995]. When programming with streams, it seems natural that references to streams can be used more than once. For instance, if a latex document is input of which the layout and the table of content are output, an example that stems from [Labath & Niehren 2013]. Or consider a transformation as in [Maneth *et al.* 2015] that reads a list on the input stream X and outputs a pair $<< X >< X >>$ with twice the input. In this case, outputting the pair on a hyperstream with compression is very natural. The shared occurrences of the bound variable X of the output are then instantiated incrementally with reading the input.

1.5 Publications

Two of the four contributions of this thesis have been peer-reviewed and published in the proceedings of international computer science conferences. Three publications were obtained of which two are included in this thesis. They can be found in the following chapters:

Chapter 3 Our work on the compilation of nested regular path queries to small deterministic [Boneva *et al.* 2020] got accepted at the International Computer Science Symposium in Russia (CSR'2020).

Chapter 6 Our study of the complexity of regular pattern matching and inclusion for compressed tree patterns with context variables [Boneva *et al.* 2019] is published in the proceedings on the International Conference on Language and Automata Theory and Applications (LATA'2019).

Not included Before studying the complexity of CQA on hyperstreams in the case of data trees in Chapter 6 we considered the case of words. We determined the complexity of regular pattern matching and regular pattern inclusion on compressed string patterns [Boneva *et al.* 2018]. These results were published in the proceedings of the International Conference on Reachability Problems (RP'2018) but not included in this thesis.

1.6 Organization of the thesis

Chapter 2 introduces the main concepts that will be used throughout this thesis. It defines the notions of nested words, patterns and queries, but also different classic automata models such as finite-state automata, nested word automata and stepwise tree automata. Regular expressions for nested words are discussed there, as well as the query languages XPath and Fxp.

In Chapter 3 we present our model of stepwise hedge automata, and study it under different aspects. In particular we show how it is related to the previously introduced automata model. We also provide a compiler from nested regular expressions to stepwise hedge automata, and later show how to obtain small deterministic stepwise hedge automata. Finally we show experimentally that deterministic nested word automata obtained for nested regular expressions can be very small when they are built from stepwise hedge automata.

We present our new algorithm for certain query answering on streams in Chapter 4. We prove the correctness of this algorithm and establish its worst-case running time.

Hyperstreams are formally introduced in Chapter 5. Then in Chapter 6, we study the complexity of certain query answering on compressed tree patterns with context variables. The latter are a type of hyperstreams into which general hyperstreams of hedges can be converted.

Finally in Chapter 7, we get back to the more general hyperstreams of hedges, and study different approximations of the certain query answering problem. We introduce linear certainty and strong certainty, and show in which cases they fit the most.

Chapter 2

Preliminaries

Contents

2.1 Words, Trees, Nested Words and Languages

2.1.1 Words

An *alphabet* Σ is a set of elements called symbols. A word w of length $n \geq 0$ over Σ is an element of Σ^n. As usual, we write $a_1 \dots a_n$ to denote the word $(a_1, \dots, a_n) \in \Sigma^n$, and $\Sigma^* = \bigcup_{i \geq 0} \Sigma^n$ for the set of words over Σ. The empty word ε is the word of length 0. The concatenation of two words $w = a_1 \dots a_n$ and $w' = b_1 \dots b_m$ is the word $a_1 \dots a_n b_1 \dots b_m$ and is noted $w \cdot w'$ or ww' when w and w' are clearly separated in the context. A language of words is a set of words. For instance, Σ^* is a language of words.

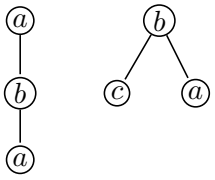


Figure 2.1: Graphical representation of the hedge in Example 1

2.1.2 Trees

In contrast to words that only have a horizontal structure, trees permit vertical structuring.

2.1.2.1 Ranked Trees

Ranked trees are trees where every symbol has a fixed number of subtrees. A ranked signature $\Sigma = (\Sigma, \mathit{arity})$ is a tuple where Σ is an alphabet, and $\mathit{arity} : \Sigma \to \mathbb{N}$ a function associating a natural with each symbol of Σ. For a symbol $f \in \Sigma$, we call this natural its *arity*. For all $m \geq 0$, we write $\Sigma^{(m)}$ to denote the set of symbols in Σ with arity m, that is $\Sigma^{(m)} = \{f \in \Sigma \mid \mathit{arity}(f) = m\}$. A symbol of arity 0 is called a constant.

The set $\mathcal{T}_\Sigma$ of ranked trees over Σ is defined using the following abstract syntax:

$$t \in \mathcal{T}_\Sigma ::= f(t_1, \ldots, t_n)$$

where $f \in \Sigma^{(n)}$ for some $n \geq 0$ and $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$.

In the above definition, $t_1, \ldots, t_n$ are the *children* of the tree $f(t_1, \ldots, t_n)$. Any *atomic tree* $a() \in \mathcal{T}_\Sigma$ will be identified with the constant $a \in \Sigma^{(0)}$.

2.1.2.2 Hedges And Unranked Trees

Unranked trees do not have the limitation in the number of children, to which ranked trees are subjected. Moreover, they can be concatenated in order to form hedges. Let Σ be an alphabet. The set of hedges $\mathcal{H}_\Sigma$ is obtained from the abstract syntax

$$H, H' \in \mathcal{H}_\Sigma ::= \varepsilon \quad | \quad \langle_a H \rangle \quad | \quad HH'$$

The set of unranked trees $\mathcal{U}_\Sigma$ is the subset of hedges of the form $\langle_a H \rangle$.

Example 1 *Let $\Sigma = \{a, b, c\}$. The hedge $\langle_a \langle_b \langle_a \rangle \rangle \rangle \langle_b \langle_c \rangle \langle_a \rangle \rangle$ is graphically represented in Figure 2.1.*

2.1.3 Nested Words

Nested words are words having both a linear and hierarchical – or nesting – structure.

Let $\langle$ and $\rangle$ be symbols that we respectively call *opening parenthesis* and *closing parenthesis*. A nested word over some alphabet Σ disjoint from $\{\langle, \rangle\}$ is a word generated by the following abstract syntax:

$$w, w' \in nWords_\Sigma \quad ::= \quad \varepsilon \quad | \quad a \quad | \quad \langle w \rangle \quad | \quad w \cdot w' \qquad \text{where } a \in \Sigma$$

A language of nested words is a set of nested words, and the set of nested words over Σ is denoted by $nWords_\Sigma$.

Our definition of nested words restricts the more general one that can usually be found [Alur & Madhusudan 2009]. First, we only consider the well-nested forms, that is, only nested words where every opening parenthesis is properly closed and every closing parenthesis is properly opened. Second, the only markers of the nesting structure are $\langle$ and $\rangle$, and not elements of general alphabets.

Nested words are a generalization of words and hedges. Any hedge over some alphabet Σ can be identified with the nested word where all the occurrences of $\langle_a$ for $a \in \Sigma$ are replaced by $\langle a$. For instance, a hedge $\langle_a \langle_b \rangle \rangle \langle_c \rangle$ is considered as equal to the nested word $\langle a \langle b \rangle \rangle \langle c \rangle$. For this reason, we consider hedges and unranked trees as nested words, and will write them using the syntax of nested words.

2.2 Patterns

2.2.1 Definition

Let $\mathcal{V}$ be a set of elements that we call *pattern variables*, and Σ an alphabet. A *nested (word) pattern* over Σ is a nested word over $\Sigma \uplus \mathcal{V}$. The set of nested patterns over Σ is denoted $nPatterns_\Sigma$. The set of pattern variables appearing in a pattern ρ is denoted $fv(\rho)$. A nested pattern is called *linear* if all of its free variables occur at most once in its definition. The set of linear nested patterns over Σ is denoted $nLinPatterns_\Sigma$. *String patterns* over Σ are the restriction of nested words over $\Sigma \uplus \mathcal{V}$ to words over $\Sigma \uplus \mathcal{V}$.

Let μ be an assignment from $\mathcal{V}$ to $nPatterns_\Sigma$. An *instance* of a pattern in $nPatterns_\Sigma$ is a nested pattern defined by the function $[\![\cdot]\!]^\mu$ so that for all $\rho, \rho' \in nPatterns_\Sigma$, $w \in nWords_\Sigma$, $X \in \mathcal{V}$:

$$\begin{aligned} [\![X]\!]^\mu &= \mu(X) \\ [\![w]\!]^\mu &= w \\ [\![\rho\rho']\!]^\mu &= [\![\rho]\!]^\mu [\![\rho']\!]^\mu \end{aligned}$$

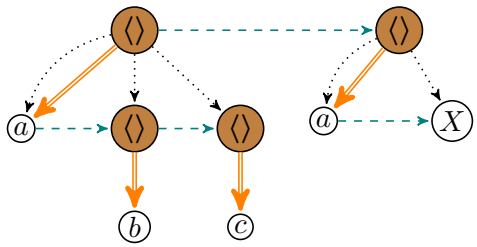


Figure 2.2: Relations between the positions of $\rho = \langle a\langle b\rangle\langle c\rangle\rangle\langle aX\rangle$

Let ρ be a nested pattern. A *ground instance* of ρ is a nested word $[\![\rho]\!]^\mu$ where $\mu : fv(\rho) \to nWords_\Sigma$ maps every pattern variable of ρ to some nested word. We write $Inst(\rho) = \{[\![\rho]\!]^\mu \mid \mu : fv(\rho) \to nWords_\Sigma\}$ to denote the set of ground instances of ρ.

2.2.2 Identification to Logical Structures

Nested patterns can also be represented as logical structures, using the standard first-child and next-sibling relations.

Example 2 *Figure 2.2 shows a nested pattern as a logical structure. Its domains are positions represented by circles, while the different relations between the positions (*first-child*,* next-sibling*,* child*) are represented by arcs between circles. The position with no arc pointing to it is called the* root*.*

Let Σ be an alphabet and $\rho \in nPatterns_\Sigma$ a nested pattern. The *set of positions* of ρ, written $Pos(\rho)$, is a set of elements on which the binary relations fc^ρ and ns^ρ are defined. These relations are respectively called *first-child* and *next-sibling*. Another useful binary relation is the *child* relation, defined as $ch^\rho = fc^\rho \circ (ns^\rho)^*$. Let $\pi, \pi' \in Pos(\rho)$ be two positions of ρ. We call π the *parent* of π' if $(\pi, \pi') \in ch^\rho$. In this case, π' is also the child of π. We call π the *previous sibling* of π' if $(\pi, \pi') \in ns^\rho$. We also say that π' is the next-sibling of π in this case. The child and next-sibling relations impose every position to have exactly one parent and at most one next sibling, except a special node that we call the *root position* that has no parent – but may have a next sibling. The root position has no previous sibling neither. We denote it by $root(\rho)$.

A *label* of ρ is an element of $\Sigma_\rho = \Sigma \cup fv(\rho)$. Any label ℓ defines the set $lab^\rho_\ell \subseteq Pos(\rho)$ of positions. Furthermore, for any position $\pi \in Pos(\rho)$, there exists at most one label ℓ called the label of π, that satisfies $\pi \in lab^\rho_\ell$. A *node* is a position with no label. The set of nodes of ρ is written $Nodes(\rho)$, while the set of labeled positions of ρ is $LPos_{\Sigma_\rho}(\rho)$. We finally denote the set of positions labeled only by elements of some set $\Sigma' \subseteq \Sigma_\rho$ by $LPos_{\Sigma'}(\rho)$.

We now show how the positions and labels are related to ρ. For any patterns ρ', ρ'' and label $a \in \Sigma \cup fv(\rho)$,

$$\rho = \varepsilon \quad \text{iff } Pos(\rho) = fc^\rho = ns^\rho = \emptyset \text{ and } \forall \ell \in \Sigma_\rho . lab^\rho_\ell = \emptyset$$

$$\rho = a \quad \text{iff} \begin{cases} Pos(\rho) = \{root(\rho)\},\ \{root(\rho)\} = lab^\rho_a,\ fc^\rho = ns^\rho = \emptyset \\ \text{and } \forall \ell \in \Sigma_\rho \setminus \{a\} . lab^\rho_\ell = \emptyset \end{cases}$$

$$\rho = \langle \rho' \rangle \quad \text{iff} \begin{cases} Pos(\rho) = \{root(\rho)\} \uplus Pos(\rho'), \quad ns^\rho = ns^{\rho'}, \\ fc^\rho = fc^{\rho'} \cup \{(root(\rho), root(\rho'))\} \text{ and } \forall \ell \in \Sigma_\rho . lab^\rho_\ell = lab^{\rho'}_\ell \end{cases}$$

$$\rho = \rho'\rho'' \quad \text{iff} \begin{cases} Pos(\rho) = Pos(\rho') \uplus Pos(\rho''),\ \forall \ell \in \Sigma_\rho . lab^\rho_\ell = lab^{\rho'}_\ell \cup lab^{\rho''}_\ell, \\ fc^\rho = fc^{\rho'} \uplus fc^{\rho''},\ \exists! \pi \in Pos(\rho') . (root(\rho'), \pi) \in (ns^\rho)^*, \\ \pi \text{ has no next-sibling and } ns^\rho = ns^{\rho'} \cup ns^{\rho''} \cup \{(\pi, root(\rho'))\} \end{cases}$$

Example 3 *Consider for instance the nested pattern $\rho = \langle a\langle b\rangle\langle c\rangle\rangle\langle aX\rangle$ graphically represented in Figure 2.2. Its positions are represented as circles, and the colored circles are the nodes. The elements of the first-child relation are linked by doubled arrows, pointing to the child. The tuples in the next-sibling relation are linked by dashed arrows, originating from the previous sibling. We also added the child relation, whose elements are linked by dotted arrows – pointing to the child. Furthermore, for all $\ell \in \Sigma_\rho$, lab^ρ_ℓ is the set of positions – circled nodes – where ℓ is written.*

We also define the *descendant* $ds^\rho = (ch^\rho)^+$ and *following-sibling* $fs^\rho = (ns^\rho)^+$ relations. Furthermore, we allow additional properties not related to fc^ρ and ns^ρ to be defined on patterns. For this, we define a set $\mathcal{L}^\rho$ of properties $\ell^\rho \subseteq Pos(\rho)$. If $\mathcal{L}^\rho \neq \emptyset$, we say that ρ has *complex labels*.

2.3 Queries

2.3.1 V-Structures and Sequenced V-Structures

We consider an infinite set $\mathcal{W}$ of elements called *query variables*, totally ordered by $\leq_\mathcal{W}$. Let Σ be an alphabet, ρ a pattern over Σ, and α a partial function from $\mathcal{W}$ to the set $LPos_\Sigma(\rho)$ of labeled positions of ρ where the label is an element of Σ.

We write $\rho * \alpha$ to denote the pattern over $\Sigma \times 2^\mathcal{W}$ where any position $\pi \in Pos(\rho)$ labeled by some $a \in \Sigma$ is replaced by a new position π' labeled by $(a, \alpha^{-1}(\pi))$, while the other positions are not changed. $\rho * \alpha$ is called a *$\mathcal{W}$-structure* over Σ [Straubing 1994].

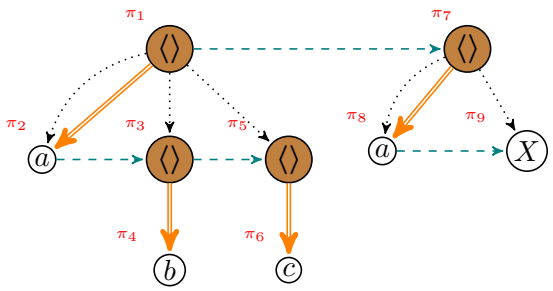


Figure 2.3: Pattern structure with node names

Example 4 *Consider the nested pattern* $\rho = \langle a\langle b\rangle\langle c\rangle\rangle\langle aX\rangle$ *represented in Figure 2.3 where the positions are written next to the circles that represent them. Let* $x, y \in \mathcal{W}$ *be variables and* α *a function that maps* x *to* π_4 *and* y *to* π_8*. Then* $\rho * \alpha = \langle (a,\emptyset)\langle (b,\{x\})\rangle\langle (c,\emptyset)\rangle\rangle\langle (a,\{y\})X\rangle$.

Let $n \geq 0$ and $\mathcal{W}' = \{x_1, \ldots, x_n\} \subsetneq \mathcal{W}$ a finite set of query variables satisfying the order $x_1 \leq_{\mathcal{W}} x_2 \leq_{\mathcal{W}} \ldots \leq_{\mathcal{W}} x_n$. Let $\Sigma' \subseteq \Sigma$ be a subset of Σ and $\alpha\colon \mathcal{W}' \to LPos_{\Sigma'}(\rho)$ a total function from $\mathcal{W}'$ to the positions labeled by elements of Σ'. We write $\neg\mathcal{W}'$ to designate the set $\{\neg x \mid x \in \mathcal{W}'\}$. We define $\rho\star\alpha$ as the nested pattern over $\Sigma \cup \mathcal{W}' \cup \neg\mathcal{W}'$ where every position $\pi \in LPos_{\Sigma'}(\rho)$ is replaced by a set of new positions $\{\pi_0, \pi_1 \ldots \pi_n\}$ such that:

- π_0 is labeled by $a \in \Sigma$ iff π is labeled by a
- for all $1 \leq i \leq n$, π_i is labeled by x_i if $\alpha(x_i) = \pi$ and by $\neg x_i$ otherwise
- for all $0 \leq i < n$, $(\pi_i, \pi_{i+1}) \in ns^{\rho\star\alpha}$
- if π has a parent π', then $\begin{cases} (\pi', \pi_0) \in ch^{\rho\star\alpha} & \text{if } \pi' \notin LPos_{\Sigma}(\rho) \\ (\pi'_n, \pi_0) \in ch^{\rho\star\alpha} & \text{otherwise} \end{cases}$
- if π has a next sibling π', then $\begin{cases} (\pi_n, \pi') \in ch^{\rho\star\alpha} & \text{if } \pi' \notin LPos_{\Sigma'}(\rho) \\ (\pi_n, \pi'_n) \in ch^{\rho\star\alpha} & \text{otherwise} \end{cases}$
- if π has a previous sibling π', then $\begin{cases} (\pi', \pi_0) \in ns^{\rho\star\alpha} & \text{if } \pi' \notin LPos_{\Sigma'}(\rho) \\ (\pi'_n, \pi_0) \in ns^{\rho\star\alpha} & \text{otherwise} \end{cases}$

$\rho \star \alpha$ is called a *sequenced* $\mathcal{W}'$*-structure* with respect to Σ'. If Σ' is not specified, then the sequenced $\mathcal{W}'$-structures are built with respect to Σ.

Example 5 *Back to Example 4, assume that* $x \leq_{\mathcal{W}} y$ *and let* $\Sigma' = \{a, b\}$*. The sequenced* $\mathcal{W}'$*-structure* $\rho \star \alpha$ *with respect to* Σ' *equals*

$$\langle a\ \neg x\ \neg y\ \langle b\ x\ \neg y\rangle\ \langle c\rangle\rangle\ \langle a\ \neg x\ y\ X\rangle$$

.

Lemma 1 *For all $\mathcal{W}' \subseteq \mathcal{W}$, any sequenced $\mathcal{W}'$-structure contains exactly one occurrence of each element of $\mathcal{W}'$.*

The property expressed in Lemma 1 is called the *canonicity* of $\mathcal{W}'$-structures.

2.3.2 Certain answers and non-answers

Let $\mathcal{W}' \subsetneq \mathcal{W}$ be a finite set of query variables, and $\rho \in nPatterns_\Sigma$ a nested pattern.

Definition 1 *A* candidate *of ρ with (query) variables in $\mathcal{W}'$ is a partial function from $\mathcal{W}'$ to $Pos(\rho)$.*

We write $[x_1/\pi_1, \ldots, x_n/\pi_n]$ to denote any candidate α mapping variables in $\mathcal{W}'$ to positions, where $\{x_1, \ldots, x_n\}$ is the subset of $\mathcal{W}'$ for which α is defined and $\alpha(x_i) = \pi_i$ for all $1 \leq i \leq n$. The set of candidates of ρ with variables in $\mathcal{W}'$ is denoted by $\mathcal{C}_{\mathcal{W}'}(\rho)$. A candidate is called *complete* if it's a total function.

Definition 2 *A query $\mathbf{Q}$ with alphabet Σ and (query) variables in $\mathcal{W}'$ is a function that associates any nested word $w \in nWords_\Sigma$ with a subset of $\mathcal{C}_{\mathcal{W}'}(w)$.*

The set of query variables of a query $\mathbf{Q}$ is also written $fv(\mathbf{Q})$. The language of $\mathbf{Q}$, denoted $L(\mathbf{Q})$, is the set of sequenced $fv(\mathbf{Q})$-structures over Σ that equals $\{w \star \alpha \mid w \in nWords_\Sigma, \alpha \in \mathbf{Q}(w)\}$.

A *boolean query* $\mathbf{Q}$ over Σ is a query with no variables, that is $fv(\mathbf{Q}) = \emptyset$. Let $\mathbf{Q}$ be a boolean query over Σ. Therefore, for any nested word $w \in nWords_\Sigma$, the only candidate that may be part of $\mathbf{Q}(w)$ is the *empty candidate*, written $[]$. Remark that the language of $\mathbf{Q}$ is a set of nested words, that is $L(\mathbf{Q}) \subseteq nWords_\Sigma$, and that if $w \notin L(\mathbf{Q})$, then $\mathbf{Q}(w) = \emptyset \neq \{[]\}$.

We next formalize the notions of certain query answers and non-answers on nested patterns. For streams, these definitions coincide with the notions of earliest query answers from [Gauwin & Niehren 2011] and fast-failure from [Benedikt *et al.* 2008], respectively.

Let $\mathbf{Q}$ be a query over Σ and $\rho \in nPatterns_\Sigma$ a pattern.

Definition 3 *A candidate α of ρ is a certain query answer for $\mathbf{Q}$ if for all assignment $\mu\colon fv(\rho) \to nWords_\Sigma$, the instance $[\![\rho]\!]^\mu \star \alpha$ is in $L(\mathbf{Q})$.*

Definition 4 *A candidate α of ρ is a certain query nonanswer for $\mathbf{Q}$ if for all assignment $\mu\colon fv(\rho) \to nWords_\Sigma$, the instance $[\![\rho]\!]^\mu \star \alpha$ is not in $L(\mathbf{Q})$.*

Remark that a certain query answer is always a complete candidate, while a certain nonanswer can be of any type.

2.4 Automata and Regular Expressions

We now recall the models of word automata, stepwise tree automata and nested word automata.

2.4.1 Word Automata

Definition 5 *A (nondeterministic) finite-state automaton (*NFA*)* $\mathcal{F}$ *is a tuple* $\mathcal{F} = (Q, \Sigma, \Delta, I, F)$ *where* Q *is a set of states,* Σ *an alphabet,* $I, F \subseteq Q$ *are respectively the sets of initial states and final states, and* $\Delta \subseteq Q \times \Sigma \cup \{\varepsilon\} \times Q$ *is a transition relation.*

The NFA $\mathcal{F}$ is said *deterministic*, or a DFA, if I is a singleton and Δ can be represented as a function from $Q \times \Sigma$ to Q. Let $q_1, q_2 \in Q$ be states of $\mathcal{F}$. The set of words that can be read by $\mathcal{F}$ from q_1 to q_2, written $L_{q_1,q_2}(\Delta)$, is defined by

$$\begin{array}{rcl} L_{q_1,q_2}(\Delta) & = & \{\varepsilon \mid \text{ if } q_1 = q_2 \text{ or } (q_1, \varepsilon, q_2) \in \Delta\} \cup \{a \in \Sigma \mid (q_1, a, q_2) \in \Delta\} \\ & & \cup \bigcup_{q_3 \in Q} L_{q_1,q_3}(\Delta) \cdot L_{q_3,q_2}(\Delta) \end{array}$$

where $L_{q_1,q_3}(\Delta) \cdot L_{q_3,q_2}(\Delta)$ stands for the concatenation of the sets $L_{q_1,q_3}(\mathcal{F})$ and L_{q_3,q_2} for some state q_3. The language of $\mathcal{F}$ is defined as the set of words that can be read from some initial state of $\mathcal{F}$ to some final $\mathcal{F}$, and is denoted by:

$$L(\mathcal{F}) = \bigcup_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(\mathcal{F}).$$

Given the alphabet Σ, a regular expression over Σ is an expression satisfying the following abstract syntax:

$$reg, reg' ::= \emptyset \quad | \quad \varepsilon \quad | \quad a \quad | \quad reg + reg' \quad | reg \cdot reg' \quad | \quad reg^*$$

where $a \in \Sigma$. It is well-known that any regular expression defines a language recognized by an NFA, and vice-versa.

Let $\mathcal{F} = (Q, \Sigma, \Delta, I, F)$ and $\mathcal{F}' = (Q, \Sigma, \Delta, I, F)$ be NFAs. The *product* of $\mathcal{F}$ and $\mathcal{F}'$ is the automaton $(\times\mathcal{F}, \mathcal{F}') = (Q \times Q', \Sigma, \Delta_{\mathcal{F}\times\mathcal{F}'}, I \times I', F \times F')$ where $\Delta_{\mathcal{F}\times\mathcal{F}'} = \{((q_1, q_1'), a, (q_2, q_2')) \mid (q_1, a, q_2) \in \Delta \text{ and } (q_1', a, q_2') \in \Delta'\}$. Note that $\mathcal{F} \times \mathcal{F}'$ recognizes the intersection of the languages of $\mathcal{F}$ and $\mathcal{F}'$, that is, $L(\mathcal{F} \times \mathcal{F}') = L(\mathcal{F}) \cap L(\mathcal{F}')$.

For every $\mathcal{F} = (Q, \Sigma, \Delta, I, F)$, one can obtain a DFA $\det \mathcal{F}$ recognizing the same language than $\mathcal{F}$. The process for obtaining $\det \mathcal{F}$ is called *determinization*. The easiest way for obtaining $\det \mathcal{F}$ is to set $\det \mathcal{F} = (2^Q, \Sigma, \det \Delta, \{I\}, \{Q' \subseteq Q \mid$

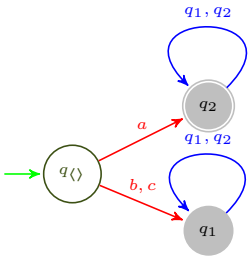


Figure 2.4: Example of dSTA over $\{a, b, c\}$

$Q' \cap F \neq \emptyset\}$) and $\det \Delta = \{(Q', a, Q'') \mid Q', Q'' \subseteq Q$ and $\exists q' \in Q', q'' \in Q'' \mid (q', a, q'') \in \Delta\}$.

2.4.2 Stepwise Tree Automata (STAs)

We now recall stepwise tree automata STAs[1], that allow to recognize languages of (unranked) trees. The definition that we give here is adapted to our syntax of unranked trees, seen as nested words.

Definition 6 *A (nondeterministic) stepwise tree automaton [Carme et al. 2004] is a tuple* $\mathcal{T} = (Q \uplus \{q_{\langle\rangle}\}, \Sigma, \Delta_\Sigma, \Delta_Q, F)$ *where* Q *is a set of states and* $q_{\langle\rangle}$ *the tree initial state,* Σ *an unranked alphabet,* $F \subseteq Q$ *the set of final states,* $\Delta_\Sigma \subseteq \{q_{\langle\rangle}\} \times \Sigma \times Q$ *the set of initial transitions and* $\Delta_Q \subseteq Q \times Q \times Q$ *the transition relation.*

A STA is called deterministic, or a dSTA, if Δ_Σ can be written as a function from $\{q_{\langle\rangle}\} \times \Sigma$ to Q, and Δ_Q as a function from $Q \times Q$ to Q. For all $a \in \Sigma$, $q_1, q_2, q_3 \in Q$, we write $q_{\langle\rangle} \xrightarrow{a} q$ whenever $(q_{\langle\rangle}, a, q) \in \Delta_\Sigma$, and $q_1 @ q_2 \to q_3$ if $(q_1, q_2, q_3) \in \Delta_Q$. Also, for all $q \in Q$, we define the tree language of q as

$$\begin{aligned} L_q(\mathcal{T}) \quad = \quad & \{\langle a \rangle \mid q_{\langle\rangle} \xrightarrow{a} q \in \Delta_\Sigma\} \cup \{\langle f\ t_1 \dots t_n \rangle \mid \exists q_0, q_1, \dots, q_n \in Q \,.\, q_{\langle\rangle} \xrightarrow{f} q_0 \in \Delta_\Sigma, \\ & t_i \in L_{q_i}(\mathcal{T}) \text{ and } q_{i-1} @ q_i \to q_{i+1} \in \Delta_Q \text{ for all } 0 < i < n\}. \end{aligned}$$

The language of $\mathcal{T}$, written $L(\mathcal{T})$, is the set of unranked trees that are in the tree language of some final state, that is:

$$L(\mathcal{T}) = \bigcup_{q \in F} L_q(\mathcal{T}).$$

Figure 2.4 illustrates a deterministic STA over $\{a, b, c\}$ with states $\{q_{\langle\rangle}, q_1, q_2\}$ and final state q_2. The tree initial state is denoted by $\xrightarrow{\langle\rangle} q_{\langle\rangle}$. The dSTA defines the initial transitions $q_{\langle\rangle} \xrightarrow{a} q_2, q_{\langle\rangle} \xrightarrow{b} q_1$ and $q_{\langle\rangle} \xrightarrow{c} q_1$. The other transitions are

[1] STAs were called stepwise hedge automata in [Comon *et al.* 2007]

$q_1@q_1 \to q_1, q_1@q_2 \to q_1, q_2@q_1 \to q_2$, and $q_2@q_2 \to q_2$. The set of trees that it recognizes are all the trees of the form $\langle aw \rangle$ where $w \in \mathcal{H}_\Sigma$.

As for word automata, one can define the product and the determinization of STAs, by process that are quite similar. Note that there also exist regular expressions for tree languages [Comon *et al.* 2007].

2.4.3 Nested Word Automata (NWAs)

Nested word automata (NWAs) are pushdown automata reading nested words, whose stacks are visible: they push a single stack symbol when reading an opening parenthesis, pop a single stack symbol when reading a closing parenthesis, and don't alter or inspect the stack otherwise.

Our notion of NWA is symbolic [D'Antoni & Alur 2014] for dealing with large or infinite alphabets, and supports factorization in the spirit of [Champavère *et al.* 2009].

Definition 7 *An NWA is a tuple* $\mathcal{N} = (Q_h, Q_t, \Sigma, \Gamma, \Delta, I, F)$ *consisting of a possibly infinite set* Σ *of internal symbols, finite sets* Q_h *and* Q_t *of states of type hedge and tree respectively, sets of initial and final states* $I, F \subseteq Q_h$*, a finite set* Γ *of stack symbols, and a finite set* Δ *of transition rules of the forms:*

hedge rules	$a^\Delta, \_^\Delta, \varepsilon^\Delta \subseteq Q_h \times Q_h$	*where* $a \in \Sigma$
opening rules	$\langle_\gamma^\Delta \subseteq Q_h \times Q_h$	*where* $\gamma \in \Gamma$
hedge ending rules	$tree^\Delta \subseteq Q_h \times Q_t$	
closing rules	$\rangle_\gamma^\Delta \subseteq Q_t \times Q_h$	

Our NWAs are symbolic, in that they come with else rules, i.e elements of $(q, q') \in \_^\Delta$ that we will denote by $q \Rightarrow q'$.

The NWAs proposed by [Debarbieux *et al.* 2015] were also symbolic in order to describe the complex labels of nodes of XML documents, which are tuples of:

1. subsets of query variables,
2. an XML type (element, attribute, document, comment, or text)
3. XML names,
4. XML namespaces.

Such descriptors needed to be removed before determinization, leading to an considerable size increase. We therefore use a simpler notion of NWAs this thesis, where the symbolic description are reduced to else rules.

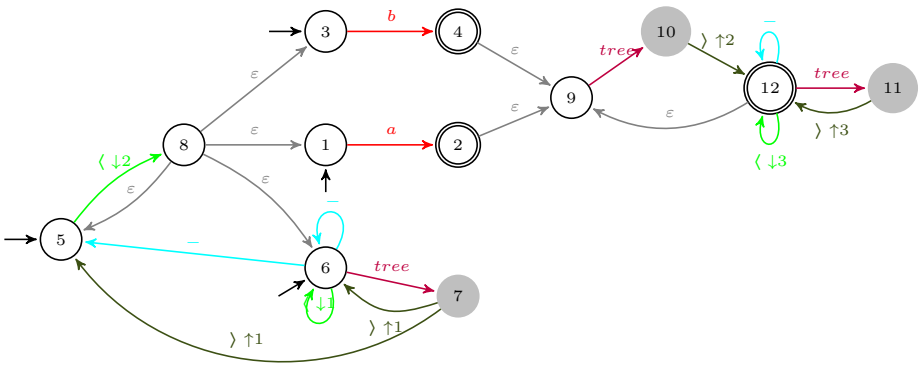


Figure 2.5: Nested word automaton $nwa(ch^*(a+b))$.

Consider a query with the query variables in $\{x, y, z\}$. The complex label of a *book*-node that is selected by variables x can then be described by:

$$elem\&book\&\_\&\{V \subseteq \{x, y, z\} \mid x \in V\}$$

An example for a complex label satisfying this descriptor is the tuple $(elem, book, inria, \{x, z\})$. We now consider this complex label as the word $elem.book.inria.x.\neg y.z$ while imposing an order on the variables. The above descriptor can then be replaced by the following regular expression, where _ may stands for arbitrary letters:

$$elem.book.\_.x.(y + \neg y).(z + \neg z)$$

Such descriptors can be compiled quite naturally to an NFA with else rules.

An example for an NWA is given in a graphical syntax in Figure 2.5. Tree states are drawn in circles that are filled in light gray (q), while hedge states are in unfilled circles (q). Initial states are drawn as →(q) and final states as ((q)). Hedge rules that have the form $(q_1, q_2) \in o^\Delta$ where $o \in \Sigma \cup \{\_, \varepsilon, tree\}$ are denoted by $q_1 \xrightarrow{o} q_2$. They are either label, else, epsilon, or tree rules depending of the type of letter o. Opening rules $(q_1, q_2) \in \langle^\Delta_\gamma$ are represented as $q_1 \xrightarrow{\langle\downarrow\gamma} q_2$ and closing rules $(q_1, q_2) \in \rangle^\Delta_\gamma$ as $q_1 \xrightarrow{\rangle\uparrow\gamma} q_2$.

Our notion of NWA supports factorization in the spirit of [Champavère *et al.* 2009]. It is obtained by distinguishing two types of states $q \in Q_h$ and $p \in Q_t$, and adding explicit type coercion rules $q \xrightarrow{tree} p$. Semantically, both kinds of states could be merged when replacing the type coercion rules by the epsilon rule $q \xrightarrow{\varepsilon} p$, but at the cost of introducing additional nondeterminism. This

may lead to quadratically larger deterministic automata when determinizing, as illustrated in Example 6.

The language of nested words between two states $q_1, q_2 \in Q_h$ is defined as the least language such that:

$$\begin{array}{rcl} L_{q_1,q_2}(\Delta) & = & \{\varepsilon \mid \text{if } q_1 = q_2 \text{ or } q_1 \xrightarrow{\varepsilon} q_2 \in \Delta\} \cup \bigcup_{q_3 \in Q_h} L_{q_1,q_3}(\Delta) \cdot L_{q_3,q_2}(\Delta) \\ & \cup & \{a \mid \text{if } q_1 \xrightarrow{a} q_2 \in \Delta \text{ or } (q_1 \dashrightarrow q_2 \in \Delta \text{ and } \neg\exists q_2'.\ q_1 \xrightarrow{a} q_2' \in \Delta)\} \\ & \cup & \{\langle h \rangle \mid \exists q_1', q_2' \in Q_h . \exists q_3 \in Q_t . \exists \gamma \in \Gamma.\ q_1 \xrightarrow{\langle\downarrow\gamma} q_1', h \in L_{q_1',q_2'}(\Delta), \\ & & \qquad q_2' \xrightarrow{tree} q_3 \in \Delta \text{ and } q_3 \xrightarrow{\rangle\uparrow\gamma} q_2 \in \Delta\}. \end{array}$$

The language of the NWA then is $L(\mathcal{N}) = \bigcup_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(\Delta)$.
Furthermore, we write NWA_Σ to denote the set of NWAs over Σ.

Definition 8 *An NWA is* deterministic *if I is a singleton or empty, ε^Δ is empty, for all $a \in \Sigma$ a^Δ and $\_^\Delta$ are partial functions from Q_h to Q_h, for all $q \in Q_h$ and $\gamma \in \Gamma$ there exists a most one $q' \in Q_h$ such that $q' \in \langle_\gamma^\Delta$, and for all $\gamma \in \Gamma$, $\rangle_\gamma^\Delta$ is a partial function from Q_h to Q_t.*

We determinize our NWAs by adapting the determinization procedures presented in [Debarbieux *et al.* 2015, Alur & Madhusudan 2009], while taking into account hedge ending and else rules, and generating only accessible states. Given an NWA $\mathcal{N} = (Q_h, Q_t, \Sigma, \Gamma, \Delta, I, F)$, the difficulty is to deal with concurrent opening rules $q \xrightarrow{\langle\downarrow\gamma_1} q_1$ and $q \xrightarrow{\langle\downarrow\gamma_2} q_2$ in Δ during determinization without mixing up the stack symbols γ_1 and γ_2. Therefore, we use binary relations as states of the determinized automaton $det(\mathcal{N}) = (Q_h^{det}, Q_t^{det}, \Sigma, \Gamma^{det}, \Delta^{det}, I^{det}, F^{det})$, that is $Q_h^{det} = 2^{Q_h \times Q_h}$, $Q_t^{det} = 2^{Q_h \times Q_t}$. The only initial state is the identity relation id_I which relates all initial states of $\mathcal{N}$ to themselves, i.e., $I^{det} = \{id_I\}$. The set of final states is $F^{det} = \{\tau \in Q_h^{det} \mid \tau \cap (I \times F) \neq \emptyset\}$. Schemas generating the transition rules in Δ^{det} are given in Figure 2.6. For a relation $\tau \in Q_h^{det} \cup Q_t^{det}$, we write $lab(\tau) = \{a \in \Sigma \mid \exists (q, q') \in \tau, q'' \in Q . q' \xrightarrow{a} q''$ wrt. $\Delta\}$. We also write $\tau \circ \tau'$ to define the relational composition of τ and τ', i.e $\tau \circ \tau' = \{(q_1, q_3) \mid \exists q_2 \in Q_h\ .(q_1, q_2) \in \tau$ and $(q_2, q_3) \in \tau'\}$. These schemas generate transition rules for the accessible relations only.

Example 6 *The NWA of Figure 2.7 is obtained by determinization. Here factorization avoids a quadratic blow up. This can be observed at state* 14*, which has* 3 *incoming tree-edges and* 10 *outgoing closing edges. Without factorization, the* 3 *tree edges could be replaced by* 3 *ε-edges whose elimination during the determinization would have produced* 30 *closing edges.*

$$\frac{\tau \in Q_h^{det}}{\tau \Rightarrow \tau \circ \_^{\Delta} \in \Delta^{det}} \qquad \frac{\tau \in Q_h^{det} \quad Q' = \{q' \mid \exists(\_, q) \in \tau.\ q \xrightarrow{\langle\downarrow\gamma} q' \in \Delta\}}{\tau \xrightarrow{\langle\downarrow\tau} id_{Q'} \in \Delta^{det}}$$

$$\frac{\tau \in Q_h^{det}}{\tau \xrightarrow{tree} \tau \circ tree^{\Delta} \in \Delta^{det}} \qquad \frac{\tau \in Q_t^{det} \quad \langle\tau\rangle^{\Delta} = \cup_{\gamma\in\Gamma}\langle_{\gamma}^{\Delta} \circ \tau \circ \rangle_{\gamma}^{\Delta}}{\tau \xrightarrow{\rangle\uparrow\tau'} \tau' \circ \langle\tau\rangle^{\Delta} \in \Delta^{det}}$$

$$\frac{\tau \in Q_h^{det} \quad a \in lab(\tau) \quad \tau' = \{(q, q') \in \_^{\Delta} \mid \nexists q''.q \xrightarrow{a} q'' \in \Delta\}}{\tau \xrightarrow{a} \tau \circ (a^{\Delta} \cup \tau') \in \Delta^{det}}$$

Figure 2.6: Determinization of NWAs.

Regarding intersections – and unions –, products of NWAs are defined analogously to NFAs, with the stack symbols in addition.

We finally introduce some notations for referring to sets of NWAs. The set of dNWAs over Σ is denoted by dNWA_Σ. A class of NWAs is a function that maps any alphabet Σ to a set of NWAs over Σ. We define the classes NWA and dNWA that map any alphabet Σ with NWA_Σ, respectively dNWA_Σ.

2.4.4 Nested Regular Expressions

We present nested regular expressions (NREs), that were introduced under the name *regular expression types* in the context of XDuce [Hosoya & Pierce 2003] up to minor details. A NRE over alphabet Σ has the following abstract syntax:

$$E, E' ::= \varepsilon \mid a \mid \neg\Sigma' \mid \emptyset \mid E \cdot E' \mid E + E' \mid E\&E' \mid E^* \mid \overline{E} \mid \langle E\rangle \mid \mu a.E$$

where $a \in \Sigma$ and $\Sigma' \subseteq \Sigma$ is finite. We restrict the recursive expressions $\mu a.E$ such that all occurrences of a in E are nested below parentheses. The sets of free and bound symbols $fn(E)$ and $bn(E)$ are defined as usual where $\mu a.E$ binds symbol a with scope E and there is no other binder.

Compared to the regular expression types in [Hosoya & Pierce 2003], there are two differences. First, our NREs treat labels as internal symbols instead of labels of parentheses. Second, they provide recursion through the μ-operator instead of using recursive equation systems. Even though not needed from the view point of expressiveness, we allow conjunctions $E\&E'$ to simplify the compilation of CoreXPath expressions with filters to NREs. NREs having no subexpressions $E\&E'$ are called conjunction-free (CF-NREs). Any NRE describes a language of nested words

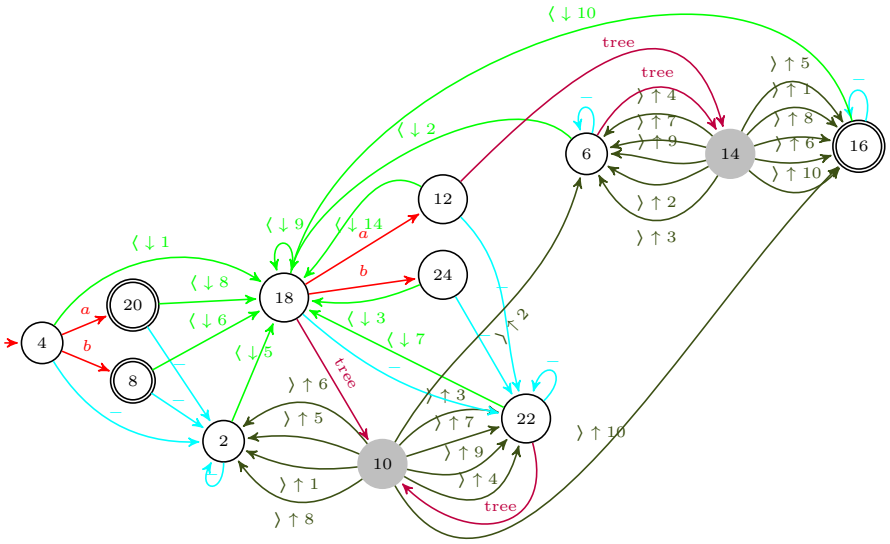


Figure 2.7: Determinized NWA with factorization

that is defined by structural induction as follows:

$$
\begin{array}{ll}
L(\varepsilon) = \{\varepsilon\} \qquad L(a) = \{a\} & L(\neg\Sigma') = \Sigma \setminus \Sigma' \\
L(E \cdot E') = L(E) \cdot L(E') & L(\overline{E}) = nWords_\Sigma \setminus L(E) \\
L(E + E') = L(E) \cup L(E') & L(E\&E') = L(E) \cap L(E') \\
L(\langle E \rangle) = \{\langle h \rangle \mid h \in L(E)\} & L(\mu a.E) = \cup_{n \geq 0} L(\mu^n a.E) \\
L(E^*) = L(E)^* & L(\emptyset) = \emptyset
\end{array}
$$

A negation $\neg\Sigma'$ stands for $\Sigma \setminus \Sigma'$. This is useful for dealing with infinite alphabets and with large finite alphabets. For all expressions E, E_1 and E_2, the notation $E[E_1/E_2]$ stands for the expression E where all the occurrences of E_1 have been replaced by E_2. The semantics of a μ-operator is then defined using the shortcuts $\mu^0 a.E = E[a/\emptyset]$ and $\mu^n a.E = E[a/\mu^{n-1} a.E]$ for all $n \geq 1$. Note that $\mu a.\, b \cdot a \cdot c + \varepsilon$ would define the string language $\{b^n \cdot c^n \mid n \geq 0\}$ which is not regular. But this expression is ruled out since the μ-bound name a is not nested below parentheses.

In the context of XML queries, we can express the child and descendant-or-self

axes of XPath expressions by using the following NREs:

$$\begin{array}{lll} ch(E) & =_{\text{df}} & T \cdot \langle E \rangle \cdot T \qquad T =_{\text{df}} \mu x.\ (\langle x \rangle + \neg\emptyset)^* \\ ch^*(E) & =_{\text{df}} & \mu x.\ (E + ch(x)) \qquad \text{where } x \notin fn(E) \\ ch^+(E) & =_{\text{df}} & \mu x.\ (ch(E) + ch(x)) \qquad \text{where } x \notin fn(E) \end{array}$$

Thereby, the XPath expression $a[\mathit{following\text{-}sibling}::b]/\mathit{descendant}::c$ can be expressed as a NRE, in which $x \in \Sigma$ serves as the selection variable, while the negation $\neg\{x\}$ expresses nonselection.

$$\langle \mathit{elem} \cdot a \cdot \neg\{x\} \cdot ch^+(\langle \mathit{elem} \cdot c \cdot x \cdot T \rangle) \rangle \cdot T \cdot \langle \mathit{elem} \cdot b \cdot \neg\{x\} \cdot T \rangle \cdot T$$

Our next objective is to distinguish NREs that can be evaluated deterministically in polynomial time, for instance by compilation to deterministic NWAs. For this, we consider the language of NREs $\mathit{nregexp}(ch, T)$ extended by the constant T and the unary constructor ch.

Definition 9 *An expression of $\mathit{nregexp}(ch, T)$ is deterministic if it does not contain a subexpression of any of the forms: $E_1 + E_2$, E^*, $T \cdot E$, $\mu a.E$.*

Note in particular that $ch(a)$ is a deterministic expression of $\mathit{nregexp}(ch, T)$. In contrast, the semantically equivalent expression $T.\langle a \rangle.T$ is not deterministic. Similarly, T is deterministic while the equivalent expression $\mu x.(\langle x \rangle + \neg\emptyset)^*$ is not. The expression $ch^*(E)$ is not deterministic since its definition relies on the μ-operator.

NREs have the same expressiveness. We next discuss on a compiler from expression an E of $\mathit{nregexp}(T, ch)$ to an NWA $\mathit{nwa}(E)$ that preserves determinism.

For instance, the NWA for the expression $ch^*(a + b)$ is shown in Figure 2.5. For regular expressions without nesting, the compiler is based on Glushkov's construction recursively on the structure of the expression while eliminating ε-edges on the fly. Such construction is known to preserve determinism [Brüggemann-Klein & Wood 1998]. For deterministic expressions $ch(E)$, we adapt ideas from [Debarbieux *et al.* 2015]. As for conjunctions, product of automata are used. It remains to discuss the compilation of expressions $\mu a.E$. We first note that we can assume w.l.o.g. that a occurs at most once in E by using the golden lemma of the μ-calculus [Arnold & Niwiński 2001], stating for all names $a_1, \ldots, a_n$ and expressions E'' in which $a_1, \ldots, a_n$ can appear free that $\mu a_1 \ldots . \mu a_n.\ E'' \equiv \mu a.E''[a_1/a, \ldots, a_n/a]$. Our construction guarantees that all transitions of the form $q \xrightarrow{a} q'$ in $\mathit{nwa}(E)$ will start with the same state q. A natural construction would remove the transitions $q \xrightarrow{a} q'$ from $\mathit{nwa}(E)$ and add ε-rules from q to all the initial states of $\mathit{nwa}(E)$, and from all final states of $\mathit{nwa}(E)$ to q'. Un-

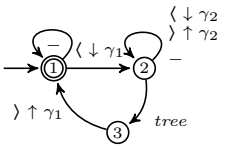


Figure 2.8: Typing NWA

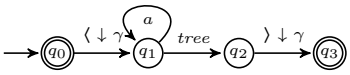


Figure 2.9: Automaton for the $\langle a^* \rangle$ expression.

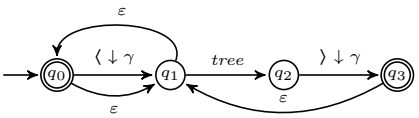


Figure 2.10: Bad automaton for $\mu a.\langle a^* \rangle$

fortunately, the resulting automaton would not be correct. Correctness is achieved by *typing* the states of the automaton by the NWA $\mathcal{N}$ presented on Figure 2.8. Automaton $\mathcal{N}$ evaluates in state 2 all (sub)-trees, and in state 1 all non nested parts of its input. Let $\mathcal{P}$ be the product automaton of $nwa(E)$ and $\mathcal{N}$, in which we remove transition $(q, 2) \xrightarrow{a} (q', 2)$ and add ε-rules from state $(q, 2)$ to all states in $I \times \{2\}$, and from all states in $F \times \{2\}$ to $(q', 2)$, where I and F are respectively the set of initial and final states of $nwa(E)$. Then $\mathcal{P}$ recognizes $L(\mu a.E)$.

Example 7 *For illustration, we consider the* NRE $E = \mu a.\langle a^* \rangle$ *whose NWA is given in Figure 2.9. Simply adding epsilon edges to capture the operator μa will not work though. It will lead to the wrong automaton in Figure 2.10. This automaton will wrongly accept the hedge $\langle\rangle\langle\rangle$, since this hedge does not belong to $L(E)$.*

Figure 2.11 illustrates the product automaton for E obtained by our construction, where only accessible states are kept.

The correctness would fail if permitting $\mu a.\ b \cdot a \cdot c$. We can only sketch the correctness argument. It depends on that the μ-bound name a must appear below parenthesis in E. Therefore, $nwa(E)$ has a stack symbol γ that will be pushed on all paths from some initial state leading to q, and popped on all paths from q' to some final state. Thus, no successful run of $nwa(\mu a.E)$ may use an added ε-edge

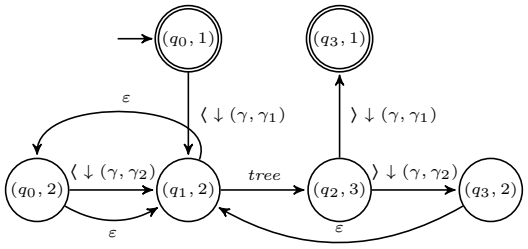


Figure 2.11: Good automaton for $\mu a.\langle a^* \rangle$

from some final state leading to q' before using an added ε-edge from some initial state leading to q. Regarding complexity, a naive implementation would lead to automaton $\mathcal{P}$ with size exponential in the number of nested μ-expressions. The exponential blow-up is avoided as instead of constructing the product by $\mathcal{N}$ for every μ-expression, it is enough to introduce two versions of every newly introduced state with *types* 1 and 2, respectively.

Theorem 1 *For any CF-*NRE E *where all the subexpressions* $\overline{E'}$ *are such that* E' *is deterministic, we can construct in time* $O(|E|^2)$ *an NWA* A *while preserving determinism such that* $L(A) = L(E)$.

This quadratic time result generalizes on a previous result for the Glushkov construction [Brüggemann-Klein 1993]. Because of automata products used to build them, NREs having conjunctions may in the worst case yield NWAs with an exponential size. Furthermore, having a subexpression $\overline{E}$ where is not deterministic necessarily leads to the determinization of the automaton for E – before complementing it –, which could induce an exponential blow-up. As a consequence of Theorem 1 small deterministic CF-NREs where the complementation operation is not used above nondeterministic expressions can be compiled to small deterministic NWAs.

2.5 FXP

Fxp is a fragment of the λXP language defined in [Sebastian 2016]. It is used as a core language for compiling forward CoreXPath, extended with string comparisons. In this thesis, we use a lighter version of Fxp, where the *following* axis is omitted. We'll see in Section 3.2 the reason of this restriction.

The version of Fxp that we consider has the abstract syntax presented in Figure 2.12. It is parameterized by an alphabet Σ, a finite set Υ of words, a set of query variables $\mathcal{W}' \subseteq \mathcal{W}$ and a set $\mathcal{L}$ of label properties ℓ that can be interpreted over positions. Besides of the usual boolean connectives, it defines formulas with axes for testing the existence of paths, and label formulas for testing whether a label property – or a variable assignment – is satisfied. It also defines the string comparisons equals_v, $\mathit{contains}_v$, $\mathit{starts\text{-}with}_v$ and $\mathit{ends\text{-}with}_v$ for all word $v \in \Upsilon$.

The models of Fxp formulas are nested words $w \in \mathit{nWords}_\Sigma$ seen as relational structures, with complex labels that define the label properties ℓ^w for all $\ell \in \mathcal{L}$. We consider that the properties lab_a for all $a \in \Sigma$ are included in $\mathcal{L}$. The nested words w must also define unary predicates equals_v^w, $\mathit{contains}_v^w$, $\mathit{starts\text{-}with}_v^w$ and $\mathit{ends\text{-}with}_v^w$ for all word $v \in \Upsilon$. A position π and a partial function from $\mathcal{W}'$ to the set of positions

Formulas	F	::=	$F \wedge F \mid F \vee F \mid \neg F \mid \mathit{true} \mid A(F) \mid B(F) \mid O_v$
Forward Axes	A	::=	$\mathit{ch} \mid \mathit{ds} \mid \mathit{fs}$
Label Formulas	B	::=	$\mathit{is}_x \mid \ell \mid B\&B$
Comparisons	O	::=	$\mathit{equals} \mid \mathit{contains} \mid \mathit{starts\text{-}with} \mid \mathit{ends\text{-}with}$

Figure 2.12: Abstract syntax of the restricted Fxp language, where $x \in \mathcal{W}'$, $v \in \Upsilon$ is a word and $\ell \in \mathcal{L}$

$$[\![\mathit{true}]\!]_{w,\pi,\alpha} =_{\text{df}} 1$$

$$[\![\mathit{is}_x(F)]\!]_{w,\pi,\alpha} =_{\text{df}} \begin{cases} [\![F]\!]_{w,\pi,\alpha} & \text{if } \alpha(x) = \pi \\ 0 & \text{otherwise} \end{cases}$$

$$[\![\ell(F)]\!]_{w,\pi,\alpha} =_{\text{df}} \ell^w(\pi) \wedge [\![F]\!]_{w,\pi,\alpha}$$

$$[\![O_v]\!]_{w,\pi,\alpha} =_{\text{df}} \begin{cases} 1 & \text{if } \pi \in O_v^w \\ 0 & \text{otherwise} \end{cases}$$

$$[\![F \wedge F']\!]_{w,\pi,\alpha} =_{\text{df}} [\![F]\!]_{w,\pi,\alpha} \wedge [\![F']\!]_{w,\pi,\alpha}$$

$$[\![F \vee F']\!]_{w,\pi,\alpha} =_{\text{df}} [\![F]\!]_{w,\pi,\alpha} \vee [\![F']\!]_{w,\pi,\alpha}$$

$$[\![\neg F]\!]_{w,\pi,\alpha} =_{\text{df}} \neg [\![F]\!]_{w,\pi,\alpha}$$

$$[\![A(F)]\!]_{w,\pi,\alpha} =_{\text{df}} \exists \pi'.A^w(\pi,\pi') \wedge [\![F]\!]_{w,\pi',\alpha}$$

$$[\![B\&B'(F)]\!]_{w,\pi,\alpha} =_{\text{df}} [\![B]\!]_{w,\pi,\alpha} \wedge [\![B']\!]_{w,\pi,\alpha} \wedge [\![F]\!]_{w,\pi,\alpha}$$

Figure 2.13: Semantics of Fxp formulas

of the nested word are also part of the model. Thus, a formula F is evaluated to a boolean with respect to a nested word w over Σ, a node $\pi \in \mathit{Pos}(w)$ and a variable assignment $\alpha : \mathcal{W}' \to \mathit{Pos}(w)$. We write $[\![F]\!]_{w,\pi,\alpha}$ to denote its semantics, given in Figure 2.13.

Chapter 3

Small Deterministic Automata for Navigational Queries

Contents

3.1 Introduction

Whether an answer is certain is computationally hard for tiny syntactic fragments of CoreXPath [Benedikt *et al.* 2008, Gauwin & Niehren 2011], but can be done in polynomial time for queries defined by deterministic NWAs [Gauwin *et al.* 2009]. A natural question is therefore, whether it is possible to compile CoreXPath queries as in the XMark benchmark of [Franceschet] to deterministic NWAs of reasonable size. Unfortunately, the existing compilers fail to do so [Debarbieux *et al.* 2015], since they are based on NWA determinization for dealing with disjunction, negation, and recursive steps. Thereby they produce huge deterministic automata even for

very simple CoreXPath queries from the benchmark, or do not terminate after some hours.

In this chapter, we consider NREs for defining queries on nested words, since there exist compilers that can map the CoreXPath queries to NREs of reasonable size, under the condition that the path query contains only forwards steps.

However, the NREs obtained by compilation from CoreXPath queries are rarely deterministic, so neither are the NWAs obtained from them by direct compilation. Neither can we apply NWA determinization to them as argued above. We show that deterministic NWAs can be obtained nevertheless based on stepwise hedge automata (SHAs), that we introduce. SHAs combine stepwise tree automata [Carme *et al.* 2004] for unranked trees with finite state automata on words (NFAs). They can be determinized in a bottom-up and left-to-right manner, simply by combining the determinization procedures for tree automata and for NFAs. Furthermore, we can compile deterministic SHAs to deterministic NWAs in polynomial time. Conversely, NWAs can be compiled to SHAs in polynomial time too, but at the cost of introducing nondeterminism.

By composing these compilers and determinization algorithms, NREs can be compiled to deterministic NWAs in the following two manners. The first method is to compile the NRE to an SHA, from there to an NWA, which is then determinized. The second way consists of compiling the NRE to an SHA, determinize it, and convert the result to a deterministic NWA. In an experimental study, we consider a collection of NREs that we constructed automatically from CoreXPath queries in the XMark benchmark. It turns out a little surprisingly that both above algorithms yield a satisfactory solution: they produce small deterministic NWAs for all NREs in our collection. The sizes of the deterministic may differ, sometimes in favor of the one or the other algorithm. We also discuss, why the NWA determinization behaves reasonably for the NWAs obtained from SHAs, while it behaved so badly for NWAs obtained directly from NREs. The reason seems to be that the former NWAs in contrast to the latter have the single entry property, which basically states that the NWA performs all its work in a bottom-up and left-to-right manner, and none when moving top-down. This conjecture is supported by practical evidence rather than some formal statement.

3.2 From FXP to Nested Regular Expressions

We show in this section that FXP formulas can be compiled to NREs expressions. These expressions would then recognize languages of sequenced $\mathcal{W}$-structures.

Let Σ be an alphabet, Υ a finite set of strings, $\mathcal{W}' \subseteq \mathcal{W}$ a set of query variables

$$
\begin{array}{ll}
enc(\ell) = exp(\ell) & enc(ds(F)) = ch^{+}(enc(F)) \\
enc(\ell \& \ell') = enc(\ell) \& enc(\ell') & enc(ch(F)) = \langle anylabvar_{\Sigma,\mathcal{W}'} \cdot T \cdot enc(F) \cdot T \rangle \cdot T \\
enc(O_v) = exp(O_v) & enc(F \wedge F') = enc(F) \& enc(F') \\
enc(ns(F)) = \langle T \rangle \cdot enc(F) & enc(F \vee F') = enc(F) + enc(F') \\
enc(fs(F)) = T \cdot enc(F) & enc(\neg F) = \overline{enc(F)}
\end{array}
$$

Figure 3.1: FXP to NREs compiler for all $\ell, \ell' \in \mathcal{L}$, $v \in \Upsilon$ and FXP formulas F, F'

and $\mathcal{L}$ a set of label properties ℓ, that is assumed to contain at least all the lab_a for $a \in \Sigma \cup \mathcal{W} \cup \mathcal{W}'$. We convert any FXP F formula to a nested regular expression $enc(F)$ recognizing sequenced $\mathcal{W}'$-structures. Our compiler has two parameters:

- a function *exp* that associates any label property and string comparison with a nested regular expression
- and a constant nested regular expression $anylabvar_{\Sigma,\mathcal{W}'}$ that is used as a wild-card for labels and variables.

The FXP to NREs compiler is given in Figure 3.1. The parameters indicate how the label properties, variables and string comparisons are represented in the sequenced $\mathcal{W}'$-structures.

It should be noticed that the nested regular expressions that are obtained by compilation are not enough. Indeed, their languages may accept elements that are not canonic (see Lemma 1). Furthermore, there should be a *schema* indicating clearly how nested words with complex labels should be encoded into sequenced $\mathcal{W}'$-structures. For this reason, the NREs expressions obtained in Figure 3.1 should be intersected with other nested regular expressions that allow to restrict their languages to sets of $\mathcal{W}'$-structures correctly encoded.

Example 8 *Consider the case of* FXP *queries on* XML *documents. Nodes of* XML *documents have, in addition to their labels, properties such that their type. An* XML *document can thus be considered as a nested word with complex labels, where the properties added to the structures are about the types of the* XML *nodes.*

Let $\Sigma = \{temp\}$, $\Upsilon = \{40\}$, $\mathcal{W}' = \{x\}$. *The following* FXP *query selects all the positions of type* element *and labeled by temp, and having a child position of type* text *labeled by* 40*:*

$$F = elem \& temp \& is_x \wedge ch(text(equals_{40}))$$

An example of nested word that satisfies this property is the tree $w = \langle temp \langle 40 \rangle \rangle$

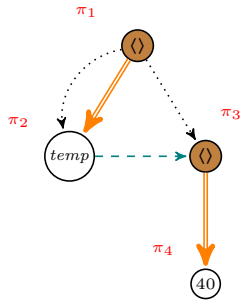


Figure 3.2: Initial structure

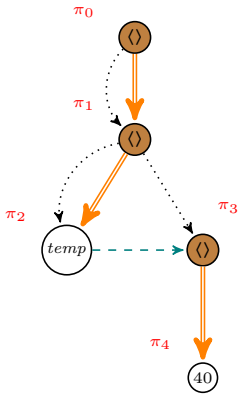


Figure 3.3: With fake root node

(represented in Figure 3.2) over $\Sigma \cup \Upsilon$ *together with its position* π_1 *and the candidate* α *that maps* x *to* π_1.

To express the fact that its node π_1 *is of type element and that its node* π_3 *is of type text, we define the properties* $elem^w = \{\pi_1\}$ *and* $text^w = \{\pi_3\}$. XML *also imposes trees to have a* fake root *of type* document*, different from the actual root of the tree. So we add a new node* π_0*, set it as the new root –* π_1 *becomes the child of* π_0 *– and add the property* $doc^w = \{\pi_0\}$. *This is illustrated in Figure 3.3.*

The information about the type of the node appears only at the logical level. We can make it appear in the nested word itself, by creating a new nested word where the XML *types are added to the labels. An example for this would be the nested word* $w' = \langle doc\langle elem \cdot temp\langle text \cdot 40\rangle\rangle\rangle$ *over the alphabet* $\Sigma' = \{doc, elem, temp, text, 40\}$. *Moreover, given a candidate* $\alpha\colon \mathcal{W}' \to LPos_\Sigma(w')$*, we can obtain a sequenced* $\mathcal{W}$*-structure where positions labeled by the elements of* $\mathcal{W}'$ *– and* $\neg\mathcal{W}'$ *– can only be found after positions labeled by elements of* Σ. *For instance, let* α' *be the candidate that maps* x *to* π'_1. *The sequenced* $\mathcal{W}'$*-structure* $w' \star \alpha' = \langle doc\langle elem \cdot temp \cdot x \cdot \langle text \cdot 40\rangle\rangle\rangle$ *represented in Figure 3.4 could be in the language of the nested regular expressions into which* F *will be compiled.*

However there may be many other sequenced $\mathcal{W}'$*-structures into which* w *is converted. In order to have only one possible correspondence for a given nested word, we set a nested regular expression defining the set of accepted regular expressions. This will limit the number of correspondences to one. The expression obtained by compilation of* F *will then be intersected – by the conjunction operator – with the schema expression. The schema expression must also ensure the canonicity of the accepted* $\mathcal{W}'$*-structure, as expressed in Lemma 1.*

We show an example of definitions of nested regular expressions for the parameters of F*:*

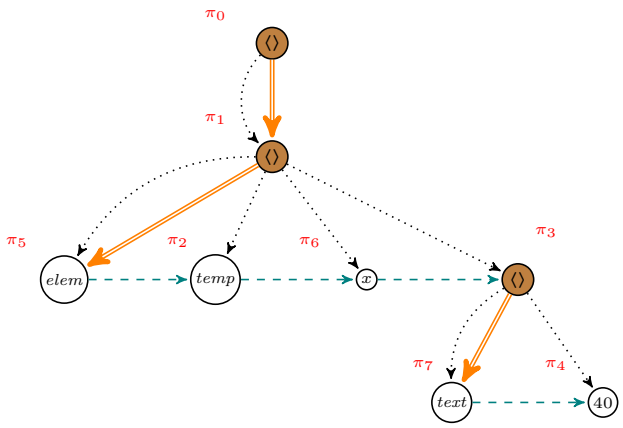


Figure 3.4: With properties expressed as labels

- $exp(elem) = \langle elem \cdot \_ \cdot \_ \cdot T \rangle \cdot T$
- $exp(temp) = \langle \_ \cdot temp \cdot \_ \cdot T \rangle \cdot T$
- $exp(text) = \langle text \cdot \_ \cdot T \rangle \cdot T$
- $exp(equals_{40}) = \langle text \cdot \_ \cdot 4 \cdot 0 \rangle \cdot T$
- $exp(x) = \langle \_ \cdot \_ \cdot x \cdot T \rangle \cdot T$
- $anylabvar_{\Sigma,\mathcal{W}'} = \_ \cdot \_ \cdot \_ \cdot + \_ \cdot \_$

where $\_$ *stands for* $\neg\emptyset$*, that is any symbol in* $\Sigma \cup \mathcal{W}' \cup \neg\mathcal{W}'$*. Note that the languages of these* NREs *expressions contain nested words that are not sequenced* $\mathcal{W}'$*-structures. To avoid this, we can take the conjunction of the* NREs *expressions obtained from* FXP *formulas with another* NREs *that defines the language of sequenced* $\mathcal{W}'$*-structures.*

To establish the soundness of this compilation, we need to define the notion of hedge rooted at some node. For any nested word w over some alphabet Σ and node $\pi \in Nodes(w)$, the hedge of w rooted by π, written $hedge(w \star \alpha)_{\geq\pi}$, is the nested word restricted to the portion of w where the positions are

- π, or
- descendant of π, or
- following siblings of π, or
- descendant of the following siblings of π.

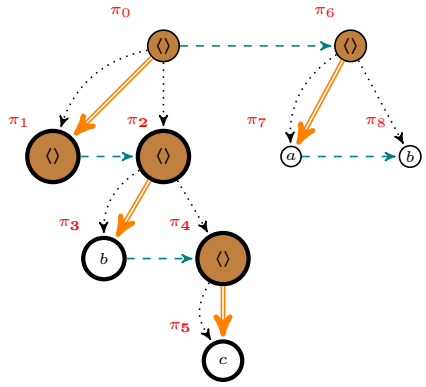


Figure 3.5: Restriction of a nested word to a hedge

Example 9 *The restriction of the nested word in Figure 3.5 rooted by π_1 is the hedge nested word where the circles representing the positions have thick borders.*

Proposition 1 (Soundness) *Any* Fxp *formula F can be converted to a nested regular expression E over $\Sigma \cup \mathcal{W}' \cup \neg\mathcal{W}'$ such that for all nested word $w \in nWords_\Sigma$, position $\pi \in Pos(w)$ and candidate $\alpha : \mathcal{W}' \to Pos(w)$, $[\![F]\!]_{w,\pi,\alpha} = true$ iff $hedge(w \star \alpha)_{\geq\pi} \in L(E)$.*

Given a position π of a hedge, the *following* axis of Fxp considers not only the descendant of the following sibling of π, but also the descendant of the following-sibling of the ancestors of π – where an ancestor is a parent, or a parent of a parent, etc. The structures formed by such set of nodes are unfortunately not nested words. In Figure 3.5 for instance, adding the nodes π_6, π_7 and π_8 to the restriction rooted by π_2 would lead to something which is not a nested word. For this reason, we ruled out the *following* axis in the version of Fxp that we consider.

3.3 Stepwise Hedge Automata (SHAs)

We propose SHAs as an extension of stepwise tree automata [Carme *et al.* 2004] to recognize not only unranked trees but also hedges and generally nested words.

Our notion of SHAs will be symbolic in using else rules, and factorized in the sense of [Champavère *et al.* 2009]: there are two types of states for hedges and trees and an operator for explicit type coercion. We also propose a novel treatment of internal letters inspired by nested word automata, so that SHAs generalize both on stepwise tree automata and on Nfas.

Definition 10 *A SHA is a tuple $S = (Q_h, Q_t, \Sigma, \Delta, I, F)$ such that Q_t and Q_h are finite set of tree states respectively hedge states, $\Omega \subseteq Q_t$, Σ an alphabet of internal*

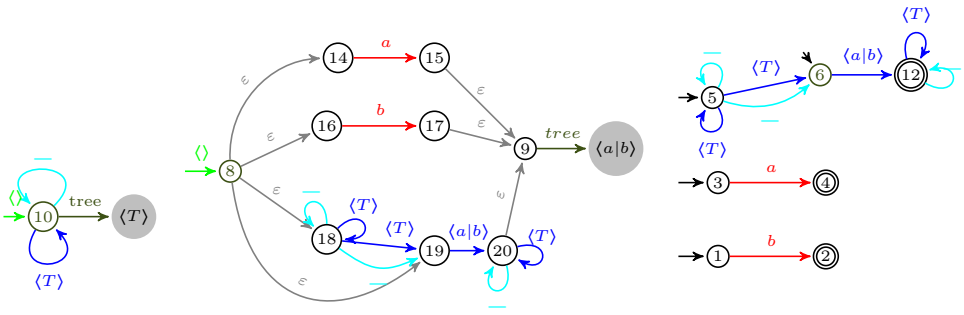


Figure 3.6: Stepwise hedge automaton $sha(ch^*(a+b))$: the stepwise tree automaton part is on the left and middle, and the NFA part on the right.

letters (that may be infinite), $I, F \subseteq Q_h$ *subsets of initial and final states respectively, and* Δ *a finite set of*

tree initial rules	$\langle\rangle^{\Delta} \subseteq Q_h$
tree final rules	$q \xrightarrow{tree} p$
named internal rules	$q \xrightarrow{a} q'$
else internal rules	$q \xrightarrow{=} q'$
epsilon rules	$q \xrightarrow{\varepsilon} q'$
apply rules	$q \xrightarrow{p} q'$

for $a \in \Sigma$, $q, q' \in Q_h$ *and* $p \in Q_t$. *All the rules of* Δ *minus the tree initial and the tree final rules constitute the subset of hedge rules of* Δ. *Note that tree initial rules are actually hedge states. For this reason, we also call them* tree initial states *by abuse of language.*

The set of SHAs over Σ is denoted by SHA_Σ.

An example for a SHA is given in graphical syntax in Figure 3.6. It recognizes all hedges which are either just a or b or contain some tree node that contains either just a or b. In the graphical syntax, the states of type tree $q \in Q_t$ are drawn in circles filled in light gray (q), while the states of type hedge $q' \in Q_h$ are drawn in unfilled circles (q'). The right part of the graph is an NFA which uses tree states as additional edge labels, while the left part is a stepwise tree automaton, and defines the tree languages of these tree states.

Let Δ_h be the restriction of Δ to the hedge rules. Then, $(Q_h, \Sigma \uplus Q_t, \Delta_h, I, F)$ is a standard NFA with ε-rules, which is symbolic [D'Antoni & Alur 2014] in providing else rules for dealing with large or infinite alphabets in addition. Therefore, we denote the initial states $q \in I$ by $\to$(q) and the final states $q \in F$ by ((q)). A rule with an internal letter $(q_1, q_2) \in a^{\Delta}$ is denoted by $q_1 \xrightarrow{a} q_2$ wrt Δ stating that a

hedge in state q_1 can be extended by the internal letter a leading to a hedge in state q_2. Similarly, an epsilon rule $(q_1, q_2) \in \varepsilon^\Delta$ is denoted by $q_1 \xrightarrow{\varepsilon} q_2$, and an else rule $(q_1, q_2) \in \_^\Delta$ is denoted by $q_1 \Rightarrow q_2$. In the same spirit, an apply rule $(q_1, q_2) \in q^\Delta$ is denoted by $q_1 \xrightarrow{q} q_2$ wrt. Δ, stating that a hedge in state q_1 can be extended by a tree in state q leading to a hedge in state q_2.

A tree initial state $q \in \langle\rangle^\Delta$ is graphically denoted by $\xrightarrow{\langle\rangle} q$ and a tree final rule $(q, p) \in \mathit{tree}^\Delta$ by $q \xrightarrow{\mathit{tree}} p$.

Any letter $a \in \Sigma$ defines two binary relations over Q_h. The set of pairs (q, q') such that the SHA in state q, reaches q' after having read a, that is

$$a^\Delta = \{(q, q') \mid q \xrightarrow{a} q' \in \Delta\} \cup \{(q, q') \mid q \Rightarrow q' \in \Delta \text{ and } \nexists q'' \in Q_h . q \xrightarrow{a} q'' \in \Delta\}$$

and the set of pairs for which the SHA cannot read a,

$$\neg a^\Delta = \{(q, q') \mid q \Rightarrow q' \in \Delta \text{ and } \exists q'' \in Q_h \mid q \xrightarrow{a} q'' \in \Delta\}.$$

Also, any tree state $p \in Q_t$ defines a binary relation over Q_h, which can be interpreted as the set of pairs (q, q') for which the SHA in state q, can read a tree evaluated in state p before reaching state q', that is

$$@_p^\Delta = \{(q, q') \mid q \xrightarrow{p} q' \in \Delta\}.$$

We define the apply relation as

$$@^\Delta = \bigcup_{p \in Q_t} @_p^\Delta.$$

Intuitively, a tree $\langle h \rangle$ can be evaluated to state $p \in Q_t$ if h can be evaluated starting with some tree initial state $\xrightarrow{\langle\rangle} q_1$ to some state q_2 such that $q_2 \xrightarrow{\mathit{tree}} p$. More formally, the hedge languages $L_{q_1,q_2}(\Delta)$ between any two hedge states $q_1, q_2 \in Q_h$ are defined as follows:

$$\begin{array}{rcl} L_{q_1,q_2}(\Delta) & = & \{\varepsilon \mid \text{if } q_1 = q_2 \text{ or } q_1 \xrightarrow{\varepsilon} q_2 \in \Delta\} \cup \bigcup_{q_3 \in Q_h} L_{q_1,q_3}(\Delta) \cdot L_{q_3,q_2}(\Delta) \\ & \cup & \{a \mid (q_1, q_2) \in a^\Delta\} \\ & \cup & \bigcup_{(q_1,q_2) \in @_p^\Delta} L_p(\Delta) \end{array}$$

This definition is mutually recursive with the definition of the tree languages $L_p(\Delta)$ of all tree states $p \in Q_t$:

$$L_p(\Delta) \;=\; \{\langle h \rangle \mid \exists q \in \langle\rangle^\Delta, q' \in Q_h .\ h \in L_{q,q'}(S),\ q' \xrightarrow{\mathit{tree}} p\}.$$

The hedge language $L(S)$ that is recognized by S is $\bigcup_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(\Delta)$.

We now define the binary relations $acc^{\Delta}_{h\times\kappa} \subseteq Q_h \times Q_\kappa$ for $\kappa \in \{h, t\}$.

The hedge state to hedge state accessibility relation $acc^{\Delta}_{h\times h}$ consists of the set of pairs of hedge states $q, q' \in Q_h$ for which $L_{q,q}(\Delta) \neq \emptyset$. It relates all the hedge states between which some nested word can be read.

The hedge state to tree state accessibility relation $acc^{\Delta}_{h\times t}$ is formed by the pairs of hedge state $q \in Q_h$ and tree state $p \in Q_t$ for which there exists a hedge state $q' \in Q_h$ satisfying the following conditions:

- $q' \xrightarrow{tree} p \in \Delta$
- $L_{q,q'}(\Delta) \neq \emptyset$

Of course, the accessibility relations can be computed efficiently, using a simple reachability algorithm on the graph representing the SHA.

Lemma 2 *For all state type $\kappa \in \{h, t\}$, $acc^{\Delta}_{h\times\kappa}$ can be computed in polynomial time.*

3.3.1 Evaluation on Nested Words

We show how an SHA operates on nested words via the following example.

Example 10 *Let Σ be the alphabet formed by all the characters that can be encoded by UTF-8, and a nested word $w = ac\langle b\rangle$ that we evaluate on the SHA in Figure 3.6, to which we refer by S. Let Δ be the transition relation of S. The evaluation starts at the initial states of S, that is $I = \{1, 3, 5, 6\}$. The only states of I from which the letter a can be read are 3 (because of the internal rule $3 \xrightarrow{a} 4$) and 5 (because of the else rules $5 \rightrightarrows 5$ and $5 \rightrightarrows 6$). Note that the else rule $5 \rightrightarrows 5$ and $5 \rightrightarrows 6$ wouldn't be eligible for reading a if there were an internal rule $5 \xrightarrow{a} q$ for some hedge state q. Now, reading a from states 3 and 5 brings A to states 4, 5 and 6. From these states, the SHA has to read the letter c. This brings it to states 5 and 6 using the else rules $5 \rightrightarrows 5$ and $5 \rightrightarrows 6$ – states 4 and 6 has no outgoing transition. We now have to read the tree nested word $w' = \langle b\rangle$ from states 5 and 6. To evaluate w', the automaton takes some tree initial rule while not forgetting the hedge states – 5 and 6 – in which it was before reading the hedge b and taking a tree rule. The hedge states in tree initial rules are 8 and 10. Reading b from state 10 – using the else rule $10 \rightrightarrows 10$ – doesn't change the state of S, and taking a tree rule brings S to the tree state $\langle T\rangle$. On the other hand, from state 8, S can reach states $14, 16, 18, 19$ with no input by taking epsilon rules. Then it reads b from one of those states and goes into states 17 (from 16), 18 and 19 (from 18). Then from state 17, S takes the epsilon rule and enters state 9, before reading a tree rule into state $\langle a|b\rangle$. Note that neither*

18 *nor* 19 *can reach a tree state while only reading* b*. So the tree states into which* w' *is evaluated are* $\langle T \rangle$ *and* $\langle a|b \rangle$*, that is* $w' \in L_{\langle T \rangle}(\Delta) \cap L_{\langle a|b \rangle}(\Delta)$*. Recall that* S *was in states* 5 *and* 6 *before reading* w'*. From state* 5*, it can read a tree evaluated into the tree state* $\langle T \rangle$ *and reach the states* 5 *and* 6*. From state* 6*, it can read a tree evaluated in state* $\langle a|b \rangle$ *and enters the state* 12*. So* $w \in L_{5,12}(\Delta)$*, and since* 5 *is an initial state and* 12 *a final state,* w *is accepted by* S*.*

3.3.2 Relation to STAs and NWAs

Clearly, STAs can be seen as SHAs with restricted capabilities, recognizing only trees. Any STA can be converted to a SHA by simply adding a tree state q' for each non tree initial state q. The tree state q' will then replace q in all the left hand sides of rules where the latter appears. Finally, the final states are the tree states introduced for the final states of the STA.

Concerning NWAs, they can be converted to SHAs, and vice versa.

3.3.2.1 From SHAs to NWAs

Given a SHA $S = (Q_h, Q_t, \Sigma, \Delta, I, F)$, we can compile it to an NWA $nwa(S) = (Q_h, Q_t, \Sigma, \Gamma, \Delta', I,\ F)$ such that $L_{q_1,q_2}(S) = L_{q_1,q_2}(nwa(S))$. We set $\Gamma = Q_h$, $\_^{\delta} = \_^{\Delta}$, $a^{\delta} = a^{\Delta}$ for all $a \in \Sigma$, $\varepsilon^{\delta} = \varepsilon^{\Delta}$, $tree^{\delta} = tree^{\Delta}$:

$$\frac{q_1 \xrightarrow{q} q_2 \in \Delta \quad p \in \langle\rangle^{\Delta}}{q_1 \xrightarrow{\langle\downarrow q_1} p \in \delta \text{ and } q \xrightarrow{\rangle\uparrow q_1} q_2 \in \delta}$$

Clearly, if S is deterministic then so is $nwa(S)$, since p is unique in this case in particular. One might be tempted to restrict the above construction rule to states p such that $L_q(S[\langle\rangle^{\Delta}/\{p\}]) \neq \emptyset$ where the set of tree initial states $\langle\rangle^{\Delta}$ is replaced by $\{p\}$. However, this would lead to huge blow-up when determinizing these NWAs, basically since this change spoils the single-entry property discussed in Definition 16.

3.3.2.2 From NWAs to SHAs

Conversely, NWAs can be compiled to SHAs, but at the cost of introducing non-determinism, since an NWA may traverse the branches of a tree top-down, while a stepwise must traverse them bottom-up. For this, the stepwise guesses the state in which the NWA will arrive from above and then evaluates the subtree starting with this state, while verifying the correctness of the guess later on. Let $A = (Q_h, Q_t, \Sigma, \delta, I, F)$ be an NWA. We build a SHA $sha(A) = (Q^s_h, Q^s_t, \Sigma, \Delta^s, I^s, F^s)$ where $Q^s_h = Q_h \times Q_h$, $Q^s_t = Q_h \times Q_t$, $I^s = \{(q, q) \mid q \in I\}$, $F^s = I \times F$ and Δ^s

$$\frac{o \in \Sigma \cup \{tree, \_, \varepsilon\} \quad q_1 \xrightarrow{o} q_2 \in \Delta \quad q \in Q_h}{(q, q_1) \xrightarrow{o} (q, q_2) \in \Delta^s} \qquad \frac{q_1 \xrightarrow{\langle\downarrow\gamma} q_2 \in \Delta}{\xrightarrow{\langle\rangle} (q_2, q_2) \in \Delta^s}$$

$$\frac{q_1 \xrightarrow{\langle\downarrow\gamma} q_2 \in \Delta \quad q_3 \in Q_t \quad q_3 \xrightarrow{\rangle\uparrow\gamma} q_4 \in \Delta \quad q \in Q_h}{(q, q_1) \xrightarrow{(q_2,q_3)} (q, q_4) \in \Delta^s}$$

Figure 3.7: NWA to stepwise conversion.

is the smallest satisfying the rule schemas in Figure 3.7. The construction is such that $L(A) = L(sha(A))$.

3.3.3 Determinization

Stepwise hedge automata have a natural notion of determinism, generalizing both on that of stepwise tree automata and on NFAs, in contrast to the earlier notion of hedge automata in [Comon *et al.* 2007, Thatcher 1967].

Definition 11 *A SHA $S = (Q_h, Q_t, \Sigma, \Delta, I, F)$ is* deterministic *if $\langle\rangle^\Delta$ and I are both singletons or empty, ε^Δ is empty, for all $a \in \Sigma$ and $p \in Q_t$, a^Δ, p^Δ, $\_^\Delta$ are partial functions from Q_h to Q_h, and $tree^\Delta$ is a partial function from Q_h to Q_t.*

The set of dSHAs over some alphabet Σ is denoted by dSHA_Σ. Naturally, $\mathrm{dSHA}_\Sigma \subseteq \mathrm{SHA}_\Sigma$. We next show that

Proposition 2 *Any SHA can be made deterministic in at most exponential time such that the hedge language is preserved.*

In a first step we eliminate ε-rules as usual for NFAs in cubic time. Given a stepwise hedge automaton $S = (Q_h, Q_t, \Sigma, \Delta, I, F)$ without ε-rules, we define an equivalent deterministic stepwise hedge automaton $det(A) = (Q_h^{det}, Q_t^{det}, \Sigma, \Delta^{det}, I^{det}, F^{det})$ such that $Q_h^{det} = 2^{Q_h}$, $Q_t^{det} = 2^{Q_t}$, $I^{det} = \{I\}$ and $F^{det} = \{Q' \subseteq Q_h \mid Q' \cap F \neq \emptyset\}$. There is a unique tree initial state in $\langle^{\Delta^{det}} = \{\langle^\Delta\}$ and no ε-rule in $\varepsilon^{\Delta^{det}} = \emptyset$. Furthermore, for all $Q_1 \subseteq Q_h$, and $Q' \subseteq Q_t$, we build new transitions

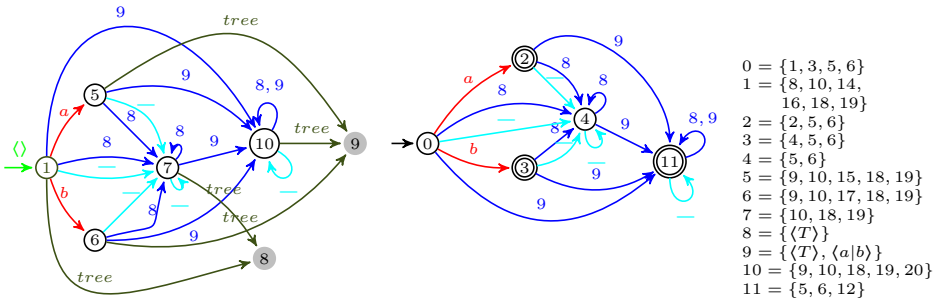


Figure 3.8: The determinized SHA $det(step(ch * (a + b)))$

in Δ^{det} using the following inference rules:

$$\frac{Q_2 = \{q_2 \mid \exists q_1 \in Q_1, q \in Q'.\ q_1 \xrightarrow{q'} q_2 \in \Delta\}}{Q_1 \xrightarrow{Q'} Q_2 \in \Delta^{det}}$$

$$\frac{a \in \Sigma \quad \{q \mid \exists q_1 \in Q_1 . q_1 \xrightarrow{a} q \in \Delta\} \neq \emptyset \quad Q_2 = \{q_2 \mid \exists q_1 \in Q_1 . (q_1, q_2) \in a^{\Delta}\}}{Q_1 \xrightarrow{a} Q_2 \in \Delta^{det}}$$

$$\frac{Q_2 = \{q_2 \mid \exists q_1 \in Q_1,\ q_1 \xrightarrow{tree} q_2 \in \Delta\}}{Q_1 \xrightarrow{tree} Q_2 \in \Delta^{det}} \qquad \frac{Q_2 = \{q_2 \mid \exists q_1 \in Q_1,\ q_1 \Rightarrow q_2 \in \Delta\}}{Q_1 \Rightarrow Q_2 \in \Delta^{det}}$$

We can show for all subsets of hedge states $Q_1, Q_2 \subseteq Q_h$ and subset of tree states $Q' \subseteq Q_t$ that $L_{Q_1,Q_2}(\Delta^{det}) = \bigcup_{q_1 \in Q_1, q_2 \in Q_2} L_{q_1,q_2}(\Delta)$ and that $L_{Q'}(\Delta^{det}) = \bigcup_{q' \in Q'} L_{q'}(S)$. Hence $L(det(S)) = \bigcup_{Q' \in F^{det}} L_{I,Q'}(\Delta^{det})$ and thus $L(det(S)) = \bigcup_{q_1 \in I, q_2 \in F} L_{q_1,q_2}(\Delta) = L(S)$.

For instance, the SHA in Figure 3.8 is obtained by determinization of the automaton in Figure 3.6. It consists of a DFA on the right and a deterministic stepwise tree automaton on the left.

The problematic notion of determinism of the hedge automata from [Martens & Niehren 2007, Comon *et al.* 2007, Thatcher 1967] is avoided by SHAs. The notion of determinism of hedge automata is rather like a notion of unambiguity. And for unambiguous automata, unique minimization fails as well as efficient complementation or universality testing.

3.3.4 Completeness and Pseudo-Completeness

We now introduce the notion of completeness, followed by a refined version of it. Let $S = (Q_h, Q_t, \Sigma, \Delta, I, F)$ be SHA.

Definition 12 *S is called complete if for all hedge state $q \in Q_h$, the following conditions hold:*

1. *for all letter $a \in \Sigma$, there exists $q' \in Q_h$ such that $(q, q') \in a^{\Delta}$*
2. *if q is accessible from a tree initial state $q_{\langle\rangle} \in \langle\rangle^{\Delta}$, there exists a tree state $p \in Q_t$ such that $q \xrightarrow{tree} p \in \Delta$*
3. *for all tree state $p \in Q_t$, there exists a hedge state $q' \in Q_h$ such that $(q, q') \in @_p^{\Delta}$.*

If S is not complete, then it's qualified as *incomplete*. Any SHA can be made complete in polynomial time by first adding new states called *sink states* and new transitions to the sink states.

The notion of completeness is used to ensure that every input can be evaluated by the automaton without ever blocking, that is without being in a situation where no transition can be taken. This is a very important property that we seek for in our approximation algorithms of CQA.

A direct consequence of completeness is that:

Lemma 3 *If S is complete, then $\bigcup_{p \in Q_t} L_p(\Delta) = \mathcal{U}_\Sigma$.*

Deciding whether or not an automaton is complete can be done by just using the syntactical properties of the automaton. However, completing a SHA increases its size. On the other hand, it's a very strong property, since it requires all the possible runs on the nondeterministic SHA to not block. One could have a looser version, as in the closely related notion of *pseudo-completeness*.

Definition 13 *S is called* pseudo-complete *if for all hedge state $q \in Q_h$, the following conditions hold:*

1. *for all letter $a \in \Sigma$, there exists $q' \in Q_h$ such that $(q, q') \in a^{\Delta}$ and*
2. *for all nested word $w \in nWords_\Sigma$, there exists a tree state $p \in Q_t$ and a hedge state $q' \in Q_h$ such that $\langle w \rangle \in L_p(\Delta)$ and $(q, q') \in @_p^{\Delta}$.*

The first condition for pseudo-completeness is completely syntactical. It can be ensured by enforcing every hedge state to have an else rule that leaves it. The

second condition, however, is not syntactical. Nevertheless, it is possible to ensure pseudo-completeness in a purely syntactical manner.

Call $\mathcal{T}$ the automaton represented by the connected component to the left in Figure 3.6. Then

Lemma 4 *S is pseudo-complete if the following conditions hold:*

- *for all $q \in Q_h$ and all letter $a \in \Sigma$, there exists $q' \in Q_h$ such that $(q, q') \in a^{\Delta}$ and*
- *S contains an automaton equivalent to $\mathcal{T}$ modulo renaming of states.*

It is also of interest to notice that:

Lemma 5 *If S is pseudo-complete, then* $\det(S)$ *is complete.*

Actually, a more interesting property is ensured by the presence of $\mathcal{T}$. Indeed, a SHA that contains $\mathcal{T}$ can evaluate any tree nested word without ever blocking.

Lemma 6 *If S contains $\mathcal{T}$, then* $\bigcup_{Q \in Q_t} L_Q(\Delta^{det}) = \mathcal{U}_\Sigma$.

Remark that the stepwise automaton represented by the connected component to the left in Figure 3.8 can evaluate any tree nested word.

It is finally interesting to notice that when a SHA is deterministic and reduced (that is all its tree states are accessible), completeness and pseudo-completeness are equivalent notions. Thus, it is more interesting to determinize pseudo-complete SHAs, as they yield complete dSHAs while having smaller sizes than dSHAs obtained by determinizing complete SHAs.

3.3.5 Universality and Intersection Nonemptiness Problems

We show that the classic decision problems of universality and intersection nonemptiness for SHAs have the same complexity than on the stepwise tree automata – and the classical tree automata.

A class of SHAs is a function that maps any alphabet Σ to a set of SHAs over Σ. We define the classes SHA and dSHA that map any alphabet Σ with SHA_Σ, respectively dSHA_Σ.

Let S be a SHA over some alphabet Σ. The language of S is said *universal* if $L(A) = nWords_\Sigma$. We also say by abuse that S is universal. For any class of SHAs $\mathcal{A} \in \{\mathrm{SHA}, \mathrm{dSHA}\}$, we define the universality problem:

Univ($\mathcal{A}$).
Input: An alphabet Σ and a SHA $S \in \mathcal{A}_\Sigma$
Output: Whether $L(S)$ is universal.

For any class of SHAs $\mathcal{A} \in \{\text{SHA}, \text{dSHA}\}$, the intersection nonemptiness problem is the following.

Inter($\mathcal{A}$).
Input: An alphabet Σ and a finite number of SHAs $S_1, \ldots, S_n \in \mathcal{A}_\Sigma$
Output: Whether $L(S_1) \cap \ldots \cap L(S_n) \neq \emptyset$

The complexities of the different problems presented above are established in the following statements.

Proposition 3 Univ(*SHA*) *is* Exp*-complete while* Univ(*dSHA*) *is in* PTime.

Proof: For the upper bound, remark that deciding the universality of a SHA is equivalent to checking whether the complement of its language is empty. The latter can be done in linear time in the size of the automaton to check. If the automaton is deterministic, one can construct an automaton recognizing the complement of its language in linear time, by just completing the dSHA and switching its final and non final states. It the automaton is nondeterministic, then an exponential algorithm can be obtained by first determinizing the input automaton, complementing it before finally testing whether no final state is accessible.

The lower bound of $univ_{\text{SHA}}$ follows from the fact that the universality problem of stepwise tree automata can be reduced in polynomial time to the universality problem of SHAs. Universality for stepwise tree automata is known to be Exp-hard. □

Proposition 4 Inter(*dSHA*) *is* Exp*-hard, while* Inter(*SHA*) *is in* Exp.

Proof: Recall that the problem of intersection nonemptiness of a sequence of deterministic stepwise tree automata is Exp-hard. The latter problem can be reduced in polynomial time to Inter(dSHA), and gives us the lower bound of Inter(dSHA). An algorithm for Inter(SHA) consists in determinizing the input automata in exponential time, building their product – still in exponential time – before testing whether a final state is accessible. □

3.4 Compiler from nRegExp to SHAs

A *tree (sub)expression* of an NRE E is an expression of the form $E' = \langle E'' \rangle$ appearing in E, where E'' is another subexpression of E. Clearly, languages of tree expressions are tree nested words. Any expression E can be compiled to a SHA $sha(E) = (Q_h, Q_t, \Sigma, \Delta, I, F)$ such that $Q_t = \{E' \mid E' \text{ tree subexpression of } E\}$ and $L_{E'}(\Delta) = L(E')$ for all tree states $E' \in Q_t$. In other words, the tree subexpressions E' of E will be identified with the tree states of $sha(E)$. The SHA $sha(E)$ can be partitioned into disjoint SHAs $sha(E) = A^{top} \cup \bigcup_{E' \in Q_t} A^{E'}$ such that $A^{top} = (Q_h^{top}, Q_t, \Sigma, \Delta^{top}, I, F)$ is called the *top-level automaton* and $A^{E'} = (Q_h^{E'}, Q_t, \Sigma, \Delta^{E'}, \emptyset, \emptyset)$ are *tree subexpressions automata* for all $E' \in Q_t$. The set of tree initial states of the top automaton $\langle\rangle^{\Delta^{top}}$ is set to $\emptyset$. Note that the transitions relation Δ is decomposed thereby into independent connected components. The automaton A^{top} can be identified with an NFA with signature $\Sigma \cup Q_t$ given that it has no tree initial state. The automata $A^{E'}$ are stepwise tree automata that recognize the tree language $L(E')$ when taking E' as final state. For this, they may have tree initial states, but will not have any initial nor final states.

The construction of automaton $sha(E)$ is by induction on the structure of E. We sketch some of the more interesting cases.

Case $E = E' \cdot E''$. We use Glushkov's construction for composing the top-level NFAs of $sha(E')$ and $sha(E'')$. The stepwise tree automata $A^{\langle E''' \rangle}$ of the subexpressions of type tree are preserved. If some subexpression $\langle E''' \rangle$ occurs more than once, then only a single copy of $A^{\langle E''' \rangle}$ is kept.

Case $E = \neg\{a_1, \ldots, a_n\}$. We use an else transition from an initial to a final state in combination with $a_1, \ldots, a_n$-transitions from the initial state into a sink. This is illustrated in Figure 3.9.

Case $E = \langle E' \rangle$. We construct $sha(E)$ from $sha(E')$. The initial states of $sha(E')$ are turned into tree initial states. We then add a new tree state $\langle E' \rangle$ and connect it to all final states of $sha(E')$ by a hedge ending rule $q \xrightarrow{tree} \langle E' \rangle$. Furthermore, the previously final state q becomes non final. Finally we add a new initial state q_i, a new final state q_f and a transition rule $\rightarrow\textcircled{q_i} \xrightarrow{\langle E' \rangle} \textcircled{q_f}$.

Case $E = \mu a.E'$. The main idea of the construction is similar to the case of NWAs. The correctness argument is quite different though. It depends on the fact, that we have independent automata for the top-level and all the tree subexpressions. That this invariant can be maintained in polynomial time requires an additional argument. We can assume without loss of

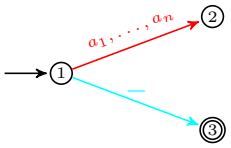


Figure 3.9: Construction of $sha(\neg\{a_1, \ldots, a_n\})$

generality that a occurs at most once in E', by the golden lemma of the μ-calculus [Arnold & Niwiński 2001], It states that for all names $a_1, \ldots, a_n$ and expressions E'' in which $a_1, \ldots, a_n$ can appear free, the equivalence $\mu a_1 \ldots . \mu a_n.\ E'' \equiv \mu a. E''[a_1/a, \ldots, a_n/a]$ holds.

By using ε-rules instead of Glushkov's construction, we can preserve the invariant that there will be at most one pair (q, q') such that $q \xrightarrow{a} q'$ in $sha(E')$. Furthermore, these transitions cannot be on top level, given that the occurrence of a in E' must be nested below parentheses. The automaton $sha(E)$ is obtained from $sha(E')$ by adding a fresh initial state q_i and a fresh final state q_f, a ε-rule from q to q_i and from q_i to all initial states of $sha(E')$, and ε-rules from all final states of $sha(E')$ to q_f, and from q_f to q'. The construction is correct since the μ-bound name a is nested below parenthesis in E'. Therefore, it can be shown that the ε-edges introduced can not be used to do unwanted order in successful runs.

Proposition 5 *For any CF-*NRE *E where all the subexpressions $\overline{E'}$ are such that E' is deterministic, we can construct in time $O(|E|^2)$ a SHA $sha(E)$ such that $L(sha(E)) = L(E)$.*

However, the construction does not preserve determinism. For the deterministic NRE $\langle a_1 \cdot \langle a_2 \cdot \ldots \cdot \langle a_n \rangle \ldots \rangle \rangle$, one would have a SHA having a tree initial state for each of the $\langle a_i \ldots \rangle$ subtree, implying nondeterminism. This is in contrast to the compiler to NWAs, which can rely on top-down determinism that is unavailable for SHAs though. Furthermore, as for NWAs, conjunctions may cause an exponential blow-up of the produced SHA.

3.5 Reducing the size of (d)SHAs

The size of automata greatly impacts on the performances of the algorithms that use them. In this section, we first present a technique for having smaller representations of smaller size. Then we introduce other techniques for eliminating useless rules and states that could be generated during the determinization of a SHA.

3.5.1 Symbolic Apply Rules

With their symbolic rules, our SHAs actually describe automata of bigger sizes. Indeed, given that the alphabets they accept are possibly infinite, we would have needed to add an infinite number of rules if not for else rules. This concept of symbolic rules can also be used for tree states, when the automaton has some special properties that we detail further.

Even if the input alphabet is infinite, the rules of SHAs only are only defined for a finite subset of the alphabet. The SHAs deal with the remaining symbols via else rules. Thus for a given SHA, every state has a set of *letters of interest*, that represent the letters of the alphabet for which the state has explicit outgoing transitions. For instance, in Figure 3.8, the letters of interest of 0 and 1 are a and b. Having a set of letters of interest doesn't imply that any letter that is not in that set can't be read from the state. Back to Figure 3.8, we see that any other letter in the alphabet that is different from a and b can be read from 1 using the else rule $1 \Rightarrow 7$.

This concept can be applied to tree states, especially when the automaton is deterministic and complete. Each state will have its *tree states of interest*, and will thus define *symbolic apply rules*, also called *apply else rules*, for the other tree states. These apply else rules will merge all the apply rules of states that are not of interest.

Definition 14 *A SHA with symbolic apply rules is a tuple* $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ *where* $\Omega \subseteq Q_t$ *and* Δ *contains the rules of the underlying SHA* $(Q_h, Q_t, \Sigma, \Delta, I, F)$*, plus apply else rules of the form* $q \xrightarrow{@}\!\!\!\!\Rightarrow q'$ *for hedge states* $q, q' \in Q_h$*. Furthermore, the states in* Ω *are such that* $\bigcup_{p \in \Omega} L_p(\Delta) = \mathcal{U}_\Sigma$.

The elements of Ω are called *else states*. Unlike else rules, apply else rules describe finite sets of apply rules for which the tree state is an else state. For every apply else rule $q \xrightarrow{@}\!\!\!\!\Rightarrow q' \in \Delta$, the set of described apply rules is

$$\{q \xrightarrow{p} q' \mid p \in \Omega \text{ and } \nexists q'' \in Q_h . q \xrightarrow{p} q'' \in \Delta\}.$$

SHAs with symbolic apply rules update the relations defined by their underlying SHAs, in particular those defined for tree states. Thus any tree state $p \in Q_t$ defines a binary relation $@_p^\Delta$ over Q_h such that:

$$@_p^\Delta = \{(q, q') \mid q \xrightarrow{p} q' \in \Delta\} \cup else\_@_p^\Delta,$$

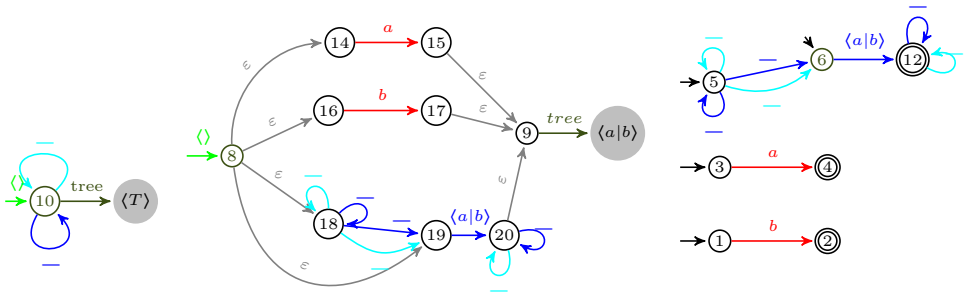


Figure 3.10: SHA with symbolic apply rules

where

$$else\_@_p^\Delta = \begin{cases} \{(q,q') \mid q \xrightarrow{@} q' \in \Delta \text{ and } \nexists q'' \in Q_h . q \xrightarrow{p} q'' \in \Delta\} & \text{if } p \in \Omega \\ \emptyset & \text{otherwise} \end{cases}$$

The relation $@^\Delta$ is of course extended with the elements brought by $else\_@_p^\Delta$ for tree states $p \in Q_t$.

The SHA in Figure 3.6 can be turned into a SHA with symbolic apply rules. Indeed, since it contains $\mathcal{T}$ – and $L_{\langle T \rangle}(\Delta) = \mathcal{U}_\Sigma$ – we can set $\Omega = \{\langle T \rangle\}$. The resulting SHA is illustrated in Figure 3.10, where the apply else rules are represented by dark blue lines with the symbol _ over them. Each rule $q \xrightarrow{@} q'$ actually denotes the rule $q \xrightarrow{\langle T \rangle} q'$.

The determinization algorithm have to be adapted a little bit for handling apply else rules. Let $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ be a SHA with symbolic apply rules, and $S_u = (Q_h, Q_t, \Sigma, \Delta, I, F)$ its underlying SHA. We note $S^{det} = (Q_h^{det}, Q_t^{det}, \Omega^{det}, \Sigma, \Delta^{det}, I^{det}, F^{det})$ where $Q_h^{det}, Q_t^{det}, I^{det}, F^{det}$ are obtained as in the same way than for S_u^{det}. For Δ^{det}, we add the following rules to those used to build S_u^{det}. For all subset of hedge states $Q_1 \subseteq Q_h$ and subset of tree states $Q \subseteq Q_t$, write $apply(Q_1, Q) = \{q \mid \exists q_1 \in Q_1, p \in Q . q_1 \xrightarrow{p} q \in \Delta\}$ and $applyelse(Q_1) = \{q \mid \exists q_1 \in Q_1 . q_1 \xrightarrow{@} q \in \Delta\}$: then

$$\frac{true}{Q_1 \xrightarrow{@} applyelse(Q_1) \in \Delta^{det}}$$

$$\frac{apply(Q_1) \neq \emptyset \quad Q_2 = \{q_2 \mid \exists q_1 \in Q_1, p \in Q . (q_1, q_2) \in @_p^\Delta\} \quad Q_2 \neq applyelse(Q_1)}{Q_1 \xrightarrow{Q} Q_2 \in \Delta^{det}}$$

Finally, we set $\Omega^{det} = \{Q \subseteq Q_t \mid Q \cap \Omega \neq \emptyset\}$.

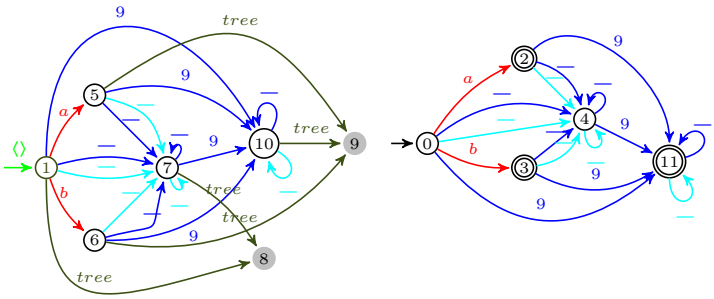


Figure 3.11: A dSHA with symbolic apply rules

We present in Figure 3.11 the dSHA with symbolic apply rules obtained by determinizing the SHA with apply else rules in Figure 3.10. Notice that both states 10 and 11 had 2 loops, and they now have only one loop. We'll see in the experiments how apply else rules help reduce the size of the dSHAs.

3.5.2 Cleaning Methods for Determinized SHAs

The determinization of SHAs (and automata in general) generates a lot of useless rules and states. A first and straightforward optimization that can be done when determinizing a SHA is to only generate accessible states. But there are other states and rules that we can avoid to generate, especially when we have additional information about the language that is accepted by the SHA. A language $\mathcal{L}$ is a *schema* of a language $\mathcal{L}'$ if $\mathcal{L}' \subseteq \mathcal{L}$. Let S be a SHA that we want to determinize. When a schema is given, then it is not necessary to generate states and rules that allow to read inputs that are not in the schema.

3.5.2.1 Canonicity Cleaning

Let $\mathcal{W}' \subsetneq \mathcal{W}$ be a finite subset of query variables. If SHA represents a query with query variables in $\mathcal{W}'$, then its language is a subset of the language of sequenced $\mathcal{W}'$-structures. In a sequenced $\mathcal{W}'$-structure, every element of $\mathcal{W}'$ appears exactly once.

Definition 15 *A SHA is called* ($\mathcal{W}'$-)canonic *if it recognizes a language of $\mathcal{W}'$-structures and, for all variable in $x \in \mathcal{W}'$, there exists no inputs having different number of occurrences of x that can be evaluated to the same state.*

Canonic automata have somehow states that are typed by the query variables, as each state q defines the set of variables that a sequenced $\mathcal{W}'$-structure should have in order be evaluated to q. Furthermore, since they accept only $\mathcal{W}'$-structures, they

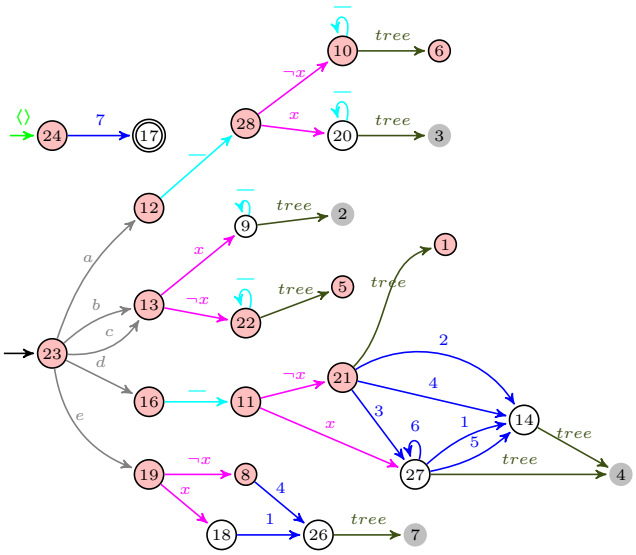


Figure 3.12: SHA on sequenced x-structures overs some alphabet containing at least a, b, c, d and e

should not allow to read inputs where one element in $\mathcal{W}'$ is read more than once. We use the latter remark to build determinized SHAs that are not necessarily canonic, but never allow to read a variable more than once. This may reduce the size of dSHAs (built from SHAs) drastically.

Let $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ be an automaton recognizing a language of sequenced $\mathcal{W}'$-structures, and write $Q_{all} = Q_h \cup Q_t$. We first define a binary predicate $mark_\Delta \subseteq Q_{all} \times \mathcal{W}'$ that associates a state q with a variable x only if q can be reached from some initial state or tree initial state without reading x on the input. Formally,

- for all hedge state $q \in Q_h$ and variable $x \in \mathcal{W}'$, if there exists a state $q' \in I \cup \langle\rangle^\Delta$ and a sequenced $\mathcal{W}'$-structure $w \in L_{q',q}(\Delta)$ that contains no occurrence of x, then $(q, x) \in mark_\Delta$
- for all tree state $p \in Q_t$ and variable $x \in \mathcal{W}'$, a sequenced $\mathcal{W}'$-structure $w \in L_p(\Delta)$ that contains no occurrence of x, then $(p, x) \in mark_\Delta$.

Example 11 *Let $\mathcal{W}' = \{x\}$, and consider the automaton in Figure 3.12, which reads sequenced $\mathcal{W}'$-structures. We call it S and note Δ its transition relation. All the states in S that can be reached while not reading x are colored in pink. Thus, the $mark_\Delta$ relation is formed of all the pairs pink-colored state with the variable x. Note that 1, 5 and 6 are tree states – not in their usual color.*

The $mark_\Delta$ can be computed by some inference rules, that somehow propagate the membership to the relation, starting from the initial states and the tree initial states, as following:

$$\frac{q \in I \cup \langle\rangle^\Delta \quad x \in \mathcal{W}'}{(q,x) \in mark_\Delta}$$

$$\frac{(q,x) \in mark_\Delta \quad q' \in Q_{all} \quad o \in \Sigma \cup \{\varepsilon, tree\} \quad (q,q') \in o^\Delta}{(q',x) \in mark_\Delta}$$

$$\frac{(q,x) \in mark_\Delta \quad (p,x) \in mark_\Delta \quad q' \in Q_h \quad (q,q') \in @_p^\Delta}{(q',x) \in mark_\Delta}$$

Once the $mark_\Delta$ predicate is built, we can derive the set of states for which any sequenced $\mathcal{W}'$-structure that are evaluated to them must contain at least an occurrence of a variable. In other words, we define the relation $safevar_\Delta \subseteq Q_{all} \times \mathcal{W}'$ that associates any state q with a variable x only if there is no sequenced $\mathcal{W}'$-structure with zero occurrence of x that can be evaluated to q. Clearly, $safevar_\Delta = (Q_{all} \times \mathcal{W}') \setminus mark_\Delta$. In the SHA of Figure 3.12, $safevar_\Delta$ consists of the set of all pairs of non pink-colored state with the variable x.

Finally, we use the $safevar_\Delta$ during the determinization process. This is done by allowing no rule that will cause some variable to be read more than once. Let $Q \subseteq Q_h$ be a hedge state in S^{det} and $Q \subseteq Q_t$ be a tree state of S^{det}. We adapt the determinization rules so that:

- for all $x \in \mathcal{W}'$, if there is a hedge state $q \in Q$ such that $(q,x) \in safevar_\Delta$, then no rule leaving $Q \xrightarrow{x} Q''$ will be created, for all $Q'' \subseteq Q_h$.
- for all $x \in \mathcal{W}'$, if there is a hedge state $q \in Q$ such that $(q,x) \in safevar_\Delta$ and a tree state $q' \in Q'$ such that $(q',x) \in safevar_\Delta$, then there will be no apply rule in Δ^{det} involving both Q and Q'.

This refinement doesn't change the language of $\mathcal{W}'$-structures of S^{det}, while avoiding the creation of a lot of useless states and rules.

3.5.2.2 Schema Cleaning by Intersections

When the schema is given with by the means of a dSHA S' that recognizes it, then we can build the deterministic SHA recognizing the intersection of the schema and the language of S. This can be done in multiple manners, e.g. by first determinizing SHA and intersecting it with S'. But intersections are costly and the resulting dSHA

can be very big. Instead, we can use this very big dSHA to produce a smaller one that integrates some information from the schema.

Note $S^{det} \times S'$ the product automaton recognizing $L(S^{det}) \cap L(S')$. Its states are tuples (Q, Q') where Q is a state of S and q a state of the schema automaton S'. We build a dSHA S'' from $S^{det} \times S'$ by only keeping the first components of the pairs of states of $S^{det} \times S'$, that is the states of S^{det}. Doing so allows to reduce a lot of transitions generated during the determinization of S, and that were not allowed by the schema.

3.6 Experimental Results for XPath Queries

The original goal of this chapter is to show that deterministic automata of reasonable size can be obtained for nested regular path queries. Deterministic NWAs are particularly targeted, since they empower some of the previous approaches of CQA and CQNA on streams. We show in this section that using SHAs in the process of building dNWAs allow to have smaller dNWAs– in contrast to dNWAs produced by determinizing the NWAs obtained by compilation of NREs. Of course, this also implies that the dSHAs obtained from NREs are not huge.

We now compare the sizes of deterministic NWAs that we can obtain by composing the various compilers in different orders.

We test the $A1, \ldots, A8$ XPATH queries (see Appendix .1) in the usual XPATH benchmark [Franceschet], which contain not only forward child, descendant and following-sibling axes, but also filters and path compositions. Note that the queries A4 until A8 contain filters, which are mapped to NREs with conjunctions. We compiled these queries automatically to nested regular expressions, then compiled these expressions to deterministic NWAs, by composing the various compilers presented earlier in all reasonable manners. A1 is the only query for which we obtain a deterministic regular expression. But since we replaced $ch(E)$ systematically by $T \cdot \langle E \rangle \cdot T$ in our experiments, all nested regular expression become nondeterministic.

The overall size of the resulting automata and the number of their rules are given in Figure 3.13. We can see that determinization applied to the NWAs for these expressions fails. Only 2 out of 8 automata have a size less than 400000, and for the others, the determinization run out of time. In contrast, 3 of the 4 other methods – that use stepwise hedge automaton intermediately – produce reasonable small deterministic NWAs.

This is not a coincidence. Intuitively, the reason is that NWAs obtained from stepwise hedge automata do all work bottom-up, where NWAs obtained directly from the regular expression do a considerable amount of work top-down. In terms

	$det(nwa(.))$	$nwa(det(sha(.)))$	$det(nwa(sha(.)))$	$nwa(det(sha(nwa(.))))$	$det(nwa(sha(nwa(.))))$
A1		398 (37)	302 (62)	398 (37)	398 (37)
A2	362600 (6782)	668 (57)	4889 (221)	1648 (127)	4105 (148)
A3	318704 (8216)	469 (44)	542 (66)	625 (56)	907 (62)
A4		487 (42)	335 (67)	487 (42)	487 (42)
A5		676 (55)	1054 (110)	856 (67)	1192 (73)
A6		548 (45)	332 (62)	548 (45)	548 (45)
A7		468 (41)	285 (54)	468 (41)	468 (41)
A8		2520 (124)	1236 (137)	1804 (118)	

Figure 3.13: Deterministic NWAs for XPath benchmark: size (#states).

	$det(nwa(.))$	$nwa(det(sha(.)))$	$det(nwa(sha(.)))$	$nwa(det(sha(nwa(.))))$	$det(nwa(sha(nwa(.))))$
$ch^3[a]$	19828 (1281)	85 (13)	157 (30)	192 (24)	352 (32)
$ch^4[a]$		177 (21)	206 (39)	664 (56)	2200 (88)
$ch^5[a]$		457 (37)	255 (48)	3336 (168)	
$ch^7[a]$		4825 (133)	353 (66)		
$ch^9[a]$			451 (84)		

Figure 3.14: Deterministic NWAs queries $ch^n[a]$ for $n = 3, 4, 5, 7, 9$: size (#states).

of [Alur *et al.* 2005] this restriction can be characterized syntactically by the single-entry property:

Definition 16 *An NWA $\mathcal{N}$ has the single-entry property, if there exists a single state $q_{entry} \in Q_h$ such that all opening rules of $\mathcal{N}$ have the form $q \xrightarrow{\langle\downarrow q} q_{entry}$.*

It can be shown that $nwa(S)$ has the single-entry property for all SHAs S for which the p's are unique in the construction rule of 3.3.2.1, i.e. such that $\langle\rangle \to p$ (see Figure 2 in Appendix .3).

For the fourth method in the last column, NWA-determinization didn't terminate on $nwa(sha(nwa(A8)))$ after a few hours.

We also tested our algorithms on collections of XPath queries with a scalable parameter, such as the queries $ch^n(a)$ for increasing n. This series is known to require many states for deterministic bottom-up evaluation. Indeed, the determinization for stepwise hedge automata $nwa(det(sha))$ leads to a size explosion. The method $det(nwa(sha(.)))$, however, still yields small deterministic automata! Generally this method produced satisfactory results in all our experiments. In quite some cases, however, $nwa(det(sha(.)))$ still behaves better.

Chapter 4

Certain Query Answering on Streams

Contents

It is clear from the results of Chapter 3 that stepwise hedge automata are a good intermediate model for obtaining deterministic nested word automata of reasonable size for nested regular path queries. In fact, dSHAs produced by determinizing SHAs obtained from NREs are also of reasonable size. We could thus also use them for CQA and CQNA on streams.

In this chapter, we present an algorithm empowered by dSHAs that decides CQA with a lower per-event time complexity than the approaches using NWAs.

4.1 Modeling Streams of Hedges

We show in this section two methods for representing streams of hedges. The difference between these methods comes from some assumptions that can be made on the content of the stream, in particular the parts representing trees.

$$X_0 \quad \langle_a X_1 \quad \langle_a\langle_b X_2 \quad \langle_a\langle_b\rangle X_3 \quad \langle_a\langle_b\rangle\rangle X_4 \quad \langle_a\langle_b\rangle\rangle\langle_c X_5 \quad \langle_a\langle_b\rangle\rangle\langle_c\rangle X_6 \quad \langle_a\langle_b\rangle\rangle\langle_c\rangle$$

Figure 4.1: Instantiations steps of a stream represented as a string pattern with parentheses

4.1.1 ... As String Patterns With Parentheses

In this approach, the stream is modeled as a pattern over $\Sigma \cup \{\langle, \rangle\}$. The elements of the known part of the stream are concatenated as they are received, while a single pattern variable represents the future elements to come. Figure 4.1 illustrates how the hedge $\langle_a\langle_b\rangle\rangle\langle_c\rangle$ over the alphabet $\Sigma = \{a, b, c\}$ is received as a stream with this model. It starts with a single variable X_0, that is instantiated with elements of $\{\langle_e \mid e \in \Sigma\} \cup \{\rangle\}$ until the penultimate step, where the pattern variable X_6 is instantiated into the empty word.

Because they can be instantiated separately, the opening and closing parentheses seem to be independent. In [Gauwin *et al.* 2009] and [Sebastian 2016], streams were modeled in this way, and NWAs were used to process them.

4.1.2 ... As Nested Patterns

One assumption that can be made by a streaming algorithm on hedges is that trees can always be represented by nested words, as seen in the subsection 2.1.3. This means that opening and closing parentheses do not have to be instantiated separately, as it is sure that every opening parenthesis will have its matching closing parenthesis. We show in Figure 4.2 how the hedge $\langle_a\langle_b\rangle\rangle\langle_c\rangle$ is streamed in such model.

The stream starts with X_0 and evolves respectively into $\langle aX_2\rangle X_1$, $\langle a\langle bX_3\rangle X_2'\rangle X_1$, $\langle a\langle b\rangle X_2'\rangle X_1$, $\langle a\langle b\rangle\rangle X_1$, $\langle a\langle b\rangle\rangle\langle cX_4\rangle X_1'$, $\langle a\langle b\rangle\rangle\langle c\rangle X_1'$ and finally $\langle a\langle b\rangle\rangle\langle c\rangle$. In this model variables can only be instantiated into nested words: corresponding opening and closing parentheses cannot be dissociated. Furthermore, due to the top-down-left-right traversal of the hedges, an order is imposed on the instantiations of variables. For instance X_2' cannot be instantiated before X_3 is completely instantiated, and X_1 cannot be instantiated before X_2' and X_3.

We note $nStreams_\Sigma$ the subset of $nPatterns_\Sigma$ that is constituted by the nested patterns representing streams.

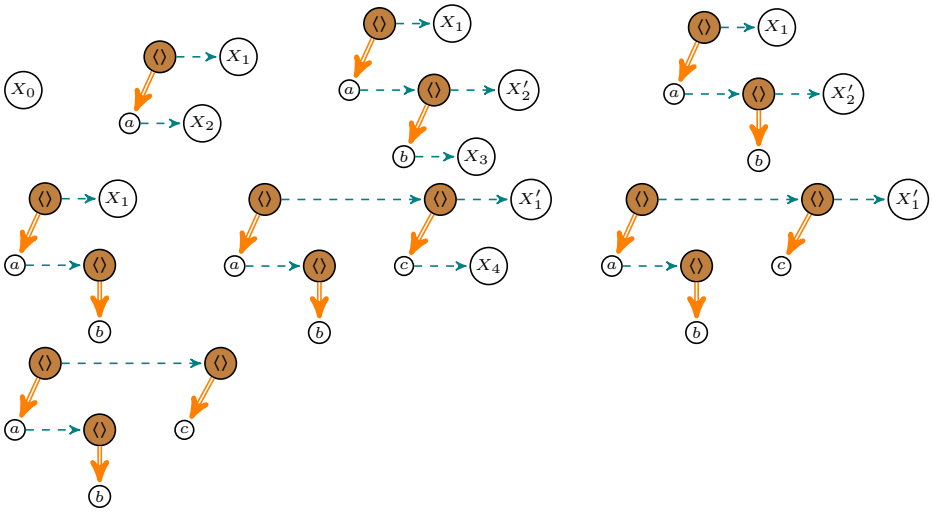


Figure 4.2: Instantiations steps of a stream of hedge as a nested pattern

4.2 About CQA Algorithms on Streams

A query answering algorithm is an algorithm that takes as input a query and a hedge, and outputs all the nodes of the hedge that satisfy the property expressed in the query. In the case of a boolean query, it just answers yes or no, depending on whether the hedge satisfies the query. Query answering algorithms can be grouped into two categories: in-memory and streaming algorithms. We focus on streaming algorithms in this section. Streaming algorithms read their input in only one pass. There are some criteria that are used in order to measure the quality of a streaming algorithm, among which:

- Latency: the responses of the algorithm are to be output at the earliest possible event.
- Memory consumption: the size of the buffer used by the algorithm should be the smallest possible. This implies that the elements that should no longer be kept into memory have to be discarded.
- Time efficiency: the number of operations per event of the algorithm should be reasonable.

In the case of a query answering algorithm working on streams, latency implies that an answer to the query should be output as soon as there are enough information to do so. Query answering algorithms meeting this requirement are called Certain Query Answering (CQA) algorithms. The low memory consumption requirement combines both CQA and the ability to remove from the buffer as soon as possible

any element that will never be part of the output of the query. Such elements are called certain non-answers. Thus, in order to have a minimal latency, a query answering algorithm must be both a CQA and a Certain Query Non-Answering (CQNA) algorithm.

Some CQA or approximation of CQA algorithms have been proposed for navigational XPath queries. These algorithms assume that the queries are represented by automata, that are either deterministic or that are determinized while reading the input hedge. Gauwin et al. proposed an exact algorithm for CQA [Gauwin *et al.* 2009] whose time efficiency is in $O(m|h||A|^2+|E(A)|+n)$ where m is the maximal number of candidates that are simultaneously alive, h is the input hedge, A the input dNWA representing the query, $E(A)$ a dNWA that is built on the fly from A and the input tree, and n the total number of candidates created while reading the input tree. The number of operations per event is quadratic in the size of the input dSHA, which is too much in practice. The dNWA E(A) that is built by the algorithm is used in order to keep information that facilitate deciding CQA. When considering approximations of CQA, [Debarbieux *et al.* 2015] proposed a very efficient algorithm that runs in $O(|Q||h|+|det(A,h)|+n)$ where Q is the number of states of the input NWA representing the query, h the input tree, $det(A,h)$ the on-the-fly determinization of A wrt. h and n the total number of candidates created when running the algorithm on its input. Unlike the algorithm of Gauwin, the input NWA is not necessarily deterministic. It's dynamically determinized, while taking care of only producing the part of the its deterministic version that is useful to evaluate the input tree. The approximation that results of this algorithm is good enough to be exact for all navigational XPath queries that have only positive filters, that is no negation.

In this chapter, we present a new and more efficient algorithm for CQA on streams that applies to queries defined by dSHAs.

We have seen in chapter 3, that bottom-up deterministic dSHAs are suitable for representing regular path queries on data trees. This is in contrast to previous approaches on XML stream processing, where various nondeterministic machines where used for representing XPATH queries (see e.g. SPEX transducer networks approach [Olteanu 2007b] and the NWA approaches from [Debarbieux *et al.* 2015] and [Mozafari *et al.* 2012]).

However, streams of hedges are received in a top-down left-to-right order, and not in bottom-up. One way to reconcile the two evaluation methods is to compile a dSHA into a dNWA, which can be done in linear time, and then to run this dNWA on the stream. When doing this in a naive manner, however, most of the evaluation work would be done late, which is incompatible with CQA. Indeed, when evaluating subtrees in a top-down left-to-right order, SHAs (and NWAs obtained from SHAs)

do not consider the information gathered before entering the subtrees. For instance, if the query asking whether there is some tree rooted by a letter a with a subtree rooted by a letter b were represented by a dSHA, then the dSHA would accept a satisfactory input only after having closed the subtree rooted by b – because of the bottom-up evaluation. In contrast, a CQA algorithm would accept the input as soon as the b letter is read. So we need to do more effort to obtain a CQA algorithm for dSHAs.

One proposal would be to apply the CQA algorithm for dNWAs of Gauwin [Gauwin *et al.* 2009], which is based on the usage of the hedge accessibility relation of dNWAs. This algorithm has the advantage to run in polynomial time. Unfortunately, however, it requires quadratic time in the size of the dNWA for each event on the stream, which is far too much in practice. In order to avoid this complexity problem, we propose an alternative CQA algorithm in the next sections, that runs in combined linear time for Boolean queries, and which can be lifted to nonBoolean queries with additional polynomial time costs that depend only on the number of query answers, but not on the size of the stream.

4.3 A Streaming Algorithm for Boolean CQA

We start with a CQA algorithm for Boolean queries represented by dSHAs. In order to simplify the situation we consider the problem of *certain co-accessibility*.

> **Existence of certainly co-accessible states.**
> *Input:* A dSHA S, a subset Q of hedge states of S, and a stream s with the same signature as S.
> *Output:* Whether there exists a state $q \in Q$ such that all the instances of s will be accepted by $S[I/\{q\}]$.

This problem is a little more general that what we will need for the boolean case, so that it will help us to find an algorithm for the nonboolean case too.

For capturing boolean certainty for selection, it is sufficient to restrict Existence of certainly co-accessible states to singleton sets Q. Indeed, a stream s is a certain answer for a boolean query $\mathbf{Q}$ represented by some dSHA S with initial state q_0 iff the output of Existence of certainly co-accessible states for $S, \{q_0\}, s$ is true. On the other hand, s is a certain non-answer for $\mathbf{Q}$ iff the output of Existence of certainly co-accessible states for $\overline{S}, \{q_0\}, s$ is true, where $\overline{S}$ is equal to S except that its final states are flipped.

We next present a streaming algorithm detecting certain co-accessibility at the earliest possible event of the stream. As inputs it receives a dSHA $S = (Q_h, Q_t, \_,$

$\Sigma, \Delta, I, F)$, a subset of hedge states $Q \subseteq Q_h$, and a hedge $s \in \mathit{nStreams}_\Sigma$. At the earliest event of the stream where the co-accessibility of some state in Q becomes certain, the algorithm will return *true* and stops. Recall that a state $q \in Q_h$ is co-accessible if a final state can be reached from q by the transitions in Δ. Without loss of generality we can assume that S is complete, given that completion can be done in linear time (based on applyelse rules).

The idea of the algorithm is to run S on s from the left to the right, while memoizing at any event the current subset of reached states and a set of *target states* that ensure co-accessibility. At the start of the stream, the algorithm tests whether only states in F are accessible from Q, returns *true* and stops if this is the case. This is done by testing if a state in Q is in the set of *safe states* of F, that is the states that can only lead to F. Otherwise, the algorithm continues with the evaluation of the automaton on the stream starting with source states Q and target states F, that is by calling $\mathit{eval\_coacc}_h(s, Q, F)$. If this evaluation successfully produces a result, say $Q' \subseteq Q_h$, then the algorithm returns the truth value of $Q' \cap F = \emptyset$. Otherwise, the algorithm raises an exception, which may have value *true* or *false*. This exception is caught and returned, and then the algorithm stops. So the algorithm either stops at the earliest event where certain co-accessibility becomes true, or at the end of the stream where co-accessibility may be certainly true or certainly false.

The algorithm for Existence of certainly co-accessible states is given in Figure 4.3. For any dSHA $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ it executes the definition of the total function:

$$\textsc{Cqa}\_\mathit{bool} : \mathit{nStreams}_\Sigma \times 2^Q_h \to \mathbb{B}$$

As subroutines it executes the definition of the partial functions:

$$\mathit{eval\_coacc}_\kappa : \mathit{nStreams}_\Sigma \times 2^{Q_h} \times 2^{Q_\kappa} \hookrightarrow 2^{Q_\kappa}$$

where κ is a type in $\{h, t\}$. This function evaluates a stream, from a set of hedge states for the source, while targeting a set of states of type κ. The result is a set of states of type κ that was reached over the stream. Whenever the existence of a certainly co-accessible state is discovered, the result of function $\mathit{eval\_coacc}_\kappa$ is undefined and an exception with value *true* is raised. The return value of function $\mathit{eval\_coacc}_\kappa$ is in $\mathbb{B}$ when reaching the end of a closed stream.

In the case where the stream is still open with some variable pattern $X \in Vars$, then X is considered as a *future*. Futures are a concept in programming languages

```
1  CQA_bool = fun(S,Q,s) // where S = (Q_h, Q_t, _, Σ, Δ, I, F) a complete dSHA,
       Q ⊆ Q_h and s ∈ nStreams_Σ
2       if Q ∩ safe_h^Δ(F) ≠ ∅ then return true
3       else try return eval_coacc_h(s,Q,F) ∩ F ≠ ∅
4            catch ex then return ex
5  eval_coacc_κ^Δ = fun(s,Q,R) // where κ ∈ {h,t}, Q ⊆ Q_h, R ⊆ Q_κ
6       if Q = ∅ then raise false
7       else case s
8         of ε then
9            case κ
10           of h then return Q
11           of t then return tree^Δ(Q)
12        of as' where a ∈ Σ and s' ∈ nStreams_Σ then
13           let Q' = a^Δ(Q) in
14            if Q' ∩ safe_κ^Δ(R) ≠ ∅ then
15              raise true // certainly inaccessible
16           else
17             return eval_coacc_κ(s',Q',R)
18        of ⟨s1⟩s2 where s1, s2 ∈ nStreams_Σ then
19            let P = eval_coacc_t(s1, ⟨⟩^Δ, down_κ^Δ(Q,R)) in
20            return eval_coacc_κ(s2, Q@^Δ P, R)
21        of X where X ∈ V then // Wait for the instantiation of X
22            when X is s' return eval_coacc_κ(s',Q,R)
23
24 safe_κ^Δ = fun(Q) // κ ∈ {h,t}, Q ⊆ Q_κ
25   let P = Q_h \ (acc_{h×κ}^Δ)^{-1}(Q_κ \ Q) in
26   return (acc_{h×κ}^Δ)^{-1}(Q) ∩ P
27 down_κ^Δ = fun(Q,R) // κ ∈ {h,t}, Q ⊆ Q_h, R ⊆ Q_κ
28        return {p ∈ Q_t | (Q@^Δ p) ∩ safe_κ^Δ(R) ≠ ∅}
```

Figure 4.3: Algorithm for EXISTENCE OF CERTAINLY CO-ACCESSIBLE STATES.

that allow to do operations on the content of a variable that is not instantiated yet. They are generally used in a concurrent context, but in our case they will be a way to wait for the instantiation of the end of the stream. Then whenever a pattern variable is reached, the algorithm pauses and waits for its instantiation before resuming.

We now consider a call $eval\_coacc_\kappa(s, Q, R)$ where $Q \subseteq Q_h$ and $R \subseteq Q_\kappa$. If Q is empty, then an exception with value $false$ is raised. This is a direct consequence of the definition of the EXISTENCE OF CERTAINLY CO-INACCESSIBE STATES problem. If the input stream is $s = \varepsilon$ then co-accessibility is not certain, since this is an invariant that our algorithm maintains. Therefore, if $\kappa = h$, the set of source states Q is returned and if $\kappa = t$ the tree states obtained for the source states $tree^\Delta(Q)$. In the case where the input stream has the form $s = as'$, the next source set $Q' = a^\Delta(Q)$ is computed. If only states of the target set R are accessible from Q' then certain co-accessibility holds and an exception with value *true* is raised. Otherwise, the

evaluation continues on stream s' with the new source states Q' and the old target states R. Whether or not some state in Q' can only access states in R is indicated by the truth value of $Q' \cap \mathit{safe}^{\Delta}_{\kappa}(R) \neq \emptyset$. The $\mathit{safe}^{\Delta}_{\kappa}$ function computes, given a set Q of type κ, the set of states of Q_h from which only elements of Q are accessible, that is

$$\mathit{safe}^{\Delta}_{\kappa}(Q) = \mathit{acc}^{\Delta}_{h\times\kappa}(Q) \cap P$$

where $P = Q_h \setminus (\mathit{acc}^{\Delta}_{h\times\kappa})^{-1}(Q_\kappa \setminus Q)$.

The most interesting case is the case where the input stream has a nested substream, i.e., $s = \langle s_1 \rangle s_2$. The algorithm first evaluates the substream s_1 starting with $\langle\rangle^{\Delta}$ as source states and the following set of tree states as target states:

$$\mathit{down}^{\Delta}_{\kappa}(Q, R) = \{p \in Q_t \mid (Q@^{\Delta}p) \cap \mathit{safe}^{\Delta}_{\kappa}(R) \neq \emptyset\}$$

A tree state p belongs to this set if there is an apply rule of the form $q@p \to q' \in \Delta$ with $q \in Q$ so that q' can only access R. If the tree states of $\mathit{down}^{\Delta}_{\kappa}(Q, R)$ can be certainly accessed on s_1 from $\langle\rangle^{\Delta}$, then R is certainly accessible from Q on s. So if the evaluation of s_1 successfully returns a set of states P (without detecting certain co-accessibility), then it has to continue on the stream s_2 with the sets of source states $Q@P$ and of target states R. If certain accessibility of $\mathit{down}^{\Delta}_{\kappa}(Q, R)$ is detected during the evaluation of s_1 from $\langle\rangle^{\Delta}$, then R is certainly accessible from Q for the evaluation of s. In this case, *true* is raised and the computation stops.

Proposition 6 (Correctness of the CQA_*bool* algorithm) *Let $s \in \mathit{nStreams}_{\Sigma}$ be a stream, $S = (Q_h, Q_t, \_, \Sigma, \Delta, \_, F)$ a complete dSHA, and $Q \subseteq Q_h$ a set of hedge states. There exists a state $q \in Q$ that is certainly co-accessible by s if and only if* $\textsc{Cqa}\_\mathit{bool}(S, Q, s) = \mathit{true}$.

To prove this proposition, we first introduce some notations. We will write for all $Q', Q'' \subseteq Q_h, P \subseteq Q_t$, $L_{Q',Q''}(\Delta) = \bigcup_{q'\in Q', q''\in Q''} L_{q',q''}(\Delta)$ and $L_P(\Delta) = \bigcup_{p\in P} L_p(\Delta)$. For the sake of simplicity, we will also write for all $q \in Q_h$ and $Q' \subseteq Q_h$, $L_{q,Q'}(\Delta)$ (resp. $L_{Q',q}(\Delta)$) to denote $L_{\{q\},Q'}(\Delta)$ (resp. $L_{Q',\{q\}}(\Delta)$). Now we show that:

Lemma 7 *Let $\kappa \in \{t, h\}$ be a type of state and $R \in Q_\kappa$ a set of states of type κ. Then if $\kappa = h$, there exists a state $q \in Q$ for which $\mathit{Inst}(s) \subseteq L_{q,R}(\Delta)$ if and only if, when evaluating $\mathit{eval}\_\mathit{coacc}^{\Delta}_{h}(s, Q, R)$,*

- *either a set of states Q' where $Q' \cap R \neq \emptyset$ is returned*
- *or an exception with value true is raised*

Furthermore, if $\kappa = t$*, then* $Inst(\langle s \rangle) \subseteq L_R(\Delta)$ *if and only if, when evaluating* $eval\_coacc_t^{\Delta}(s, \langle\rangle^{\Delta}, R)$*,*

- *either a set of states* Q' *where* $Q' \cap R \neq \emptyset$ *is returned*
- *or an exception with value true is raised*

Proof: The proof is by induction on the structure of s.

Case $s = \varepsilon$ **and** $\kappa = h$**:** then $Inst(s) = \{\varepsilon\}$ and $eval\_coacc_h^{\Delta}(s, Q, R) = Q$. Furthermore, since A admits no epsilon-rule – because of its determinism – there exists a state $q \in Q$ for which $\varepsilon \in L_{q,R}(\Delta)$ iff $q \in R$, that is $Q \cap R \neq \emptyset$.

Case $s = \varepsilon$ **and** $\kappa = t$**:** then $eval\_coacc_t^{\Delta}(s, \langle\rangle^{\Delta}, R) = tree^{\Delta}(\langle\rangle^{\Delta})$. Furthermore, since A admits no epsilon-rule, $\langle\varepsilon\rangle \in L_R(\Delta)$ if and only if $tree^{\Delta}(\langle\rangle^{\Delta}) \cap R \neq \emptyset$.

Case $s = as'$ where $a \in \Sigma$, $s' \in nPatterns_{\Sigma}$: then $Inst(s) = \{aw \mid w \in Inst(s')\}$. Assume $\kappa = h$, so for all $q \in Q$, $Inst(s) \subseteq L_{q,R}(\Delta)$ iff

- either all the words starting by a are elements of $L_{q,R}(\Delta)$, that is $\{aw' \mid w' \in nStreams_{\Sigma}\} \subseteq L_{q,R}(\Delta)$ (i)
- or $Inst(s') \subseteq L_{a^{\Delta}(q),R}(\Delta)$ (ii)

Let $Q' = a^{\Delta}(Q)$. Since S is complete, then case (i) is satisfied if and only if there exists a state $q' \in Q'$ from which only states in R can be reached, that is the condition $Q' \cap safe_{\kappa}^{\Delta}(R) \neq \emptyset$ at line 14 is true, and an exception with value *true* is raised at line 15. For case of (ii), if (i) is not satisfied, then the recursive call $eval\_coacc_{hedge}^{\Delta}(s', Q', R)$ is made, and by the induction hypothesis, $Inst(s') \subseteq L_{Q',R}(\Delta)$ iff either an exception with value *true* is raised, or a set Q'' satisfying $Q'' \cap R \neq \emptyset$ is returned.

The proof is similar for $\kappa = t$.

Case $\langle s_1 \rangle s_2$ where $s_1, s_2 \in nStreams_{\Sigma}$: then $Inst(s) = \{\langle w_1 \rangle w_2 \mid w_1 \in Inst(s_1), w_2 \in Inst(s_2)\}$. Let $\kappa = h$, $D = down_{\kappa}^{\Delta}(\{q\}, R)$. So for all $q \in Q$, $Inst(s) \subseteq L_{q,R}(\Delta)$ iff

- either all the nested words starting with some prefix $\langle w_1 \rangle \in Inst(\langle s_1 \rangle)$ are members of $L_{q,R}(\Delta)$, that is $\{\langle w_1 \rangle w' \mid w_1 \in Inst(s_1)$ and $w' \in nStreams_{\Sigma}\} \subseteq L_{q,R}(\Delta)$ (i)
- or $Inst(s_2) \subseteq L_{Q@^{\Delta}P,R}(\Delta)$, where $P = \{p \in Q_t \mid L_p(\Delta) \cap Inst(\langle s_1 \rangle) \neq \emptyset\}$ (ii)

By the induction hypothesis, case (i) is satisfied if and only if either the set $eval\_coacc_t^{\Delta}(s_1, \langle\rangle^{\Delta}, D)$ is successfully computed and has a non-empty intersection with D or an exception with value *true* is raised during its evaluation. In case (ii), if case (i) is false, we have by the induction hypothesis that $eval\_coacc_h^{\Delta}(s_1, \langle\rangle^{\Delta}, D)$ returns a set P having an empty intersection with D – an exception with value *false* will not be raised because this would imply that $P = \emptyset$, which is impossible given that S is complete. Furthermore, by the induction hypothesis, $eval\_coacc_h^{\Delta}(s_2, Q@^{\Delta}P, R)$ either raises an exception with value *true*, or returns a set Q'' having a non-empty intersection with R. The case where $\kappa = t$ is proved similarly.

We won't deal with the case where $s = X \in \mathcal{V}$, since it reduces to the above cases when X is instantiated.

□

Proof of Proposition 6: The algorithm first computes $Q \cap safe_{\kappa}^{\Delta}(F)$ at line 2. If it is non-empty, then any nested word can be evaluated from Q to F, in particular the instances of s. Otherwise, the algorithm tries to compute $Q' = eval\_coacc_h(s, Q, F)$. By Lemma 7, the cases where it succeeds to compute it and finds that $Q' \cap F \neq \emptyset$, or fails to do so while catching an exception with value *true*, are reached if and only if there exists a state $q \in Q$ such that $Inst(s) \subseteq L_{q,F}(\Delta)$. Thus the algorithm returns true if and only if there exists a state $q \in Q$ that is certainly co-accessible by s. □

We next show that CQA_*bool* decides the problem of certainty of selection.

Proposition 7 *Let $\mathbf{Q}$ be a boolean query with alphabet Σ, represented by a complete dSHA S with set of initial states I. A stream $s \in nStreams_{\Sigma}$ is certain for selection for $\mathbf{Q}$ if and only if* CQA_$bool(S, I, s) = true$.

Proof: Let Q_h be the set of hedge states of S, $F \subseteq Q_h$ its set of final states and $s \in nStreams_{\Sigma}$ a stream. Given that S is deterministic, I consists of a single state $q_0 \in Q_h$. By Proposition 6, CQA_$bool(S, I, s)$ equals *true* if and only if it is certain that only states in F can be accessed after having read s from q_0, which is equivalent to $Inst(s) \subseteq L_{q_0,F}(\Delta) = L(\mathbf{Q})$. Thus s is certain for selection for $\mathbf{Q}$ if and only if CQA_$bool(S, I, s) = true$. □

4.4 Certain Query Answering for Monadic Queries

We now adapt the algorithm for the boolean case to the monadic case. Let $\mathbf{Q}$ a boolean query over alphabet Σ represented by a complete dSHA S with transitions

relation Δ. We consider a stream $s \in nStreams_\Sigma$. Proposition 7 shows that calling the CQA_*bool* function with the arguments S, I and s is enough for deciding the certainty of s. For every call $eval\_coacc_\kappa^\Delta(s', Q, R)$ made during the execution of CQA_$bool(S, I, s)$ where $s' \in nStreams_\Sigma$ is a substream of s and κ a type, Q is necessarily a singleton, thanks to the determinism and the completeness of S. Actually, every call $eval\_coacc_\kappa^\Delta(s', Q, R)$ is equivalent to asking the following question: is it certain that only states in R can be reached by the single state $q \in Q$ that bears the empty candidate $[]$, after having read s' ? This interpretation follows directly from the definition of boolean queries, which are queries that select the empty candidate.

Monadic queries select candidates other than the empty candidate. A monadic query $\mathbf{Q}$ with alphabet Σ and query variable $x \in \mathcal{W}$ should thus associate to x positions of the input on which it is evaluated. But the associations of variables to positions are not given in the input, and the automaton S' associated to $\mathbf{Q}$ have somehow to guess them. This leads to a kind of non deterministic run, even if S' is deterministic. More precisely, it is as if S' were running on multiple inputs simultaneously, each one corresponding to a different annotation of the variable x.

The CQA_*bool* algorithm can be adapted in order to decide whether there is a candidate that is certain for selection. It will have to work on sets of states, where each element bears an object describing a set of candidates.

4.4.1 Position-annotated patterns

Before pointing on the differences between the monadic and boolean CQA algorithm, we introduce special nested patterns where positions are part of their labels.

Definition 17 *Given an alphabet Σ and a nested pattern $\rho \in nPatterns_\Sigma$, the* position-annotated pattern *obtained from ρ is the nested pattern $\widetilde{\rho}$ over $\Sigma \times Pos(\rho)$ such that $ch^{\widetilde{\rho}} = ch^\rho$ and $ns^{\widetilde{\rho}} = ns^\rho$ and for every position $\pi \in Pos(\rho)$:*

- *$\pi \in Nodes(\rho)$ iff $\pi \in Nodes(\widetilde{\rho})$*
- *for all $X \in \mathcal{V}$, $\pi \in LPos_{\{X\}}(\rho)$ iff $\pi \in LPos_{\{X\}}(\widetilde{\rho})$*
- *for all $a \in \Sigma$, $\pi \in lab_a^\rho$ iff $\pi \in lab_{(a,\pi)}^{\widetilde{\rho}}$*

We write $pan_\Sigma(\rho) = \widetilde{\rho}$, and a^π to denote the elements (a, π) of $\Sigma \times Pos(\rho)$.

Figure 4.4 illustrates a nested pattern (to the left) with its position-annotated version (to the right).

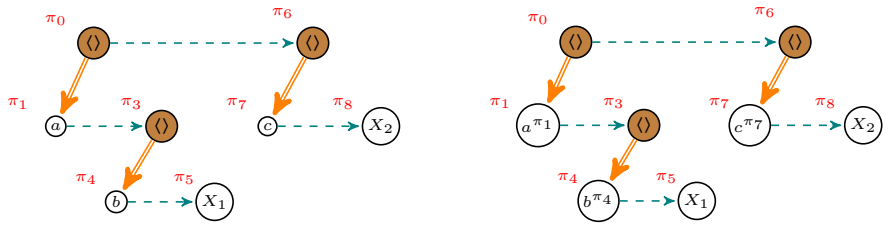


Figure 4.4: A pattern and its position-annotated version

4.4.2 Main differences with Boolean CQA

Carrying candidates in states. In contrary to boolean CQA where only the empty candidate could be selected, we need in monadic CQA to assign candidates to states. Thus, from now and for the rest of this chapter, we use the letters Q, P and their derivatives to refer to functions that map states in $Q_h \cup Q_t$ to sets of candidates. These functions are called *run snapshots*.

We'll also need to define the *join* operation on sets of candidates. The join of two sets of candidates C_1 and C_2 is the new set of candidates

$$C_1 \bowtie C_2 = \{\alpha_1 \cup \alpha_2 \mid \alpha_1 \in C_1 \text{ and } \alpha_2 \in C_2 \text{ and } dom(\alpha_1) \cap dom(\alpha_2) = \emptyset\}.$$

Canonic automata. The input automata that are considered by the monadic CQA algorithm are canonic, in the sense of Definition 15 in Section 3.5.2.1. Since exactly one variable can be bound in the monadic case, this implies that every state of the automata must either carry only monadic candidates, or carry only the boolean candidate. To obtain such kind of automata for a monadic query **Q** with query variables $\{x\}$, we can build the product of the complete dSHA obtained by compilation from **Q** – and eventually determinization – with the dSHA in Figure 4.5.

The latter dSHA, denoted *one-x*, recognizes the language of all sequenced $\{x\}$-structures. We note $Q_h^{\text{one-}x} = \{3, 4, 5\}$ its set of hedge states, $Q_t^{\text{one-}x} = \{0, 1, 2\}$ its set of tree states and $F^{\text{one-}x} = \{4\}$ its set of final states. Notice that the tree states $0, 1, 2$ are respectively the states where tree having respectively $0, 1$ or 2 occurrences of x are evaluated. The hedge state 4 is its only final state given that any hedge evaluated in 4 has exactly one occurrence of x.

Any dSHA representing a monadic query and obtained by a product with *one-x* is called *one-x-canonic*.

Presence of candidates that are certain for selection. At each event, the algorithm tests whether there is some candidate that is certain for selection. This test doesn't identify which candidates are certain for selection, but just reports the presence of such candidates. As we will see in Theorem 2, this test is less expensive

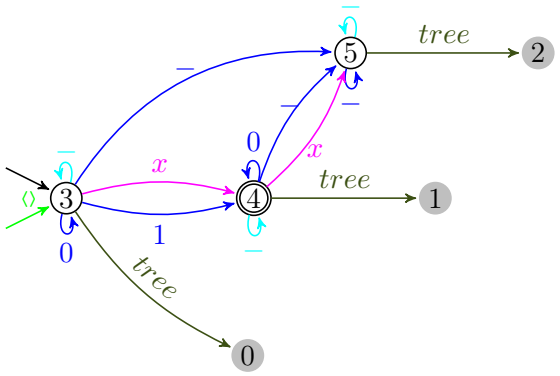


Figure 4.5: Automaton *one-x* with set of apply else rules $\{0, 1, 2\}$

– in time – than the task of seeking for the candidates that are certain for selection. The part of the algorithm that is in charge of this test is very similar to the CQA_*bool* function. We thus use it to restrict to the minimum the number of times that the procedure for retrieving candidates is called.

Searching for and outputting candidates. We introduce a class of objects named *candidate managers*, that will carry out the task of gathering and outputting the candidates that are certain for selection. We use them in a separate thread, where they perform their task every time the certainty of some candidates is reported by the CQA algorithm. Their *select* function is actually responsible for this task.

Killing candidates. In order to not process again candidates that are already known to be certain for selection or rejection, we use a side-effecting function that can kill candidates, that is mark them as useless for the operations to come. A dead candidate is thus a candidate that is to be ignored by the algorithm. A candidate that is not dead is called *alive*. For all run snapshot Q, the function *adom* returns the set of states that are not associated to a set of dead candidates, that is

$$adom(Q) = \{q \mid \exists C.(q, C) \in Q \text{ and } \exists \alpha \in C.\ \alpha \text{ alive}\}.$$

By abuse of language, we also say that a set of candidates is alive if at least one of its candidates is.

Explicit Stack. The algorithm for boolean CQA is a recursive algorithm, and thus uses implicitly the recursion stack. For monadic CQA, the algorithm that we will present will also be recursive, but will have to maintain an explicit stack. Before entering any subtree in the input, both the run snapshot and the sets of target states are pushed on the stack. These data will be used to resume runs when closing a subtree in the input, but also by the candidate manager when it will search and output the candidates that are certain for selection.

4.4.3 Description of the Algorithm

The algorithm for monadic CQA is called *CQA_sel* and presented in Figure 4.6. It has 5 parameters:

- the query variable $x \in \mathcal{W}$
- $Q_h^{one\text{-}x}$, $Q_t^{one\text{-}x}$, $F^{one\text{-}x}$, respectively sets of hedge states, tree states and final states of *one-x*
- the dSHA $S = (Q_h, Q_t, \Omega, \Sigma \uplus \{x, \neg x\}, \Delta, I, F)$ representing the query and recognizing a language of sequenced $\{x\}$-structures. Notice that its signature includes the query variable $x \in \mathcal{W}$ and its negation $\neg x$. S has also to be complete and *one-x*-canonic.
- Functions $\mathit{schema\text{-}state}_\kappa \colon Q_\kappa \to Q_\kappa^{one\text{-}x}$ for $\kappa \in \{h, t\}$ mapping every state of S to the only state of *one-x* from which it was built. Since S is the result of a product involving *one-x*, the states in Q_h can be seen as pairs of states where one of the components is a state in $Q_h^{one\text{-}x}$. The latter state, that we call *schema state*, is returned by the $\mathit{schema\text{-}state}_\kappa$ functions.
- The function $\mathit{hedge\text{-}schema\text{-}state} \colon Q_t \to Q_h^{one\text{-}x}$ mapping every tree state $p \in Q_t$ to the only hedge schema state having a tree transition to $\mathit{schema\text{-}state}_t(p)$. Indeed for all tree state $q_{one\text{-}x} \in Q_t^{one\text{-}x}$ there exists exactly one hedge state $q'_{one\text{-}x} \in Q_h^{one\text{-}x}$ such that $q'_{one\text{-}x} \xrightarrow{tree} q_{one\text{-}x}$ is a transition in *one-x*.

The input of the algorithm is a stream $s \in \mathit{nStreams}_\Sigma$. From Line 9 to Line 11 are respectively initialized the run snapshot $Q_{\langle\rangle} = \langle\rangle^\Delta \times \{[]\}$ that maps the tree initial state of the S to the singleton containing the empty candidate, a stack – data structure – and a *candidate manager*. At Line 12 starts the definition of the main subroutine of the algorithm, the function $\mathit{eval\_sel}_\kappa$ where $\kappa \in \{h, t\}$. It's a transposition of the $\mathit{eval\_coacc}_\kappa^\Delta$ algorithm in Figure 4.3 to the monadic case, but takes as input a position-annotated stream $\widetilde{s}$ – instead of a stream –, a run snapshot Q – instead of set of states – where $\mathit{dom}(Q) \subseteq Q_h$, a set of *query-successful* states $R \subseteq Q_\kappa$ and a set $T \subseteq Q_\kappa$ of *out-of-schema* states. Query-successful states are constituted of the states that can only lead to final states, while out-of-schema states consist of the states that will only lead to states whose schema states are non final.

The definition of certain answers require that all the instantiations of the stream annotated with some candidate belong to the language of the query. However, in the case of sequenced $\{x\}$-structures, the instances that are considered are constrained

```
1  //Query variable x ∈ 𝒲
2  //Sets Q_h^{one-x}, Q_t^{one-x} and F^{one-x} from one-x
3  //Complete and one-x−canonic dSHA S = (Q_h, Q_t, _, Σ ⊎ {x, ¬x}, Δ, I, F)
4  //Functions schema-state_κ: Q_κ → Q_κ^{one-x} for κ ∈ {h, t}
5  //Function hedge-schema-state: Q_t → Q_h^{one-x}
6
7  CQA_sel =
8    fun(s) // where s ∈ nStreams_Σ
9     let Q_⟨⟩ = ⟨⟩^Δ × {[]} in
10    let stack = newstack() in
11    let candidate_manager = thread new_candidate_manager^Δ(stack) in
12    let eval_sel_κ =
13     fun(s̃, Q, R, T) // where κ ∈ {h,t}, s̃ ∈ nStreams_{Σ×Pos(s)}, Q: Q_h ↪ 2^Cand,
                          R ⊆ Q_κ query−succesful states, T ⊆ Q_κ out−of−schema states
14      if adom(Q) = ∅ then raise false
15      else case s̃
16       of ε then
17         case κ
18         of h then return fusion(Q)
19         of t then return fusion({(tree^Δ(q), C) | (q, C) ∈ Q and C alive})
20       of a^π s̃′ where a ∈ Σ, π ∈ Pos(s), s̃′ ∈ nStreams_{Σ×Pos(s)} then
21         let Q_1 = fusion({(a^Δ(q), C) | (q, C) ∈ Q and C alive}) in
22         let Q_2 = readVar^Δ(Q_1, π, x) in
23         let Ψ = {q ∈ adom(Q_2) | schema-state_h(q) ∈ hedge-schema-state(R)} in
24         if Ψ ∩ safe_κ^Δ(R ∪ T) ≠ ∅ then
25           candidate_manager.select(Q_2)
26         return eval_sel_κ(s̃′, Q_2, R, T)
27       of ⟨s̃_1⟩s̃_2 where s̃_1, s̃_2 ∈ nStreams_{Σ×Pos(s)} then
28         stack.push((Q, R, T))
29         let R_d = safe-acc-down_κ^Δ(Q, R, T),
30             T_d = safe-down_κ^Δ(Q, R, T) in
31         let P = eval_sel_t(s̃_1, Q_⟨⟩, R_d, T_d) in
32         let _ = stack.pop() in let
               Q′ = {(q@^Δ p, C_q ⋈ C_p) | (q, C_q) ∈ Q, (p, C_p) ∈ P, C_q, C_p alive and C_q ⋈ C_p ≠ ∅}
33         in return eval_sel_κ(s̃_2, Q′, R, T)
34       of X where X ∈ 𝒱 then
35         when X is s̃′ return eval_sel_κ(s̃′, Q, R, T)
36   in
37    try
38      let s̃_0 = thread pan_Σ(s) in
39      let Q_I = I × {[]},
40          R_0 = F,
41          T_0 = Q_h \ schema-state_h^{-1}(F^{one-x}) in
42      return eval_sel_h(s̃_0, Q_I, R_0, T_0)
43    catch ex then skip
```

Figure 4.6: CQA algorithm for monadic queries.

```
1  adom = fun(Q) // Active domain of Q
2     return {q | ∃C.(q,C) ∈ Q and C alive}
3
4  readVar^Δ = fun(Q,π,x)
5     return fusion({(¬x^Δ(q),C) | (q,C) ∈ Q and C alive}
6                   ∪{(x^Δ(q),C ∪ {[x/π]}) | (q,C ⊎ {[]}) ∈ Q})
7
8  fusion = fun(Q) // Q : Q_h ↪ 2^Cand
9     return {(q, ⋃_{(q,C)∈Q} C) | q ∈ dom(Q)}
10
11 safe^Δ_κ = fun(Ψ) // κ ∈ {h,t}, Ψ ⊆ Q_κ
12    let Ψ' = Q_h \ (acc^Δ_{h×κ})^{-1}(Q_κ \ Ψ) in
13    return (acc^Δ_{h×κ})^{-1}(Ψ) ∩ Ψ'
14
15 safe-down^Δ_κ = fun(Q,R,T) // κ ∈ {h,t}, Q: Q_h ↪ 2^Cand, T ⊆ Q_κ
16    return {p ∈ Q_t | ∃q ∈ adom(Q) ∩ (acc^Δ_{h×κ})^{-1}(R). q@^Δp ∈ safe^Δ_κ(T)}
17
18 safe-acc-down^Δ_κ = fun(Q,R,T) // κ ∈ {h,t}, Q: Q_h ↪ 2^Cand, T ⊆ Q_κ
19    return {p ∈ Q_t | ∃q ∈ adom(Q). q@^Δp ∈ safe^Δ_κ(R ∪ T) ∩ (acc^Δ_{h×κ})^{-1}(R)}
```

Figure 4.7: Utility functions for the monadic CQA algorithm

to the ones where only one occurrence of x appears, that is the elements of $L(\textit{one-}x)$. So in order to decide certainty, instances that are not accepted by *one-x* do not have to be accepted by S. This is the reason why the states in $R \cup T$ represent the target states. Thus a candidate at some state $q \in \textit{dom}(Q)$ is certain for selection if R is accessible from q – instances that are accepted by *one-x* can be accepted by the query – and only states in $R \cup T$ are accessible from q – all the instances will be accepted by S or rejected by *one-x*.

The result of a call of $\textit{eval\_sel}_\kappa$ is a run snapshot, the one in which $\widetilde{s}$ has been evaluated from Q and sets of target states R and T. Furthermore the return value of $\textit{CQA\_sel}(s)$ is $\textit{eval\_sel}_h(\widetilde{s}_0, Q_I, R_0, T_0)$, where $\widetilde{s}_0$ is the position-annotated equivalent of s, $Q_I = \{(q, \{[]\}) \mid q \in I\}$ is the run snapshot that maps the initial state of S to the singleton containing the empty candidate, $R_0 = F$ and $T_0 = Q_h \setminus \textit{schema-state}_h^{-1}(F^{\textit{one-}x})$ is the set of hedge states of S whose schema states are not final in *one-x*. The task of creating $\widetilde{s}_0$ dynamically from s can be carried out by another thread.

Now consider a call $\textit{eval\_sel}_\kappa(\widetilde{s}, Q, R, T)$. As for the boolean case, an exception with value false is raised when there is no state in $\textit{dom}(Q)$ that carries an alive candidate. If this condition is not satisfied then the remaining actions of $\textit{eval\_sel}_\kappa$ depend on the form of $\widetilde{s}$.

If $\widetilde{s}$ is empty, then the result depends on the type κ. If κ is the hedge type, then Q is returned. For the tree type however, since the tree has to be closed, the

candidates of the hedge states in Q are moved to the tree states that they can access. Using the function $fusion$ defined on run snapshots Q' so that

$$fusion(Q') = \{(q, \bigcup_{(q,C)\in Q'} C) \mid q \in dom(Q')\}$$

the run snapshot $fusion(K)$ is returned, where $K = \{(tree^{\Delta}(q), C) \mid (q, C) \in Q\}$. Although Q is a function, the relation K is not necessarily functional, since $tree^{\Delta}$ – and by the way o^{Δ} for all $o \in \Sigma \cup \{x, \neg x\}$ – is not injective. So there may exist two tuples $(q, C), (q, C') \in K$ for which $C \neq C'$. We thus apply the $fusion$ function on K before returning it, so that all the sets of candidates associated to the same state are merged.

In the case where $\widetilde{s}$ is prefixed by a node $a^{\pi} \in Nodes(\widetilde{s})$ where $a \in \Sigma$ is a letter and π a position, we first compute the run snapshot $Q_1 = fusion(\{(a^{\Delta}(q), C) \mid (q, C) \in Q$ and C alive$\})$ reached by S when reading a from Q, while omitting states with dead candidates. Then, since the query is monadic, the algorithm combines two runs: one where the query variable x is read and another one where it is not read. This is done when applying the $readVar^{\Delta}$ function to Q_1, which returns a run snapshot Q_2. Not reading the query variable is simulated by actually reading its negation $\neg x$. Next, we check whether or not there is some candidate that is certain for selection. This is done without retrieving the candidates that are certain for selection, in a way that is similar to the boolean case, by testing $\Psi \cap safe_{\kappa}^{\Delta}(R \cup T) \neq \emptyset$ where $\Psi = \{q \in adom(Q_2) \mid schema\text{-}state_h(q) \in hedge\text{-}schema\text{-}state(R)\}$. The $safe_{\kappa}^{\Delta}$ function is defined in Figure 4.7 exactly as in the boolean case. Remark that a necessary – but not sufficient – condition for a stream representing a sequenced $\{x\}$-structure to be certain for selection for a monadic query is to have read the variable x once. This is ensured by gathering in Ψ all the states that have the same schema states as the states in R. If the test returns true, then the algorithm sends a message to the candidate manager, which searches for the candidates that are certain for selection and outputs them. After that, the main function takes the control back and makes a recursive call from the snapshot run, and with the suffix of $\widetilde{s}$. Note that the actions of the candidate manager are side-effecting, since candidates can be killed by it, and the content of the stack can be modified.

When $\widetilde{s}$ is of the form $\langle \widetilde{s}_1 \rangle \widetilde{s}_2$ where $\widetilde{s}_1$ are $\widetilde{s}_2$ position-annotated streams, then the algorithm pushes (Q, R, T) on the stack, evaluates $\widetilde{s}_1$ from $Q_{\langle\rangle}$ with type parameter t, query-successful states $R_d = safe\text{-}acc\text{-}down_{\kappa}^{\Delta}(Q, R, T)$ and out-of-schema states $T_d = safe\text{-}down_{\kappa}^{\Delta}(Q, R, T)$. The functions $safe\text{-}acc\text{-}down_{\kappa}^{\Delta}$ and $safe\text{-}down_{\kappa}^{\Delta}$ defined in Figure 4.7 are adaptations of the $down_{\kappa}^{\Delta}$ function to the monadic case. For a new

```
1   // Same parameters than CQA_sel
2
3   new_candidate_manager_Δ = fun(stack) // Takes a stack as input
4     let select = fun(Q)
5         let L_0 = stack.stackToList() in
6         let Q' = eval_stack^Δ(L_0, Q) in
7         let M = {α | ∀(q,C) ∈ Q', α ∈ C ⇒ q ∈ F ∨ schema-state_h(q) ∉ F^one-x} in
8         output_and_kill(M) // Side effecting function that outputs
              and kills candidates
9         let L'_0 = recompute-targets^Δ(L_0.reverse()) in
10        stack.refresh(L'_0) // set the content of stack to the content of
              L'_0
11    in return (select)
12
13  eval_stack^Δ = fun(L, Q)
14    case L
15      of nil then // empty stack
16         return fusion({(q',C) | ∃q ∈ Q. (q,C) ∈ Q, C alive and q' ∈ acc^Δ_{h×h}(q)})
17      of (Q_head, _, _) :: L' then
18        let Q' = fusion({(q',C) | ∃q ∈ Q. (q,C) ∈ Q, C alive and q' ∈ acc^Δ_{h×t}(q)})
              in
19        let Q'' = fusion({(q@^Δq', C ⋈ C') | (q,C) ∈ Q_head, (q',C') ∈ Q',
20                                          C, C' alive and C ⋈ C' ≠ ∅}) in //
                                              Composing Q_head with Q'
21        return eval_stack^Δ(L', Q'')
22
23  recompute-targets^Δ = fun(L) // list of triples (Q,R,T)
24    case L
25      of nil then
26        return nil
27      of (Q,R,T) :: L' where R,T ⊆ Q_κ and κ ∈ {h,t} then
28        let R' = safe-acc-down^Δ_κ(Q,R,T),
29            T' = safe-down^Δ_κ(Q,R,T),
30            L_rec = recompute-targets^Δ(L') in
31            return L_rec.append(Q,R',T')
```

Figure 4.8: Candidate manager

subtree, they allow compute the new sets of target states, using the tree states from which it is possible to reach only R (resp. to reach only T). Setting P as the result of the recursive call, the algorithm then pops the stack, composes the run snapshots Q and P – while taking into account the canonicity of $\mathcal{W}$-structures and omitting the dead candidates – before making another recursive call on $\widetilde{s}_2$.

Finally, in the case where $\widetilde{s} = X \in \mathcal{V}$, the algorithms waits for the instantiation of the future X into $\widetilde{s}'$ before making a recursive call with $\widetilde{s}'$, Q and R.

Let's now discuss a bit about the candidate manager, whose definition is presented in Figure 4.8. It has the charge of searching and outputting certain for selection candidates, and runs on a different thread. It is initialized with a reference

to a stack. For instance *CQA_sel* initializes a candidate manager with the stack that it has declared. Candidates managers declare a side-effecting function *select*, which takes a run snapshot as input and outputs the set of all candidates that are certain for selection. Those candidates are also marked as dead, so that they will no longer be taken into account by the future process. The *select* function uses an $eval\_stack^{\Delta}$ function, that takes as input a list L of pairs of run snapshots and target states, and a run snapshot Q. The list L is obtained by transforming the input stack of the candidate manager into a list whose first element is the top of the stack. Then according to the form of L, one of the two following cases may occur.

- If L is the empty list, that is when the stack is empty, the run snapshot where every set of candidate in Q is associated with the set of hedge states accessible from the state that carries it is returned.
- If L is a list whose head and tail are respectively the triplet $(Q_{head}, \_, \_)$ and the list L', then the candidates in Q are moved to the tree states accessible from Q, yielding a run snapshot Q'. Furthermore, Q_{head} is composed with Q', and a recursive call is made with the resulting run snapshot and L'.

Back to the *select* function, a candidate is output and marked only if all the states in which it can be evaluated are either final states of S, or states whose schema states are non final in *one-x*.

The candidate manager is also responsible for updating the content of the stack, whenever it is required. Indeed, when candidates are marked, the sets of target states stored in the stack become obsolescent, since the states used to compute them may now be associated to dead candidates. This is why the *select* function also calls the *recompute-targets*$^{\Delta}$ function, which recompute the values of the target states after candidates are marked.

4.4.4 Correctness and Complexity of the Algorithm

We show in this part that the *CQA_sel* algorithm indeed decides certainty for selection, and that it does it in a quite efficient way.

For this purpose, we will have to study the evolution of run snapshots evolve during the evaluation of a stream. We will also have to introduce new notations for manipulating position-annotated streams: this is because the algorithm for the monadic case is more interested into position-annotated streams than plain streams. For all run snapshots Q, Q' and element $e \in \Sigma \cup \{tree, x, \neg x\}$, we define the operations

$$e^{\Delta^{rs}}(Q) = \{(q', C) \mid \exists q.\ e^{\Delta}(q) = q' \text{ and } (q, C) \in Q\}$$

and

$$Q@^{\Delta^{rs}}Q' = \{(q@^{\Delta}q', C \bowtie C') \mid (q, C) \in Q, (q', C') \in Q' \text{ and } C \bowtie C' \neq \emptyset\}.$$

We recall that $Q_{\langle\rangle}$ denotes the run snapshot $\langle\rangle^{\Delta} \times \{[]\}$ that associates the set of tree initial states to the singleton containing the empty candidate.

We now define the evaluation of a position-annotated nested word by a SHA. For all run snapshot position-annotated nested words $w, w' \in nWords_{\Sigma}$, run snapshot Q and candidate $\alpha \in \mathcal{C}_{\{x\}}(w) \cup \mathcal{C}_{\{x\}}(w')$, the evaluation by S from Q and with α is such that:

$$\begin{array}{l}
[\![\varepsilon]\!]^S_\alpha(Q) = Q \\
[\![a^\pi]\!]^S_\alpha(Q) = \begin{cases} \{(q, C \cup \{[x/\pi]\}) \mid (q, C \uplus \{[]\}) \in x^{\Delta^{rs}} \circ a^{\Delta^{rs}}(Q)\} & \text{if } \alpha(x) = \pi \\ (\neg x)^{\Delta^{rs}} \circ a^{\Delta^{rs}}(Q) & \text{if } \alpha(x) \neq \pi \end{cases} \\
[\![\langle\widetilde{w}\rangle]\!]^S_\alpha(Q) = Q@^{\Delta^{rs}}Q' \text{ if } Q' = tree^{\Delta^{rs}}([\![\widetilde{w}]\!]^S_\alpha(\langle\rangle^{\Delta} \times \{[]\})) \\
[\![\widetilde{w}\widetilde{w}']\!]^S_\alpha(Q) = [\![\widetilde{w}']\!]^S_\alpha([\![\widetilde{w}]\!]^S_\alpha(Q))
\end{array}$$

Finally, we denote the evaluation of $\widetilde{w}$ on S from Q with all the possible candidates by

$$[\![\widetilde{w}]\!]^S(Q) = \bigcup_{\alpha \in \mathcal{C}_{\{x\}}(w)} [\![\widetilde{w}]\!]^S_\alpha(Q).$$

For deciding whether a candidate is returned by a monadic query on some nested word, it is sufficient to evaluate the position-annotated version of the nested word on some SHA that represents the query, starting from the initial run snapshot and with the candidate, before testing whether the a final state carries the candidate in the returned run snapshot.

Lemma 8 *Let $\boldsymbol{Q}$ be a monadic query over Σ with variable $\{x\} \subseteq \mathcal{W}$ represented by a query SHA S with set of initial states I and set of final states F. Let $Q_I = I \times \{[]\}$. For all nested word $w \in nWords_{\Sigma}$:*

$$\boldsymbol{Q}(w) = \{\alpha \in \mathcal{C}_{\{x\}}(w) \mid \exists (q, C) \in [\![\widetilde{w}]\!]^S_\alpha(Q_I).\ \alpha \in C, q \in F.\}$$

where $\widetilde{w} = pan_{\Sigma}(w)$.

Proof: Let w be a nested word, $\widetilde{w} = pan_{\Sigma}(w)$ a position-annotated nested word and $\alpha \in \mathcal{C}_{\{x\}}(w)$ a candidate. We know that $\alpha \in \mathbf{Q}(w)$ iff $w \star \alpha \in L(\mathbf{Q})$, that is $w \star \alpha \in L(S)$. Then we have

Claim 1 *$w \star \alpha \in L(S)$ iff there exists $q \in F$ and a set of candidates C so that $(q, C \uplus \alpha) \in [\![\widetilde{w}]\!]^S_\alpha(Q_I)$.*

The proof of Claim 1 is by an easy induction on the structure of w – and thus on the structure of $\widetilde{w}$.

It follows directly from Claim 1 that $\mathbf{Q}(w) = \{\alpha \in \mathcal{C}_{\{x\}}(w) \mid \exists (q, C) \in [\![\widetilde{w}]\!]^S_\alpha(Q_I).\ q \in F, \alpha \in C\}$.

□

Next, we show that the $eval\_sel_\kappa$ function simulates the evaluation of a position-annotated nested word with all the possible candidates. To do so, we consider a version of the CQA_sel function where the candidate manager is not used. Thus no candidates will be killed, and the candidates that could have been reported as certain for selection at some earlier point of time are still taken into account and used for further computations. Even if removing such candidates has no effect on the correctness of the algorithm – as we'll see –, conserving them into memory makes the proof much clearer.

From here and until stated otherwise, we consider a version of the $eval\_sel_\kappa$ function wherein the statement at Line 25 in Figure 4.6 is removed, or replaced by some arbitrary neutral statement that has no effect at all on the further steps of the algorithm.

Lemma 9 *For all position-annotated nested word $\widetilde{w}$, run snapshot Q, type $\kappa \in \{h, t\}$ and sets of states $R, T \subseteq Q_\kappa$:*

$$eval\_sel_\kappa(\widetilde{w}, Q, R, T) = \begin{cases} fusion([\![\widetilde{w}]\!]^S(Q)) & \text{if } \kappa = h \\ fusion(tree^{\Delta^{rs}}([\![\widetilde{w}]\!]^S(Q))) & \text{if } \kappa = t \end{cases}$$

Proof: We start by the following claim:

Claim 2 *For all run snapshot Q, stream s and set of candidates M mapping x to elements of some set $K \supseteq \mathcal{C}_{\{x\}}(s)$, it holds that $[\![\widetilde{s}]\!]^S(Q) \supseteq \bigcup_{\alpha \in M} [\![\widetilde{s}]\!]^S_\alpha(Q)$.*

Proof: The proof of the claim is by induction on $\widetilde{s}$. □

Now we prove the lemma by induction on the structure of $\widetilde{w}$. The proof is quite the same regardless the type. We thus only present the cases where the type is h. Recall that since $fusion$ only turns relational run snapshots into functional run snapshots, $fusion(Q') = fusion(fusion(Q')) = fusion(\ldots(fusion(Q'))\ldots)$ for all run snapshot Q'.

Case $\widetilde{w} = \varepsilon$**.** Then $eval\_sel_h(\widetilde{w}, Q, R) = fusion(Q)$. Furthermore, $Pos(w) = \emptyset$ and so $[\![\widetilde{w}]\!]^S(Q) = [\![\widetilde{w}]\!]^S_\emptyset(Q) = Q$.

Case $\widetilde{w} = a^\pi \widetilde{w}'$ where $a \in \Sigma$, $\pi \in Pos(w)$ and $\widetilde{w}'$ is a position-annotated pattern. Setting $Q_1 = fusion(\{(a^\Delta(q), C) \mid (q, C) \in Q\})$ and $Q_2 =$

$readVar^{\Delta}(Q_1, \pi, x)$, we have that $eval\_sel_h(\widetilde{w}, Q, R) = eval\_sel_h(\widetilde{w}', Q_2, R)$. By the induction hypothesis, $eval\_sel_h(\widetilde{w}', Q_2, R) = fusion([\![\widetilde{w}']\!]^S(Q_2))$. On the other hand side, let α_0 be the candidate so that $\alpha_0(x) = \pi$, and $Q_2' = \bigcup_{\alpha \in \mathcal{C}_{\{x\}}(w)} [\![a^\pi]\!]^S_\alpha(Q)$. Then we have $[\![\widetilde{w}]\!]^S(Q) = [\![\widetilde{w}']\!]^S(Q_2') \cup [\![\widetilde{w}']\!]^S_{\alpha_0}(Q_2') = [\![\widetilde{w}']\!]^S(Q_2')$ by Claim 2. Remark that $[\![a^\pi]\!]^S_{\alpha_0}(Q) = \{(q, C \cup \alpha) \mid (q, C \uplus \{[]\}) \in x^{\Delta^{rs}} \circ a^{\Delta^{rs}}(Q)\}$ and for all candidates $\alpha \in \mathcal{C}_{\{x\}}(w)$ that are different from α_0, $[\![a^\pi]\!]^S_\alpha(Q) = (\neg x)^{\Delta^{rs}} \circ a^{\Delta^{rs}}(Q)$. Then $fusion(Q_2') = fusion([\![a^\pi]\!]^S_{\alpha_0}(Q) \cup \neg x^{\Delta^{rs}} \circ a^{\Delta^{rs}}(Q)) = Q_2$. We finally have $eval\_sel_h(\widetilde{w}, Q, R, T) = eval\_sel_h(\widetilde{w}', Q_2, R, T) = fusion([\![\widetilde{w}']\!]^S(Q_2')) = fusion([\![\widetilde{w}]\!]^S(Q))$.

Case $\widetilde{w} = \langle \widetilde{w}_1 \rangle \widetilde{w}_2$ where $\widetilde{w}_1, \widetilde{w}_2$ are position annotated-streams. Then we have that $eval\_sel_h(Q, \widetilde{w}, R, T) = eval\_sel_h(\widetilde{w}_2, Q_1, R, T)$ where $Q_1 = Q@^{\Delta^{rs}} P_1$, $P_1 = eval\_sel_t(\widetilde{w}_1, Q_{\langle\rangle}, R_d, T_d)$, $R_d = safe\text{-}acc\text{-}down_h^{\Delta}(Q, R, T)$ and $T_d = safe\text{-}down_h^{\Delta}(Q, R, T)$. On the other hand, let $P_2 = \bigcup_{\alpha \in \mathcal{C}_{\{x\}}(\widetilde{w})} tree^{\Delta^{rs}}([\![\widetilde{w}_1]\!]^S_\alpha(Q_{\langle\rangle}))$. It follows from Claim 2 that $P_2 = tree^{\Delta^{rs}}([\![\widetilde{w}_1]\!]^S(Q_{\langle\rangle}))$. Setting $Q_2 = Q@^{\Delta^{rs}} P_2$, we have by Claim 2 that $[\![\widetilde{w}]\!]^S(Q) = [\![\widetilde{w}_2]\!]^S(Q_2) \cup \bigcup_{\alpha \in \mathcal{C}_{\{x\}}(w_1)} [\![\widetilde{w}_2]\!]^S_\alpha(Q_2) = [\![\widetilde{w}_2]\!]^S(Q_2)$. By the induction hypothesis, $P_1 = fusion(P_2)$, implying that $Q_1 = Q_2$. The induction hypothesis also implies that $eval\_sel_h(\widetilde{w}_2, Q_1, R, T) = fusion([\![\widetilde{w}_2]\!]^S(Q_2))$. So $eval\_sel_h(Q, \widetilde{w}, R, T) = fusion([\![\widetilde{w}]\!]^S(Q))$.

□

The explicit stack used by the *CQA_sel* function plays an important role when it comes to retrieve the candidates that are certain for selection. We show in the next proposition an invariant on the elements that are pushed on it.

But first let us define a *suffix stream* s' of a stream s as stream whose instances are all instances of s, that is:

Definition 18 *A stream s' is called a suffix stream of another stream s whenever $Inst(s') \subseteq Inst(s)$.*

Proposition 8 *Let a position-annotated stream $\widetilde{s}$ and a natural $i > 0$ so that $\widetilde{s} = \widetilde{w}_0\langle \ldots \langle \widetilde{w}_{i-1}\langle \widetilde{s}_i \rangle X_{i-1} \rangle \ldots \rangle X_1$, where $\widetilde{w}_0, \ldots, \widetilde{w}_i$ are position-annotated nested words and $X_0, \ldots, X_{i-1} \in \mathcal{V}$ pattern variables. For all run snapshot Q, sets of tree states R, T and suffix stream $\widetilde{s}_i'$ of $\widetilde{s}_i$, the stack when calling $eval\_sel_t(\widetilde{s}_i', Q, R, T)$ has the same content than the list $(Q_{i-1}, R_{i-1}, T_{i-1}) :: \ldots :: (Q_0, R_0, T_0) :: nil$, where $Q_j = fusion([\![\widetilde{w}_j]\!]^S(Q_{\langle\rangle}))$, $R_j = safe\text{-}acc\text{-}down_t^{\Delta}(Q_{j-1}, R_{j-1}, T_{j-1})$, $T_j = safe\text{-}down_t^{\Delta}(Q_{j-1}, R_{j-1}, T_{j-1})$ for $0 < j < i$, $Q_0 = fusion([\![\widetilde{w}_0]\!]^S(I \times \{[]\}))$, $R_0 = F$ and $T_0 = Q_h \setminus schema\text{-}state_h^{-1}(F^{one\text{-}x})$.*

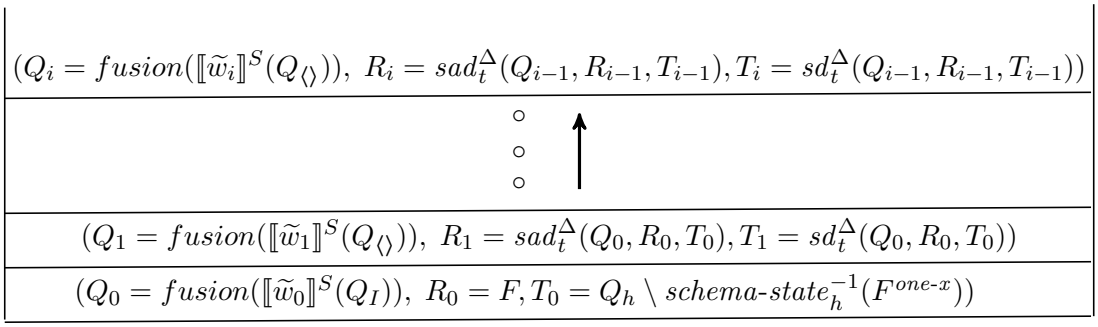


Figure 4.9: Content of *stack* on any call $eval\_sel_t(\widetilde{s}'_{i+1}, Q, R, T)$ where $\widetilde{s}'_{i+1}$ is a suffix stream of $\widetilde{s}_{i+1}$, during the evaluation of $\widetilde{w}_1\langle \dots \langle \widetilde{w}_i\langle \widetilde{s}_{i+1}\rangle X_i\rangle \dots \rangle X_1$. Here sad_κ^Δ and sd_κ^Δ stand respectively for $safe\text{-}acc\text{-}down_\kappa^\Delta$ and $safe\text{-}down_\kappa^\Delta$

Proof: Let $Q_I = I \times \{[]\}$. Figure 4.9 describes the situation. The proof is by induction on i.

Case $i = 1$, meaning that $\widetilde{s} = \widetilde{w}_0\langle\widetilde{s}_1\rangle X_0$. The content of the stack is empty at the beginning. Then the algorithm calls $eval\_sel_h(\widetilde{w}_0, Q_I, R_0, T_0)$ where $R_0 = F$ and $T_0 = Q_h \setminus schema\text{-}state_h^{-1}(F^{one\text{-}x})$. During this call, the algorithm processes all the symbols in $\widetilde{w}_0$. After having processed the last symbol of $\widetilde{w}_0$, the remaining part of $\widetilde{s}$ to process is $\langle\widetilde{s}_1\rangle$. In this case, it pushes the triplet constituted by the returned run snapshot after the last symbol of $\widetilde{w}_0$, R_0 and T_0. Lemma 9 tells us that the returned run snapshot after the last symbol of $\widetilde{w}_0$ is $[\![\widetilde{w}_0]\!]^S(Q_I)$. Then the algorithm calls $eval\_sel_t(\widetilde{s}_1, Q_{\langle\rangle}, R_1, T_1)$, where $R_1 = safe\text{-}acc\text{-}down_h^\Delta([\![\widetilde{w}_0]\!]^S(Q_I), R_0, T_0)$ and $T_1 = safe\text{-}down_h^\Delta([\![\widetilde{w}_0]\!]^S(Q_I), R_0, T_0)$. Thus the content of the stack right after this call is then $([\![\widetilde{w}_0]\!]^S(Q_I), R_0, T_0)$. The content of the stack for all the future calls on suffix streams of $\widetilde{w}_1$ will remain globally unchanged, even if the stack may grow from time to time on some nested calls. But then every newly pushed triplet would subsequently be popped.

case $i > 1$. Assume that $\widetilde{s}_i = \widetilde{w}_i\langle\widetilde{s}_{i+1}\rangle X_i$, where $\widetilde{w}_i$ is a position-annotated nested word, $\widetilde{s}_{i+1}$ a position-annotated stream and X_i a pattern variable. So $\widetilde{s} = \widetilde{w}_1\langle \dots \langle \widetilde{w}_{i-1}\langle \widetilde{w}_i\langle \widetilde{s}_{i+1}\rangle X_i\rangle X_{i-1}\rangle \dots \rangle X_1$. Using the induction hypothesis and following the same reasoning than the above case, we can show that the content of the stack when a call $eval\_sel_t(\widetilde{s}', \_, \_, \_)$ is made for any suffix stream $\widetilde{s}'$ of $\widetilde{s}_{i+1}$ has the same content than the list $(Q_i, R_i, T_i) :: \dots :: (Q_0, R_0, T_0) :: nil$, where $Q_j = fusion([\![\widetilde{w}_j]\!]^S(Q_{\langle\rangle}))$, $R_j =$

safe-acc-down$_t^{\Delta}(Q_{j-1}, R_{j-1}, T_{j-1})$, and T_j = *safe-down*$_t^{\Delta}(Q_{j-1}, R_{j-1}, T_{j-1})$ for $0 < j < i+1$, $Q_0 = fusion([\![\widetilde{w}_0]\!]^S(Q_I))$, $R_0 = F$ and $T_0 = Q_h \setminus$ *schema-state*$_h^{-1}(F^{one\text{-}x})$.

□

So far, the $[\![\cdot]\!]^S_\alpha(Q)$ function has been defined for position-annotated nested words. We now extend it to position-annotated streams, so that for some pattern variable $X \in \mathcal{V}$:

$$[\![X]\!]^S_\alpha(Q) = acc^{\Delta^{rs}}(Q).$$

The notation $[\![\widetilde{s}]\!]^S(Q)$ is also lifted with respect to the above extension. The next lemma gives a characterization of the candidates that are certain for selection for a query $\mathbf{Q}$ on a position-annotated stream. It basically states that a necessary and sufficient condition for a candidate to be certain for selection for a query is that all the possible runs of the stream annotated with the candidate must either lead to a final state, or to a state whose schema state is not final for *one-x*.

Lemma 10 *Let $s \in nWords_\Sigma$ be a nested word, $\widetilde{s} = pan_\Sigma(s)$ its position-annotated version, $\alpha \in \mathcal{C}_{\{x\}}(s)$ a complete candidate and $\boldsymbol{Q}$ a monadic query over Σ with variables $\{x\}$ represented by a complete and one-x-canonic dSHA S with set of initial (resp. final) states I (resp F). α is certain for selection for $\boldsymbol{Q}$ by s iff for all $q \in Q_h$, $(\exists C \subseteq \mathcal{C}_{\{x\}}(s).\ (q, C \uplus \{\alpha\}) \in [\![\widetilde{s}]\!]^S_\alpha(I \times \{[]\})) \Rightarrow q \in F \vee$ schema-state$_h(q) \notin F^{one\text{-}x}$.*

Proof: First let us remark the following equality stating that the set of run snapshots obtained by making the union all the run snapshots of all the instances is equal to the set of run snapshots where every pattern variable is replaced by the accessibility relation:

Claim 3 $\bigcup_{\mu \in M} [\![[\![s \star \alpha]\!]^\mu]\!]^S_\alpha(I \times \{[]\}) = [\![\widetilde{s}]\!]^S_\alpha(I \times \{[]\})$

Proof: The proof is straightforward. □

Now according to Definition 3, α is a certain answer for $\mathbf{Q}$ if for all assignment $\mu : fv(s) \to nWords_\Sigma$, the instance $[\![s]\!]^\mu \star \alpha$ is in $L(\mathbf{Q})$. Let $M = \{\mu\colon fv(s) \to nWords_{\Sigma \cup \{x, \neg x\}}\}$ be the set of assignment mapping the free variables of s to nested words over $\Sigma \cup \{x, \neg x\}$, and $Q_I = I \times \{[]\}$. Since $L(\mathbf{Q})$ is a language of $\{x\}$-structures,

it follows that α is a certain answer for $\mathbf{Q}$ iff

$$\bigcup_{\mu \in M} [\![s \star \alpha]\!]^\mu \cap L(\textit{one-x}) \subseteq L(\mathbf{Q})$$
$\Leftrightarrow$ $(\forall \mu \in M.\ [\![s \star \alpha]\!]^\mu \in L(\textit{one-x}) \Rightarrow [\![s \star \alpha]\!]^\mu \in L(\mathbf{Q}))$
$\Leftrightarrow$ $(\forall \mu \in M.\ [\![s \star \alpha]\!]^\mu \in L(\textit{one-x}) \Rightarrow \exists (q, C) \in Q^S_\mu.\ q \in F, \alpha \in C)$ by Lemma 8, where $Q^S_\mu = [\![[\![s \star \alpha]\!]^\mu]\!]^S_\alpha(Q_I)$
$\Leftrightarrow$ $(\forall \mu \in M.\ (\exists (q_{\textit{one-x}}, C') \in Q^{\textit{one-x}}_\mu.\ q_{\textit{one-x}} \in F^{\textit{one-x}}, \alpha \in C')) \Rightarrow \exists (q, C) \in Q^S_\mu.\ q \in F, \alpha \in C)$ by Lemma 8, where $Q^{\textit{one-x}}_\mu = [\![[\![s \star \alpha]\!]^\mu]\!]^{\textit{one-x}}_\alpha(Q_I)$ and $I^{\textit{one-x}}$ is the set of initial states of *one-x*
$\Leftrightarrow$ $\forall \mu \in M.\ \exists (q, C) \in Q^S_\mu.\ (q \in F \vee \textit{schema-state}_h(q) \notin F^{\textit{one-x}}) \wedge \alpha \in C$
$\Leftrightarrow$ $\forall \mu \in M, q \in Q_h.\ \exists C \subseteq \mathcal{C}_{\{x\}}(s).\ (q, C \uplus \{\alpha\}) \in Q^S_\mu \Rightarrow (q \in F \vee \textit{schema-state}_h(q) \notin F^{\textit{one-x}})$ by the determinism of S
$\Leftrightarrow$ $\forall q \in Q_h.\ \exists C \subseteq \mathcal{C}_{\{x\}}(s).\ (q, C \uplus \{\alpha\}) \in [\![\widehat{s}]\!]^S_\alpha(Q_I) \Rightarrow q \in F \vee \textit{schema-state}_h(q) \notin F^{\textit{one-x}}$ by Claim 3

.

□

We can now show that the select function of the candidate manager outputs and kills only candidates that are certain for selection.

Proposition 9 (Correctness of the select function of the candidate manager) *Let $\mathbf{Q}$ be a monadic query with alphabet Σ and query variables in $\{x\}$, $s \in \textit{nStreams}_\Sigma$ a stream and α a candidate. Then α is dead if and only if α is certain for selection for $\mathbf{Q}$.*

Proof: Let $Q_I = I \times \{[]\}$. It is sufficient to notice that the $\textit{eval\_stack}^\Delta$ function defined in the candidate manager computes exactly $\bigcup_{\alpha \text{ alive}} [\![\widehat{s}]\!]^S_\alpha(Q_I)$, that is, $[\![\widehat{s}]\!]^S(Q_I)$ restricted to the only candidates that have not been proved to be certain for selection yet. Thus at Line 7, the *select* function computes the set of candidates that are certain for selection by Lemma 10. It then marks and outputs them at Line 8. □

Proposition 9 shows one of the effects of the candidate manager on the algorithm is to clean the stack and the run snapshot given to the *select* function by getting rid of the candidates that are already certain for selection. Thus candidates that are alive, those for which certainty cannot be decided yet, are not touched by the candidate manager. The other effect of the candidate manager is to recompute the target sets. Note that those values may change only if a candidate that is marked is on the stack. Knowing this, we can now safely put back in place Line 25, since removing from the run snapshots states associated only to certain for selection candidates has no impact on the further processing.

Proposition 10 (Correctness and completeness of *CQA_sel*) *When*

CQA_sel is parameterized by a complete and one-x-canonic dSHA S representing a monadic query $\boldsymbol{Q}$ with query variables in $\{x\}$, then a candidate α of a stream s is killed and output by CQA_sel(s) via its candidate manager if and only if α is certain for selection for $\boldsymbol{Q}$ on s.

Proof: Given that the algorithms in Figures 4.3 and 4.6 work quite similarly, it can be derived from Proposition 9 the condition at Line 24 of Figure 4.6 is somehow equivalent to the condition at Line 14 of Figure 4.3. Recall that the condition $Q' \cap \mathit{safe}^{\Delta}_{\kappa}(R) \neq \emptyset$ at Line 14 of Figure 4.3 is true if and only if there is a state in Q' from which only states in R can be reached. Thus the condition $\Psi \cap \mathit{safe}^{\Delta}_{\kappa}(R \cup T) \neq \emptyset$ in Line 24 of Figure 4.6 where $\Psi = \{q \in \mathit{adom}(Q_2) \mid \mathit{schema\text{-}state}_h(q) \in \mathit{hedge\text{-}schema\text{-}state}(R)\}$ is in the same way true iff there is a state in $\mathit{adom}(Q_2)$ with a schema state equal to one of the schema states of R, and from which only states in R are reachable and $R \cup T$ can be reached. Proposition 8 established that the target states at some level are derived from the target states of the upper level, and that the target states at the top-level are the set of final states plus those having schema states that are not final in *one-x*. So the condition at Line 24 of Figure 4.6 is true iff and only if there is a state in the active domain of Q_2 from which states that are final or out-of-the-schema can be reached. Thus, by Proposition 10, the condition at Line 24 of Figure 4.6 is true iff there is a candidate that is certain for selection and that has not been killed yet. In the case when the condition is true, the statement at Line 25 is executed, and by Proposition 9, all the candidates that are certain for selection and that have not been killed so far are killed. □

We finally study the time complexity of the *CQA_sel* function. Thanks to its ability of testing whether there are some new candidates that are certain for selection without actually retrieving any candidate, the algorithm has quite a good complexity, compared to the other ones of the state of the art.

Theorem 2 (Time complexity) *Let s be a stream and $\boldsymbol{Q}$ a monadic query represented by a complete and one-x-canonic dSHA S. The certainty for selection of any candidate $\alpha \in \mathcal{C}_{\{x\}}(s)$ for $\boldsymbol{Q}$ can be decided at the earliest event in time $O(|s||S| + |\boldsymbol{Q}(s)||\mathit{maxdepth}(s)||\mathit{acc}^{\Delta}_{h\times t}|)$.*

Proof: We set $S = (Q_h, Q_t, \Sigma, \Delta, I, F)$, and first study the per-event complexity. As for the boolean case, the number of operations that is done for each element of $\widetilde{s}$ is in $O(|A|)$. We assume that a representation of the join of two sets of candidates $C \bowtie C'$ can be computed in constant time.

Case $\widetilde{s} = \varepsilon$: Then either Line 18 or Line 19 is executed, depending on κ. Each of them can be done in $O(|S|)$.

Case $\widetilde{s} = a^{\pi}\widetilde{s}'$ for $a^{\pi} \in \mathit{Nodes}(\widetilde{s})$: computing Q_1 at Line 21, Q_2 at Line 22 and Ψ at Line 23 can be done in time $O(|S|)$. Furthermore, testing in Line 24 whether there is a certain for selection candidate is also in $O(|S|)$. In the case where the test returns true, an additional cost for gathering and retrieving the certain for selection candidates will apply. We study this case later.

Case $\widetilde{s} = \langle \widetilde{s}_1 \rangle \widetilde{s}_2$**:** It takes a number of operations in $O(|S|)$ to compute P at Line 31 and a constant time to push and pop the stack. Then Q' at Line 32 is computed in $O(|S|)$.

Case $\widetilde{s} = X \in \mathcal{V}$**:** reduces to one of the above cases.

This shows that there are at least $O(|S|)$ operations for each event, setting the complexity for evaluating the stream and testing the existence of candidates that are certain for selection to $O(|s||S|)$. Now let's analyze the complexity of retrieving the candidates that are certain for selection. The *select* function of the candidate manager is called in this case. This function uses the $\mathit{eval\_stack}^{\Delta}$ and $\mathit{recompute\text{-}targets}^{\Delta}$ functions in Figure 4.8 as subroutines, which recursively call themselves as many times as there are elements in the stack. For instance, let's consider $\mathit{eval\_stack}^{\Delta}$. When L is the empty list, then computing the returned run snapshot at Line 16 can be done in time $O(|\mathit{acc}^{\Delta}_{h\times h}|)$. Furthermore, in the case where $L = (Q_{\mathit{head}}, \_) :: L'$ it takes $O(|\mathit{acc}^{\Delta}_{h\times t}|)$ and $O(|A|)$ operations to compute respectively Q' at Line 18 and Q'' at Line 19. Note that $|\Delta| \leq |\mathit{acc}^{\Delta}_{h\times t}|$. The number of times that the retrieval of candidates is done depends on the number of solutions $|\mathbf{Q}(s)|$ of the query, while each call to $\mathit{eval\_stack}^{\Delta}$ takes $O(|\mathit{maxdepth}(s)||\mathit{acc}_{\Delta}|)$, where $\mathit{maxdepth}(s)$ is the maximum depth of the input and thus the maximum depth of the stack. The additional cost is then in $O(|\mathbf{Q}(s)||\mathit{maxdepth}(s)||\mathit{acc}^{\Delta}_{h\times t}|)$. Hence the overall complexity of this algorithm is in $O(|s||S| + |\mathbf{Q}(s)||\mathit{maxdepth}(s)||\mathit{acc}^{\Delta}_{h\times t}|)$. □

4.5 Experiments

We have developed a prototype of our boolean and monadic CQA algorithms using the Scala programming language. It has been tested on the navigational queries of the benchmark of [Franceschet] and on additional queries with comparisons to constant values, presented in Section .2 of the appendix.

However it is not optimized enough to be competitive, and for this reason we have not added its running time for the different tests that we have made.

Chapter 5

Hyperstreams and Certain Query Answering

Contents

This chapter presents the concept of hyperstream more formally, and introduces some definitions that we'll use later. It also defines the certain query answering problem for hyperstreams.

5.1 Hyperstreams

Hyperstreams are descriptors of patterns. They allow to have references to parts of a given pattern, and to reuse these references several times. Such patterns with references can be seen as *incomplete* versions of *singleton context-free grammars* [Plandowski 1995], where the rules of some nonterminals may be missing.

We propose here to represent hyperstreams by particular grammars. For instance, the hyperstream in Figure 5.1 has terminals in $\{a, b, c\}$ and nonterminals in $\{\mathcal{S}, X, Y, Z\}$:

The nonterminals are the references of the hyperstream. The two patterns of the above hyperstream are given by grammar rules for the references $\mathcal{S}$ and X, while there is no rule for the other two references Y and Z. The missing rules for these references can be added in the future one by one by the hyperstreaming environment. A hyperstream is called a singleton context-free grammar – or equivalently a *straight line program* [Babai & Szemeredi 1984] – if it has no missing rule. It is well-known

$$\mathcal{S} \;\to\; \langle aX\langle bb\rangle Y\rangle aX, \qquad X \;\to\; \langle Y\rangle cZa.$$

Figure 5.1: Hyperstream represented by a grammar

that a singleton context-free grammar defines a word. Well-nesting is an additional criterion to be checked. The object of interest here is the set of nested words that can be obtained by completing a hyperstream to a singleton context-free grammar, or equivalently, the set of instances of the nested pattern.

Sharing. Hyperstreams can be identified with *compressed patterns*. A nonterminal can occur multiple times in the right-hand sides of rules, allowing to *share* its content. The hyperstream in Figure 5.1 for instance represents the nested pattern

$$\langle a\underline{\langle Y\rangle cZa}\langle bb\rangle\, Y\rangle\, a\underline{\langle Y\rangle cZa}$$

in a compressed manner, while sharing the two underlined factors that were replaced for the two occurrences of X. A hyperstream with no sharing is called *compression-free*.

Variables of all Orders. The values associated to nonterminals can be of different types. They could be for instance restricted to trees, words, hedges or be general nested words. In Figure 5.1, the values allowed for $\mathcal{S}$ and X are general nested words. However, we could restrict Y to trees and Z to hedges.

It is also possible to have values of *higher-order*, that are functions or functions of functions, etc. A second order value could be a function that takes a nested pattern and returns another nested pattern. For instance, $\lambda X.\langle XaX\rangle$ is a function that takes a pattern X and returns a new pattern $\langle XaX\rangle$. Note the usage of the λ symbol for abstraction. This function can be applied to a nested pattern, say $\langle b\rangle c$, and return $(\lambda X.\langle XaX\rangle)@(\langle b\rangle c) = \langle\langle b\rangle ca\langle b\rangle c\rangle$. The @ symbol is used to apply the function.

5.1.1 Hyperstreams of Nested Words

A hyperstream that represents a nested pattern is called a *nested hyperstream*. Formally,

Definition 19 *A nested hyperstream is a tuple $G = (N, \Sigma, R, \mathcal{S})$ such that $N \subsetneq \mathcal{V}$ is a finite set of nonterminals, alphabet Σ is a finite set, rule set R is a partial function from $dom(R) \subseteq N$ to $nPatterns_\Sigma$ such that $fv(R(X)) \subseteq N$ for all $X \in dom(R)$, and $\mathcal{S} \in N$ is the starting symbol. Furthermore, the directed graph which links all $X \in dom(R)$ to all $X' \in fv(R(X))$ must be acyclic.*

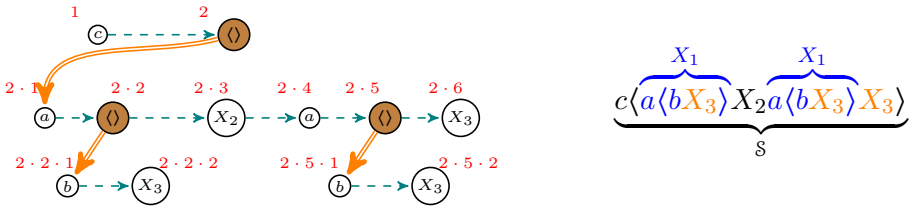


Figure 5.2: Pattern $pat(G)$ described by G

The set of nested hyperstreams over Σ is denoted by Hyp_Σ. The set of *free variables* of G is $fv(G) = N \setminus dom(R)$. The variables in $dom(R)$ are *bound* in G. The size of G is the sum of the sizes of the right-hand sides of the rules plus the number of its nonterminals: $|G| = \sum_{X \in dom(R)} |R(X)| + |N|$. Any nested hyperstream $G \in Hyp_\Sigma$ represents a nested pattern $pat(G) \in nPatterns_\Sigma$ in a compressed manner, by recursively replacing every nonterminal X by its definition $R(X)$:

$$pat(G) = R(\mathcal{S})[X/pat(G[\mathcal{S}/X])].$$

G is called *linear* if $pat(G)$ is linear. Note that a linear hyperstream may still use compression but for describing a linear pattern. The set of linear hyperstreams is denoted $LinHyp_\Sigma$.

It is called *compression-free* if for any sequence of distinct variables $X_1 \ldots, X_n \in dom(R)$ the concatenated nested pattern $R(X_1) \ldots R(X_n)$ is linear.

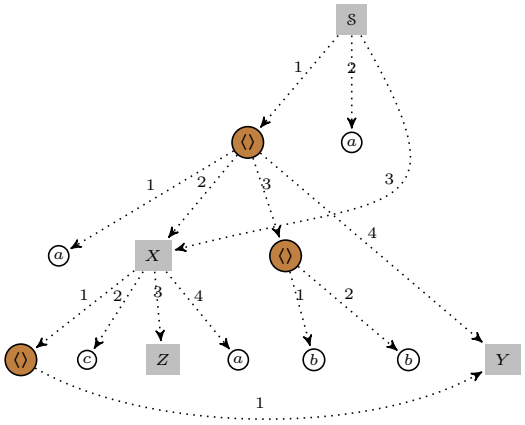
We will freely identify compression-free nested hyperstreams G with the nested pattern $pat(G)$, given that G can be converted into $pat(G)$ in linear time (due to the absence of compression). In this sense, the inclusion $nPatterns_\Sigma \subseteq Hyp_\Sigma$ holds.

Example 12 *Let* $G = (N, \Sigma, R, \mathcal{S})$ *be nested hyperstream where* $N = \{\mathcal{S}, X_1, X_2, X_3\}$, $\Sigma = \{a, b, c\}$, $R = \{\mathcal{S} \to c\langle X_1 X_2 X_1 X_3\rangle, X_1 \to a\langle bX_3\rangle\}$. *It describes the pattern represented in Figure 5.2 and is neither compression-free – the bound variable* X_1 *has two occurrences– nor linear – the free variable* X_3 *occurs more than once in* $pat(G)$.

A class of nested hyperstreams $\mathcal{G}$ is a function that maps any signature Σ to a subset of nested hyperstreams $\mathcal{G}_\Sigma \subseteq Hyp_\Sigma$. For instance, *Hyp*, *LinHyp*, *nPatterns* and *nLinPatterns* are classes of nested hyperstreams.

Note that hyperstreams of words are special hyperstreams where the values of the nonterminals are restricted to words.

Logical Structure. As for nested patterns, a nested hyperstream can be described by a logical structure. Unfortunately, the first-child and next-sibling relations will not be enough. A shared nonterminal could have different next-siblings, and different



$$\begin{aligned}\mathcal{S} &\to \langle aX\langle bb\rangle Y\rangle aX,\\ X &\to \langle Y\rangle cZa.\end{aligned}$$

Figure 5.3: Logical structure of a hyperstream

parents. Instead, we consider a set of child relations ch_i indexed by integers $i > 0$ that indicate the position of the child with respect to its parent.

Example 13 *Figure 5.3 shows the logical structure of the hyperstream in Figure 5.1. The positions labeled by nonterminals are represented by rectangles where the label is written inside. The ch_i relations are represented by dotted arrows labeled by i.*

For all nested hyperstream G, we enforce the positions of $\rho = pat(G)$ to be defined as words over the set of naturals $\mathbb{N}$ such that:

$$\begin{aligned}&root(\rho) = 1\\ &\forall(\pi,\pi') \in ch^\rho, \pi' = \pi \cdot 1\\ &\forall(\pi,\pi') \in ns^\rho, \exists \pi'' \in \mathbb{N}^* \text{ and } i > 0 \text{ such that } \pi = \pi'' \cdot i \text{ and } \pi' = \pi'' \cdot i + 1\end{aligned}$$

In Figure 5.2, the positions of $pat(G)$ are marked in red.

5.1.2 Hyperstreams of Ranked Trees With Context Variables

We are going to study hyperstreams of ranked trees in which *contexts* variables are allowed. We formalize these objects here after. Throughout this subsection, the terms trees and ranked trees both refer to ranked trees.

5.1.2.1 Contexts

We consider the two types in $\mathbf{T} = \{tree, context\}$. We assume that any ranked signature contains at least one constant and one symbol of arity at least 2.

For defining contexts, we fix an arbitrary nonempty set $\mathcal{V}^{tree}$ disjoint from Σ, whose elements are variables of type *tree*. We take an approach based on the λ-terms

$$\begin{array}{rcll} p, p', p_1, \ldots, p_n \in \mathcal{P}_\Sigma^{tree} & ::= & x \mid f(p_1, \ldots, p_n) \mid P@p & \\ P \in \mathcal{P}_\Sigma^{context} & ::= & X \mid \lambda x.p' & \text{where } x \text{ occurs exactly once in } p' \end{array}$$

Figure 5.4: Tree and context patterns where $x \in \mathcal{V}^{tree}$, $X \in \mathcal{V}^{context}$, $f \in \Sigma^{(n)}$ and $n \geq 0$.

but will not consider values of higher types.

Definition 20 *The set of contexts $\mathcal{C}_\Sigma$ is the set of all terms $\lambda x.t$ for some ranked tree $t \in \mathcal{T}_{\Sigma \cup \{x\}}$ where $x \in \mathcal{V}^{tree}$ occurs exactly once in t, and this with arity* 0*.*

We will identify contexts modulo α-renaming so that the choice of variable x does not matter. This means that the contexts $\lambda x.t$ and $\lambda x'.t[x/x']$ are equal for all $x, x' \in \mathcal{V}^{tree}$. The variable serves as the hole marker of the context.

A ranked tree in $\mathcal{T}_\Sigma$ is a value of the first-order type *tree*, and a context in $\mathcal{C}_\Sigma$ a value of the linear second-order type $context = tree \multimap tree$. This is the subtype of the more usual function type $tree \to tree$ that is restricted to linear functions using their argument exactly once. Any context $\lambda x.t \in \mathcal{C}_\Sigma$ denotes a linear function since x must occur exactly once in t by definition. The set of all values of both types is:

$$Val_\Sigma = \mathcal{T}_\Sigma \cup \mathcal{C}_\Sigma$$

5.1.2.2 Patterns of Ranked Trees

We next extend ranked trees and contexts to patterns by adding variables of both types. For this we assume a set $\mathcal{V} = \uplus_{\tau \in \mathbf{T}} \mathcal{V}^\tau$ with two kinds of variables. Variables $x, y, z \in \mathcal{V}^{tree}$ have type *tree* and variables $X, Y \in \mathcal{V}^{context}$ type *context*.

Patterns for ranked trees $p \in \mathcal{P}_\Sigma^{tree}$ and patterns for contexts $P \in \mathcal{P}_\Sigma^{context}$ are defined in Figure 5.4. The set of all patterns is $\mathcal{P}_\Sigma = \uplus_{\tau \in \mathbf{T}} \mathcal{P}_\Sigma^\tau$. In both kinds of patterns, tree variables x may now occur freely but can also be bound in the scope of a λ-binder as before. Context variables X can also occur in both kinds of patterns, but will always be free. For instance, the tree pattern $X@(\lambda y.f(y, a)@x)$ in $\mathcal{P}_\Sigma^{context}$ contains the free context variable X, the bound tree variable y and the free tree variable x. Up to β-reduction this pattern is equal to $X@f(a, x)$ which also belongs to $\mathcal{P}_\Sigma^{context}$.

The set of free variables $fv(p)$ and $fv(P)$ and of bound variables $bv(p)$ and $bv(P)$ are defined as usual for λ-terms. A pattern is called *linear* if each of its free variables has at most one free occurrence.

The set $\mathcal{P}_\Sigma^{gr,\tau}$ of ground patterns of type $\tau \in \mathbf{T}$ is the subset of patterns in $\mathcal{P}_\Sigma^\tau$ without free variables. The set of all ground patterns is denoted by $\mathcal{P}_\Sigma^{gr} = \mathcal{P}_\Sigma^{gr,tree} \cup \mathcal{P}_\Sigma^{gr,context}$. Clearly, any ranked tree is a ground pattern of type *tree* and any context is a ground pattern of type *context*, i.e., $\mathcal{T}_\Sigma \subseteq \mathcal{P}_\Sigma^{gr,tree}$ and $\mathcal{C}_\Sigma \subseteq \mathcal{P}_\Sigma^{gr,context}$. The converse is not true. The ground pattern $\lambda x.x@f(a)$ for instance is not a tree. However, it becomes equal to the tree $f(a)$ by β-reduction. The situation is similar for ground pattern for contexts, which can always be reduced to a context by exhaustive β-reduction. The ground context pattern $\lambda x.\lambda y.y@f(x)$ for instance can be β-reduced to the context $\lambda x.f(x)$. In general, each β-reduction step replaces some redex of the form $(\lambda x.p)@p'$ in a bigger pattern by $p[x/p']$ if $x \notin bv(p)$ and otherwise renames x beforehand. Exhaustive β-reduction can be done in any order, but always leads to the same result. We denote the β-reduced form of a tree pattern $p \in \mathcal{P}_\Sigma^{gr,tree}$ by $norm_\beta(p)$ and of a context pattern $P \in \mathcal{P}_\Sigma^{gr,context}$ by $norm_\beta(P)$. The overall reduction requires at most a linear number of steps, since all λ-bound variables in patterns are constrained to occur exactly once (in the scope of the λ-binder). As a consequence, we have $norm_\beta(\mathcal{P}_\Sigma^{gr,tree}) = \mathcal{T}_\Sigma$ and $norm_\beta(\mathcal{P}_\Sigma^{gr,context}) = \mathcal{C}_\Sigma$.

A substitution $\mu\colon V \to \mathcal{P}_\Sigma^{gr}$ on a subset of variables V is called well-typed if it maps tree variables to $\mathcal{P}_\Sigma^{gr,tree}$ and context variables to $\mathcal{P}_\Sigma^{gr,context}$. For any pattern $p \in \mathcal{P}_\Sigma^{tree}$, the grounding $\mu(p) \in \mathcal{P}_\Sigma^{gr,tree}$ is obtained from p by replacing free variables v by $\mu(v)$. The set of all instances of p is obtained by β-normalizing all groundings of p:

$$Inst(p) = \{norm_\beta(\mu(p)) \mid \mu\colon fv(p) \to \mathcal{P}_\Sigma^{gr} \text{ well-typed}\}.$$

Clearly any instance of p is a ranked tree, that is $Inst(p) \subseteq \mathcal{T}_\Sigma$.

Example 14 *Consider the tree pattern $p = X@(X@x)$ and the substitution μ where $\mu(X) = \lambda x.f(b,x)$ and $\mu(x) = a$. The β-normal form of $\mu(p) = \mu(X)@(\mu(X)@\mu(x))$ is the tree $norm_\beta(\mu(p)) = f(b,f(b,a))$ belonging to $Inst(p)$.*

5.1.2.3 Compressed Tree Patterns

We now show how to define patterns with grammar compression for both types by using a variant of singleton tree grammars with contexts. The term compressed tree pattern is used specially to designate such objects.

Definition 21 *A* compressed pattern *of type $\tau \in \boldsymbol{T}$ is an acyclic context-free tree grammar $G = (N, \Sigma, R, \mathcal{S})$ where $N \subseteq \mathcal{V}$ is a finite set of nonterminals, $\mathcal{S} \in N$ of type τ is the start symbol, R is a partial well-typed function from N to patterns in $\mathcal{P}_\Sigma$ with free variables in N. The set of all compressed patterns of type τ is denoted by $\mathcal{P}_\Sigma^{comp,\tau}$.*

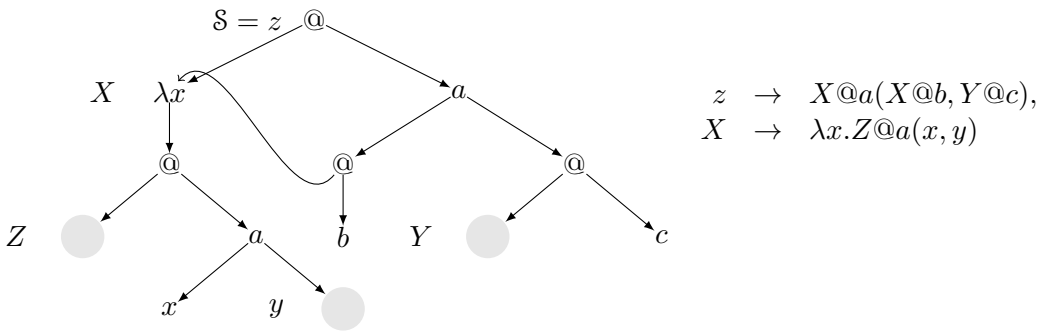


Figure 5.5: Graph and rules of the compressed tree pattern G_0.

For instance, consider the compressed tree pattern $G_0 \in \mathcal{P}_\Sigma^{comp,tree}$ with the nonterminals $N = \{z, X, Y, Z, y\}$, with $\mathcal{S} = z$ and with two rules $R(z) = X@a(X@b, Y@c)$, and $R(X) = \lambda x.Z@a(x, y)$. We illustrate G_0 by the graph in Figure 5.5. Each nonterminal is annotated to the left of the corresponding node. Note that the circled empty nodes correspond to the nonterminals without any rule. The compressed pattern G_0 is acyclic, in that no variable on the left hand side of some rule does appear in any subsequent rule. In other words, the graph of G_0 is a DAG. It should also be noticed that the tree language of the grammar G_0 is $\emptyset$. What interests us instead is its tree pattern:

$$pat(G_0) = (\lambda x.Z@a(x, y))@a((\lambda x.Z@a(x, y))@b, Y@c)$$

The grammar serves to represent this pattern in a compressed manner, by sharing the context pattern referred to by X. By exhaustive β-reduction of $pat(G_0)$ we obtain the following tree pattern with context variables but without λ-binders:

$$norm_\beta(pat(G_0)) = Z@a(a(Z@a(b, y), Y@c), y)$$

A compressed tree pattern is called compression-free if the structure of its grammar is a tree, that is, every nonterminal appears at most once in all the right-hand sides of the rules. We define the free variables of a compressed tree pattern G as the free variables of $pat(G)$, and the bound variables of G as the nonterminals in $dom(R)$.

In what follows we will identify tree patterns $p \in \mathcal{P}_\Sigma^{tree}$ with the compressed tree pattern $ctp_\Sigma(p) = (\{\mathcal{S}\}, \Sigma, \{\mathcal{S} \rightarrow p\}, \mathcal{S})$, which has a single rule mapping a fixed start symbol $\mathcal{S}$ to p. Note that $ctp_\Sigma(p)$ is compression-free. In this sense, $\mathcal{P}_\Sigma^{tree} \subseteq \mathcal{P}_\Sigma^{comp,tree}$. A compressed tree pattern G is called *linear* if its tree pattern $pat(G)$ is linear.

5.2 Certain Query (Non) Answering on Hyperstreams

We next formalize the notions of certain query answers and non-answers on hyperstreams. We fix an alphabet Σ and a finite set $\mathcal{W}' \subseteq \mathcal{W}$ of query variables for this section.

5.2.1 Definitions

Let $\mathbf{Q}$ be a query over Σ with variables in $\mathcal{W}'$ and $G \in Hyp_\Sigma$ a nested hyperstream.

Definition 22 *A candidate α of $pat(G)$ is a certain answer for $\boldsymbol{Q}$ if for all assignment $\mu : fv(G) \to nWords_\Sigma$, the instance $[\![pat(G)]\!]^\mu \star \alpha$ is in $L(\boldsymbol{Q})$.*

Analogously,

Definition 23 *A candidate α of $pat(G)$ is a certain non-answer for $\boldsymbol{Q}$ if for all assignment $\mu : fv(G) \to nWords_\Sigma$, the instance $[\![pat(G)]\!]^\mu \star \alpha$ is not in $L(\boldsymbol{Q})$.*

Note that the candidate maps the query variables to positions of the pattern described by the nested hyperstream. This is due to the fact that shared nonterminals define as many positions as they are referred to in a nested hyperstream. In Figure 5.3, each of the position labeled by X and its children define two positions in the pattern that is described.

We introduce the problems of certain query answering and non-answering for classes of nested hyperstreams $\mathcal{G}$ and of query automata $\mathcal{A} \in \{\text{dSHA}, \text{SHA}, \text{dNWA}, \text{NWA}\}$. For all query automaton, the notation $\mathcal{Q}(A)$ denotes the query whose language equals the language of A. We also write $Seq_{\Sigma,\mathcal{W}'}$ to denote the set $\Sigma \cup \mathcal{W}' \cup \neg\mathcal{W}'$.

Certain query answering: $\text{Cert}^{ans}_{\Sigma,\mathcal{W}'}(\mathcal{G}, \mathcal{A})$.
Input: a nested hyperstream $G \in \mathcal{G}_\Sigma$, a candidate $\alpha \in \mathcal{C}_{\mathcal{W}'}(pat(G))$, and a query automaton $A \in \mathcal{A}_{Seq_{\Sigma,\mathcal{W}'}}$
Output: whether α is a certain answer for query $\mathcal{Q}(A)$ on $pat(G)$

Certain query non-answering: $\text{Cert}^{\neg ans}_{\Sigma,\mathcal{W}'}(\mathcal{G}, \mathcal{A})$.
Input: a nested hyperstream $G \in \mathcal{G}_\Sigma$, a candidate $\alpha \in \mathcal{C}_{\mathcal{W}'}(pat(G))$, and a query automaton $A \in \mathcal{A}_{Seq_{\Sigma,\mathcal{W}'}}$
Output: whether α is a certain non-answer for query $\mathcal{Q}(A)$ on $pat(G)$

Remark that certain query answering is equivalent to the *regular pattern inclusion*, which consists in determining whether a language of nested words (the set

of instances of the pattern described by the hyperstream) is included in a regular language of nested words (the language of the query). The set of instances of the pattern described by the hyperstream is not necessarily regular, because of the sharing possibilities.

On the other hand, certain query non answering is equivalent to the *regular pattern matching* problem, which determines whether two languages of nested words have an empty intersection, given that one of them is regular (the language of the query) and the other is not (the language of the instances of the pattern described by the hyperstream).

In Chapter 6, we formally define regular pattern matching and regular pattern inclusion. We study them in the case of compressed (ranked) tree patterns with context variables, to which the case of nested hyperstreams will be reduced.

5.2.2 From the Non-Boolean Cases to the Boolean Cases

Boolean CQA and CQNA are the special cases $\textsc{Cert}^{ans}_{\Sigma,\emptyset}(\mathcal{G},\mathcal{A})$ respectively $\textsc{Cert}^{ans}_{\Sigma,\emptyset}(\mathcal{G},\mathcal{A})$, i.e. where only the empty candidate can be tested for certainty. We show that the more general problems of certain query answering and certain query non answering can be reduced to their boolean versions in polynomial time, by decompressing partially the nested hyperstream.

5.2.2.1 Partial Decompression

Lemma 11 (Partial Decompression Lemma) *For any nested hyperstream $G \in Hyp_\Sigma$ and any candidate $\alpha \in \mathcal{C}_{\mathcal{W}'}(pat(G))$, we can compute in* PTIME *some $G' \in Hyp_{Seq_{\Sigma,\mathcal{W}'}}$ such that $pat(G) \star \alpha = pat(G')$. In particular, if $pat(G)$ was linear then $pat(G')$ is linear.*

Let $G = (N, \Sigma, R, \mathcal{S})$ be a nested hyperstream and $\alpha \in \mathcal{C}_{\mathcal{W}'}(patG)$ a candidate of $pat(G)$, fixed in this section. We show that we can compute in PTIME a nested hyperstream G_α over $Seq_{\Sigma,\mathcal{W}'}$ such that $pat(G)\star\alpha = pat(G')$. Thus G_α will describe a sequenced $\mathcal{W}'$-structure. We assume that $\mathcal{S} \in dom(R)$ as otherwise the lemma is trivial.

The main ingredients of the proof will be illustrated on the example $G_{ex} = (\{\mathcal{S}_{ex}, X_1, X_2\}, \{a, b\}, R_{ex}, \mathcal{S}_{ex})$ in Figure 5.6 where $R_{ex}(\mathcal{S}_{ex}) = X_1 a X_1 X_1$, $R_{ex}(X_1) = \langle b X_2 \rangle$. The pattern $pat(G_{ex})$ described by G_{ex} is illustrated in Figure 5.6 (to the right). We also use for the example $\mathcal{W}' = \{x\}$ and $\alpha_{ex} = [x/3 \cdot 1]$. Note that the position $3 \cdot 1$ in $pat(G_{ex})$ is shared in G, and that $pat(G_{ex}) \star \alpha_{ex} = \langle b \cdot \neg x \cdot X_2 \rangle a \cdot \neg x \langle b \cdot x \cdot X_2 \rangle \langle b \cdot \neg x \cdot X_2 \rangle$.

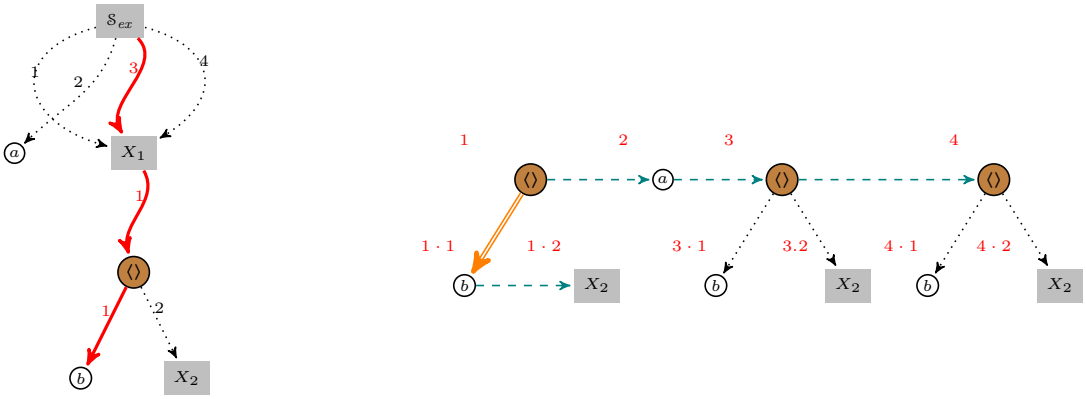


Figure 5.6: Nested hyperstream G and $pat(G)$

First we introduce new definitions that we'll use in the sequel. It is clear that for any non terminal X, there exists at most one position in the nested hyperstream G labeled by X. We write $pos^G(X)$ to denote it. For any natural $j > 0$ and positions $\pi, \pi' \in ch_j^G$ such that π' is labeled by some element $e \in \Sigma \cup N$, we write $chlab_j^G(\pi) = e$. Furthermore, for any position $\pi \in Pos(G)$, the degree of π denoted by $deg^G(\pi)$ is the greatest natural i for which there exists a position $\pi' \in Pos(G)$ such that $(\pi, \pi') \in ch_i^G$. We now define the *addresses* of G that are non-empty words over the alphabet $\mathbb{N}$ of natural numbers, similarly to the standard Dewey notation for trees but applied to the acyclic graph structure of G. For any position $\pi \in Pos(G)$ that is either a node or labeled by a nonterminal $X \in N$, the address of π with respect to G is:

$$Addr(\pi) = \begin{array}{l} \{i \mid 0 < i \leq deg^G(\pi)\} \\ \cup \{i \cdot d \mid chlab_i^G(\pi) \in dom(R) \text{ and } d \in Addr(ch_i^G(\pi))\} \end{array}$$

The set of addresses of $\mathcal{S}$ is denoted by $Addr$ On the example, $d_1 = 1$, $d_2 = 3 \cdot 1 \cdot 1$ and $d_3 = 4 \cdot 1$ are addresses of $\mathcal{S}_{ex}$.

For any address $d \in Addr$ we associate its *path* that is a word whose letters are pairs of the form (X, i) for $X \in dom(R)$ and $0 < i \leq deg^G(pos^G(X))$: $path(k_1 \dots k_n) = (X_1, k_1), \dots, (X_n, k_n)$ where $X_1 = \mathcal{S}$ and for any $2 \leq i \leq n$,

$$X_i = \begin{cases} chlab_{k_{i-1}}^G(\pi) & \text{if } \pi = pos^G(X_{i-1}) \text{ is labeled by a nonterminal} \\ \langle\rangle & \text{if } \pi = pos^G(X_{i-1}) \text{ and } ch_{k_{i-1}}^G(\pi) \text{ is a node} \end{cases}$$

On the example, $path(d_2) = (\mathcal{S}_{ex}, 3)(X_1, 1)(\langle\rangle, 1)$. Let $Paths_\Sigma$ be the set of paths of G that lead to a letter in Σ. We first establish that there is a one-to-one correspondence between $Paths_\Sigma$ and $LPos_\Sigma(pat(G))$, the positions of $pat(G)$ labeled by

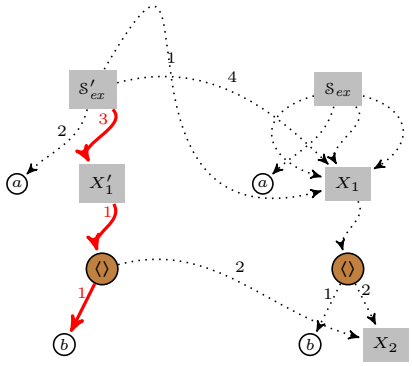


Figure 5.7: $3 \cdot 1 \cdot 1$ is sharing-free in G'

elements of Σ.

Claim 4 *There is a bijection* $pospath : LPos_\Sigma(pat(G)) \to Paths_\Sigma$ *s.t. for any* $m \in LPos_\Sigma(pat(G))$, $pospath(m)$ *can be computed in* PTIME *in the size of* G.

This claim exploits the fact that the positions of patterns described by nested hyperstreams are words over naturals. Given a word over $\mathbb{N}$, we can compute the path associated to the address of that word by going through the structure of the nested hyperstream while counting the number of positions already reached. This is done in polynomial time since once a nonterminal is entirely traversed, we can memorize the size of the pattern that it describes. This allows to not go through the potential exponential size of the whole described pattern. On the example, $path(3 \cdot 1) = (\mathcal{S}_{ex}, 3)(X_1, 1)(\langle\rangle, 1)$ is a path (in red).

The next ingredient for the proof of the partial decompression lemma is to show that given some address, G can be transformed into G' having the same pattern, but in which this address is sharing free. An address $d \cdot U \in Addr$ is called *sharing-free* if there does not exist a different address $d' \cdot U \in Addr$ s.t. $path(d)$ and $path(d')$ have the same symbol in their last position.

Claim 5 *For any address* $d \in Addr$ *we can compute in* PTIME *in the sizes of* G *and* d *a compressed string pattern* G'' *such that* $pat(G'') = pat(G)$ *and* d *is sharing-free in* G''.

The new nested hyperstream is built by making a copy of G and setting a new starting symbol, so that the given address is no longer shared. The elements that

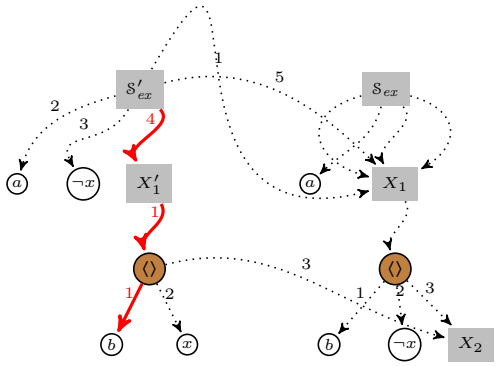


Figure 5.8: Adding query variables

are not accessible from the new starting symbol can be ignored. This is illustrated in Figure 5.7. The new nested hyperstream G'_{ex} has $\mathcal{S}'_{ex}$ as its starting symbol, and the address $3 \cdot 1 \cdot 1$ is no longer shared.

Finally, as last ingredient of the proof of the partial decompression lemma, we show how for a given candidate α, a nested hyperstream over Σ is transformed to a nested hyperstream on $Seq_{\Sigma,\mathcal{W}'}$ in which the positions associated with query variables by α are sharing-free.

Using the bijection $pospath : LPos_\Sigma(pat(G)) \to Paths_\Sigma$ defined in Claim 4:

Claim 6 *If all paths in $pospath(\alpha(\mathcal{W}'))$ are sharing-free in G, we can compute in* PTime *a nested hyperstream G'' over $Seq_{\Sigma,\mathcal{W}'}$ such that $pat(G'') = pat(G) \star \alpha$.*

This is illustrated in Figure 5.8. Note that the addresses are shifted, due to the elements of $\mathcal{W}' \cup \neg\mathcal{W}'$ that are added.

Proof of Lemma 11. We first compute the set of paths $D = pospath(\alpha(\mathcal{W}'))$ in PTime by Claim 4. Note that the cardinality of D is at most $|\mathcal{W}'|$, so it is of constant size. Then we compute a nested hyperstream G' with $pat(G) = pat(G')$ such that all the addresses corresponding to the paths in D are sharing-free in G''. Since there are constantly many such addresses, this can be done in PTime by iterating Claim 5 a constant number of times. Then using Claim 6 we compute the nested hyperstream G_α over $Seq_{\Sigma,\mathcal{W}'}$ such that $pat(G_\alpha) = pat(G') \star \alpha = pat(G) \star \alpha$.

5.2.2.2 Reductions to the Boolean Cases

We finally reduce the general certainty problems to their boolean versions in polynomial time.

Proposition 11 *For all $\mathcal{B}$ in $\{ans, \neg ans\}$, all $\mathcal{A}$ in $\{dSHA, SHA, NWA, dNWA\}$ and all $\mathcal{G}$ in $\{nPatterns, nLinPatterns, Hyp, LinHyp\}$, there is a* PTIME *reduction from* $\text{CERT}^{\mathcal{B}}_{\Sigma,\mathcal{W}'}(\mathcal{G}, \mathcal{A})$ *to* $\text{CERT}^{\mathcal{B}}_{Seq_{\Sigma,\mathcal{W}'},\emptyset}(\mathcal{G}, \mathcal{A})$.

Proof: Let $G \in \mathcal{G}_\Sigma$ be a nested hyperstream and $\alpha \in \mathcal{C}_{\mathcal{W}'}(pat(G))$ a candidate of the pattern described by G. If $\mathcal{G} \in \{Hyp, LinHyp\}$, then by Lemma 11 we can compute in PTIME a nested hyperstream $G' \in \mathcal{G}_{Seq_{\Sigma,\mathcal{W}'}}$ such that $pat(G) \star \alpha = pat(G')$. If $\mathcal{G} \in \{nPatterns, nLinPatterns\}$, then the same property holds trivially, since no decompression is to be done.

In the following, we write $Struct^{\mathcal{W}'}$ to denote the set of all sequenced $\mathcal{W}'$-structures. We recall that by Lemma 1, any variable of $\mathcal{W}'$ has exactly one positive occurrence in a sequenced $\mathcal{W}'$-structure.

We start with $\mathcal{B} = ans$. Let $A \in \mathcal{A}_{Seq_{\Sigma,\mathcal{W}'}}$ be a query automaton, B a dSHA or dNWA over $Seq_{\Sigma,\mathcal{W}'}$ – depending on whether A is a SHA or an NWA– such that $L(B) = (Seq_{\Sigma,\mathcal{W}'})^* \setminus Struct^{\mathcal{W}'}$, and $C \in \mathcal{A}_{Seq_{\Sigma,\mathcal{W}'}}$ such that $L(C) = L(A) \cup L(B)$. In the case where $\mathcal{G}$ is a class of nondeterministic automata, i.e. $\mathcal{G} \in \{\text{SHA}, \text{NWA}\}$, we can chose C to be the union of A and B. On the other hand, when $\mathcal{G}$ is a class of deterministic automata, that is $\mathcal{G} \in \{\text{dSHA}, \text{dNWA}\}$, C can be built the product of A and B. The automaton B can be constructed in time $O(2^{|\mathcal{W}'|})$ which is constant since $\mathcal{W}'$ is a parameter of the certainty problem rather than being part of its input. Then α is a certain answer for $\mathcal{Q}(A)$ on G iff $Inst(G \star \alpha) \cap Struct^{\mathcal{W}'} \subseteq \mathcal{L}(A)$ iff $Inst(G') \cap Struct^{\mathcal{W}'} \subseteq \mathcal{L}(A)$ iff $Inst(G') \subseteq \mathcal{L}(A) \cup L(B)$ iff $Inst(G') \subseteq \mathcal{L}(C)$ iff the empty candidate is a certain answer for $\mathcal{Q}(C)$ on G'. Based on this fact, the PTIME reduction from $\text{CERT}^{ans}_{\Sigma,\mathcal{W}'}(\mathcal{G}, \mathcal{A})$ to $\text{CERT}^{ans}_{Seq_{\Sigma,\mathcal{W}'},\emptyset}(\mathcal{G}, \mathcal{A})$ holds. Also, α is a certain non-answer of $\mathcal{Q}(A)$ on G iff $Inst(G \star \alpha) \cap \mathcal{L}(A) = \emptyset$ iff $Inst(G') \cap \mathcal{L}(A) = \emptyset$ iff the empty Σ-assignment is a certain non-answer of $\mathcal{Q}(A)$. Thus $\text{CERT}^{\neg ans}_{\Sigma,\mathcal{W}'}(\mathcal{G}, \mathcal{A}) \leq_p$ is reducible in polynomial time to $\text{CERT}^{\neg ans}_{Seq_{\Sigma,\mathcal{W}'},\emptyset}(\mathcal{G}, \mathcal{A})$. □

This shows that the non-boolean versions of the certainty problems are not harder than their boolean counterparts. The next chapters will thus study the boolean cases.

Chapter 6

Complexity of Certain Query Answering

Contents

6.1 Introduction

The following generic problems for patterns were widely studied in the literature:

Pattern matching: Is a given algebraic value an instance of a given pattern?

Pattern unification: Do two given patterns have some common instance?

Regular pattern matching: Does some instance of the given pattern belong to the given regular language?

B18: [Boneva *et al.* 2018]
C07: [Comon *et al.* 2007]

	DFAs	NFAs
MATCH	PSPACE-c B18	PSPACE-c B18
INCL	PSPACE-c B18	PSPACE-c B18

Figure 6.1: (Compressed) string patterns.

	DFAs	NFAs
MATCH	PTIME B18	PTIME B18
INCL	PTIME B18	PSPACE-c B18

Figure 6.2: Linear restriction.

	DTAs	NTAs
MATCH	NP-c C07,Th 3,Prop 18	EXP-c C07, Th 3, Prop 18
INCL	coNP-c Th 3, Prop 18	EXP-c Lem 17, Th 3, Prop 18

Figure 6.3: (Compressed) tree patterns without context variables.

	DTAs	NTAs
MATCH	PTIME Prop 21	PTIME Prop 21
INCL	PTIME Lem 16, Prop 21	EXP-c Lem 17, Th 6

Figure 6.4: Linear restriction.

	DTAs	NTAs
MATCH	PSPACE-c Th 6, Prop 19	EXP-c Th 6, Prop 19
INCL	PSPACE-c Th 6, Prop 19	EXP-c Th 6, Prop 19

Figure 6.5: (Compressed) tree patterns with (constrained) context variables.

	DTAs	NTAs
MATCH	PTIME Prop 21	PTIME Prop 21
INCL	PTIME Lem 16, Prop 21	EXP-c Lem 17 , Th 6

Figure 6.6: Linear restriction.

Regular pattern inclusion: Do all instances of the given pattern belong to the given regular language?

As inputs, these problems receive descriptors of patterns, values, and regular languages. The problem of string pattern matching is well known to be NP-complete for NFAs [Angluin 1980] but in PTIME for DFAs, with and without compression [Gascón *et al.* 2008]. The more general problem of string unification is of quite different nature. It is known to be in PSPACE [Plandowski 2004].

We have shown in a previous work [Boneva *et al.* 2018] that regular inclusion and matching for string patterns are PSPACE-complete, both for DFAs and NFAs, with and without compression. See Figure 6.1 for an overview. When restricted to linear string patterns, the complexity goes down to polynomial time in 3 of the 4 cases, as summarized in Figure 6.2. The problem which remains PSPACE-complete is regular inclusion on linear string patterns for NFAs.

The complexity landscape of regular matching and inclusion for ranked tree patterns without context variables looks quite different to the case of string patterns, see Figs. 6.3 and 6.4. Here, regular languages are defined by tree automata, which may either be nondeterministic (NTAs) or (bottom-up) deterministic (DTAs). Regular matching of ranked tree patterns without context variables for NTAs was named the *ground instance intersection problem* in [Comon *et al.* 2007], where it was shown to be NP-complete for DTAs and EXP-complete for NTAs. Furthermore it was shown

that the restriction to linear patterns is in PTIME, both for DTAs and NTAs. Regular inclusion for ranked tree patterns has not been studied so far to the best of our knowledge. We show that it is coNP-complete for DTAs and EXP-complete for NTAs even when restricted to linear tree patterns. Only for DTAs, the problem of regular inclusion for linear ranked tree patterns is in PTIME. Compression can be added to ranked tree patterns (see Section 5.1.2.3) by using singleton tree grammars [Schmidt-Schauß 2018]. But as we will see, this doesn't affect the above results.

The prime reason for the asymmetry of the complexity landscapes in the case of strings and trees is that string patterns cannot be encoded as tree patterns with a monadic signature without adding context variables. For instance, the string pattern $aZZbY$ corresponds to the tree pattern $a(Z@(Z@b(Y)))$ with context variable Z, tree variable Y and application symbol @. The interest of adding context variables to ranked tree patterns was already noticed when generalizing string pattern matching to context pattern matching [Gascón *et al.* 2008], which are both NP-complete, with or without compression. The same was noticed when generalizing string unification to context unification, that are both in PSPACE [Jez 2014]. Since we are interested in a proper generalization of regular matching and inclusion from string to ranked tree patterns, we propose to study these problems for tree patterns with context variables.

In this chapter, we relate regular matching of ranked tree patterns to inhabitation problems of tree automata in a systematic manner. The naive semi-decision procedure for regular matching guesses some context for all the context variables in the tree pattern and then checks whether the instance of the pattern obtained thereby matches the regular language, i.e. whether it is recognized by the tree automaton defining this language. In order to avoid infinite guesses, our decision algorithm will guess for all context variables a function of type $Q \to 2^Q$ where Q is the set of states of the tree automaton, and then test whether this function is inhabited by some context with respect to the automaton. In order to make this approach work, we need to study the problem of context inhabitation on its own right.

Context inhabitation is a special case of second-order linear λ-definability, except that the input function is represented in a succinct manner. More generally, λ-*definability* is known to be decidable up to the order of three [Zaionc 2005], while it is undecidable in general [Loader 2001, Joly 2003]. Context inhabitation for tree automata can also be understood as a generalization of transition inhabitation for word automata, which is sometimes called the membership problem of the transition monoid [Kozen 1977]. We show that context inhabitation for NTAs is EXP-complete. The lower bound is obtained by a reduction from the nonemptiness problem of

intersections of a finite number of NTAs [Seidl 1990], and the upper bound by an algorithm using determinization. We then show that context inhabitation for DTAs is PSPACE-complete. We obtain the PSPACE upper bound by a nontrivial reduction to the nonemptiness problem of intersections of a finite number of DFAs (for words) [Kozen 1977]. The fact that automata on words are enough for this purpose rather than automata for ranked trees explains the otherwise surprising PSPACE upper bound.

We also study the complexity of regular matching and inclusion for compressed ranked tree patterns with context variables. All our results are based in a systematic manner on the close relationship between regular matching of tree patterns with context variables and context inhabitation for tree automata. They are summarized in Figs. 6.5 and 6.6. The only change compared to compressed string patterns is for NTAs, where the complexity increases from PSPACE-complete to EXP-complete. The main reason for this change is that the context inhabitation for NTAs is EXP-complete. In contrast, for DTAs context inhabitation remains PSPACE-complete, so that there is no difference to the case of DFAs.

Next we extend regular matching and inclusion with regular constraints on the possible instantiations of the variables of the pattern.

Regular pattern matching with regular constraints: Does some instantiation of the variables satisfying the given regular constraints produce an instance of the given pattern that belongs to the given regular language?

Regular pattern inclusion with regular constraints: Do all instantiations of the variables satisfying the given regular constraints produce some instance of the given pattern that belong to the given regular language?

We show that the extended problems with regular constraints can be compiled to the original problems without constraints in polynomial time. Our reduction preserves the determinism of the automata and the linearity of the patterns. Therefore all our complexity results on regular matching and inclusion listed above remain valid when adding regular constraints.

Finally we show an application of these results to regular matching and inclusion for compressed patterns on unranked trees with tree and hedge variables (but without context variables). The idea is that unranked tree patterns can be encoded to ranked tree patterns, while mapping hedge variables to context variables. We contribute a reduction of unranked regular matching and inclusion to the ranked case but with regular constraints. In order to deal with the unranked symbols, we cannot bound the maximal arity of the ranked signature. Therefore we have to con-

sider the uniform variant of all problems, where the signature is part of the input rather than being fixed as a parameter.

Most of this chapter is dedicated to (compressed) patterns of ranked trees. Thus we will abuse the term tree to refer to a ranked tree, when it's clear from the context.

6.2 Σ-Algebras

We first present the interpretation of the values of types tree and context in arbitrary Σ-algebras (including tree automata, as we will see later on). Trees will be interpreted as elements of the domain of the Σ-algebra, and contexts as linear functions on this domain.

Definition 24 *A Σ-algebra $\Delta = (\Sigma, D, .^\Delta)$ consists of a ranked signature Σ, a set D called the domain, and a mapping $.^\Delta$ that interprets symbols $f \in \Sigma^{(n)}$ as functions $f^\Delta : D^n \to D$. The domain of Δ is $dom^\Delta = D$.*

We next define the interpretation of values in a Σ-algebra. The *interpretation of a tree* $t = f(t_1, \ldots, t_n) \in \mathcal{T}_\Sigma$ is the domain element $[\![t]\!]^\Delta = f^\Delta([\![t_1]\!]^\Delta, \ldots, [\![t_n]\!]^\Delta)$. This interpretation can be extended to trees over the signature $\Sigma \cup D$ by interpreting any symbol $d \in D$ by itself, i.e., $d^\Delta = d$. The *interpretation of a context* $C = \lambda x.t \in \mathcal{C}_\Sigma$ is the function $[\![C]\!]^\Delta \colon D \to D$ with $[\![C]\!]^\Delta(d) = [\![t[x/d]]\!]^\Delta$ for all $d \in D$. The elements of D and functions of type $D \to D$ that can be obtained by Δ-interpretation of some tree or context are called Δ-*inhabited*:

$$[\![\mathit{Val}_\Sigma]\!]^\Delta = [\![\mathcal{T}_\Sigma]\!]^\Delta \cup [\![\mathcal{C}_\Sigma]\!]^\Delta$$

The set $\mathcal{T}_\Sigma$ of trees can be identified with the free Σ-algebra $(\Sigma, \mathcal{T}_\Sigma, .^{\mathcal{T}_\Sigma})$ whose interpretation function satisfies $f^{\mathcal{T}_\Sigma}(t_1, \ldots, t_n) = f(t_1, \ldots, t_n)$ for all symbols $f \in \Sigma^{(n)}$ and trees $t_1, \ldots, t_n \in \mathcal{T}_\Sigma$. We note that $[\![\mathcal{T}_\Sigma]\!]^{\mathcal{T}_\Sigma} = \mathcal{T}_\Sigma$ while $[\![\mathcal{C}_\Sigma]\!]^{\mathcal{T}_\Sigma}$ is a proper subset of functions of type $\mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$. In other words, the interpretation over the Σ-algebra $\mathcal{T}_\Sigma$ converts any context $C \in \mathcal{C}_\Sigma$ into the function on trees $[\![C]\!]^{\mathcal{T}_\Sigma} \colon \mathcal{T}_\Sigma \to \mathcal{T}_\Sigma$ that it defines, i.e., if $C = \lambda x.t$ for some tree t in which x occurs once, then $[\![C]\!]^{\mathcal{T}_\Sigma}(t') = t[x/t']$ for all $t' \in \mathcal{T}_\Sigma$.

6.3 Inhabitation for Tree Automata

One of the insights of this chapter will be that inhabitation is closely related to regular matching and inclusion for tree patterns, depending on the class of tree

automata and the type of variables. Therefore, here we study inhabitation problems of tree automata on their own right.

6.3.1 Tree Automata

We start by recalling the standard notion of tree automata for ranked trees, their notion of bottom-up determinism, and their relationship to Σ-algebras also in the nondeterministic case.

Definition 25 *A (nondeterministic) tree automaton (*NTA*) over a ranked signature Σ is a tuple $A = (Q, \Sigma, F, \Delta)$ where Q is a finite set of states, $F \subseteq Q$ is the set of final states, and $\Delta \subseteq \cup_{n\geq 0}\Sigma^{(n)} \times Q^{n+1}$ is the transition relation.*

A rule $(f, q_1, \ldots, q_n, q) \in \Delta$ is written as $f(q_1, \ldots, q_n) \to q$. We will identify any transition relation Δ of some NTA as a Σ-algebra $(\Sigma, 2^Q, .^\Delta)$, that interprets function symbols $f \in \Sigma^{(n)}$ as the n-ary functions f^Δ that satisfy for any subsets of states $Q_1 \ldots, Q_n \subseteq Q$:

$$f^\Delta(Q_1, \ldots, Q_n) = \{q \mid \exists q_1 \in Q_1 \ldots \exists q_n \in Q_n.\ f(q_1, \ldots, q_n) \to q \text{ in } \Delta\}.$$

It should always be clear from the context whether we consider Δ as a Σ-algebra or as a transition relation.

The *regular language* $L(A)$ recognized by A is defined as the set of all trees in $\mathcal{T}_\Sigma$ whose evaluation in the Σ-algebra Δ yields some final state in F:

$$L(A) = \{t \in \mathcal{T}_\Sigma \mid [\![t]\!]^\Delta \cap F \neq \emptyset\}.$$

The more general concept of inhabitation from Σ-algebras can now be applied to tree automata, yielding the following definition:

Definition 26 (Inhabitation) *Let $A = (Q, \Sigma, F, \Delta)$ be a tree automaton.*

- *A subset $Q' \subseteq Q$ is called Δ-inhabited by a tree $t \in \mathcal{T}_\Sigma$ if $Q' = [\![t]\!]^\Delta$.*
- *A function $S\colon 2^Q \to 2^Q$ is called Δ-inhabited by a context $C \in \mathcal{C}_\Sigma$ if $S = [\![C]\!]^\Delta$.*

An NTA is called *(bottom-up) deterministic* or equivalently a DTA if no two distinct rules of Δ have the same left-hand side, i.e., if Δ is a partial function from $\bigcup_{n\geq 0} \Sigma^{(n)} \times Q^n$ to Q. The *determinization* of an NTA A is the tree automaton $\det(A) = (2^Q,\ \Sigma,\ \det(F), \det(\Delta))$ where $\det(F) = \{Q' \subseteq Q \mid Q' \cap F \neq \emptyset\}$ and $\det(\Delta) = \{f(Q_1, \ldots, Q_n) \to f^\Delta(Q_1, \ldots, Q_n) \mid f \in \Sigma^{(n)}, Q_1, \ldots, Q_n \subseteq Q\}$. It is

well known that $\det(A)$ is a DTA with $L(A) = L(\det(A))$. Furthermore, for any tree $t \in \mathcal{T}_\Sigma$ it holds that $[\![t]\!]^{\det(\Delta)} = \{[\![t]\!]^\Delta\}$.

An NTA is called *complete* if for all $f \in \Sigma^{(n)}$ and $q_1, \ldots, q_n \in Q$, there exists a state q so that the rule $f(q_1, \ldots, q_n) \to q$ is in Δ.

Let NTA_Σ be the set of all NTAs with signature Σ, and DTA_Σ the set of all DTAs with signature Σ. Clearly, $\text{DTA}_\Sigma \subseteq \text{NTA}_\Sigma$. A class of automata is a function that maps any signature Σ to a subset of NTA_Σ. In particular, NTA and DTA are classes of automata mapping signature Σ to the sets of automata NTA_Σ and DTA_Σ.

In the next subsections, we introduce and study the decision problem of context inhabitation, and its relationship to the problem of intersection nonemptiness. We distinguish the cases of NTAs and DTAs. In both cases, we consider the non-uniform version where the signature Σ is fixed as a parameter of the problem, and the uniform version where the signature is given with the input.

An overview of the results on context inhabitation is given in Figure 6.8. Context inhabitation for nondeterministic tree automata $\text{INHAB}_\Sigma^{context}(\text{NTA})$ is EXP-complete, while the deterministic restriction, $\text{INHAB}_\Sigma^{context}(\text{DTA})$ is PSPACE complete. This might be surprising given that intersection nonemptiness is EXP-complete for tree automata, while it is PSPACE-complete for finite automata on words, in both cases independently of determinism (see Figure 6.7).

Indeed, we will establish a close correspondence for tree automata between the problem of context inhabitation $\text{INHAB}_\Sigma^{context}(\text{NTA})$ and the problem of intersection nonemptiness $\text{INTER}_\Sigma(\text{DTA})$. The surprising result will come from another close relationship between context inhabitation for deterministic tree automata $\text{INHAB}_\Sigma^{context}(\text{DTA})$ and the intersection nonemptiness problem of deterministic finite automata for words $\text{INTER}_\Sigma(\text{DFA})$.

6.3.2 Intersection NonEmptiness

For any class of automata $\mathcal{A}$ and any signature Σ, the non-uniform version of intersection nonemptiness for a finite number of automata is the following problem.

$\text{INTER}_\Sigma(\mathcal{A})$.
Input: a finite number of automata $A_1, \ldots, A_n \in \mathcal{A}_\Sigma$ where $n \geq 0$.
Output: whether $\cap_{i=1}^n L(A_i) \neq \emptyset$.

The uniform variant of this problem where the signature Σ is passed as an input is called $\text{INTER}(\mathcal{A})$. Analogous problems can be defined for classes of finite automata on words, i.e. nondeterministic finite automata (NFAs) and deterministic finite automata (DFAs).

	words		trees	
deterministic	DFAs:	PSPACE-c [Kozen 1977]	DTAs:	EXP-c [Seidl 1990]
nondeterministic	NFAs:	PSPACE-c [Kozen 1977]	NTAs:	EXP-c [Seidl 1990]

Figure 6.7: Emptiness of intersection of a finite number of automata.

	DTAs	NTAs
Tree	PTIME Th 3	EXP-c Th 3
Context	PSPACE-c Th 5	EXP-c Th 4

Figure 6.8: Inhabitation for Tree Automata.

In Figure 6.7 we recall the complexities of the problems $\text{INTER}_\Sigma(\mathcal{A})$ in the cases of deterministic and nondeterministic automata on trees and words, i.e. for $\mathcal{A} \in \{\text{NTA}, \text{DTA}, \text{NFA}, \text{DFA}\}$. The results hold both for the uniform and the non-uniform variants. In the case of trees, the hardness result requires our assumption that the signature Σ contains at least one constant and one symbol of arity greater than or equal to 2.

6.3.3 Tree Inhabitation

Before moving to context inhabitation, we reconsider known results on the easier problem of tree inhabitation, that will be instructive for what follows.

For any class of automata $\mathcal{A}$ and any signature Σ, tree inhabitation is the following problem:

$\text{INHAB}^{tree}_\Sigma(\mathcal{A})$.
Input: a tree automaton $A = (Q, \Sigma, F, \Delta) \in \mathcal{A}_\Sigma$, $Q' \subseteq Q$.
Output: whether Q' is Δ-inhabited by some tree in $\mathcal{T}_\Sigma$.

The uniform variant of this problem where the signature Σ is passed as an input is called $\text{INHAB}^{tree}(\mathcal{A})$. The complexity of tree inhabitation is folklore, in both cases, uniform or not. An overview of the results is given in Figure 6.8. An algorithm for solving the problem for NTAs can be based on determinization. This algorithm will be instructive for context inhabitation as well, so we include it in the proof.

Proposition 12 *Tree inhabitation* $\text{INHAB}^{tree}(\text{NTA})$ *is in* EXP. *The restriction to deterministic tree automata* $\text{INHAB}^{tree}(\text{DTA})$ *is in* PTIME.

Proof: Let Σ be a ranked signature. If A is a DTA then $Q' \subseteq Q$ is Δ-inhabited by some tree, if either $Q' = \emptyset$ and A is not complete, or Q' is a singleton and the unique state of Q' is accessible wrt. Δ. Hence $\text{INHAB}^{tree}(\text{DTA})$ is in polynomial time. For NTA*s* the EXP upper bound can be obtained by determinization. If $A = (Q, \Sigma, F, \Delta)$ is an NTA then by definition $Q' \subseteq Q$ is Δ-inhabited by some tree $t \in \mathcal{T}_\Sigma$ if $[\![t]\!]^\Delta = Q'$. This is equivalent to that $[\![t]\!]^{\det(\Delta)} = \{Q'\}$. Thus Q' is Δ-inhabited iff $\{Q'\}$ is $\det(\Delta)$-inhabited in $\det(A)$. This can be tested in polynomial time from $\det(A)$, which in turn can be computed in exponential time from A. Thus $\text{INHAB}^{tree}(\text{NTA})$ is in EXP. □

The worst case exponential blow up coming with determinization cannot be avoided for solving tree inhabitation of NTAs, as we show next.

Theorem 3 (Folklore) *Tree inhabitation* $\text{INHAB}^{tree}_\Sigma(\text{NTA})$ *is* EXP*-complete, while its restriction to deterministic tree automata* $\text{INHAB}^{tree}_\Sigma(\text{DTA})$ *is in* PTIME.

Proof: The upper bounds were shown in Proposition 12. The lower EXP lower bound for NTA*s* follows from a reduction from intersection nonemptiness of a finite number of deterministic tree automata $\text{INTER}_\Sigma(\text{DTA})$, which is well known to be EXPTIME-complete [Seidl 1990]. The relationship to this nonemptiness problem is instructive for context inhabitation later on, so we also present this reduction.

Let $A_1, \ldots, A_n$ be a sequence of DTAs with signature Σ. We want to know whether $\cap_{i=1}^n L(A_i) \neq \emptyset$. Suppose that $A_i = (Q_i, \Sigma, F_i, \Delta_i)$. Without loss of generality, we can assume that each of them has a single final state $F^i = \{q_f^i\}$ [1]. Let A be the disjoint union of all A_i, that is $A = (Q, \Sigma, F, \Delta)$ where $Q = \uplus_{i=1}^n Q_i$, $\Delta = \uplus_{i=1}^n \Delta_i$ and $F = \{q_f^1, \ldots, q_f^n\}$. Since all A_i are deterministic, it holds that (*) F is Δ-inhabited if and only if $\bigcap_{i=1}^n L(A_i) \neq \emptyset$. In order to see (*), let $t \in \mathcal{T}_\Sigma$ be a tree. It then holds that:

$$
\begin{array}{lll}
t \in \bigcap_{i=1}^n L(A_i) & \text{iff} & \text{for all } i \in \{1, \ldots, n\} : \; q_f^i \in [\![t]\!]^{\Delta_i} \\
& \text{iff} & \text{for all } i \in \{1, \ldots, n\} : \; \{q_f^i\} = [\![t]\!]^{\Delta_i} \quad (A_i \text{ is deterministic}) \\
& \text{iff} & \{q_f^1, \ldots, q_f^n\} = [\![t]\!]^\Delta \\
& \text{iff} & \{q_f^1, \ldots, q_f^n\} \text{ is } \Delta\text{-inhabited by } t.
\end{array}
$$

The property (*) shows that $\text{INTER}_\Sigma(\text{DTA})$ can be reduced to $\text{INHAB}^{tree}_\Sigma(\text{NTA})$ in polynomial time, so $\text{INHAB}^{tree}_\Sigma(\text{NTA})$ is EXP hard. □

[1] Otherwise, we fix a nonconstant $g \in \Sigma \setminus \Sigma^{(0)}$ and a constant $a \in \Sigma^{(0)}$. We then compute automata A_i' with $L(A_i') = g(L(A_i), a, \ldots, a)$. These can be constructed in PTIME from A_i such that they have a unique final state. Furthermore, $\cap_{i=1}^n L(A_i) \neq \emptyset$ if and only if $\cap_{i=1}^n L(A_i') \neq \emptyset$.

6.3.4 Context Inhabitation

Since the bound variable of a context occurs exactly once, contexts are interpreted as union homomorphisms in the transition algebras of a tree automata. These will play a key role for defining the problem of context inhabitation and for studying its complexity.

Definition 27 *A union homomorphism on 2^Q is a function $S\colon 2^Q \to 2^Q$ such that $S(\emptyset) = \emptyset$ and $S(Q' \cup Q'') = S(Q') \cup S(Q'')$ for all $Q', Q'' \subseteq Q$.*

Lemma 12 *For any context $C \in \mathcal{C}_\Sigma$ and* NTA *$A = (Q, \Sigma, F, \Delta)$ the Δ-inhabited value $[\![C]\!]^\Delta$ is a union homomorphism on 2^Q.*

Proof: Any context $C \in \mathcal{C}_\Sigma$ has the form $\lambda x.t$ such that x occurs exactly once in t. The proof is by induction on the structure of t.

- Case $t = x$. We then have that $[\![C]\!]^\Delta(Q') = [\![\lambda x.x]\!]^\Delta(Q') = [\![Q']\!]^\Delta = Q'$ for all $Q' \subseteq Q$. In particular $[\![C]\!]^\Delta(\emptyset) = \emptyset$. Furthermore for any two subsets $Q', Q'' \subseteq Q$, it holds that $[\![C]\!]^\Delta(Q' \cup Q'') = Q' \cup Q'' = [\![C]\!]^\Delta(Q') \cup [\![C]\!]^\Delta(Q'')$. Thus $[\![C]\!]^\Delta$ is a union homomorphism.

- Case $t = f(t_1, \ldots, t_n)$ and x occurs exactly once in t, say in t_k but not elsewhere.

  Let $S_k = [\![\lambda x.t_k]\!]^\Delta$ and $Q_i = [\![t_i]\!]^\Delta$ for all $i \neq k$. Clearly S_k is Δ-inhabited. We then have by the induction hypothesis that S_k is a union homomorphism. Furthermore, we have that for all $Q' \subseteq Q$, $[\![C]\!]^\Delta(Q') = [\![\lambda x.f(t_1, \ldots, t_n)]\!]^\Delta(Q') = [\![f(t_1, \ldots, \lambda x.t_k, \ldots, t_n)]\!]^\Delta(Q')$. By definition of algebra evaluation, we have $[\![f(t_1, \ldots, \lambda x.t_k, \ldots, t_n)]\!]^\Delta(Q') = \{q \mid \exists q_1 \in Q_1, \ldots, q_k \in S_k(Q'), \ldots, \exists q_n \in Q_n.\ f(q_1, \ldots, q_n) \to q \text{ in } \Delta\}$. This implies that for any two subsets $Q', Q'' \subseteq Q$, $[\![C]\!]^\Delta(Q' \cup Q'') = \{q \mid \exists q_1 \in Q_1, \ldots, q_k \in S_k(Q') \cup S_k(Q''), \ldots \exists q_n \in Q_n.\ f(q_1, \ldots, q_n) \to q \text{ in } \Delta\} = [\![C]\!]^\Delta(Q') \cup [\![C]\!]^\Delta(Q'')$. In particular, $[\![C]\!]^\Delta(\emptyset) = \emptyset$. So $[\![C]\!]^\Delta$ is a union homomorphism.

□

The following example shows that Lemma 12 would fail if it were generalized from contexts to nonlinear second-order λ-terms.

Example 15 *Consider $N = \lambda x.f(x, x)$ over the signature $\Sigma = \{a, f\}$ where a is a constant and f a symbol of arity 2, and the* NTA *$A = (Q, \Sigma, F, \Delta)$ with $Q = \{q_1, q_2, q_{ok}\}$, $F = \{q_{ok}\}$ and $\Delta = \{a \to q_1, a \to q_2, f(q_1, q_2) \to q_{ok}\}$.*

We have $[\![N]\!]^{\Delta}(\{q_1\}) = [\![N]\!]^{\Delta}(\{q_2\}) = \emptyset$*, while* $[\![N]\!]^{\Delta}(\{q_1, q_2\}) = \{q_{ok}\}$*. Hence,* $[\![N]\!]^{\Delta}(\{q_1, q_2\}) \neq [\![N]\!]^{\Delta}(\{q_1\}) \cup [\![N]\!]^{\Delta}(\{q_2\})$*, so* $[\![N]\!]^{\Delta}$ *is not a union homomorphism.*

Any function $s\colon Q \to 2^Q$ defines the union homomorphism $\hat{s}\colon 2^Q \to 2^Q$ such that $\hat{s}(Q') = \bigcup_{q \in Q'} s(q)$ for all $Q' \subseteq Q$. Conversely, any union homomorphism is determined by the images of all singletons.

Lemma 13 (Succinct representations of union homomorphisms) *If* $S\colon 2^Q \to 2^Q$ *is a union homomorphism then* $S = \hat{s}$ *for the function* $s\colon Q \to 2^Q$ *such that* $s(q) = S(\{q\})$ *for all* $q \in Q$*.*

Proof: This is straightforward from the definitions. □

As a consequence, the number of union homomorphisms is equal to the number of functions of type $Q \to 2^Q$ which is exponential. In contrast the number of functions of type $2^Q \to 2^Q$ is doubly exponential. This is the reason why second-order inhabitation is a more difficult problem than context inhabitation that we formalize next.

Context inhabitation receives as input a function $s\colon Q \to 2^Q$ that represents the union homomorphism $\hat{s}\colon 2^Q \to 2^Q$. Note that the representation is exponentially smaller than the union homomorphism it represents. Using this succinct representation of a union homomorphism as an input rather than the union homomorphism itself will permit to relate the complexity of regular matching to context inhabitation.

For any ranked signature Σ and class of automata $\mathcal{A}$, we define the following decision problem.

INHAB$^{context}_{\Sigma}(\mathcal{A})$.
Input: an automaton $A = (Q, \Sigma, F, \Delta) \in \mathcal{A}_\Sigma$ and a function $s\colon Q \to 2^Q$.
Output: whether the union homomorphism $\hat{s}$ is Δ-inhabited by some context in $\mathcal{C}_\Sigma$.

The uniform variant of the problem where the signature Σ is given with the input is denoted by INHAB$^{context}(\mathcal{A})$. Based on the properties of union homomorphisms, we next show that $\hat{s}$ is Δ-inhabited if and only if its restriction to singletons is.

Proposition 13 *Let* $A = (Q, \Sigma, F, \Delta)$ *be an* NTA *and* $s\colon Q \to 2^Q$*. Then* $\hat{s}$ *is* Δ*-inhabited iff there exists* $C \in \mathcal{C}_\Sigma$ *such that for all* $q \in Q$*,* $s(q) = [\![C]\!]^{\Delta}(\{q\})$*.*

Proof: The forward implication is straightforward. For the backward direction, let $C \in \mathcal{C}_\Sigma$ be a context with $s(q) = [\![C]\!]^{\Delta}(\{q\})$ for all $q \in Q$. Since $\hat{s}$ is a union homo-

morphism, we have for all $Q' \subseteq Q$ that $\hat{s}(Q') = \bigcup_{q \in Q'} s(q) = \bigcup_{q \in Q'} [\![C]\!]^{\Delta}(\{q\}) = [\![C]\!]^{\Delta}(Q')$ since $[\![C]\!]^{\Delta}$ is a union-homomorphism by Lemma 12. Thus $\hat{s}$ is Δ-inhabited. □

Context inhabitation is a special case of second-order linear λ-definability where the second-order input function is a union homomorphism, except that the union homomorphism is represented in a succinct manner. Note that λ-*definability* is known to be decidable up to the order of three [Zaionc 2005], while it is undecidable in general [Loader 2001, Joly 2003]. We now determine the complexity of context inhabitation for NTAs and then for DTAs.

Proposition 14 INHAB^{context}(NTA) *is in* EXP.

Proof: As in the case of tree inhabitation, the problem can be solved based on determinization, but in a more tricky manner. Let Σ be a ranked signature, $A = (Q, \Sigma, F, \Delta)$ an NTA where $Q = \{q_1, \ldots, q_n\}$ and $s\colon Q \to 2^Q$. We fix $x \in \mathcal{V}_{tree}$. For each $i \in \{1, \ldots, n\}$, let $\Delta_i = \Delta \cup \{x \to q_i\}$ and $A_i = (Q, \Sigma \uplus \{x\}, F, \Delta_i)$. Let $\tilde{A}$ be the product DTA $\tilde{A} = \det(A_1) \times \ldots \times \det(A_n)$ with transition relation $\tilde{\Delta}$, recognizing the intersection of the languages of the DTAs $\det(A_i)$. Note that the number of states of $\tilde{A}$ is at most $(2^n)^n = 2^{n^2}$, which is exponential.

Claim 7 *Let $p \in \mathcal{T}_{\Sigma \uplus \{x\}}$ be a tree having exactly one occurrence of x. Then $[\![p]\!]^{\tilde{\Delta}} = \{(s(q_1), \ldots, s(q_n))\}$ if and only if $[\![p]\!]^{\Delta_i} = s(q_i)$ for all $1 \leq i \leq n$.*

Recall that for any context $C = \lambda x.p$, the set $[\![p]\!]^{\Delta_i}$ contains all the states to which C can be evaluated when starting at the hole marker x with state q_i. Let B be the DTA with signature $\mathcal{T}_{\Sigma \uplus \{x\}}$ recognizing the set of all trees having exactly one occurrence of x. We assume w.l.o.g. that B has a single final state q_f. Now consider the product DTA $\tilde{A} \times B$ recognizing the language $L(\tilde{A}) \cap L(B)$ of all the elements of $L(\tilde{A})$ having exactly one occurrence of x. Then it follows from Claim 7 that the tuple $(s(q_1), \ldots, s(q_n), q_f)$ is an accessible state of $\tilde{A} \times B$ if and only if there exists a context $\lambda x.p \in \mathcal{C}_\Sigma$ such that $[\![\lambda x.p]\!]^{\Delta}(\{q_i\}) = s(q_i)$. By Proposition 13 the latter is equivalent to that $\hat{s}$ is Δ-inhabited. Testing whether $(s(q_1), \ldots, s(q_n), q_f)$ is accessible in $\tilde{A} \times B$ is in polynomial time in the size of $\tilde{A} \times B$, which is in EXP. □

Theorem 4 INHAB$^{context}_{\Sigma}$(NTA) *is* EXP*-complete.*

Proof: The EXP upper bound was shown in Proposition 14 even for the uniform variant of the problem. For EXP-hardness of INHAB$^{context}_{\Sigma}$(NTA) for any Σ – with at least one constant and one symbol of arity at least 2 – we use a reduction from

$\text{INTER}_\Sigma(\text{DTA})$. Let $A_1, \ldots, A_n$ be DTAs where $A_i = (Q_i, \Sigma, F_i, \Delta_i)$ for $1 \leq i \leq n$, and such that their sets of states are pairwise-disjoint. We consider a fresh constant x not in Σ, a fresh symbol \$ of arity 2, and write $\Sigma' = \Sigma \cup \{x, \$\}$. Let $q_1, q_1^f, \ldots, q_n, q_n^f$ be fresh states, i.e. not in $Q_1 \cup \cdots \cup Q_n$. We build the NTA $B = (Q, \Sigma', F, \Delta)$ obtained by setting $Q = Q_1 \cup \ldots \cup Q_n \cup \{q_1, q_1^f, \ldots, q_n, q_n^f\}$, $F = \{q_1^f, \ldots, q_n^f\}$ and $\Delta = \Delta_1 \cup \ldots \cup \Delta_n \cup \{x \to q_i \mid 1 \leq i \leq n\} \cup \{\$(q, q_i) \to q_i^f \mid q \in F_i\}$. Now let $s \colon Q \to 2^Q$ be the function so that for all $q \in Q$,

$$s(q) = \begin{cases} \{q_i^f\} & \text{if } q = q_i \text{ for } 1 \leq i \leq n \\ \emptyset & \text{otherwise} \end{cases}$$

Then $\bigcap_{i=1}^{n} L(A_i) \neq \emptyset$ if and only if $\hat{s}$ is Δ-inhabited. Indeed, there exists a tree $t \in \bigcap_{i=1}^{n} L(A_i)$ iff $[\![t]\!]^{\Delta_i} \cap F_i \neq \emptyset$ for all $1 \leq i \leq n$, iff $[\![\$(t, x)[x/\{q\}]]\!]^{\Delta} = s(q)$ for all $q \in Q$ iff $[\![\lambda x.\$(t, x)]\!]^{\Delta}(\{q\}) = s(q)$ for all $q \in Q$. By Proposition 13, the latter is equivalent to $\hat{s}$ is Δ-inhabited. This concludes the proof of the theorem. □

We finally show that context inhabitation is in PSPACE for DTAs, even though this is a problem concerning automata for trees and not words. Indeed, inhabitation for DTAs can be reduced to the nonemptiness of intersection for words $\text{INTER}_\Sigma(\text{DFA})$, which is PSPACE-complete [Kozen 1977]. The PSPACE upper bound holds even for the uniform variant of the problem. Showing this requires two additional tricks in the proof, but is worth the effort since the uniform version of context inhabitation will be needed for solving the uniform version of regular matching, to which the non-uniform version of regular matching with constraints will be reduced. The constraints will allow us to solve regular matching in the case of unranked trees, the original motivation of the present work.

Lemma 14 *The problem* $\text{INHAB}^{context}(\text{DTA})$ *can be reduced in polynomial time to its restriction where the input function* $s \colon Q \to 2^Q$ *always maps to singletons.*

Proof: If there exists $q \in Q$ such that $s(q)$ contains more than one element then s cannot be inhabited for any DTA. It remains to remove cases where $s(q) = \emptyset$ for some $q \in Q$. The main idea to deal with empty sets is to complete A to A' by adding a sink state q_{sink} and to replace function s by s' such that $s'(q) = s(q)$ if $s(q) \neq \emptyset$ and $s'(q) = \{q_{sink}\}$ otherwise. Inhabitation of s with respect to A is then equivalent to inhabitation of s' with respect to A'. However, this construction may take exponential time in the maximal arity of function symbols of A with is not fixed for the uniform problem. This problem can be circumvented by a trick, permitting to complete A only partially.

Here is how it works. We consider a DTA $A = (Q, \Sigma, F, \Delta)$ and a function $s : Q \to 2^Q$. We construct another DTA $A' = (Q', \Sigma', F', \Delta')$ and a function $s' : Q' \to 2^{Q'} \setminus \emptyset$ such that $\hat{s}$ is Δ-inhabited if and only if $\hat{s'}$ is Δ'-inhabited.

The first idea would be to set A' as the completion of A. We then have $\Sigma' = \Sigma$ and $Q' = Q \cup \{q_{sink}\}$ where q_{sink} is some fresh sink state. Furthermore, the set of rules Δ' subsumes Δ and all the rules $f(q_1, \ldots, q_n) \to q_{sink}$ with $q_1, \ldots, q_n \in Q'$ for which $f(q_1, \ldots, q_n)$ is not a left-hand side of any rule in Δ. The function s' is defined such that $s'(q) = s(q)$ if $s(q) \neq \emptyset$ and $s'(q) = \{q_{sink}\}$ otherwise. One can then see for any context $C \in \mathcal{C}_\Sigma$ that $\hat{s}$ is Δ-inhabited by C if and only if $\hat{s'}$ is Δ'-inhabited by C. The size of Δ' is in $O(|\Delta| + |\Sigma||Q|^n)$ where n is the maximal arity of function symbols in Σ. Unfortunately, the maximal arity is not fixed in the uniform version since Σ is part of the input. Therefore, this reduction requires exponential space in the worst case, while polynomial time was claimed.

The second idea is to perform some kind of partial completion, so that only polynomially many rules need to be added. For this, we define the signature $\Sigma' = \Sigma \cup \{g\}$ where g is a fresh monadic function symbol. For any context $C \in \mathcal{C}_\Sigma$ we define a context in $\mathcal{C}_{\Sigma'}$ by $C' = \lambda x.C@g(x)$. The state set of A' remains $Q' = Q \cup \{q_{sink}\}$ where q_{sink} is some fresh state as before. The set of rules Δ' extends Δ by the following rules for all $q \in Q$:

$$h(q) \to \begin{cases} q_{sink} & \text{if } s(q) = \emptyset \\ q & \text{else} \end{cases}$$

Furthermore, we add the following rule for all rules $f(q_1, \ldots, q_n) \to q'$ of Δ and all $1 \leq i \leq n$:

$$f(q_1 \ldots, q_{i-1}, q_{sink}, q_{i+1}, \ldots, q_n) \to q_{sink}$$

It can then be shown for any context $C \in \mathcal{C}_\Sigma$, that $\hat{s}$ is Δ-inhabited by C if and only if $\hat{s'}$ is Δ'-inhabited by C'. Now the construction of A' is in time $O(|A|^2 + |s|)$ which is polynomial even if the maximal arity of function symbols in Σ is not bounded. □

Proposition 15 $\text{INHAB}^{context}(\text{DTA})$ *is in* PSPACE.

Proof: Let Σ be a ranked signature, $A = (Q, \Sigma, F, \Delta)$ a DTA where $Q = \{q_1, \ldots, q_n\}$ and all the states are accessible, $s \colon Q \to 2^Q$ a function and x a fresh constant not in Σ. We assume w.l.o.g that $s(q_i) \neq \emptyset$ for all $1 \leq i \leq n$ (see Lemma 14). If $|s(q_i)| > 1$ for some $1 \leq i \leq n$, then $\hat{s}$ is not Δ-inhabited, given that A is deterministic. The following lines consider the case where all the images by s are singletons. First we reduce the inhabitation of $\hat{s}$ to the nonemptiness of the intersection of $n + 1$ DTAs $A_1, \ldots, A_{n+1}$. In a second step, we reduce the nonemptiness of the intersection of

$A_1, \ldots, A_{n+1}$ to the nonemptiness of the intersection of n DFAs $A'_1, \ldots, W_i$.

We write $\Sigma_x = \Sigma \cup \{x\}$. For any $i \in \{1, \ldots, n\}$, let $A_i = (Q, \Sigma_x, s(q_i), \Delta_i)$ be the tree automaton on Σ_x having the same states as A, whose set of final states is $s(q_i)$, and whose transition relation is $\Delta_i = \Delta \cup \{x \to q_i\}$. We also write A_{n+1} to denote the simple DTA that accepts all trees $t \in \mathcal{T}_{\Sigma_x}$ having exactly one occurrence of x. We first show that

Claim 8 *There exists a context $\lambda x.p \in \mathcal{C}_\Sigma$ such that $[\![\lambda x.p]\!]^\Delta = \hat{s}$ if and only if $\bigcap_{i=1}^{n+1} L(A_i) \neq \emptyset$.*

Proof: On one hand, if there is a context $\lambda x.p \in \mathcal{C}_\Sigma$ such that $[\![\lambda x.p]\!]^\Delta = \hat{s}$, then by Proposition 13 we have $[\![p[x/\{q_i\}]]\!]^\Delta = s(q_i)$ for any $1 \leq i \leq n$. This implies that $p \in L(A_i)$ for any $1 \leq i \leq n$, and since $\lambda x.p$ is a context, p contains exactly one occurrence of x and thus belongs to $L(A_{n+1})$. Hence $\bigcap_{i=1}^{n+1} L(A_i) \supseteq \{p\} \neq \emptyset$. On the other hand, assume $\bigcap_{i=1}^{n+1} L(A_i) \neq \emptyset$ and let $p \in \bigcap_{i=1}^{n+1} L(A_i)$. Given that $p \in L(A_{n+1})$, it contains exactly one occurrence of x. Furthermore, since the automata A_j are all deterministic with unique final states $s(q_j)$, and $p \in \bigcap_{j=1}^{n} L(A_j)$, we have $[\![p]\!]^{\Delta_i} = s(q_i)$ for $1 \leq i \leq n$. This implies that $[\![p[x/\{q_i\}]]\!]^\Delta = s(q_i)$ for $1 \leq i \leq n$, and thus $[\![\lambda x.p]\!]^\Delta = \hat{s}$. $\square$ According to Claim 8, deciding whether or not $\hat{s}$ is Δ-inhabited is equivalent to determining if the DTAs A_i have an nonempty intersection. Next we show a PSPACE algorithm to decide $\bigcap_{i=1}^{n+1} L(A_i) \neq \emptyset$, by reduction to INTER(DFA).

Let Σ_Q be the alphabet that contains the symbol $\underline{x}$, and for any rule $f(q'_1, \ldots, q'_{k-1}, q'_k, q'_{k+1}, \ldots, q'_m) \to q''$ in Δ and any $1 \leq k \leq m$, Σ_Q contains the symbol $\underline{f(q'_1, \ldots, q'_{k-1}, \star, q'_{k+1}, \ldots, q'_m)}$, where m is the arity of f. Formally,

$$\Sigma_Q = \{\underline{x}\} \cup \left\{ \begin{array}{l} \underline{f(q'_1, \ldots, q'_{k-1}, \star, q'_{k+1}, \ldots, q'_m)} \mid m \text{ is an arity in } \Sigma, \\ f \in \Sigma^{(m)}, 1 \leq k \leq m \text{ and } \exists q'_k, q'_{m+1} \in Q. \\ f(q'_1, \ldots, q'_{k-1}, q'_k, q'_{k+1}, \ldots, q'_m) \to q'_{m+1} \in \Delta \end{array} \right\}$$

The notation introduced for the elements of Σ_Q allows us to distinguish them from the trees in $\mathcal{T}_{\Sigma_x \cup Q}$. This is because the elements of Σ_Q are considered as atomic symbols. Now let the alphabet $\mathfrak{S} = \Sigma_Q \cup \{\bot\}$. For some $i \in \{1, \ldots, n\}$ and a tree $t \in \mathcal{T}_{\Sigma_x} \cap L(A_{n+1})$ over Σ_x containing exactly one occurrence of x, we define inductively the *run path* $rp_i(t)$ of t with respect to the DTA A_i as a word over $\mathfrak{S}$ such that:

- if $t = x$, then $rp_i(t) = \underline{x}$

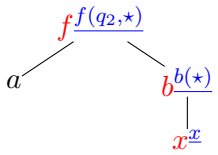


Figure 6.9: Run path (in blue) of the tree $f(a, b(x))$ with respect to A_1.

- if $t = f(t_1, \ldots, t_{k-1}, t_k, t_{k+1}, \ldots, t_m)$ for some arity $m \geq 0$, symbol $f \in \Sigma^{(m)}$, integer $k \in \{1, \ldots, m\}$ so that t_k contains the only occurrence of x in t, then

$$rp_i(t) = \begin{cases} rp_i(t_k)\underline{f(q'_1, \ldots, q'_{k-1}, \star, q'_{k+1}, \ldots, q'_m)} \text{ if } \{q'_j\} = [\![t_j]\!]^{\Delta_i} \text{ for all } j \neq k, \\ 1 \leq j \leq m \text{ and } f(q'_1, \ldots, q'_{k-1}, \star, q'_{k+1}, \ldots, q'_m) \in \Sigma_Q \\ \\ rp_i(t_k)\bot \text{ otherwise} \end{cases}$$

Example 16 *For instance, consider that* $\Sigma = \{f^{(2)}, b^{(1)}, c^{(1)}, a^{(0)}\}$ *and the transition relation* Δ_1 *of the* DTA A_1 *is such that* $\Delta_1 = \{x \to q_1, a \to q_2, b(q_1) \to q_3, f(q_2, q_3) \to q_4\} \cup \Delta'$ *where* Δ' *consists of the remaining rules that make* Δ_1 *complete. Then the run path of the tree* $f(a, b(x))$ *with respect to* A_1 *is* $\underline{x}\ \underline{b(\star)}\ \underline{f(q_2, \star)}$ *as illustrated in Figure 6.9. On the other hand, the run path of* $f(c(a), b(x))$ *with respect to* A_1 *is* $\underline{x}\ \underline{b(\star)}\ \bot$*, as the subtree* $c(a)$ *cannot be evaluated.*

Claim 9 *Let* $t \in \mathcal{T}_{\Sigma_x} \cap L(A_{n+1})$ *be a tree over* Σ_x *containing exactly one occurrence of* x*. Then* $rp_i(t) = rp_j(t)$ *for all* $i, j \in \{1, \ldots, n\}$*.*

Proof: Let $i, j \in \{1, \ldots, n\}$. The proof is by induction on the structure of t.

Case $t = x$**.** Then $rp_i(t) = rp_j(t) = \underline{x}$ by definition.

Case $t = f(t_1, \ldots, t_k, \ldots, t_m)$ for some arity m, symbol $f \in \Sigma^{(m)}$, integer $k \in \{1, \ldots, m\}$ so that t_k contains the only occurrence of x in t. By definition,

- $rp_i(t) = rp_i(t_k)a_i$
- and $rp_j(t) = rp_j(t_k)a_j$

where a_i and a_j are such that $a_i \in \{\underline{f(q'_1, \ldots, q'_{k-1}, \star, q'_{k+1}, \ldots, q'_m)}, \bot\}$, $a_j \in \{\underline{f(q''_1, \ldots, q''_{k-1}, \star, q''_{k+1}, \ldots, q''_m)}, \bot\}$ for states $q'_l \in Q$, $q''_l \in Q$, $l \neq k$ and $l \in \{1, \ldots, m\}$. By the induction hypothesis, $rp_i(t_k) = rp_j(t_k)$. Furthermore, we show that $a_i = a_j$. Let $l \in \{1, \ldots, m\}$ be different from k. Since t_l contains no occurrence of x, we have $[\![t_l]\!]^{\Delta_i} = [\![t_l]\!]^{\Delta} = [\![t_l]\!]^{\Delta_j}$. Two cases may

occur, depending on the run of A on t_l. Either the run blocks, that is $[\![t_l]\!]^{\Delta} = \emptyset$, or it doesn't, implying that $[\![t_l]\!]^{\Delta} = \{q'_l\}$ for some state $q'_l \in Q$. Now if for all $l \in \{1, \ldots, m\}$ different from k, $[\![t_l]\!]^{\Delta}$ equals some singleton $\{q'_l\}$, then by definition $a_i = \underline{f(q'_1, \ldots, q'_{k-1}, \star, q'_{k+1}, \ldots, q'_m)} = \underline{f(q''_1, \ldots, q''_{k-1}, \star, q''_{k+1}, \ldots, q''_m)} = a_j$. And if there is some $l \in \{1, \ldots, m\}$ different from k such that $[\![t_l]\!]^{\Delta} = \emptyset$, then by definition $a_i = \bot = a_j$. In both cases, $a_i = a_j$. Thus $rp_i(t) = rp_j(t)$.

□ Next we build DFAs that accept run paths. Let q_0 and $q_\bot$ be fresh states, and note $Q_{\text{DFA}} = Q \cup \{q_0, q_\bot\}$. For all $1 \leq i \leq n$, we build the DFA W_i having Q_{DFA} as its set of states, $\mathfrak{S}$ as its alphabet, $\{q_0\}$ as its set of initial states, $s(q_i)$ as its set of final states and δ_i as its transition function, so that

- $\delta_i(q_0, \underline{x}) = q_i$ (1)
- for all $f(q'_1, \ldots, q'_m) \to q'_{m+1} \in \Delta_i$ where $f \in \Sigma^{(m)}$ for some arity m, we have $\delta_i(q'_1, \underline{f(\star, q'_2, \ldots, q'_m)}) = q'_{m+1}$, $\delta_i(q'_2, \underline{f(q'_1, \star, q'_3, \ldots, q'_m)}) = q'_{m+1}$, $\ldots, \delta_i(q'_m, \underline{f(q'_1, \ldots, q'_{m-1}, \star)}) = q'_{m+1}$ (2)
- for all $q \in Q_{\text{DFA}}$, $\delta_i(q, \bot) = q_\bot$ (3)
- for all state $q \in Q_{\text{DFA}}$ and symbol $\underline{f(q'_1, \ldots, q'_{i-1}, \star, q'_{i+1}, \ldots, q'_m)} \in \Sigma_Q$ where $m \geq 0$ and $1 \leq i \leq m$, if no rule in Δ_i having $f(q'_1, \ldots, q'_{i-1}, q, q'_{i+1}, q'_m)$ as its left-hand side exists, then $\delta_i(q, \underline{f(q'_1, \ldots, q'_{i-1}, \star, q'_{i+1}, \ldots, q'_m)}) = q_\bot$ (4).

Note that any DFA W_i has a size that is polynomial in $|A|$. Now let $p \in \mathcal{T}_{\Sigma_x} \cap L(A_{n+1})$ be a tree over Σ_x containing exactly one occurrence of x.

Claim 10 *For all $i \in \{1, \ldots, n\}$ and state $q \in Q$, $[\![p]\!]^{\Delta_i} = \{q\}$ if and only if $rp_i(p)$ is evaluated to q by the* DFA *W_i.*

Proof: Let $i \in \{1, \ldots, n\}$ and $q \in Q$. The proof is by induction on the structure of p. The backward direction is shown by contraposition.

Case p=x. Then we have $rp_i(p) = \underline{x}$. First let's assume that $[\![p]\!]^{\Delta_i} = \{q\}$. So we have $q = q_i$, since $[\![p]\!]^{\Delta_i} = [\![x]\!]^{\Delta_i} = \{q_i\}$. Furthermore, W_i in its initial state q_0 reads $\underline{x}$ and enters by (1) in state $q_i = q$. Thus $rp_i(p)$ is evaluated to q by W_i.

For the backwards direction, assume that $[\![p]\!]^{\Delta_i} \neq \{q\}$. This implies that $[\![x]\!]^{\Delta_i} = \{q_i\} \neq \{q\}$, that is $q_i \neq q$. On the other hand, starting from q_0, W_i evaluates $\underline{x}$ to $q_i \neq q$ according to (1).

Case $p = f(p_1, \ldots, p_k)$ where $f \in \Sigma^{(k)}$ and $p_1, \ldots p_k \in \mathcal{T}_{\Sigma_x}$. Then there exists a unique $l \in \{1, \ldots, k\}$ such that p_l contains exactly one occurrence of x, and for all $j \in \{1, \ldots, k\}$, if $j \neq l$ then $p_j \in \mathcal{T}_\Sigma$ – that is p_j contains only symbols in Σ.

First assume that $[\![p]\!]^{\Delta_i} = \{q\}$. Then there exist states $\gamma_1, \ldots, \gamma_k$ s.t. for all $1 \leq j \leq k$, $[\![p_j]\!]^{\Delta_i} = \{\gamma_j\}$. By the induction hypothesis, $[\![p_l]\!]^{\Delta_i} = \{\gamma_l\}$ if and only if $rp_i(p_l)$ is evaluated to γ_l by W_i. By definition $rp_i(p) = rp_i(p_l)\underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}$. We also have the equalities $[\![p]\!]^{\Delta_i} = [\![f(p_1, \ldots, p_k)]\!]^{\Delta_i} = \{q\}$. So the rule $f(\gamma_1, \ldots, \gamma_k) \to q$ exists in Δ_i. By (2), we also have $\delta_i(\gamma_l, \underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}) = q$. So the DFA W_i in state q_0 first reads the word $rp_i(p_l)$ to get in state γ_l, before finally entering state q after having read $\underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}$. So $rp_i(p)$ can be evaluated to q by W_i.

For the backwards direction, assume that $[\![p]\!]^{\Delta_i} \neq \{q\}$. Two cases may occur:

Case $[\![p]\!]^{\Delta_i} = \emptyset$. Then

- either $[\![p_j]\!]^{\Delta_i} = \emptyset$ for some $j \in \{1, \ldots, k\}$ (i),
- or there exist states $\gamma_1, \ldots, \gamma_k$ s.t. for all $j \in \{1, \ldots, k\}$, $[\![p_j]\!]^{\Delta_i} = \{\gamma_j\}$, but there is no rule in Δ_i having $f(\gamma_1, \ldots, \gamma_k)$ as its left-hand side (ii).

In (i), if $j \neq l$ we have by definition that $rp_i(p) = rp_i(p_l)\bot$. According to rule (3), whatever the state in which the DFA W_i is after having read $rp_i(p_l)$, W_i goes to state $q_\bot$ when reading $\bot$. And since $q_\bot \neq q$, the claim holds. On the other hand, if $j = l$ and $[\![p_{j'}]\!]^{\Delta_i} = \{\gamma_{j'}\}$ for all $j' \in \{1, \ldots, k\}$ different from j, then $rp_i(p) = rp_i(p_l)\underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}$. By the induction hypothesis, W_i evaluates $rp_i(p_l)$ to a state that is not in Q. The only states in Q_{DFA} that are not in Q are q_0 and $q_\bot$, and given that $rp_i(p_l) \neq \varepsilon$ and q_0 has no looping transition – W_i can't stay in state q_0 after having read p_l –, it follows that the only possible state to which $rp_i(p_l)$ has been evaluated by W_i is $q_\bot$. All the transitions in δ_i that leave $q_\bot$ end up in $q_\bot$ by the rule (4). Thus W_i evaluates $rp_i(p)$ in state $q_\bot \neq q$, and the claim holds. In (ii), $rp_i(p) = rp_i(p_l)\underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}$. By the induction hypothesis, W_i evaluates $rp_i(p_l)$ to state γ_l. But since no rule $f(\gamma_1, \ldots, \gamma_k) \to q'$ exists in Δ_i, W_i in state γ_l – after having read $rp_i(p_l)$ – goes to state $q_\bot$ after reading $\underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}$, according to rule (4). Thus the claim holds.

Case $[\![p]\!]^{\Delta_i} = \{q'\} \neq \{q\}$. Then there exist states $\gamma_1, \ldots, \gamma_k$ s.t. for all $1 \leq j \leq k$, $[\![p_j]\!]^{\Delta_i} = \{\gamma_j\}$. Moreover, there is a rule $f(\gamma_1, \ldots, \gamma_k) \to q' \in \Delta_i$, but no rule $f(\gamma_1, \ldots, \gamma_k) \to q$ in Δ_i. Thus by (2), we have that $\delta(\gamma_l, \underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}) = q'$. This implies that the DFA W_i in state q_0, first reads $rp_i(p_l)$ to get in state γ_l, then reads the symbol $\underline{f(\gamma_1, \ldots, \gamma_{l-1}, \star, \gamma_{l+1}, \ldots, \gamma_k)}$ to enter state $q' \neq q$. Thus the claim holds.

□

We next state:

Claim 11 $\bigcap_{i=1}^{n+1} L(A_i) \neq \emptyset$ *if and only if* $\bigcap_{i=1}^{n} L(W_i) \neq \emptyset$.

Proof: Let $p \in \Sigma_x \cap L(A_{n+1})$ be a tree containing exactly one occurrence of x. By Claim 10, for all $i \in \{1, \ldots, n\}$, $[\![p]\!]^{\Delta_i} = s(q_i)$ if and only if $rp_i(p)$ is evaluated to the single element of $s(q_i)$ by W_i. So $p \in L(A_i)$ if and only if $rp_i(p) \in L(W_i)$ for $1 \leq i \leq n$. Claim 9 has established that $rp_j(p) = rp_k(p)$ for all $j, k \in \{1, \ldots, n\}$. It then follows that $p \in \bigcap_{i=1}^{n} L(A_i)$ if and only if $rp_1(p) = \ldots = rp_n(p) \in \bigcap_{i=1}^{n} L(W_i)$. So $\bigcap_{i=1}^{n+1} L(A_i) \neq \emptyset$ if and only if $\bigcap_{i=1}^{n} L(W_i) \neq \emptyset$. □ It follows from Claim 8 and Claim 11 that $\hat{s}$ is Δ-inhabited if and only if $\bigcap_{i=1}^{n} L(W_i) \neq \emptyset$. Thus $\textsc{Inhab}^{context}(\textsc{Dta})$ is reducible in polynomial time to $\textsc{Inter}(\textsc{Dfa})$. Hence $\textsc{Inhab}^{context}(\textsc{Dta})$ is in PSPACE.

□

Theorem 5 $\textsc{Inhab}_{\Sigma}^{context}(\textsc{Dta})$ *is* PSPACE*-complete.*

Proof: The upper bound follows from Proposition 15. The lower bound can be shown by reduction from the nonemptiness problem of the intersection of a finite number of DFAs. Any finite word $a_1, \ldots, a_{m-1}, a_m$ can be encoded by a "string" tree $a_m(a_{m-1}(\ldots a_1(x)\ldots))$, where $a_1, \ldots, a_m$ are unary symbols and x is a fresh constant symbol. Similarly, any DFA A can be transformed in linear time to a DTA A' that accepts exactly the string encodings of words from $L(A)$, and which transitions are trivial encodings of transitions of A plus an additional transition $x \to q^x$ for some fresh state q^x. Now given DFAs $A_1, \ldots, A_n$ which we assume w.l.o.g. to have pairwise disjoint sets of states, and single final states, let $A'_1, \ldots, A'_n$ be the respective corresponding DTAs as described above. We write q_i^f to denote the only final state of the DTA A'_i, for $1 \leq i \leq n$. Let B be the union of the A'_i excluding the rules of the form $x \to q_i^x$, and Δ be the transition relation of B. Remark that B is deterministic. Then the intersection of the languages of $A_1, \ldots, A_n$ is non-empty iff

$\hat{s}$ is Δ-inhabited, where $\hat{s}$ is defined by $s(q) = \{q_i^f\}$ if $q = q_i^x$ for any $1 \leq i \leq n$, and $s(q) = \emptyset$ otherwise. □

6.4 Evaluation of Compressed Tree Patterns over NTAs

Our next objective is to evaluate compressed tree patterns efficiently over the Σ-algebra of some NTA for a given variable assignment into this algebra. In particular, we want to avoid any kind of decompression when doing so.

The precise formalization of this statement needs a little care, since we have to work with representations of variable assignments as inputs rather than with variable assignment themselves. Let $A = (Q, \Sigma, F, \Delta)$ be an NTA and $\sigma\colon V \to [\![Val_\Sigma]\!]^\Delta$ a well-typed variable assignment into the Σ-algebra $\Delta = (\Sigma, 2^Q, .^\Delta)$. The problem is that the context variables X in V are mapped to union homomorphisms $\sigma(X)\colon 2^Q \to 2^Q$ (see Definition 27) which may be of exponential size, but can be represented in polynomial space by a function $s(X)\colon Q \to 2^Q$ with $\sigma(X) = \widehat{s(X)}$.

Definition 28 *A function s represents a variable assignment $\sigma\colon V \to [\![Val_\Sigma]\!]^\Delta$ into the Σ-algebra of the* NTA *$A = (Q, \Sigma, F, \Delta)$ if $dom(s) = dom(\sigma)$, $\sigma(X) = \widehat{s(X)}$ for all context variables $X \in dom(s)$, and $\sigma(x) = s(x)$ for all tree variables $x \in dom(s)$. In this case, we write $\sigma = \hat{s}$.*

A similar result to the following lemma can be found for instance in [Lohrey *et al.* 2012].

Lemma 15 *For any* NTA *$A = (Q, \Sigma, F, \Delta)$, compressed tree pattern $G = \mathcal{P}_\Sigma^{comp,tree}$, and representation s of a variable assignment $\hat{s}$ into the Σ-algebra Δ with $fv(G) \subseteq dom(s)$ we can compute the Δ-value of the pattern $[\![pat(G)]\!]^{\Delta,\hat{s}}$ in polynomial time from Σ, Δ, G, and s.*

Proof: The algorithm evaluates the pattern inductively along the partial order on the nonterminals of G; the latter exists because G is acyclic. For any $v \in V$, let G_v be the compressed tree pattern equal to G except that the start symbol is changed to v. Then we can show for all $v \in V$ that $[\![pat(G_v)]\!]^{\Delta,\hat{s}}$ can be computed in polynomial time from Σ, Δ, G, and s. In particular this holds for $[\![pat(G)]\!]^{\Delta,\hat{s}} = [\![pat(G_S)]\!]^{\Delta,\hat{s}}$. □

6.5 Regular Matching and Inclusion

We now study the complexity of regular matching and inclusion for classes of compressed tree patterns with context variables.

A class of compress tree patterns $\mathcal{G}$ is a function that maps any signature Σ to a subset of compressed tree patterns $\mathcal{G}_\Sigma \subseteq \mathcal{P}_\Sigma^{comp,tree}$. Typical examples are the classes $\mathcal{P}^{tree}$ and $\mathcal{P}^{comp,tree}$ given that $\mathcal{P}_\Sigma^{tree} \subseteq \mathcal{P}_\Sigma^{comp,tree}$. To see this recall that we identify any tree pattern p with the compression-free compressed tree pattern $ctp_\Sigma(p) = (\{\mathcal{S}\}, \Sigma, \{\mathcal{S} \to p\}, \mathcal{S})$ where $\mathcal{S}$ is the fixed start symbol.

For any class $\mathcal{G}$ of compressed tree patterns, any class $\mathcal{A}$ of NTAs, and for any ranked alphabet Σ we define two decision problems:

REGULAR PATTERN INCLUSION: $\text{INCL}_\Sigma(\mathcal{G}, \mathcal{A})$.
Input: a compressed tree pattern $G \in \mathcal{G}_\Sigma$ and a tree automaton $A \in \mathcal{A}_\Sigma$.
Output: whether $Inst(pat(G)) \subseteq L(A)$.

REGULAR PATTERN MATCHING: $\text{MATCH}_\Sigma(\mathcal{G}, \mathcal{A})$.
Input: a compressed tree pattern $G \in \mathcal{G}_\Sigma$ and a tree automaton $A \in \mathcal{A}_\Sigma$.
Output: whether $Inst(pat(G)) \cap L(A) \neq \emptyset$.

The uniform versions of these problems where the signature Σ is given with the input are called $\text{INCL}(\mathcal{G}, \mathcal{A})$ and respectively $\text{MATCH}(\mathcal{G}, \mathcal{A})$.

6.5.1 Lower Bounds

We first establish the lower bounds for regular matching by reduction from automata intersection problems. In the second step, we establish the lower bounds for the dual problem of regular inclusion. In the deterministic case, the lower bounds for regular matching can be lifted to regular inclusion based on automaton complementation. In the nondeterministic case, another lower bound result needs to be established.

Proposition 16 (Regular matching) $\text{MATCH}_\Sigma(\mathcal{P}^{tree}, \text{NTA})$ *is* EXP*-hard, while* $\text{MATCH}_\Sigma(\mathcal{P}^{tree}, \text{DTA})$ *is* PSPACE*-hard.*

Proof: We first notice that $\text{MATCH}_\Sigma(\mathcal{P}^{tree}, \text{NTA})$ generalizes the ground instance intersection problem from [Comon *et al.* 2007] by adding compression and context variables. The latter problem is known to be EXP-complete for NTAs, so the EXP-hardness of $\text{MATCH}_\Sigma(\mathcal{P}^{tree}, \text{NTA})$ follows. In order to clarify the role of nondeterminism here, we recall the proof of this result, which is based on a reduction from intersection nonemptiness of a finite number of DTAs $\text{INTER}_\Sigma(\text{DTA})$.

The reduction is as follows. Given a sequence of DTAs $A_1, \ldots, A_n$ over the same signature Σ we can construct in P an NTA A over $\Sigma \cup \{f\}$ that recognizes the language $f(L(A_1), \ldots, L(A_n))$, where f is a fresh function symbol of arity n. The transition relation of A is the union of the transition relations of $A_1, \ldots, A_n$ extended

with rules $f(q_1^f, \ldots, q_n^f) \to q_{ok}$ where q_i^f is the final state of A_i, whose uniqueness can be assumed without loss of generality. Note that A is nondeterministic. We fix a tree variable $x \in \mathcal{V}$ arbitrarily. The regular tree pattern matching task

$$Inst(f(\underbrace{x, \ldots, x}_{n})) \cap L(A) = \emptyset$$

is then equivalent to the intersection emptiness task $L(A_1) \cap \ldots \cap L(A_n) = \emptyset$. To finish the reduction, we note that one can reduce the problem with signature $\Sigma \cup \{f\}$ to the same problem with signature Σ by simulating the new symbol f by the function symbol of arity at least 2 and the constant available in Σ by assumption.

It should be noticed that A is inherently nondeterministic by construction. Therefore, this EXP-hardness proof does not apply to $\text{MATCH}_\Sigma(\mathcal{P}^{tree}, \text{DTA})$. And indeed, as we will see this problem is not EXP-hard but PSPACE-complete.

The PSPACE-hardness of $\text{MATCH}_\Sigma(\mathcal{P}^{tree}, \text{DTA})$ follows from the special case of regular string matching, which was shown to be PSPACE-complete for deterministic finite automata (DFAs) [Boneva *et al.* 2018].

□

Lemma 16 (Duality via Complementation) *For any class of compressed tree patterns $\mathcal{G}$, the problems* $\text{INCL}_\Sigma(\mathcal{G}, \text{DTA})$ *and* $\text{coMATCH}_\Sigma(\mathcal{G}, \text{DTA})$ *are equivalent modulo polynomial time reductions.*

Proof: For any compressed tree pattern G and DTA A, we have $Inst(pat(G)) \subseteq L(A)$ iff $Inst(pat(G)) \cap \overline{L(A)} = \emptyset$ iff $Inst(pat(G)) \cap L(\overline{A}) = \emptyset$, where $\overline{A}$ is the complement automaton for A that can be computed in polynomial time since A is a DTA. □

As a consequence of Lemma 16 the problem $\text{INCL}_\Sigma(\mathcal{G}, \text{NTA})$ is equivalent to $\text{coMATCH}_\Sigma(\mathcal{G}, \text{NTA})$ modulo NTA determinization, which however requires exponential time. We now show that regular inclusion for NTAs is EXP-hard even for linear tree patterns. Even the class of tree patterns $\mathcal{V}^{tree}$ in which each pattern consists simply of a tree variable is enough. More formally this is the class of compressed tree patterns such that $\mathcal{V}_\Sigma^{tree} = \{ctp_\Sigma(x) \mid x \in \mathcal{V}^{tree}\}$ for all signatures Σ.

Lemma 17 $\text{INCL}_\Sigma(\mathcal{V}^{tree}, \text{NTA})$ *is* EXP*-hard.*

Proof: Let A be an NTA. The instance set of any pattern $x \in \mathcal{V}^{tree}$ is equal to $\mathcal{T}_\Sigma$. This set is included in $L(A)$ if and only if A is universal. The universality problem for NTAs is well known to be EXPTIME-complete. □

Proposition 17 (Regular inclusion) $\textsc{Incl}_\Sigma(\mathcal{P}^{tree}, \textsc{Dta})$ *is* Pspace*-hard, while* $\textsc{Incl}_\Sigma(\mathcal{P}^{tree}, \textsc{Nta})$ *is* Exp*-hard.*

Proof: Lemma 16 states that $\textsc{Incl}_\Sigma(\mathcal{P}^{tree}, \textsc{Dta}) = \textsc{coMatch}_\Sigma(\mathcal{P}^{tree}, \textsc{Dta})$ modulo polynomial time reductions. By Proposition 16, $\textsc{Match}_\Sigma(\mathcal{P}^{tree}, \textsc{Dta})$ is Pspace-hard and since Pspace is closed by complement, $\textsc{coMatch}_\Sigma(\mathcal{P}^{tree}, \textsc{Dta})$ is Pspace-hard too. Hence $\textsc{Incl}_\Sigma(\mathcal{P}^{tree}, \textsc{Dta})$ is Pspace-hard.

In the case of Ntas, the Exp-hardness of $\textsc{Incl}_\Sigma(\mathcal{P}^{tree}, \textsc{Nta})$ follows immediately from Lemma 17. □

6.5.2 Upper Bounds

All upper bounds will be obtained in a systematic manner by some algorithm that instead of guessing trees or contexts in Val_Σ will guess Δ-inhabited values in $[\![Val_\Sigma]\!]^\Delta$. For the guessing, a subroutine will be applied that decides tree or context inhabitation.

We start with a characterization of regular matching and inclusion, on which our decision procedure will rely.

Lemma 18 (Characterization) *Let* $A = (Q, \Sigma, F, \Delta)$ *be an* Nta *and* $p \in \mathcal{P}^{tree}_\Sigma$ *a tree pattern.*

Regular matching: *$Inst(p) \cap L(A) \neq \emptyset$ holds iff there exists some well-typed variable assignment to Δ-inhabited values $\sigma\colon fv(p) \to [\![Val_\Sigma]\!]^\Delta$ such that $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$.*

Regular inclusion: *$Inst(p) \subseteq L(A)$ holds iff all well-typed variable assignments to Δ-inhabited values $\sigma\colon fv(p) \to [\![Val_\Sigma]\!]^\Delta$ satisfy $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$.*

Proof: We start with the case of regular matching. For the forward direction, we assume $Inst(p) \cap L(A) \neq \emptyset$. By definition of instances, there exists a well-typed assignment $\mu\colon fv(p) \to Val_\Sigma$ such that $norm_\beta(\mu(p)) \in L(A)$. Let $\sigma = [\![.]\!]^\Delta \circ \mu$. Clearly $\sigma\colon fv(p) \to [\![Val_\Sigma]\!]^\Delta$ is a well-typed variable assignment. Since $[\![p]\!]^{\Delta,\sigma} = [\![\mu(p)]\!]^\Delta = [\![norm_\beta(\mu(p))]\!]^\Delta$ it follows that $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$.

For the inverse direction, we fix a well-typed variable assignment to Δ-inhabited values $\sigma\colon fv(p) \to [\![Val_\Sigma]\!]^\Delta$ such that $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$. By Δ-inhabitation there exists a well-typed variable assignment $\mu\colon fv(p) \to Val_\Sigma$ such that $\sigma = [\![.]\!]^\Delta \circ \mu$. Hence, $[\![\mu(p)]\!]^\Delta \cap F \neq \emptyset$, so that $[\![norm_\beta(\mu(p))]\!]^\Delta \cap F \neq \emptyset$. Thus $norm_\beta(\mu(p)) \in L(A)$, that is $norm_\beta(\mu(p)) \in Inst(p) \cap L(A)$.

The case of regular inclusion is similar. For the forward direction, we assume $Inst(p) \subseteq L(A)$ and fix a variable assignment to Δ-inhabited values $\sigma\colon fv(p) \to$

$[\![Val_\Sigma]\!]^\Delta$. By Δ-inhabitation, there exists a variable assignment $\mu\colon fv(p) \to Val_\Sigma$ such that $\sigma = [\![.]\!]^\Delta \circ \mu$. Since $norm_\beta(\mu(p)) \in Inst(p)$ it follows from $Inst(p) \subseteq L(A)$ that $norm_\beta(\mu(p)) \in L(A)$. Therefore, it follows from $[\![p]\!]^{\Delta,\sigma} = [\![\mu(p)]\!]^\Delta = [\![norm_\beta(\mu(p))]\!]^\Delta$ that $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$.

For the inverse direction, we assume that any variable assignment to Δ-inhabited values $\sigma\colon fv(p) \to [\![Val_\Sigma]\!]^\Delta$ satisfies $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$. We fix an element of $t \in Inst(p)$, which must be of the form $t = norm_\beta(\mu(p))$ for some $\mu\colon fv(p) \to Val_\Sigma$. The variable assignment $\sigma = [\![.]\!]^\Delta \circ \mu$ then maps to Δ-inhabited values, so that by assumption $[\![p]\!]^{\Delta,\sigma} \cap F \neq \emptyset$. Since $[\![p]\!]^{\Delta,\sigma} = [\![\mu(p)]\!]^\Delta = [\![norm_\beta(\mu(p))]\!]^\Delta = [\![t]\!]^\Delta$ it follows that $t \in L(A)$. □

We now show how to decide regular matching and inclusion based on algorithms with oracles for solving inhabitation problems. Given two complexity classes Ξ_1 and Ξ_2, we will write $\Xi_1(\Xi_2)$ for problems solvable in Ξ_1 when having an oracle in Ξ_2. We recall in particular that $\mathrm{NP}(\Xi) \subseteq \textsc{Exp}(\Xi)$, $\mathrm{coNP}(\Xi) \subseteq \textsc{Exp}(\Xi)$ and that $\textsc{Exp}(\textsc{Exp}) \subseteq \textsc{Exp}$. As a consequence, $\mathrm{NP}(\textsc{Exp}) \subseteq \textsc{Exp}$ and $\mathrm{coNP}(\textsc{Exp}) \subseteq \textsc{Exp}$. We also equip $\mathbf{T}$ with the total order $\leq_{\mathbf{T}}$ defined by *tree* $\leq_{\mathbf{T}}$ *context*.

Proposition 18 *Let $\mathcal{G}$ be a class of compressed tree patterns and $\mathcal{A}$ a class of* NTA*s. Let τ be the maximal type of free variables in a pattern in $\mathcal{G}$ wrt. $\leq_{\mathbf{T}}$ and suppose that* $\textsc{Inhab}^\tau(\mathcal{A})$ *belongs to complexity class Ξ. In this case,* $\textsc{Match}(\mathcal{G}, \mathcal{A})$ *belongs to* $\mathrm{NP}(\Xi)$ *and* $\textsc{Incl}(\mathcal{G}, \mathcal{A})$ *to* $\mathrm{coNP}(\Xi)$.

Proof: Let Σ be a ranked signature, $G = (N, \Sigma, \_, \mathcal{S})$ a compressed tree pattern of type *tree* in class $\mathcal{G}$, and $A = (Q, \Sigma, F, \Delta)$ be a tree automaton in class $\mathcal{A}$. According to Lemma 18, $pat(G)$ matches $L(A)$ iff some well-typed variable assignment $\sigma\colon fv(G) \to [\![Val_\Sigma]\!]^\Delta$ satisfies $[\![pat(G)]\!]^{\Delta,\sigma} \cap F \neq \emptyset$. For all context variables $X \in fv(G)$, the value $\sigma(X)$ belongs to $[\![\mathcal{C}_\Sigma]\!]^\Delta$ so it is a union homomorphism. Therefore, σ can be associated to a function s representing it in the sense of Definition 28. In order to find a suitable value for $\sigma(X)$, we guess a function $s(X)\colon Q \to 2^Q$ of which there are exponentially many (while the number of functions of type $2^Q \to 2^Q$ is doubly exponential) and test whether $\widehat{s(X)}$ is Δ-inhabited. The procedure is analogous for tree variables $x \in fv(G)$, except that sets of states $s(x) \subseteq Q$ are guessed and tested for inhabitation. The inhabitation test is an instance of $\textsc{Inhab}^\tau(\mathcal{A})$ which can be done by an Ξ oracle by assumption. Therefore, the guessing can be done by an algorithm in $\mathrm{NP}(\Xi)$. After having found Δ-inhabited values for all the free variables of G, the computation of $[\![pat(G)]\!]^{\Delta,\sigma}$ which equals to $[\![pat(G)]\!]^{\Delta,\hat{s}}$ can be done in polynomial time by Lemma 15, so the characterization of regular matching can be tested by an algorithm in $\mathrm{NP}(\Xi)$.

For $\textsc{Incl}(\mathcal{G}, \mathcal{A})$, the procedure is almost the same, except that by Lemma 18 we now have to guess a representation of a variable assignment $s\colon fv(G) \to [\![Val_\Sigma]\!]^\Delta$ such that $[\![pat(G)]\!]^{\Delta,\hat{s}} \cap F = \emptyset$ in order to contradict regular inclusion. This can be done by an algorithm in $\textsc{coNP}(\Xi)$. □

We next establish the complexity of the regular matching and inclusion problems.

Theorem 6 $\textsc{Match}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Dta})$ *and* $\textsc{Incl}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Dta})$ *are* Pspace*-complete, while* $\textsc{Match}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Nta})$ *and* $\textsc{Incl}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Nta})$ *are* Exp*-complete.*

Proof: The hardness results were shown in Proposition 16 and 17, so only the upper bounds remain to be proven. Let Σ be ranked signature.

On on hand, since $\textsc{Inhab}^{context}(\textsc{Dta})$ is in Pspace by Theorem 5, it follows from Proposition 18 that $\textsc{Match}(\mathcal{P}^{comp,tree}, \textsc{Dta})$ is in NP(Pspace) and thus in NPspace $\subseteq$ Pspace by Savitch's Theorem [Savitch 1970]. It also follows that $\textsc{Incl}(\mathcal{P}^{comp,tree}, \textsc{Dta})$ is in coNP(Pspace) which is in coNPspace = NPspace and thus in Pspace too. This allows to conclude that the problems $\textsc{Match}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Dta})$ and $\textsc{Incl}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Dta})$ are in Pspace.

On the other hand, since $\textsc{Inhab}^{context}(\textsc{Nta})$ is in Exp by Theorem 5, it follows by Proposition 18 that $\textsc{Match}(\mathcal{P}^{comp,tree}, \textsc{Nta})$ is in NP(Exp) and that $\textsc{Incl}(\mathcal{P}^{comp,tree}, \textsc{Nta})$ is in coNP(Exp). Hence both problems are in Exp, which imply that $\textsc{Match}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Nta})$ and $\textsc{Incl}_\Sigma(\mathcal{P}^{comp,tree}, \textsc{Nta})$ are also in Exp. □

6.6 Adding Regular Constraints

So far, regular matching and inclusion consider all the possible instances of the compressed tree pattern given as input, but it may be interesting to consider only instances satisfying some constraints. This is the case when schemas are defined for XML documents. In this part, we generalize the regular matching and inclusion problems by allowing constraints restricting how free variables are instantiated. Let Σ be a ranked signature and G a compressed tree pattern over Σ. An instantiation constraint $\mathfrak{c}$ on G is a total function that maps every free tree variable of G to a Dta over Σ and every free context variable of G to a Dta over $\Sigma \uplus \{x_\mathfrak{c}\}$ where $x_\mathfrak{c} \in \mathcal{V}^{tree}$. Furthermore, Dtas associated with context variables are allowed to recognize only languages of trees having exactly one occurrence of $x_\mathfrak{c}$. Note that $x_\mathfrak{c}$ is used to indicate the position of the *hole* in the contexts, that is the variable to be instantiated. A well-typed variable assignment $\sigma : fv(G) \to \mathcal{P}^{gr}_\Sigma$ satisfies $\mathfrak{c}$ if for every free tree variable $x \in fv(G)$, $norm_\beta(\sigma(x)) \in L(\mathfrak{c}(x))$ and for every free context

variable $X \in fv(G)$, $norm_\beta(\sigma(X)@x_{\mathfrak{c}}) \in L(\mathfrak{c}(X))$. We can now define the set of instances of G that satisfy $\mathfrak{c}$ as the set:

$$Inst^{\mathfrak{c}}(G) = \{norm_\beta(\sigma(pat(G))) \mid \sigma : fv(G) \to \mathcal{P}^{gr}_\Sigma \text{ well-typed and satisfies } \mathfrak{c}\}.$$

For any class of compressed tree patterns $\mathcal{G}$ and of NTAs $\mathcal{A}$ and any ranked signature Σ, the problems of regular pattern inclusion and matching with constraints are the following:

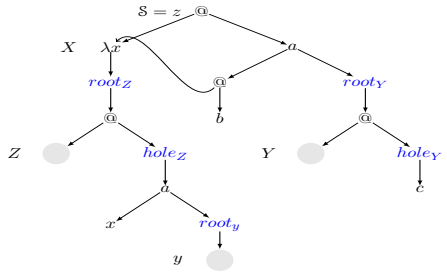
REGULAR PATTERN INCLUSION WITH CONSTRAINTS: $\text{cINCL}_\Sigma(\mathcal{G}, \mathcal{A})$.
Input: a compressed tree pattern $G \in \mathcal{G}_\Sigma$, a tree automaton $A \in \mathcal{A}_\Sigma$ and an instantiation constraint $\mathfrak{c} : fv(G) \to \text{DTA}_\Sigma \cup \text{DTA}_{\Sigma \uplus \{x_{\mathfrak{c}}\}}$.
Output: whether $Inst^{\mathfrak{c}}(pat(G)) \subseteq L(A)$.

REGULAR PATTERN MATCHING WITH CONSTRAINTS: $\text{cMATCH}_\Sigma(\mathcal{G}, \mathcal{A})$.
Input: a compressed tree pattern $G \in \mathcal{G}_\Sigma$, a tree automaton $A \in \mathcal{A}_\Sigma$ and an instantiation constraint $\mathfrak{c} : fv(G) \to \text{DTA}_\Sigma \cup \text{DTA}_{\Sigma \uplus \{x_{\mathfrak{c}}\}}$.
Output: whether $Inst^{\mathfrak{c}}(pat(G)) \cap L(A) \neq \emptyset$.

The uniform versions of these problems, where the signature can vary with the input, are written $\text{cMATCH}(\mathcal{G}, \mathcal{A})$ and $\text{cINCL}(\mathcal{G}, \mathcal{A})$. It can easily be seen that regular matching (resp. regular inclusion) is a special case of regular matching with constraints (resp. regular inclusion with matching), and that an algorithm for the general case can be used to solve the special case. What is more interesting is that regular matching with constraints (resp. regular inclusion with constraints) can also be reduced to uniform regular matching (resp. uniform regular inclusion), as stated in the next proposition:

Proposition 19 *For any class $\mathcal{G}$ of compressed tree patterns and any class of tree automata $\mathcal{A} \in \{\text{NTA}, \text{DTA}\}$, $\text{cMATCH}(\mathcal{G}, \mathcal{A})$ and $\text{cINCL}(\mathcal{G}, \mathcal{A})$ are reducible in polynomial time to respectively $\text{MATCH}(\mathcal{G}, \mathcal{A})$ and $\text{INCL}(\mathcal{G}, \mathcal{A})$.*

Proof: Let $\mathcal{G}$ be a class of compressed tree patterns, $\mathcal{A}$ a class of tree automata, Σ a ranked signature, $G \in \mathcal{G}_\Sigma$ a compressed tree pattern, $A = (Q, \Sigma, F, \Delta) \in \mathcal{A}_\Sigma$ a tree automaton and $\mathfrak{c}$ an instantiation constraint on G. The general idea is to build a new compressed tree pattern wherein there are places marked as *test zones*, that is, places that tell the automaton where constraints should be tested. Then we restrict the instances of this compressed tree pattern to the instances that satisfy $\mathfrak{c}$



$$\begin{aligned} z &\rightarrow root_X(X@hole_X(a(root_X(X@hole_X(b)), root_Y(Y@hole_Y(c))))),\\ X &\rightarrow \lambda x.root_Z(Z@hole_Z(a(x, root_y(y)))) \end{aligned}$$

Figure 6.10: $G' = mark_\Sigma(G)$ built from the compressed tree pattern G in Figure 5.5.

using two new automata, before testing matching and inclusion. We first associate to every free tree variable $x \in \mathcal{V}^{tree} \cap fv(G)$ a fresh unary symbol $root_x$ and to every free context variable $X \in \mathcal{V}^{context} \cap fv(G)$ two fresh unary symbols $root_X, hole_X$. These symbols, called markers, are used to delimit the test zones. Let $\Theta = \{root_\nu \mid \nu \in fv(G)\} \cup \{hole_X \mid X \in fv(G) \cap \mathcal{V}^{context}(G)\}$ be the set of markers. We define a function $mark_\Sigma$ that associates every compressed tree pattern G_1 over Σ with a new compressed tree pattern G_2 over Θ that is almost equal to G_1, except that

- every occurrence of a free tree variable $x \in fv(G_1)$ in G_1 is replaced by $root_x(x)$ in G_2
- every subterm $X@p$ of G_1 where $X \in fv(G_1)$ is a free context variable and $p \in \mathcal{P}_\Sigma$ a pattern is replaced by $root_X(X@hole_X(p))$ in G_2

Figure 6.10 illustrates the compressed tree pattern G' obtained after applying the $mark_\Sigma$ function on the compressed tree pattern of Figure 5.5.

Let the automaton A' over Θ built from A, so that $L(A') = \{mark_\Sigma(t) \mid t \in L(A)\}$. A' can be built in linear time from A, in a way that preserves an eventual determinism. We now build a new NTA B that will allow to test the constraints specified in $\mathfrak{c}$. Let q_{wait} be a fresh state. The state q_{wait} is the state in which B waits before testing a constraint, but also its final state. For every part of its input, B guesses whether it's in a test zone, and guesses the constraint to test. Thus, if B is reading the test zone of some free variable $\nu \in fv(G)$, it runs the automaton $\mathfrak{c}(\nu)$. If the constraint in $\mathfrak{c}(\nu)$ is satisfied, B returns to q_{wait} and waits for the next constraint to test. However, if no constraint is satisfied in a test zone, B blocks and doesn't get back to q_{wait}. For all $\nu \in fv(h)$, define Q_ν as the set of states of $\mathfrak{c}(\nu)$ and Δ_ν as its transition relation. We set $B = (Q_B, \Theta, \{q_{wait}\}, \Delta_B)$ where $Q_B = \{q_{wait}\} \cup \bigcup_{\nu \in fv(G)} Q_\nu$. The transition relation Δ_B is defined as the union of Δ_ν

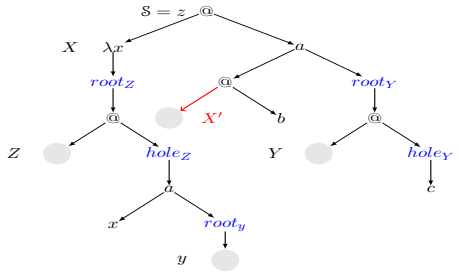


Figure 6.11: Compressed tree pattern built from G' in Figure 6.10 and used to build C.

for all $\nu \in fv(G)$, plus the following updates:

1. for all $f \in \Sigma^{(n)}$ where $n \geq 0$, add $f(\underbrace{q_{wait}, \ldots, q_{wait}}_{n}) \to q_{wait}$ to Δ_B
2. for all $X \in fv(G) \cap \mathcal{V}^{context}$, replace the only rule $x_{\mathfrak{c}} \to q_X$ by $hole_X(q_{wait}) \to q_X$ in Δ_B
3. for all $\nu \in fv(G)$ and final state q_ν^f of $\mathfrak{c}(\nu)$, add $root_\nu(q_\nu^f) \to q_{wait}$ to Δ_B.

Note the rule (2) that allows to simulate the reading of $x_{\mathfrak{c}}$ by constraint automata of contexts. So $x_{\mathfrak{c}}$ is not in the signature of B. Furthermore, B checks only whether the constraints that have been tested are satisfied, but cannot guarantee that all the constraints are tested. For this, one could have built an automaton that tests whether all the occurrences of all the variables of G' are instantiated. However, the instance set of G' is not a regular language in general. Instead, a DTA C that just tests whether all the variables of G' have one occurrence that is instantiated is enough. C is built in a way that it recognizes all the trees that have the same skeleton than G'. By same skeleton, we mean that the language of trees recognized by C is inspired from the instance set of G, except that all non-linearities in G' are removed. By replacing for instance the occurrences of variables – bound or free – in G' that are not first occurrences by fresh variables, we have a new compressed tree pattern whose instance set is a regular language. The DTA C recognizes this language. We illustrate in Figure 6.11 a compressed tree pattern obtained with this construction. Notice the new free variable X' replacing the second occurrence of X. Now remark that for some tree $t \in \mathcal{T}_\Sigma$, $t \in Inst^{\mathfrak{c}}(pat(G))$ if and only if $mark_\Sigma(t) \in Inst(pat(G')) \cap L(B) \cap L(C)$.

The main problem with our reduction is that B is not deterministic, although it is built from the DTAs $\mathfrak{c}(\nu)$. In order to solve it, we consider a new ranked signature Σ' where symbols $f \in \Sigma$ are associated to the variables ν, such that the tuple (f, ν)

is used only in some instantiation of ν. More formally, $\Sigma' = \Theta \cup \Sigma \cup (\Sigma \times fv(G))$. We modify G', A', B, and C to take into account the new signature Σ'. For G' we build a new compressed tree pattern G'' in linear time, that is equal to it but has the extended signature Σ'. Note that G'' preserves an eventual linearity of G. For B, we construct in linear time a DTA B' over Σ' equal to B except that every rule $f(q_1, \ldots, q_n) \to q \in \Delta_B$ – where $n \geq 0$ – that originates from a DTA $\mathfrak{c}(\nu)$ for some $\nu \in fv(G)$ is *replaced* by $(f, \nu)(q_1, \ldots, q_n) \to q$. This way, the set of rules of B is partitioned, according to their automata $\mathfrak{c}(\nu)$ of origin. Assuming – w.l.o.g – that the state sets of the automata $\mathfrak{c}(\nu)$ for $\nu \in fv(G)$ are disjoint, B' is indeed deterministic. Another consequence is that all letters of an instance of a free variable $\nu \in fv(G)$ must be annotated by the free variable ν itself. Unlike B, B' does not need to guess the constraint to test, as this is now indicated in the input. Finally, for A' and C, we build automata A'' and C' over Σ' – in polynomial time – so that for any rule $f(q_1, \ldots, q_n) \to q$ – where $n \geq 0$ – of their transition relations and any free variable $\nu \in fv(G)$, a new rule $(f, \nu)(q_1, \ldots, q_n) \to q$ is added.

Now observe that

Claim 12 *There exists a bijection $\varphi : Inst^{\mathfrak{c}}(pat(G)) \to Inst(pat(G'')) \cap L(B') \cap L(C')$ such that for all $t \in Inst^{\mathfrak{c}}(pat(G))$, $t \in L(A)$ if and only if $\varphi(t) \in L(A'')$.*

Proof: We construct φ as the function that transforms an element of $t \in Inst^{\mathfrak{c}}(pat(G))$ satisfying the constraints in $\mathfrak{c}$ to an element of $t' \in Inst(pat(G''))$ in which all the constraints in $\mathfrak{c}$ are satisfied – modulo the change of signature from Σ to Σ' –, thus implying that $t' \in L(B') \cap L(C')$. We first introduce a function ann_ν for all variable $\nu \in \mathcal{V}$, such that for all tree variable x, n-ary function symbol f and trees $t_1, \ldots, t_n$ where $n \geq 0$:

$$ann_\nu(x) = hole_\nu(x)$$
$$ann_\nu(f(t_1, \ldots, t_n)) = (f, \nu)(ann_\nu(t_1), \ldots, ann_\nu(t_n))$$

Then we define φ so that for all well-typed substitution $\mu : fv(G) \to \mathcal{P}^{gr}_\Sigma$, the image of the grounding $p = norm_\beta(\mu(pat(G)))$ is such that

- every subterm of p obtained by instantiating some tree variable x of G is replaced by $root_x(ann_x(\mu(x)))$
- every subterm of p obtained by instantiating some context variable X is replaced by $root_X(ann_X(t))$, where $\mu(X) = \lambda x_{\mathfrak{c}}.t$

For example, if we set $\Sigma = \{f^{(2)}, a^{(0)}, b^{(0)}\}$, $G = (\{z, x, X\}, \Sigma, \{z \to f(x, X@b)\}, z)$, $\mu(x) = a$ and $\mu(X) = \lambda x_{\mathfrak{c}}.f(a, x_{\mathfrak{c}})$, then the pattern in Figure 6.12 gives the value of $\varphi(norm_\beta(\mu(pat(G))))$.

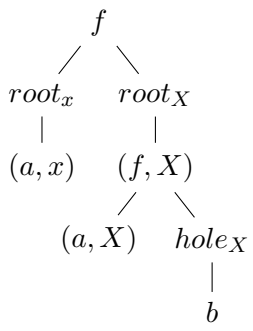


Figure 6.12: Example of image value by φ

Furthermore, for all $t \in Inst^{\mathfrak{c}}(pat(G))$, $t \in L(A)$ if and only if $\varphi(t) \in L(A'')$. □

Using Claim 12, we show that one can build an automaton D (resp. D') with signature Σ' such that $Inst^{\mathfrak{c}}(pat(G)) \cap L(A) \neq \emptyset$ (resp. $Inst^{\mathfrak{c}}(pat(G)) \subseteq L(A)$) if and only if $Inst(pat(G'')) \cap L(D) \neq \emptyset$ (resp. $Inst(pat(G'')) \subseteq L(D')$). This allows to reduce uniform regular matching (resp. inclusion) with constraints to uniform regular matching (resp. inclusion).

Claim 13 cMatch$(\mathcal{G}, \mathcal{A})$ *is reducible in polynomial time to* Match$(\mathcal{G}, \mathcal{A})$.

Proof: Let $t \in Inst^{\mathfrak{c}}(pat(G))$ be a constrained instance of G by $\mathfrak{c}$. By Claim 12, $t \in L(A)$ iff $\varphi(t) \in L(A'')$. Given that $\varphi(t) \in Inst(pat(G'')) \cap L(B') \cap L(C')$, it follows that $Inst^{\mathfrak{c}}(pat(G)) \cap L(A) \neq \emptyset$ iff $Inst(pat(G'')) \cap (L(B') \cap L(C') \cap L(A'')) \neq \emptyset$. One can compute a product automaton D in polynomial time from A'' and B' and C' so that $L(D) = L(A'') \cap L(B') \cap L(C')$. Furthermore, if A'' is deterministic, then D is also deterministic – knowing that B' and C' are deterministic. Thus $Inst^{\mathfrak{c}}(pat(G)) \cap L(A) \neq \emptyset$ iff $Inst(pat(G'')) \cap L(D) \neq \emptyset$, hence cMatch$(\mathcal{G}, \mathcal{A})$ is reducible in polynomial time to Match$(\mathcal{G}, \mathcal{A})$. □

Claim 14 cIncl$(\mathcal{G}, \mathcal{A})$ *is reducible in polynomial time to* Incl$(\mathcal{G}, \mathcal{A})$.

Proof: Let $t \in Inst^{\mathfrak{c}}(pat(G))$ be a constrained instance of G by $\mathfrak{c}$. By Claim 12, $t \in L(A)$ iff $\varphi(t) \in L(A'')$. Given that $\varphi(t) \in Inst(pat(G')) \cap L(B') \cap L(C')$, it follows that $Inst^{\mathfrak{c}}(pat(G)) \subseteq L(A)$ iff $Inst(pat(G'')) \cap L(B') \cap L(C') \subseteq L(A'')$. Let the product automaton $B' \times C'$ recognizing the language $L(B') \cap L(C')$. Then $Inst^{\mathfrak{c}}(pat(G)) \subseteq L(A)$ iff $Inst(pat(G'')) \cap L(B' \times C') \subseteq L(A'')$, that is $Inst(pat(G'')) \subseteq L(A'') \cup L(\overline{B' \times C'})$ where $\overline{B' \times C'}$ is the automaton recognizing the complement of $L(B' \times C')$. The Dta $B' \times C'$ can be complemented in linear time to obtain $\overline{B' \times C'}$, since it is deterministic. Moreover a product automaton

Hedge patterns	$H, H' \in \mathcal{P}^h_\Gamma \quad ::= \quad Y \mid \langle aH \rangle \mid \varepsilon \mid Z \mid HH'$
Encoding	$[\![\varepsilon]\!]^{context} = \lambda y.y, \quad [\![H]\!]^{tree} = [\![H]\!]^{context}@\#,$ $[\![Y]\!]^{context} = Y, \quad [\![Z]\!]^{context} = Z,$ $[\![\langle aH \rangle]\!]^{context} = \lambda y.a([\![H]\!]^{context}@\#, y),$ $[\![HH']\!]^{context} = \lambda y.([\![H]\!]^{context}@([\![H']\!]^{context}@y)).$

Figure 6.13: Encoding of a hedge pattern $H \in \mathcal{P}^h_\Gamma$ into a context pattern $[\![H]\!]^{context} \in \mathcal{P}^{context}_\Sigma$, where $Y \in \mathcal{V}^u$, $Z \in \mathcal{V}^h$, $a \in \Gamma$, and ε is the empty word.

D' recognizing $L(A'') \cup L(\overline{B' \times C'})$ can be built in polynomial time from A'' and $B' \times C'$, so that D' is deterministic if A'' is deterministic. Thus $Inst^c(pat(G)) \subseteq L(A)$ iff $Inst(pat(G')) \subseteq L(D')$, hence cIncl$(\mathcal{G}, \mathcal{A})$ is reducible in polynomial time to Incl$(\mathcal{G}, \mathcal{A})$. □

The Proposition thus follows from Claims 13 and 14. □

6.7 Encoding Patterns for Unranked Trees

The original motivation of the present work was to understand the problems of regular matching and inclusion for hyperstreams of hedges. We next show that these problems can be solved using reductions to the corresponding problems of (ranked) tree patterns with context variables.

We assume a set of variables for unranked trees $Y \in \mathcal{V}^u$ and a set of hedge variables $Z \in \mathcal{V}^h$. The set of hedge patterns $H \in \mathcal{P}^h_\Gamma$ with these two types of variables is then defined by the abstract syntax in Figure 6.13. The set $\mathcal{P}^u_\Gamma$ of *patterns for unranked trees* is the subset of hedge patterns of the forms $\langle aH \rangle$ or $Y \in \mathcal{V}^u$. A well-typed variable assignment $\sigma : V \to \mathcal{H}_\Gamma$ where $V \subseteq \mathcal{V}^u \uplus \mathcal{V}^h$ is a function that maps variables from $\mathcal{V}^u$ to unranked trees in $\mathcal{U}_\Gamma$ and variables from $\mathcal{V}^h$ to hedges in $\mathcal{H}_\Gamma$. The application $\sigma(H)$ is the hedge obtained from H by replacing all variables Y by the unranked tree $\sigma(Y)$ and all variables Z by the hedge $\sigma(Z)$. The instance set of H is denoted $Inst(H) = \{\sigma(H) \mid \sigma : fv(H) \to \mathcal{H}_\Gamma \text{ well-typed}\}$. Note that $Inst(H) \subseteq \mathcal{U}_\Gamma$ for any unranked tree pattern $H \in \mathcal{P}^u_\Gamma$.

We next show in Fig. 6.13 how to encode hedge patterns into (ranked) context patterns over a ranked signature $\Sigma = \Sigma^{(2)} \uplus \Sigma^{(0)}$ where $\Sigma^{(2)} = \Gamma$, $\Sigma^{(0)} = \{\#\}$ and $\#$ is a fresh symbol not in Γ. Our encoding is an extension of the *first-child-next-sibling* encoding [Comon *et al.* 2007]. For instance, the hedge pattern $H_0 = \langle aZbcY \rangle$ is encoded into the context pattern $[\![H_0]\!]^{context} = \lambda y.a(Z@(b(\#, c(\#, Y@\#))), y)$. The concatenation operation on hedges is simulated by the application operation of contexts. The set of context variables used in the encoding is $\mathcal{V}^{context} = \mathcal{V}^u \uplus \mathcal{V}^h$. Finally, we define for any unranked tree $H \in \mathcal{P}^u_\Gamma$ its encoding as a tree pattern

$[\![H]\!]^{tree} \in \mathcal{P}_\Sigma^{tree}$ by $[\![H]\!]^{tree} = [\![H]\!]^{context}@\#$.

In order to show the soundness of this encoding (Lemma 19 below), we need to restrict the instantiation operation. Intuitively, we cannot allow arbitrary substitutions to be applied to $[\![H]\!]^{tree}$ because then the resulting tree pattern might not be a correct encoding of an unranked tree. A variable assignment $\sigma : V \to Val_\Sigma$ is called *unranked* if it maps unranked tree variables to $[\![\mathcal{U}_\Gamma]\!]^{context}$ and hedge variables to $[\![\mathcal{H}_\Gamma]\!]^{context}$. The *unranked-restricted instance set* of a tree pattern p is defined by $Inst^{unr}(p) = \{norm_\beta(\sigma(p)) \mid \sigma : fv(p) \to Val_\Sigma \text{ well-typed and unranked}\}$.

Lemma 19 *$norm_\beta([\![Inst(H)]\!]^{tree}) = Inst^{unr}([\![H]\!]^{tree})$ for any $H \in \mathcal{P}_\Gamma^u$.*

Proof: Let $H \in \mathcal{P}_\Gamma^u$ be an unranked tree pattern. The proof is by induction on the structure of H.

Case $H = \langle a \rangle$ where $a \in \Gamma$. Then the following equalities $Inst(H) = \{\langle a \rangle\}$ and $[\![Inst(H)]\!]^{tree} = \{\lambda y.a((\lambda y.y)@\#, y)@\#\}$ hold. This implies that $norm_\beta([\![Inst(H)]\!]^{tree}) = \{a(\#, \#)\} = Inst^{unr}([\![H]\!]^{tree})$, since H contains no variable to instantiate.

Case $H = Y \in \mathcal{V}^u$. Then $Inst(H) = \{a(H') \mid a \in \Gamma \text{ and } H' \in \mathcal{H}_\Gamma\}$ and $[\![Inst(H)]\!]^{tree} = \{(\lambda y.a([\![H']\!]^{tree}, y))@\# \mid a \in \Gamma \text{ and } H' \in \mathcal{H}_\Gamma\}$. This implies that $norm_\beta([\![Inst(H)]\!]^{tree}) = \{a(norm_\beta([\![H']\!]^{tree}), \#) \mid a \in \Gamma \text{ and } H' \in \mathcal{H}_\Gamma\} = Inst^{unr}([\![H]\!]^{tree})$ since no unranked tree $\langle aH' \rangle \in Inst(H)$ contains a variable to instantiate.

Case $H = \langle bH' \rangle$ where $b \in \Gamma$ and $H' \in \mathcal{P}_\Gamma^h$. Then $Inst(H) = \{\langle bH'' \rangle \mid H'' \in Inst(H')\}$ and $[\![Inst(H)]\!]^{tree} = \{(\lambda y.b([\![H'']\!]^{tree}, y))@\# \mid H'' \in Inst(H')\}$. So $norm_\beta([\![Inst(H)]\!]^{tree}) = \{b(norm_\beta([\![H'']\!]^{tree}), \#) \mid H'' \in Inst(H')\}$. By the induction hypothesis, $norm_\beta([\![Inst(H')]\!]^{tree}) = Inst^{unr}([\![H']\!]^{tree})$, which implies that $norm_\beta([\![Inst(H)]\!]^{tree}) = \{b(t, \#) \mid t \in Inst^{unr}([\![H']\!]^{tree})\} = Inst^{unr}([\![H]\!]^{tree})$.

□

Let $\Sigma = \Sigma^{(2)} \cup \Sigma^{(0)}$ be a ranked signature constituted of binary symbols taken from an alphabet Γ and a constant $\#$, that is $\Sigma^{(2)} = \Gamma$, $\Sigma^{(0)} = \{\#\}$ and $\# \notin \Gamma$. Let $\mathcal{P}_\Gamma^{comp,u} \subseteq Hyp_\Gamma$ be the set of hyperstreams of unranked trees over Γ. For a class of automata $\mathcal{A} \in \{\text{DTA}, \text{NTA}\}$ we define the problems of regular matching and inclusion of hyperstreams of unranked trees:

Unranked regular matching: $\textsc{Match}_\Gamma(\mathcal{P}^{comp,u}, \mathcal{A})$.
Input: a hyperstream of unranked tree $H \in \mathcal{P}_\Gamma^{comp,u}$ and an automaton $A \in \mathcal{A}_\Sigma$
Output: whether $Inst^{unr}([\![H]\!]^{tree}) \cap L(A) \neq \emptyset$.

Unranked regular inclusion: $\textsc{Incl}_\Gamma(\mathcal{P}^{comp,u}, \mathcal{A})$.
Input: a hyperstream of unranked tree $H \in \mathcal{P}_\Gamma^{comp,u}$ and an automaton $A \in \mathcal{A}_\Sigma$
Output: whether $Inst^{unr}([\![H]\!]^{tree}) \subseteq L(A)$.

The uniform versions of these problems where the signature Γ is given with the input are called $\textsc{Match}(\mathcal{P}^{comp,u}, \mathcal{A})$ and respectively $\textsc{Incl}(\mathcal{P}^{comp,u}, \mathcal{A})$. Note that using tree automata in the above definitions is not a restriction, as it is well known [Comon *et al.* 2007] that for any unranked tree language L recognizable by a hedge automaton, there exists a tree automaton that recognizes the first-child-next-sibling encoding of the trees in L.

Proposition 20 *For any class of automata $\mathcal{A} \in \{\textsc{Dta}, \textsc{Nta}\}$ there exist reductions in polynomial time from $\textsc{Match}(\mathcal{P}^{comp,u}, \mathcal{A})$ to $\textsc{Match}(\mathcal{P}^{comp,tree}, \mathcal{A})$ and from $\textsc{Incl}(\mathcal{P}^{comp,u}, \mathcal{A})$ to $\textsc{Incl}(\mathcal{P}^{comp,tree}, \mathcal{A})$.*

Proof: Let Γ be an alphabet, $\Sigma = \Sigma^{(2)} \cup \Sigma^{(0)}$ a ranked signature constituted of binary symbols taken from Γ and a constant $\#$, that is $\Sigma^{(2)} = \Gamma$, $\Sigma^{(0)} = \{\#\}$ and $\# \notin \Gamma$. Let $H \in \mathcal{P}_\Gamma^{comp,u}$ be a hyperstream of unranked tree, $\mathcal{A}$ a class of automata and $A \in \mathcal{A}_\Sigma$ a tree automaton. Thanks to Lemma 19, $norm_\beta([\![Inst(H)]\!]^{tree}) = Inst^{unr}([\![H]\!]^{tree})$, and thus deciding whether $norm_\beta([\![Inst(H)]\!]^{tree}) \cap L(A) \neq \emptyset$ is equivalent to deciding whether $Inst^{unr}([\![H]\!]^{tree}) \cap L(A) \neq \emptyset$. Notice that *unr* is actually an instantiation constraint. It associates every free tree variable with the universal Dta over Σ. Context variables are mapped to the Dta that recognizes all the trees over $\Sigma \uplus \{y\}$ having only one occurrence of y, which is furthermore either the only node of the tree, or the second son of its parent, as enforced by the encoding. We have thus reduced the problem of regular matching of hyperstreams of unranked trees to the problem of regular matching with constraints – on ranked patterns – in polynomial time. Then the regular matching problem with constraints is reduced to uniform regular matching using Proposition 19. We use an analogous procedure for the inclusion problem. □

Theorem 7 *For any alphabet Γ having at least two symbols, the problems*

$\textsc{Match}_\Gamma(\mathcal{P}^{comp,u}, \textsc{Dta})$ *and* $\textsc{Incl}_\Gamma(\mathcal{P}^{comp,u}, \textsc{Dta})$ *are* Pspace*-complete while* $\textsc{Match}_\Gamma(\mathcal{P}^{comp,u}, \textsc{Nta})$ *and* $\textsc{Incl}_\Gamma(\mathcal{P}^{comp,u}, \textsc{Nta})$ *are* Exp*-complete.*

Proof: The upper bounds follow via the polynomial time reduction from Proposition 20 and the complexities in Proposition 18. The lower bounds can be obtained by reducing the equivalent problems on ranked patterns to the version on hyperstreams of unranked trees, and further using the results in Propositions 16 and 17. □

6.8 Linearity Restriction

We now study the complexity of regular matching and inclusion for the class $Lin\mathcal{P}^{comp,tree}$ that maps ranked signatures Σ to the set of linear compressed tree patterns $Lin\mathcal{P}^{comp,tree}_\Sigma$.

Proposition 21 $\textsc{Match}(Lin\mathcal{P}^{comp,tree}, \textsc{Nta})$ *is in* PTime.

Proof: Let Σ be a ranked signature, $G = (N, \Sigma, R, \mathcal{S}) \in Lin\mathcal{P}^{comp,tree}_\Sigma$ a linear compressed tree pattern and $A = (Q, \Sigma, F, \Delta)$ an Nta. Given that the instance set of the linear pattern $pat(G)$ is regular, one could think of building an Nta that recognizes $Inst(pat(G))$, but since $pat(G)$ may be exponential in the size of G, this approach does not work in polynomial time.

Instead we evaluate the pattern G directly in the Σ-algebra Δ, while mapping context variables to the accessibility relation of Δ. So let $acc_\Delta \colon Q \to 2^Q$ be the function that maps every $q \in Q$ to the set of states accessible from state q with respect to Δ. We consider the well-typed assignment s that maps all tree variables x in $fv(G)$ to $s(x) = Q$ and all context variables $X \in fv(G)$ to $s(X) = acc_\Delta$. The following then holds:

Claim 15 $Inst(pat(G)) \cap L(A) \neq \emptyset$ *if and only if* $[\![pat(G)]\!]^{\Delta,\widehat{s}} \cap F \neq \emptyset$.

Proof: For the forward direction, assume $Inst(pat(G)) \cap L(A) \neq \emptyset$. According to Lemma 18, there exists a well-typed assignment $\sigma : fv(G) \to [\![Val_\Sigma]\!]^\Delta$ such that $[\![pat(G)]\!]^{\Delta,\sigma} \cap F \neq \emptyset$. For all tree variable $x \in fv(G)$ (resp. context variable $X \in fv(G)$), the construction of s guarantees that $\sigma(x) \subseteq s(x) = Q$ (resp. for all $q \in Q$, $\sigma(X)(q) \subseteq s(X)(q) = acc_\Delta(q)$). This implies that $[\![pat(G)]\!]^{\Delta,\hat{s}} \cap F \neq \emptyset$ too.

For the inverse direction, let $p_\mathcal{S} \in \mathcal{P}^{tree}_\Sigma$ be such that $R(\mathcal{S}) = p_\mathcal{S}$ in G and assume $[\![pat(G)]\!]^{\Delta,\widehat{\sigma}} \cap F \neq \emptyset$. We prove the property by induction on the structure of $p_\mathcal{S}$.

- Case $p_\mathcal{S} = x \in \mathcal{V}$. Then $[\![pat(G)]\!]^{\Delta,\widehat{s}} = s(x) = Q$. Since A is reduced and that all the states of the Nta are accessible, it holds that for all $q \in [\![pat(G)]\!]^{\Delta,\widehat{s}}$

there exists a tree $t \in \mathcal{T}_\Sigma$ such that $q \in [\![t]\!]^\Delta$. Let $q_f \in [\![pat(G)]\!]^{\Delta,\widehat{s}} \cap F$. There exists a tree $t_f \in L(A)$ such that $q_f \in [\![t_f]\!]^\Delta$, hence $t_f \in Inst(pat(G)) \cap L(A)$.

- Case $p_{\mathcal{S}} = X@x$. We have $[\![pat(G)]\!]^{\Delta,\widehat{s}} = \widehat{s(X)}(s(x))$. Let the state $q_f \in \widehat{s(X)}(s(x)) \cap F$ be in the intersection of $[\![pat(G)]\!]^{\Delta,\widehat{s}}$ and F. Since $s(X) = acc_\Delta$, there is a state $q_r \in Q$ such that $q_f \in acc_\Delta(q_r)$, and thus a context $\lambda x.p_f \in \mathcal{C}_\Sigma$ such that $q_f \in [\![\lambda x.p_f]\!]^\Delta(\{q_r\})$. Furthermore, $q_r \in s(x) = Q$ is an accessible state of A, and so there is a tree t_r such that $q_r \in [\![t_r]\!]^\Delta$. Notice that $q_f \in [\![(\lambda x.p_f)@t_r]\!]^\Delta = [\![\lambda x.p_f]\!]^\Delta(\{q_r\})$, and thus $(\lambda x.p_f)@t_r \in Inst(pat(G)) \cap L(A)$.

- The cases $p_{\mathcal{S}} = t$ and $p_{\mathcal{S}} = X@t$ where $t \in \mathcal{T}_\Sigma$ and $X \in \mathcal{V}^{context}$ are respectively special instances of the first and second cases.

- Case $p_{\mathcal{S}} = f(\mathcal{S}_1, \ldots, \mathcal{S}_n)$ where $f \in \Sigma^{(n)}$, $\mathcal{S}_1, \ldots, \mathcal{S}_n \in N \setminus fv(G)$ are starting symbols for some linear compressed tree patterns $G_1, \ldots, G_n \in Lin\mathcal{P}^{comp,tree}$ and for all different $i, j \in \{1, \ldots, n\}$, $fv(G_i) \cap fv(G_j) \neq \emptyset$. Here we have assumed without loss of generality that any compressed tree pattern is built only from smaller compressed tree patterns. Thus if there were some constant symbol or free variable v occurring in $p_{\mathcal{S}}$, one could just create a new compressed tree pattern G' from G where the occurrences of v in $\mathcal{S}$ are replaced by a new nonterminal $\mathcal{S}_v$, and with the additional rule $\mathcal{S}_v \to v$. But for the sake of simplicity, we suppose that G is already in the form we want it to be. Thus every $\mathcal{S}_i$ can be considered as the start symbol of the compressed tree pattern G_i. We have that $[\![pat(G)]\!]^{\Delta,\widehat{s}} = [\![f(pat(G_1), \ldots, pat(G_n))]\!]^{\Delta,\widehat{s}} = \{q \mid \exists q_1 \in [\![pat(G_1)]\!]^{\Delta,\widehat{s}}, \ldots, \exists q_n \in [\![pat(G_n)]\!]^{\Delta,\widehat{s}}.\ f(q_1, \ldots, q_n) \to q \text{ in } \Delta\}$. Let $q_f \in [\![pat(G)]\!]^{\Delta,\widehat{s}} \cap F \neq \emptyset$. Then by the induction hypothesis, there exists $t_1 \in Inst(pat(G_1)), \ldots, t_n \in Inst(pat(G_n))$ such that $q_f \in [\![f(t_1, \ldots, t_n)]\!]^{\Delta,\hat{s}}$, and thus $f(t_1, \ldots, t_n) \in Inst(pat(G)) \cap L(A)$. Hence the property holds.

□

Thanks to Claim 15, one can simply test $[\![pat(G)]\!]^{\Delta,\widehat{s}} \cap F \neq \emptyset$ in order to decide whether $Inst(pat(G)) \cap L(A) \neq \emptyset$. By Lemma 15, it takes polynomial time in the sizes of Δ, G and s to compute $[\![pat(G)]\!]^{\Delta,\widehat{s}}$. It follows that the problem $\textsc{Match}_\Sigma(Lin\mathcal{P}^{comp,tree}, \textsc{Nta})$ is in PTime. □

We next consider regular inclusion for linear tree patterns. Proposition 21 and the duality via complementation (Lemma 16) yield for Dtas that regular inclusion for linear patterns is in PTime too. So it remains to consider the case of regular inclusion for Ntas. By Lemma 17, this problem is Exp-hard even without context variables and without compression. Therefore regular inclusion for Ntas and (compressed) linear patterns with or without context variables is Exp-complete.

Chapter 7

Approximating CQA on Hyperstreams

Contents

The results in Chapter 6 show that CQA is a very hard problem in general, requiring most of the time an exponential time. They do not provide solutions but they indicate the aspects that make the problem hard.

In this chapter, we introduce new approximations of CQA and CQNA for which the complexity is smaller. Unlike Chapter 6, we will use stepwise hedge automata to represent languages of hedges. We also restrict the approximations to the simpler case of boolean queries, to which all the other complex cases can be reduced.

Our study starts by defining transitions for SHAs. Chapter 6 has shown that they are a central point in the CQA problem. Afterwards, we define safety approximations. These functions are member of a special monoid, and we use them to define our approximations of CQA. Finally we illustrate some approximations of CQA, and show that the choice of the safety approximation impact on their quality.

7.1 Transitions for SHAs

As for NTAs, we introduce transitions for SHAs. Let Δ be a transition relation with hedge states Q_h and signature Σ. A transition τ is a set of pairs of states. The

transition of a nested word $w \in nWords_\Sigma$ with respect to Δ is the set of state pairs $(q, q') \in Q_h \times Q_h$ such that $w \in L_{q,q'}(\Delta)$.

While the set of nested words is infinite, the set of transitions is finite. The most basic technique of our algorithms will be to replace variables in hyperstreams by transitions and then evaluate the hyperstream in the transition monoid of a SHA, the prime example of a nesting monoid. Proposition 22 will show that this can always be done in polynomial time even for hyperstreams with compression.

Definition 29 *A* nesting monoid *is an algebra $(M, \cdot, e, \nu)$ such that $(M, \cdot, e)$ is a monoid (a set M with an associative operator $\cdot : M \times M \to M$ with neutral element $e \in M$) and $\nu : M \to M$ a function called the nesting operator.*

Most typically, the set of nested words $nWords_\Sigma$ is a nesting monoid, where $\cdot$ is the concatenation and e the empty word, and $\nu(w) = \langle w \rangle$ the nesting operator. As a consequence, the set of nested patterns $nPatterns_\Sigma$ is equally a nesting monoid.

Let $\mathcal{M} = (M, \cdot, e, \nu)$ and $\mathcal{M}' = (M', \cdot', e', \nu')$ be two nesting monoids. A *morphism* from $\mathcal{M}$ to $\mathcal{M}'$ is a mapping $\mathfrak{m} : M \to M'$ such that $\mathfrak{m}(m_1 \cdot m_2) = \mathfrak{m}(m_1) \cdot' \mathfrak{m}(m_2)$ for all $m_1, m_2 \in M$, $\mathfrak{m}(e) = e'$, and $\mathfrak{m}(\nu(m)) = \nu'(\mathfrak{m}(m))$. Any function $\mathfrak{f} : \Sigma \to M$ can be lifted to a morphism on nesting monoids $eval_\mathfrak{f} : nWords_\Sigma \to M$ such that for all $w, w' \in nWords_\Sigma$ and $a \in \Sigma$:

$$\begin{aligned} &eval_\mathfrak{f}(\varepsilon) = e, && eval_\mathfrak{f}(a) = \mathfrak{f}(a), \\ &eval_\mathfrak{f}(ww') = eval_\mathfrak{f}(w) \cdot eval_\mathfrak{f}(w') && eval_\mathfrak{f}(\langle w \rangle) = \nu(eval_\mathfrak{f}(w)) \end{aligned}$$

Furthermore, for any substitution $\sigma : \mathcal{V} \to M$ and morphism between nesting monoids $\mathfrak{m} : nWords_\Sigma \to M$, there exists a unique morphism between nesting monoids $[\![\cdot]\!]^{\sigma,\mathfrak{m}} : nPatterns_\Sigma \to M$ such that $[\![X]\!]^{\sigma,\mathfrak{m}} = \sigma(X)$ for all $X \in \mathcal{V}$ and $[\![w]\!]^{\sigma,\mathfrak{m}} = \mathfrak{m}(w)$ for all $w \in nWords_\Sigma$.

The set of all transitions $\mathcal{T}_Q = 2^{Q_h \times Q_h}$ forms a monoid whose neutral element is the identity transition $\{(q, q) \mid q \in Q_h\}$ and whose composition operator $\circ$ is the composition operator of binary relations on Q_h. Let Δ be the transition function of a SHA with hedge states Q_h, tree states Q_t and alphabet Σ. We can now turn $\mathcal{T}_Q$ into a nesting monoid $\mathcal{T}_{Q_h,\Delta} = (\mathcal{T}_Q, \circ, e, \nu^\Delta)$ such that for all $\tau \in \mathcal{T}_Q$:

$$\nu^\Delta(\tau) = \{(q_1, q_2) \mid \exists (q, q') \in \tau.\ q \in \langle\rangle^\Delta,\ q' \in Q_t,\ q_1 \xrightarrow{q'} q_2 \in \Delta\}.$$

The transition $trans^\Delta(w)$ of a nested word $w \in nWords_\Sigma$ is the following element of $\mathcal{T}_Q$:

$$trans^\Delta(w) = \{(q, q') \mid w \in L_{q,q'}(\Delta)\}.$$

Note also that $trans^{\Delta}$ is a morphism between the nesting monoids $nWords_{\Sigma}$ and $\mathcal{T}_{Q_h,\Delta}$. We next show that nested hyperstreams can be evaluated in the transition monoid $\mathcal{T}_{Q_h,\Delta}$ in polynomial time.

Proposition 22 *For any nested hyperstream $G \in Hyp_{\Sigma}$ and substitution $\sigma : fv(G) \to \mathcal{T}_Q$, the transition $[\![pat(G)]\!]^{\sigma,trans^{\Delta}}$ can be computed in polynomial time in $|G|$ and $|\delta|$.*

Note that $pat(G)$ may be of exponential size in $|G|$ if G is not compression-free. Therefore, we have to avoid computing $pat(G)$ to prove the proposition in the general case. This can be done by evaluating G along its DAG structure.
Proof: We can represent transitions as boolean $|Q_h| \times |Q_h|$ matrices and then perform all operation of the nesting monoid $\mathcal{T}_{Q_h,\Delta}$ in polynomial time. In particular we can compute $\tau \circ \tau'$ by using boolean matrix multiplication. The proof is straightforward by induction on the acyclic structure of G. □

7.2 Eliminating Hard Constraints: Linear Certainty

We fix an alphabet Σ for the rest of this chapter.
We saw in Chapter 6 that for hyperstreams, the most influential factor in the complexity of regular matching and inclusion is nonlinearity. That brings us to consider a new notion of certainty, where the nonlinearities are simply ignored.

For any nested pattern ρ, we define $lin(\rho)$ as the nested pattern that is equal to ρ except that all the occurrences of variables in ρ that are not first occurrences are replaced by fresh variables. This way, $lin(\rho)$ is always a linear nested pattern. For instance, for $\rho = aX_1\langle bX_2\rangle cX_1aX_2$ we have $lin(\rho) = aX_1\langle bX_2\rangle cX_1'aX_2'$ where X_1' and X_2' are the fresh pattern variables replacing the second occurrences of X_1 respectively X_2.

Definition 30 *Let $\boldsymbol{Q}$ be a boolean query with alphabet Σ. We call the nested hyperstream G a* linearly certain query answer (resp. non-answer) *for query $\boldsymbol{Q}$ if $lin(pat(G))$ is a certain query answer (resp. non-answer) for query $\boldsymbol{Q}$.*

By abuse of language, we say that a nested word w is a certain answer for a boolean query $\mathbf{Q}$ whenever the empty candidate on w is a certain answer for $\mathbf{Q}$. The next proposition relates linearly certain to certain answers.

Proposition 23 (Soundness) *If G is a linearly certain answer (resp. non-answer) for query $\boldsymbol{Q}$, then G is a certain answer (resp. non-answer) for query $\boldsymbol{Q}$.*

Proof: Let $\rho = pat(G)$. Clearly, $Inst(\rho) \subseteq Inst(lin(\rho))$. If G is a linearly certain answer of $\mathbf{Q}$, then $Inst(lin(\rho)) \cap L(\mathbf{Q}) = \emptyset$ and it follows that $Inst(\rho) \cap L(\mathbf{Q}) = \emptyset$ and so G is a certain non-answer of $\mathbf{Q}$. If G is a linearly certain answer of $\mathbf{Q}$, then $Inst(lin(\rho)) \subseteq L(\mathbf{Q})$, thus $Inst(\rho) \subseteq L(\mathbf{Q})$, hence G is a certain answer of $\mathbf{Q}$. □

Now define the linear certainty problem.

Definition 31 (Linear Certainty) *For any class $\mathcal{A}$ of SHAs and any alphabet Σ, we define the following decision problems:*

LinCert$^{ans}_{\Sigma}(\mathcal{A})$.
Input: A nested hyperstream $G \in Hyp_{\Sigma}$ and a SHA $S \in \mathcal{A}_{\Sigma}$.
Output: The truth value of whether G is a linearly certain query answer for $L(S)$.

LinCert$^{\neg ans}_{\Sigma}(\mathcal{A})$.
Input: A nested hyperstream $G \in Hyp_{\Sigma}$ and a SHA $S \in \mathcal{A}_{\Sigma}$.
Output: The truth value of whether G is a linearly certain query non-answer for $L(S)$.

The first positive result is that linear certainty makes the finding of certain non answers easy, no matter if the automaton representing the query is deterministic or not.

Proposition 24 LinCert$^{\neg ans}_{\Sigma}(SHA)$ *is in* PTime.

Proof: Let $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ be a SHA, $G \in Hyp_{\Sigma}$ a hyperstream and σ a substitution that associates to any free variable $X \in fv(G)$ of type *tree* the transition $@^{\Delta}$, while free variables $X \in fv(G)$ of type *hedge* are mapped to the transition $acc^{\Delta}_{h \times h}$. Then the following claim holds:

Claim 16 *G is a linearly certain query non-answer of S if and only if $[\![pat(G)]\!]^{\sigma, trans^{\Delta}} \cap (I \times F) = \emptyset$.*

The above claim can be shown by an induction on the structure of G. According to it, we just have to check $[\![pat(G)]\!]^{\sigma, trans^{\Delta}} \cap (I \times F) = \emptyset$ in order to know whether or not G is a linearly certain query non-answer for $L(S)$. This can be done in polynomial time, using Proposition 22. It follows that LinCert$^{\neg ans}_{\Sigma}$(SHA) is in PTime. □

For certain answers, linear certainty doesn't have a big impact in the case of queries represented by non deterministic SHAs. Indeed, the problem remains as

hard as CQA. This is not surprising, as removing nonlinearities for regular inclusion didn't change the complexity for queries represented by nondeterministic tree automata.

Proposition 25 LinCert$^{ans}_{\Sigma}$(*SHA*) *is* ExpTime*-hard.*

Proof: The lower bound is obtained by reduction from the problem of universality for SHAs, which is Exp-complete by Proposition 3. Indeed, the language of trees of a SHA S with alphabet Σ is universal – that is $L^{tree}(S) = nWords^{tree}_{\Sigma}$ – if and only if the nested pattern X is a linearly certain query answer of $L(S)$, where $type(X) = tree$. Thus LinCert$^{ans}_{\Sigma}$(SHA) is Exp-hard. □

Unlike nondeterministic SHAs, linear certainty for dSHAs is an interesting case in practice. We next show that it's in PTime. This is an interesting result from a practical point of view, given that dSHAs of reasonable size can be obtained for nested regular path queries.

Lemma 20 LinCert$^{ans}_{\Sigma}$(*dSHA*) *is reducible in polynomial time to* LinCert$^{\neg ans}_{\Sigma}$(*dSHA*).

Proof: We complete S in linear time (by adding symbolic rules) to S' and complement S' to $\overline{S'}$ by simply flipping the final states. Then observe that G is a certain query answer for $L(S)$ if and only if it is a certain query non-answer for $L(\overline{S'})$. And given that $\overline{S'}$ can computed in linear time in the size of S, we conclude that LinCert$^{ans}_{\Sigma}$(dSHA) is reducible in PTime to LinCert$^{\neg ans}_{\Sigma}$(dSHA). □

Proposition 26 LinCert$^{ans}_{\Sigma}$(*dSHA*) *is in* PTime.

Proof: Due to determinism, LinCert$^{ans}_{\Sigma}$(dSHA) can be reduced in polynomial time to LinCert$^{\neg ans}_{\Sigma}$(dSHA) by Lemma 20, and this problem is in PTime by Proposition 24. □

7.3 Safety Approximations

Since nested regular path queries are converted to nondeterministic SHAs where the expressions are nondeterministic, and that determinization of SHAs may sometimes blow up, it is relevant to have an efficient decision procedure for LinCert$^{ans}_{\Sigma}$(SHA). Unfortunately, this is the one case of the four – LinCert$^{ans}_{\Sigma}$(SHA), LinCert$^{ans}_{\Sigma}$(dSHA), LinCert$^{\neg ans}_{\Sigma}$(SHA), LinCert$^{\neg ans}_{\Sigma}$(dSHA) – which remains hard. Therefore, we next search for a refinement of linear certainty for dSHA queries that can be decided more efficiently, while further approximating the set of linearly

certain answers. This will lead us to the notion of strong certainty in Section 7.4, which will be based on an approximation of the set of safe states of a SHA that we introduce next.

Let Δ be the transition relation of some SHA with hedge states in Q_h and alphabet Σ. Let $\mathcal{S}_{Q_h} = \{\mathfrak{s} \mid \mathfrak{s} \colon 2^{Q_h} \to 2^{Q_h}\}$. We define the function $safe^{\Delta} \in \mathcal{S}_{Q_h}$ so that for all subset $Q' \subseteq Q_h$ of hedge states, the set of safe states $safe^{\Delta}(Q')$ contains all those states from which Δ may always reach some state in Q' when reading any nested word:

$$safe^{\Delta}(Q') = \{q \mid \forall w \in nWords_{\Sigma}.\ \exists q' \in Q'.\ w \in L_{q,q'}(\Delta)\}.$$

Definition 32 *We call an element $\mathfrak{s} \in \mathcal{S}_{Q_h}$ a* safety approximation *for Δ if $\mathfrak{s}(Q') \subseteq safe^{\Delta}(Q')$ for all subsets $Q' \subseteq Q_h$.*

We next show that one can decide the universality of SHAs using safety approximations.

Lemma 21 *If $\mathfrak{s}$ is a safety approximation for Δ and $I, F \subseteq Q_h$ such that $I \cap \mathfrak{s}(F) \neq \emptyset$, then any SHA $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ satisfies $L(S) = nWords_{\Sigma}$.*

Proof: By assumption I is non-disjoint from $\mathfrak{s}(F)$. Since $\mathfrak{s}$ is a safety approximation, it follows that I is non-disjoint from $safe^{\Delta}(F)$. Hence there exists a state $q \in I$ such that $q \in safe^{\Delta}(F)$. Let $w \in nWords_{\Sigma}$ be a nested word. We have to show that $w \in L(S)$. Since $q \in safe^{\Delta}(F)$ there exists a state $q' \in F$ such that $w \in L_{q,q'}(\Delta)$. Since $q \in I$ and $q' \in F$, it follows that $w \in L(S)$. □

7.3.1 Safety Approximation by Accessibility

We next propose two concrete safety approximations. The first will be based on the accessibility relation computed from a transition relation Δ.

Lemma 22 *$acc^{\Delta}_{h \times h}$ can be computed in polynomial time from Δ.*

We define the function $safe\text{-}acc^{\Delta} \in \mathcal{S}_{Q_h}$ such that for all subset of hedge states $Q' \subseteq Q_h$,

$$safe\text{-}acc^{\Delta}(Q') = \{q \mid \forall q' \in Q_h.\ (q, q') \in acc^{\Delta}_{h \times h} \Rightarrow q' \in Q'\}$$

Our next objective is to show that $safe\text{-}acc^{\Delta}$ is a safety approximation for Δ under the condition that Δ is pseudo-complete (see Definition 13).

Proposition 27 *If Δ is pseudo-complete, then $safe\text{-}acc^{\Delta}$ is a safety approximation for Δ.*

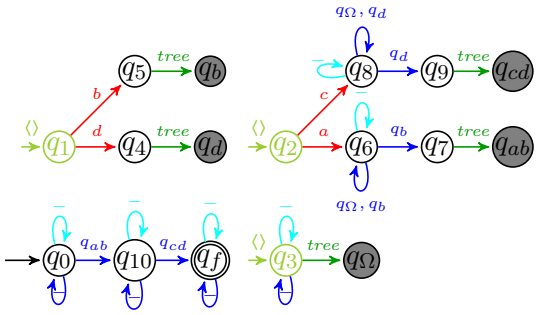


Figure 7.1: SHA with transition function Δ

Proof: Let $Q' \subseteq Q_h$ be a subset of states. We show that $\mathit{safe\text{-}acc}^\Delta(Q') \subseteq \mathit{safe}^\Delta(Q')$. Let $q \in \mathit{safe\text{-}acc}^\Delta(Q')$. By definition of safe^Δ, we have to show for all $w \in \mathit{nWords}_\Sigma$ that there exists $q' \in Q'$ such that $w \in L_{q,q'}(\Delta)$. So let $w \in \mathit{nWords}_\Sigma$ be an arbitrary nested word. Since Δ is pseudo-complete, there exists a state $q' \in Q_h$ such that $w \in L_{q,q'}(\Delta)$. Hence, $(q, q') \in \mathit{acc}^\Delta_{h\times h}$. Since $q \in \mathit{safe\text{-}acc}^\Delta(Q')$ this implies that $q' \in Q'$ as required. □

7.3.2 Safety Approximation by Accessibility and Self Loops

The second approximation will be based on the notion of self looping states beside of state accessibility.

Definition 33 *A hedge state $q \in Q_h$ is* self-looping by Δ *if:*

- *for all $a \in \Sigma$, $(q, q) \in a^\Delta$ and*
- *either for all accessible tree state $p \in Q_t$ – for which there is a state $q \in \langle\rangle^\Delta$ such that $(q, p) \in \mathit{acc}^\Delta_{h\times t}$ – it holds that $(q, q) \in @^\Delta_p$*
- *or $(q, q) \in @^\Delta_{q_\Omega}$ for all else state $q_\Omega \in \Omega$.*

We denote the set of self-looping states by Δ by $\mathit{self\text{-}looping}(\Delta)$. We now define the transition relation $\mathit{loop}(\Delta)$, which modifies a transition function Δ by removing some transitions of self-looping states to others. We define:

$$\mathit{loop}(\Delta) = \Delta \setminus \{q \xrightarrow{o} q' \in \Delta \mid o \in \Sigma \cup Q_t,\ q \text{ self-looping by } \Delta \text{ and } q' \neq q\}$$

For illustration, the SHA in Figure 7.2 with transition function Δ becomes the SHA in Figure 7.1 with transition function $\mathit{loop}(\Delta)$. The self-looping states of this SHA are q_0, q_6, q_8, q_{10} and q_f.

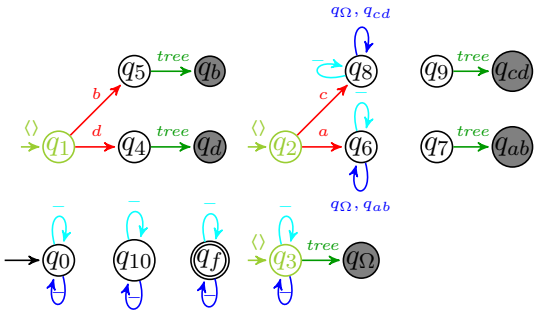


Figure 7.2: SHA with transition function $loop(\Delta)$

Lemma 23 *If Δ is pseudo-complete then safe-acc$^{loop(\Delta)}$ is a safety approximation for Δ.*

Proof: First notice, that $loop(\Delta)$ is pseudo-complete if Δ is, since any run starting in a self-looping state can be continued in the state. Proposition 27 shows that *safe-acc*$^{loop(\Delta)}$ is a safety approximation for $loop(\Delta)$ so that for all subset of hedge states $Q' \subseteq Q_h$:

$$\textit{safe-acc}^{loop(\Delta)}(Q') \subseteq \textit{safe}^{loop(\Delta)}(Q')$$

Since $loop(\Delta)$ is a restriction of Δ it holds that $\textit{safe}^{loop(\Delta)}(Q') \subseteq \textit{safe}^{\Delta}(Q')$. Hence, *safe-acc-loop*$_\Delta$ is a safety approximation for Δ. □

7.4 Strong Certainty

In this section, we show that one can approximate linear certainty using safety approximations. The complexity of these approximations is lower than the general complexity of linear certainty. Furthermore, we show how the choice of a safety approximation impacts the quality of the approximation.

7.4.1 Parameterized Strong Certainty

We now introduce a notion of $\mathfrak{s}$-strong certainty for approximating the set of linearly certain query answers based on an safety approximation $\mathfrak{s}$. Note that $\mathfrak{s}$-strong certainty – in contrast to previous certainty notions – will depend on the automaton defining the query, and not only on the query itself.

For the rest of this section, let Δ be the transition relation of some SHA with alphabet Σ, hedge states Q_h and tree states Q_t. For any nested word $w \in nWords_\Sigma$ we define an inverse transition $itrans^{\Delta}(w) \in \mathcal{S}_{Q_h}$ such that for all subsets of hedge

states $Q' \subseteq Q_h$:

$$itrans^{\Delta}(w)(Q') = \{q \in Q_h \mid \exists q' \in Q'.\ (q, q') \in trans^{\Delta}(w)\}.$$

This set contains all those states from which Q' can be reached over w. Clearly, $itrans^{\Delta}(w)(Q')$ can be computed in polynomial time when given Δ, w, and Q'. We next show that $itrans^{\Delta}$ is a morphism between nesting monoids. For this we first note that $\mathcal{S}_{Q_h}$ is a monoid with function composition as composition operator, and the identity function as the neutral element. Second, we turn $\mathcal{S}_{Q_h}$ into a nesting monoid $\mathcal{S}_{Q_h,\Delta}$ – that we call the *safety monoid*. In order to do so, we define the nesting function ν^{Δ} of the nesting monoid $\mathcal{S}_{Q_h}$ such that for all $\mathfrak{s} \in \mathcal{S}_{Q_h}$ and subset of states $Q' \subseteq Q_h$:

$$\nu^{\Delta}(\mathfrak{s})(Q') = \{q \in Q_h \mid \exists p \in Q_t, q' \in Q'.(q, q') \in @_p^{\Delta} \text{ and } \mathfrak{s}(\chi^{\Delta}(\{p\})) \cap \langle\rangle^{\Delta} \neq \emptyset\}$$

where

$$\forall Q'' \subseteq Q_t.\ \chi^{\Delta}(Q'') = \{q \in Q_h \mid \exists q'' \in Q''.q \xrightarrow{tree} q'' \in \Delta\}.$$

The function χ^{Δ} returns the set of hedge states from which Q' is reachable by a tree transition. It is thus equivalent to $(tree^{\Delta})^{-1}$.

Lemma 24 *$itrans^{\Delta} : nWords_{\Sigma} \to \mathcal{S}_{Q_h,\Delta}$ is a morphism between nesting monoids.*

Proof: □

For any element of the safety monoid $\mathfrak{s}$ we define a constant variable assignment to this element $const_{\mathfrak{s}} : \mathcal{V} \to \mathcal{S}_{Q_h}$, that is $const_{\mathfrak{s}}(X) = \mathfrak{s}$ for all variables X. For any $\rho \in nPatterns_{\Sigma}$, we define:

$$[\![\rho]\!]^{\mathfrak{s},\Delta} = [\![\rho]\!]^{const_{\mathfrak{s}},itrans^{\Delta}}$$

. We first note that $[\![\rho]\!]^{\mathfrak{s},\Delta}$ is preserved by linearisation.

Lemma 25 $[\![\rho]\!]^{\mathfrak{s},\Delta} = [\![lin(\rho)]\!]^{\mathfrak{s},\Delta}$.

Proof: Since all variables are mapped to the same element of the safety monoid, the introduction of fresh variables as in $lin(\rho)$ does not affect the value of $[\![\rho]\!]^{\mathfrak{s},\Delta}$. □

Definition 34 *Let $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ be a SHA. We call $G \in Hyp_{\Sigma}$ a* $\mathfrak{s}$-strongly certain answer *for query definition S if $\mathfrak{s}$ is a safety approximation for Δ and $[\![pat(G)]\!]^{\mathfrak{s},\Delta}(F) \cap I \neq \emptyset$. We call G a* strongly certain answer *for S if it is $safe\text{-}acc^{loop(\Delta)}$-strongly certain answer.*

We next prove the soundness of $\mathfrak{s}$-strong certainty by showing that it implies linear certainty, which we proved to be sound by Proposition 23.

Lemma 26 *Let $\mathfrak{s}$ be a safety approximation of Δ. Then for any $\rho \in nPatterns_\Sigma$ and $I, F \subseteq Q_h$ such that $I \cap [\![\rho]\!]^{\mathfrak{s},\Delta}(F) \neq \emptyset$, the SHA $S = (Q_h, Q_t, \_, \Sigma, \Delta, I, F)$ satisfies $Inst(\rho) \subseteq L(S)$.*

Proof: The proof is by induction on the structure of nested patterns $\rho \in nPatterns_\Sigma$. Let $I, F \subseteq Q_h$ such that $I \cap [\![\rho]\!]^{\mathfrak{s},\Delta}(F) \neq \emptyset$. Let $S = (Q_h, Q_t, \_, \Sigma, \Delta, I, F)$. We have to show that $Inst(\rho) \subseteq L(S)$.

Case $\rho = \varepsilon$. We then have $Inst(\rho) = \{\varepsilon\}$ and $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) = itrans^\Delta(\varepsilon)(F) = F$. So $I \cap F \neq \emptyset$, i.e., some final state is initial, which implies that $\varepsilon \in L(S)$. Thus $Inst(\rho) = \{\varepsilon\} \subseteq L(S)$.

Case $\rho = a \in \Sigma$. This implies that $Inst(\rho) = \{a\}$ and furthermore $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) = itrans^\Delta(a)(F)$. So if $I \cap itrans^\Delta(a)(F) \neq \emptyset$, then $a \in L(S)$ and thus $Inst(\rho) = \{a\} \subseteq L(S)$.

Case $\rho = Y \in \mathcal{V}$. It follows that $Inst(\rho) = nWords_\Sigma$ and $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) = \mathfrak{s}(F) \subseteq safe^\Delta(F)$. So $L(S) = nWords_\Sigma$ by Lemma 21.

Case $\rho = \langle \rho' \rangle$ where $\rho' \in nPatterns_\Sigma$. In this case $Inst(\rho) = \{\langle w \rangle \mid w \in Inst(\rho')\}$. Given that $itrans^\Delta$ is a morphism between the nesting monoids $nWords_\Sigma$ and $\mathcal{S}_{Q_h,\Delta}$ – Lemma 24 –, it follows that $[\![\rho]\!]^{\mathfrak{s},\Delta} = \nu^\Delta([\![\rho']\!]^{\mathfrak{s},\Delta})$. Since we assumed that $I \cap [\![\rho]\!]^{\mathfrak{s},\Delta}(F) \neq \emptyset$, there exist by definition of ν_Δ hedge states $q \in I, q' \in F$ and a tree state $q_t \in Q_t$ such that $(q, q') \in @_{q_t}^\Delta$ and $[\![\rho']\!]^{\mathfrak{s},\Delta}(\chi^\Delta(\{q_t\})) \cap \langle\rangle^\Delta \neq \emptyset$. Now let $I' = \langle\rangle^\Delta$ and $F' = \chi^\Delta(\{q_t\})$. It follows from the induction hypothesis that $Inst(\rho') \subseteq L(S')$ where $S' = (Q_h, Q_t, \_, \Sigma, I', F')$. Remark that $L_{q_t}(\Delta) = \{\langle w \rangle \mid w \in L(S')\}$. And since $L_{q_t}(\Delta) \subseteq L_{q,q'}(\Delta)$, we finally have that $Inst(\rho) = \{\langle w \rangle \mid w \in Inst(\rho')\} \subseteq L_{q_t}(\Delta) \subseteq L_{q,q'}(\Delta) \subseteq L(S)$.

Case $\rho = \rho_1\rho_2$ where $\rho_1, \rho_2 \in nPatterns_\Sigma$. Then $Inst(\rho) = \{w_1w_2 \in nWords_\Sigma \mid w_i \in Inst(\rho_i)\}$ and $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) = [\![\rho_1]\!]^{\mathfrak{s},\Delta}(Q')$ where $Q' = [\![\rho_2]\!]^{\mathfrak{s},\Delta}(F)$. By assumption we have that I is non-disjoint with $[\![\rho]\!]^{\mathfrak{s},\Delta}(F)$ so $Q' \neq \emptyset$. The induction hypothesis applied to $Q' = [\![\rho_2]\!]^{\mathfrak{s},\Delta}(F)$ yields that $Inst(\rho_2) \subseteq L(S_2)$ where $S_2 = (Q_h, Q_t, \_, \Sigma, \Delta, Q', F)$, and applied to $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) = [\![\rho_1]\!]^{\mathfrak{s},\Delta}(Q')$ it yields $Inst(\rho_1) \subseteq L(S_1)$ where $S_1 = (Q_h, Q_t, \_, \Sigma, \Delta, I, Q')$. Hence, $Inst(\rho) \subseteq L(S)$.

□

Proposition 28 (Soundness) *Let $\mathfrak{s}$ be a safety approximation for the transition relation Δ of some SHA S. Then any $\mathfrak{s}$-strongly certain query answer for S is a linearly certain answer for query $L(S)$.*

Proof: Let $G \in Hyp_\Sigma$ be a nested hyperstream and $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$. Then:

G is a $\mathfrak{s}$-strongly certain answer for $L(S)$

$\Leftrightarrow$ $[\![pat(G)]\!]^{\mathfrak{s},\Delta}(F) \cap I \neq \emptyset$ by Definition 34

$\Leftrightarrow$ $[\![lin(pat(G))]\!]^{\mathfrak{s},\Delta}(F) \cap (I) \neq \emptyset$ by Lemma 25

$\Rightarrow$ $Inst(lin(pat(G))) \subseteq L(S)$ by Lemma 26

$\Leftrightarrow$ G is a linearly certain answer for query $L(S)$ by Definition 30

□

The next two propositions show that $\mathfrak{s}$-strong certainty has a lower complexity than linear certainty for any reasonable choice of $\mathfrak{s}$, and that it is tractable for nested patterns – but not for hyperstreams in general.

Proposition 29 *If $\mathfrak{s}(Q')$ can be computed for all $Q' \in Q_h$ with polynomial space then whether a nested hyperstream $G \in Hyp_\Sigma$ is a $\mathfrak{s}$-strongly certain query answer for a SHA $S \in SHA_\Sigma$ with set of hedge states Q_h can be decided in* PSPACE *in the sizes of S and G.*

Proof sketch. The idea is that a PSPACE algorithm can generate and evaluate the nested word $pat(G)$ on the fly from the right to the left, even if it is of exponential size in G. Consider a SHA $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ and a nested hyperstream $G = (N, \Sigma, R, \mathcal{S}) \in Hyp_\Sigma$. We want to compute $[\![G]\!]^{\mathfrak{s},\Delta}(F)$. If $\mathcal{S} \notin dom(R)$, that is $\mathcal{S}$ is a free variable of G, we return $[\![G]\!]^{\mathfrak{s},\Delta} = \mathfrak{s}(F)$. Otherwise, $R(\mathcal{S})$ is defined. Suppose that $R(\mathcal{S}) = \langle XX \rangle$ and let $G[X] = (N, \Sigma, R, X)$. By definition, $[\![G]\!]^{\mathfrak{s},\Delta}(F) = \nu^\Delta(G)(F)$, and we can compute with polynomial space the following elements. For each tree state $p \in Q_t$, compute the set of hedge states Q_p so that there is an apply rule from a state of Q_p to a state of F using p. Then compute the set Q_1 of hedge states that have a tree rule with p. After this, compute $Q_2 = [\![G[X]]\!]^{\mathfrak{s},\Delta}(Q_1)$ then $Q_3 = [\![G[X]]\!]^{\mathfrak{s},\Delta}(Q_2)$, and check whether Q_3 has a nonempty intersection with $\langle\rangle^\Delta$. If it is the case, then return Q_p, otherwise return $\emptyset$. The number of computation steps is linear in the size of $pat(G)$, which may be exponential in the size of G, but the space needed for this computation is bounded by a polynomial in the size of G.

Proposition 30 *If $\mathfrak{s}(Q')$ can be computed for all $Q' \in Q_h$ in polynomial time then for any compression-free hyperstream $G \in Hyp_\Sigma$ and any SHA S with set of hedge*

states Q_h, it can be decided in polynomial time whether G is a $\mathfrak{s}$-strongly certain answer for S.

Proof: For compression-free hyperstreams G we can generate $pat(G)$ in liner time in the size of G. Given that $\mathfrak{s}(Q')$ can be computed in polynomial time for any subset $Q' \subseteq Q_h$, as well as $itrans^{\Delta}(a)(Q')$ for any $a \in \Sigma$, we can can compute $[\![pat(G)]\!]^{\mathfrak{s},\Delta}$ in polynomial time. □

This shows that the complexity of $\mathfrak{s}$-strong certainty lies between P and PSPACE for any reasonable choice of safety approximation $\mathfrak{s}$. The precise complexity, however, depends strongly on the precise choice of $\mathfrak{s}$, the structure of the allowed hyperstreams and the nature of the automaton (deterministic or not).

7.4.2 Examples of Concrete Strong Certainty

We introduce three approximations and discuss the use cases in which they are relevant.

7.4.2.1 Safe-Acc-Strong Certainty

We now consider the strong certainty with respect to the safety approximation $safe\text{-}acc^{\Delta}$. We show that it captures linear certainty for deterministic SHAs. The key insight is that the inverse of Lemma 26 holds for deterministic SHAs while it fails without determinism.

Lemma 27 *Let $\mathfrak{s} = safe\text{-}acc^{\Delta}$ and $S = (Q_h, Q_t, \Omega, \Sigma, \Delta, I, F)$ be a dSHA. For any $\rho \in nPatterns_{\Sigma}$, $Inst(\rho) \subseteq L(S)$ implies $I \cap [\![\rho]\!]^{\mathfrak{s},\Delta}(F) \neq \emptyset$.*

Proof: Let $\rho \in nPatterns_{\Sigma}$ be a nested pattern. The proof is by induction on the structure of ρ. We prove the statement of the lemma for all dSHAs. Now assume that $Inst(\rho) \subseteq L(S)$ and show $I \cap [\![\rho]\!]^{\mathfrak{s},\Delta}(F) \neq \emptyset$.

Case $\rho = \varepsilon$. Then $\varepsilon \in L(S)$ and $I \cap F \neq \emptyset$. Moreover, $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) = itrans^{\Delta}(\varepsilon)(F) = F$ and given that $I \cap F \neq \emptyset$, we have $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) \cap I \neq \emptyset$.

Case $\rho = a \in \Sigma$. Then $a \in L(S)$ and there exists a transition $q \xrightarrow{a} q' \in \Delta$ with $q \in I$ and $q' \in F$. Note that $I = \{q\}$ since S is deterministic – I cannot contain more than one element. Also q' is unique since S is deterministic, so there cannot be more than one rule for a starting from q. Hence $itrans^{\Delta}(a)(F) = I$, so $I \cap [\![a]\!]^{\mathfrak{s},\Delta}(F) = I \neq \emptyset$.

Case $\rho = Y \in \mathcal{V}$. Then a direct consequence of the initial assumption is that $L(S) = nWords_{\Sigma}$. In particular I must be nonempty and thus a singleton

since S is deterministic, say $I = \{q\}$ for some hedge state $q \in Q_h$. Let $w \in nWords_\Sigma$ be a nested word. There must exist a state $q' \in F$ such that $w \in L_{q,q'}(\Delta)$. Since S is deterministic, there exists no other hedge state $q'' \in Q_h$ such that $w \in L_{q,q''}(\Delta)$. Hence $q \in \mathit{safe\text{-}acc}^\Delta(F) = [\![Y]\!]^{\mathfrak{s},\Delta}(F)$, and it follows that I is non-disjoint to $[\![\rho]\!]^{\mathfrak{s},\Delta}(F)$.

Case $\rho = \langle\rho'\rangle$ where $\rho' \in nPatterns_\Sigma$. Then $[\![\rho]\!]^{\mathfrak{s},\Delta} = \nu^\Delta([\![\rho']\!]^{\mathfrak{s},\Delta})$ and $\mathit{Inst}(\rho) = \{\langle w\rangle \mid w \in \mathit{Inst}(\rho')\} \subseteq L(S)$. By determinism there exists a hedge state $q_0 \in Q_h$ such that $I = \{q_0\}$. Furthermore, for all $w \in \mathit{Inst}(\rho')$, there exists a state $q_w \in F$ such that $\langle w\rangle \in L_{q_0,q_w}(\Delta)$. Since S is deterministic, this implies that $\langle\rangle^\Delta = \{q_{\langle\rangle}\} \neq \emptyset$ where $q_{\langle\rangle} \in Q_h$. Let $Q_{\rho'} \subseteq Q_h$ be the set of states reached by the instances of ρ' when read from $q_{\langle\rangle}$. We then have that $\mathit{Inst}(\rho') \subseteq \bigcup_{q \in Q_{\rho'}} L_{q_{\langle\rangle},q}(\Delta) = L(S')$ where S' is the same automaton than S except that its set of initial state is $\langle\rangle^\Delta$ and its set of final states $Q_{\rho'}$. By the induction hypothesis, $[\![\rho']\!]^{\mathfrak{s},\Delta}(Q_{\rho'}) \cap \langle\rangle^\Delta \neq \emptyset$. Notice that $\mathit{tree}^\Delta(Q_{\rho'}) \neq \emptyset$ and that there are states $q \in F$, $q_t \in Q_t$ such that $q_0 \xrightarrow{q_t} q \in \Delta$. This implies that $\nu^\Delta([\![\rho']\!]^{\mathfrak{s},\Delta})(F) \cap I \neq \emptyset$, that is $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) \cap I \neq \emptyset$.

Case $\rho = \rho_1\rho_2$ where ρ_1 and ρ_2 are non-empty nested patterns. This yields $\{w_1w_2 \mid w_i \in \mathit{Inst}(\rho_i)\} \subseteq L(S)$. We set $I = \{q_0\}$. Due to the determinism of S, we have that for all nested words $w_1 \in \mathit{Inst}(\rho_1)$, $w_2 \in \mathit{Inst}(\rho_2)$, there exist unique states $q_{w_1} \in Q_h$ and $q_{w_2} \in F$ so that $w_1 \in L_{q_0,q_{w_1}}(\Delta)$ and $w_2 \in L_{q_{w_1},q_{w_2}}(\Delta)$. Let $Q_{\rho_1} = \bigcup_{w_1 \in \mathit{Inst}(\rho_1)} \{q_{w_1}\}$ and for all $w_1 \in \mathit{Inst}(\rho_1)$, let $Q_{w_1} = \{q_{w_2} \in Q_h \mid w_2 \in \mathit{Inst}(\rho_2)$ and $w_2 \in L_{q_{w_1},q_{w_2}}(\Delta)$. Notice that Q_{w_i} are subsets of F. We set $S_{\rho_1} = (Q_h, Q_t, \_, \Sigma, \Delta, I, Q_{\rho_1})$ and for all $w_1 \in \mathit{Inst}(\rho_1)$, $S_{w_1} = (Q_h, Q_t, \_, \Sigma, \Delta, \{q_{w_1}\}, Q_{w_1})$. These SHAs are all deterministic. Clearly, $\mathit{Inst}(\rho_1) \subseteq L(S_{\rho_1})$ and for all $w_1 \in \rho_1$, $\mathit{Inst}(\rho_2) \subseteq L(S_{w_1})$. By the induction hypothesis, $[\![\rho_1]\!]^{\mathfrak{s},\Delta}(Q_{\rho_1}) \cap I \neq \emptyset$ and for all $w_1 \in \mathit{Inst}(\rho_1)$, $[\![\rho_2]\!]^{\mathfrak{s},\Delta}(Q_{w_1}) \cap \{q_{w_1}\} \neq \emptyset$. Given that $Q_{w_i} \subseteq F$, it follows that $[\![\rho_1]\!]^{\mathfrak{s},\Delta}([\![\rho_2]\!]^{\mathfrak{s},\Delta}(F)) \cap I \neq \emptyset$, that is $[\![\rho]\!]^{\mathfrak{s},\Delta}(F) \cap I \neq \emptyset$.

□

Proposition 31 *Let S be a dSHA whose transition relation is pseudo-complete. Then any nested hyperstream is a safe-acc strongly certain answer for S if and only if it is a linearly certain answer for $L(S)$.*

Proof: By Proposition 28 $\mathit{safe\text{-}acc}^\Delta$-strong certainty for S and pseudo-completeness imply linear certainty for $L(S)$. Conversely, if G is a linearly certain answer for $L(S)$

	dSHAs	SHAs
Answers	PTIME	PTIME $\leq \cdot \leq$ PSPACE

Figure 7.3: Strong certainty.

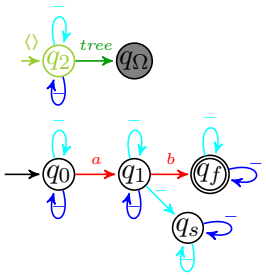


Figure 7.4: Query automaton

then $Inst(lin(pat(G))) \subseteq L(S)$. Let $\mathfrak{s} = safe\text{-}acc^\Delta$. Since S is deterministic, we can apply Lemma 27 showing that I is non-disjoint with $[\![lin(pat(G))]\!]^{\mathfrak{s},\Delta}(F)$. The latter is equal to $[\![pat(G)]\!]^{\mathfrak{s},\Delta}(F)$ by Lemma 25, so G is a *safe-acc* strongly certain answer for S. □

As a consequence of Propositions 31 and 26 it can be decided in polynomial time whether a nested hyperstream is a *safe-acc* strongly certain query answer for a dSHA.

We would also like to mention that the notion of safe-acc-strong certainty is very satisfactory for first-order queries with regular paths.

7.4.2.2 (Safe-acc-loop) Strong Certainty

We now show on an example that safe-acc-strong certainty is satisfactory only for dSHAs on linear hyperstreams, and that it is too weak for nondeterministic SHAs.

This weakness is resolved by safe-acc-loop strong certainty, that we abusively call strong certainty.

Example 17 *Let the SHA S represented in Figure 7.4 with set of else states $\Omega = \{q_\Omega\}$. The alphabet of A contains at least the symbols a and b, and the automaton recognizes the language of all hedges having a letter a followed – not necessarily immediately – by a letter b. The letters a and b should not be nested, that is they are not surrounded by matching opening and closing parentheses. Note that S is pseudo-complete.*

Now consider the linear pattern $\rho = X_1 a X_2 b X_3$, which is clearly a certain answer for $L(S)$, and let us test its safe-acc-strong certainty. Note that the set of initial

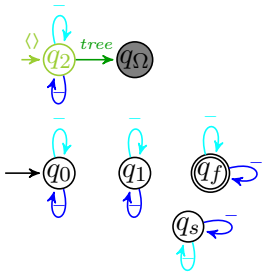


Figure 7.5: Query automaton for safe-acc-loop

states and final states of S are respectively $\{q_0\}$ and $\{q_f\}$. Let Q_h be the set of hedge states of S and Δ its transition relation. We write $\mathfrak{s} = \mathit{safe\text{-}acc}^\Delta$. Then

$$\begin{aligned}
[\![\rho]\!]^{\mathfrak{s},\Delta}(\{q_f\}) &= [\![\rho]\!]^{\mathfrak{s},\Delta}(F) = [\![X_1 a X_2 b X_3]\!]^{\mathfrak{s},\Delta}(\{q_f\}) \\
&= [\![X_1 a X_2 b]\!]^{\mathfrak{s},\Delta}(\underbrace{(\mathit{safe\text{-}acc}^\Delta(\{q_f\}))}_{\{q_f\}}) \\
&= [\![X_1 a X_2]\!]^{\mathfrak{s},\Delta}(\underbrace{(\mathit{itrans}^\Delta(b)(\{q_f\}))}_{\{q_1,q_f\}}) \\
&= [\![X_1 a]\!]^{\mathfrak{s},\Delta}(\underbrace{(\mathit{safe\text{-}acc}^\Delta(\{q_1,q_f\}))}_{\{q_f\}}) \\
&= [\![X_1]\!]^{\mathfrak{s},\Delta}(\underbrace{(\mathit{itrans}^\Delta(a)(\{q_f\}))}_{\{q_f\}})
\end{aligned}$$

We then have that $[\![\rho]\!]^{\mathfrak{s},\Delta}(\{q_f\}) = \{q_f\} \neq \{q_0\}$, which means that ρ is not a safe-acc-strongly certain answer for S.

This example exhibits the weakness of safe-acc-strong certainty in some simple cases of non determinism. Here the problem is due to the fact that $q_1 \notin \mathit{safe\text{-}acc}^\Delta(\{q_1, q_f\})$, because of the transition leaving q_1 and ending in q_s. This transition introduces non determinism in the automaton, and does not have any impact on the language of S.

This is where safe-acc-loop-strong certainty comes in handy. Figure 7.5 shows the automaton on which the safe-acc-loop safety approximation operates. This shows that $\mathit{safe\text{-}acc\text{-}loop}_\Delta(\{q_1, q_f\}) = \{q_1, q_f\}$, and thus $[\![\rho]\!]^{\mathit{safe\text{-}acc\text{-}loop}_\Delta,\Delta}(\{q_f\}) = \{q_0\}$. Hence ρ is a safe-acc-loop-strongly certain answer for S, but not a safe-acc-strongly certain answer.

	dSHAs	SHAs
Answers	PTIME	PTIME

Figure 7.6: Pruning strong certainty.

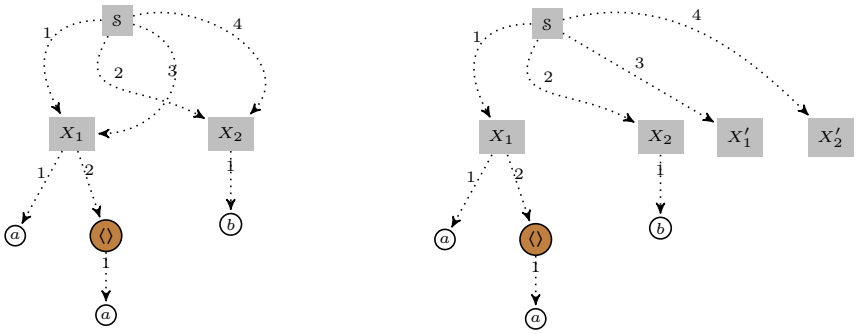


Figure 7.7: A hyperstream and its pruned version

7.4.2.3 Pruning strong certainty

Unfortunately, strong certainty does not seem to be tractable for nested hyperstreams with compression. Indeed, we haven't found a polynomial-time algorithm for deciding strong-certainty for hyperstreams with compression. Avoiding the decompression of the hyperstream while determining whether it's strongly certain is the obstacle to such an algorithm, and we are not sure that it can be circumvented.

One solution to overcome the eventual hardness of strong certainty is to map nested hyperstreams to compression-free nested hyperstreams, leading to a further approximation. This can be done simply by only keeping one occurrence for each shared part, and by replacing the other occurrences by fresh variables. We call this transformation a *pruning*. The choice of the first occurrence is arbitrary: we could have used other criteria for transforming the hyperstream into a compression-free one. We call a hyperstream G pruning strongly certain for S it its pruning is strongly certain for S. Since strong certainty can be decided in polynomial time for compression-free hyperstreams by Proposition 30, pruning strong certainty can be decided in polynomial time for general hyperstreams (see Figure 7.6).

Example 18 *Figure 7.7 illustrates a nested hyperstream with compression (to the left) and its pruned version (to the right). The second occurrences of X_1 and X_2 are replaced by fresh free variables X'_1 and X'_2.*

Proposition 32 *Let S be a SHA and G a nested hyperstream. Deciding whether the pruning of G is a strongly certain answer for S can be done in polynomial time.*

7.5 Outlook

When the hyperstream evolves, it is useful to not do computations that have been done in the past. Our algorithm keeps in memory the evaluation in the nesting monoid of the instantiated parts of the hyperstream. However, it does again the composition at the places of the hyperstream where there are free variables. One can imagine a version of the algorithm where only necessary compositions are done, with respect to the ones that were made in the past.

Strong certainty for NWAs could have also been considered. However, because of their stack symbols that should match for every pair of matching parentheses, we conjecture that strong certainty runs in exponential time in the depth of the nested word. This can be the case even for compression-free hyperstreams.

From a practical point of view, it would have been interesting to test our theoretical results on a running implementation. Unfortunately we didn't have the time to do so, but hope to achieve it soon.

Chapter 8

Conclusion

We have defined, studied and classified different problems of certain query answering on different kinds of hyperstreams.

For the less general case of streams of hedges, we have designed a new certain query answering algorithm whose worst-case running time is the lowest of the state of the art, to the best of our knowledge. This algorithm is based on deterministic stepwise hedge automata, to which navigational forward path queries were compiled. Our experiments showed that stepwise hedge automata obtained from forward navigational queries do not blow-up in size when they are determinized, and this justifies their usage in a CQA algorithm. However, we needed to design an algorithm on top of stepwise hedge automata in order to be able to compute certain answers. On the other hand, we provide a way to obtain small deterministic nested word automata from navigational forward XPath queries, for approaches based on these machines.

In a second step, we study the problems of regular matching and regular inclusion for compressed patterns of ranked trees with context variables. These problems are closely related to CQA and CQnA on hyperstreams of nested words. Using tree automata, we show that they are Exp-hard. We then consider various settings by changing the type of the variables, not allowing nonlinearities, or restricting the classes of automata – nondeterministic or deterministic. This allows us to identify the cases where CQA may be feasible.

Finally, we present different approximations of CQA and CQnA. Nonlinearity was one of the factors of hardness of the problem, at least for deterministic automata, so we introduced a variant of CQA in which the hyperstream are considered as linear, even if they are not. As expected, CQnA becomes tractable in this case, but not CQA for nondeterministic automata. We then introduced the class of strong certainty approximations. These approximations are parameterized by special functions called safety approximations. The quality of the approximations highly depends on the chosen safety approximations and the automaton representing the query.

Most of our theoretical results need confirmation by experiments. We have developed a prototype for streaming evaluation, but for lack of time we haven't tested it on a large set of examples. Of course, a prototype is needed for testing the

performance of strong certainty in practice. Furthermore, we have not succeeded in proving that strong certainty is hard for hyperstreams with compression, nor have found a polynomial time algorithm in this case. This remains an open problem.

For the future, it would be interesting to support queries that are out of the navigational forward XPath fragment, in order to have a larger coverage for real-world queries. It may be interesting to consider hyperstreams of graphs. One would then need a query language for such objects, by extending XPath for instance.

Appendices

.1 Navigational Forward XPath Queries of [Franceschet]

A1 /site/closed_auctions/closed_auction/annotation/description/text/keyword

A2 //closed_auction//keyword

A3 /site/closed_auctions/closed_auction//keyword

A4 /site/closed_auctions/closed_auction[annotation/description/text/keyword]/date

A5 /site/closed_auctions/closed_auction[descendant::keyword]/date

A6 /site/people/person[profile/gender and profile/age]/name

A7 /site/people/person[phone or homepage]/name

A8 /site/people/person[address and (phone or homepage) and (creditcard or profile)]/name

.2 Additional Forward XPath Queries

$A1_1$ //bidder/personref[@person='person0']

$A1_1$a //bidder/personref[startswith(@person,'person0')]

$A1_1$b //bidder/personref[contains(@person,'person0')]

$A1_1$c //bidder/personref[ends-with(@person,'person0')]

$A1_1$d //bidder/personref[@person='person0']

$A1_2$ //@person

$A1_3$ /site/regions/africa//@*

$A1_5$ /site/regions/*

$A1_6$ //closed_auction/annotation//keyword

.3 Deterministic NWAs for the expression $ch^*(a+b)$

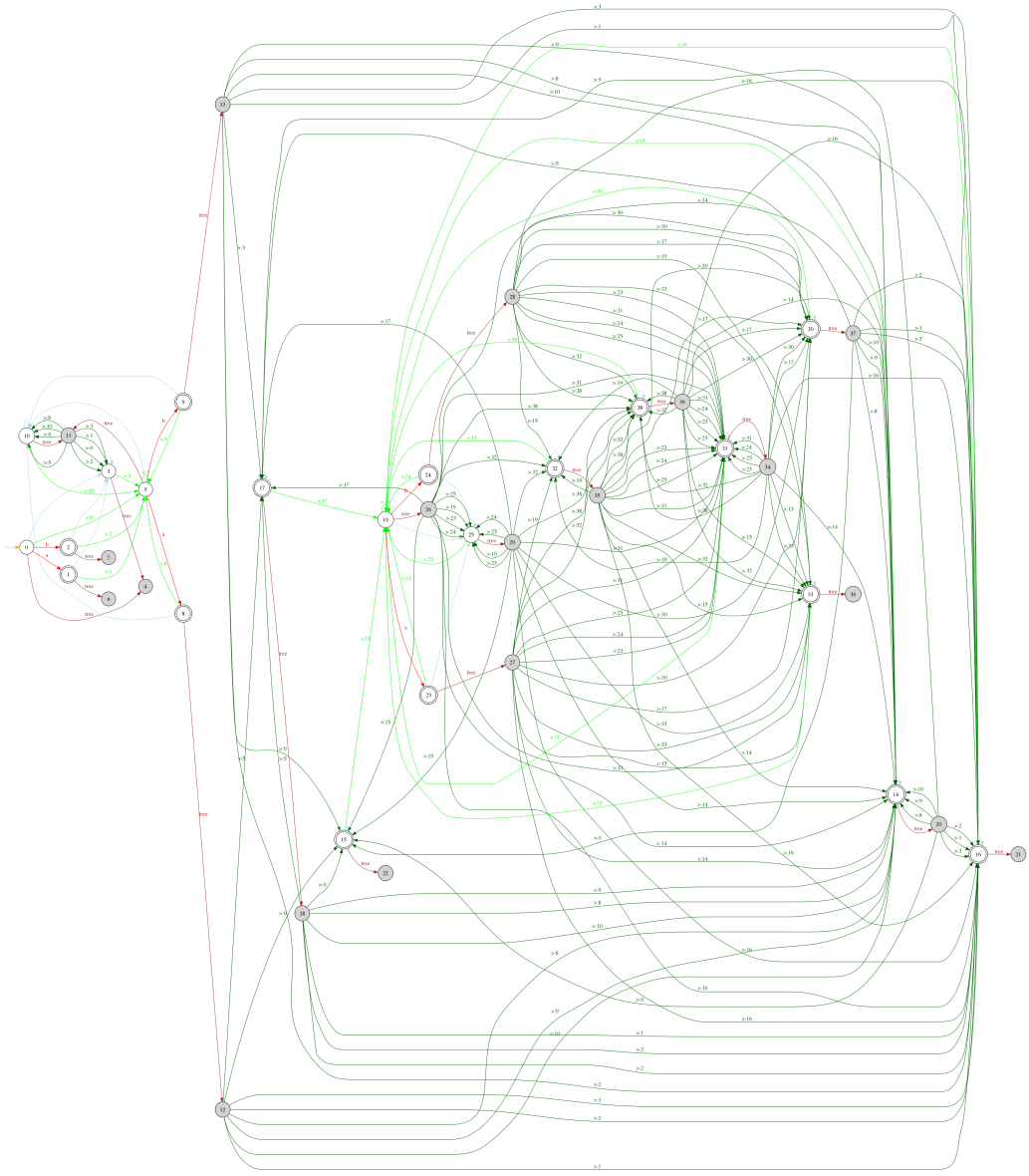


Figure 1: Deterministic NWA $det(nwa(ch^*(a+b))$ with size 271, obtained by directly determinizing the NWA in Figure 2.5 which had size 34.

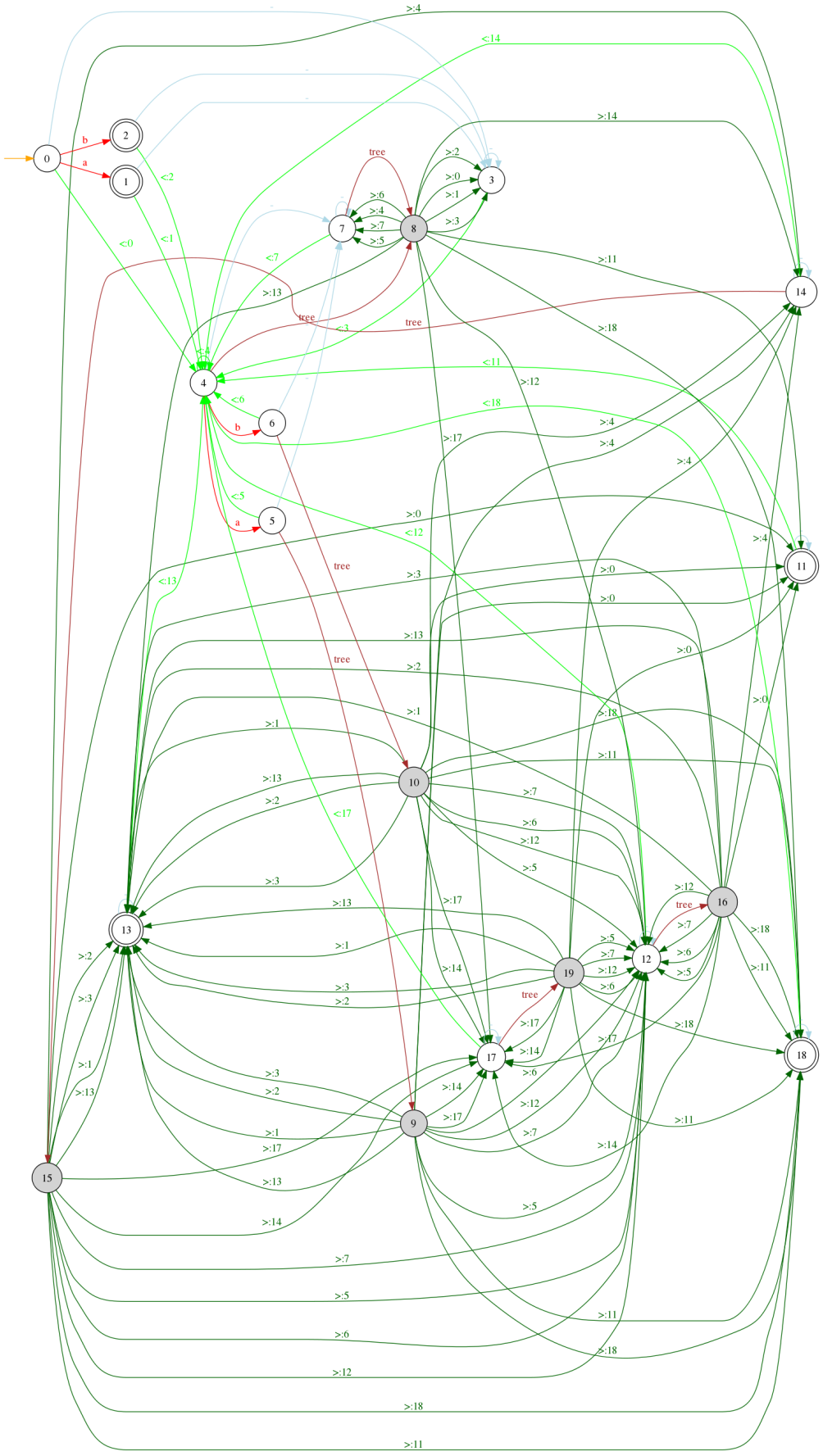


Figure 2: Deterministic NWA $det(nwa(sha(ch^*(a+b))))$ obtained by compiling $ch^*(a+b)$ to a SHA, then converting it to an NWA before determinizing it. Its size is 159.

Bibliography

[Abadi *et al.* 2005] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing and Stanley B. Zdonik. *The Design of the Borealis Stream Processing Engine.* In CIDR, 2005. (Cited on page 3.)

[Abiteboul *et al.* 2008] Serge Abiteboul, Omar Benjelloun and Tova Milo. *The Active XML project: an overview.* VLDB J., vol. 17, no. 5, pages 1019–1040, 2008. (Cited on page 16.)

[Alur & Madhusudan 2009] Rajeev Alur and P. Madhusudan. *Adding nesting structure to words.* Journal of the ACM, vol. 56, no. 3, pages 1–43, 2009. (Cited on pages 11, 12, 21 and 30.)

[Alur *et al.* 2005] Rajeev Alur, Viraj Kumar, P. Madhusudan and Mahesh Viswanathan. *Congruences for Visibly Pushdown Languages.* In Automata, Languages and Programming, 32nd International Colloquium, volume 3580 of *Lecture Notes in Computer Science*, pages 1102–1114. Springer Verlag, 2005. (Cited on pages 12 and 60.)

[Angluin 1980] D. Angluin. *Finding patterns common to a set of strings.* Journal of Computer and System Sciences, vol. 21, pages 46–62, 1980. (Cited on page 104.)

[Arnold & Niwiński 2001] André Arnold and Damian Niwiński. *Complete lattices and fixed-point theorems.* In Rudiments of μ-calculus, volume 146 of *Studies in Logic and the Foundations of Mathematics*, chapter 1, pages 1–39. North-Holland, 2001. (Cited on pages 33 and 53.)

[Babai & Szemeredi 1984] L. Babai and E. Szemeredi. *On The Complexity Of Matrix Group Problems I.* In Proceedings of the 25th Annual Symposium on-Foundations of Computer Science, 1984, SFCS '84, pages 229–240, Washington, DC, USA, 1984. IEEE Computer Society. (Cited on page 89.)

[Baelde *et al.* 2019] David Baelde, Anthony Lick and Sylvain Schmitz. *Decidable XPath Fragments in the Real World.* In Proceedings of the 38th ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems, PODS '19, page 285–302, New York, NY, USA, 2019. Association for Computing Machinery. (Cited on page 16.)

[Benedikt & Koch 2008] Michael Benedikt and Christoph Koch. *XPath leashed.* ACM Comput. Surv., vol. 41, no. 1, pages 3:1–3:54, 2008. (Cited on page 16.)

[Benedikt *et al.* 2008] Michael Benedikt, Alan Jeffrey and Ruy Ley-Wild. *Stream Firewalling of XML Constraints.* In ACM SIGMOD International Conference on Management of Data, pages 487–498. ACM-Press, 2008. (Cited on pages 6, 16, 25 and 37.)

[Benzaken *et al.* 2003] Véronique Benzaken, Giuseppe Castagna and Alain Frisch. *CDuce: an XML-centric general-purpose language.* ACM SIGPLAN Notices, vol. 38, no. 9, pages 51–63, 2003. (Cited on page 3.)

[Benzaken *et al.* 2013] Véronique Benzaken, Giuseppe Castagna, Dario Colazzo and Kim Nguyen. *Optimizing XML querying using type-based document projection.* ACM Trans. Database Syst., vol. 38, no. 1, pages 4:1–4:45, 2013. (Cited on page 8.)

[Bojańczyk *et al.* 2006] Mikolaj Bojańczyk, Mathias Samuelides, Thomas Schwentick and Luc Segoufin. *Expressive power of pebbles automata.* In International Colloquium on Automata Languages and Programming (ICALP'06), Lecture Notes in Computer Science, pages 157–168. Springer Verlag, 2006. (Cited on page 15.)

[Boneva *et al.* 2018] Iovka Boneva, Joachim Niehren and Momar Sakho. *Certain Query Answering on Compressed String Patterns: From Streams to Hyperstreams.* In Reachability Problems - 12th International Conference, RP 2018, Marseille, France, September 24-26, 2018, Proceedings, pages 117–132, 2018. (Cited on pages 17, 104 and 124.)

[Boneva *et al.* 2019] Iovka Boneva, Joachim Niehren and Momar Sakho. *Regular Matching and Inclusion on Compressed Tree Patterns with Context Variables.* In LATA 2019 - 13th International Conference on Language and Automata Theory and Applications, Saint Petersburg, Russia, March 2019. (Cited on pages 13 and 17.)

[Boneva *et al.* 2020] Iovka Boneva, Joachim Niehren and Momar Sakho. *Nested Regular Expressions can be Compiled to Small Deterministic Nested Word Automata.* In 15th International Computer Science Symposium in Russia, Ekaterinburg, Russia, July 2020. (Cited on pages 11 and 17.)

[Brüggemann-Klein & Wood 1998] Anne Brüggemann-Klein and Derick Wood. *One-Unambiguous Regular Languages.* Information and Computation, vol. 142, no. 2, pages 182–206, May 1998. (Cited on page 33.)

[Brüggemann-Klein 1993] Anne Brüggemann-Klein. *Regular Expressions into Finite Automata.* Theoretical Computer Science, vol. 120, no. 2, pages 197–213, November 1993. (Cited on page 35.)

[Carme *et al.* 2004] Julien Carme, Joachim Niehren and Marc Tommasi. *Querying Unranked Trees with Stepwise Tree Automata.* In 19th International Conference on Rewriting Techniques and Applications, volume 3091 of *Lecture Notes in Computer Science*, pages 105–118. Springer Verlag, 2004. (Cited on pages 12, 27, 38 and 42.)

[Carney *et al.* 2002] Don Carney, Uundefinedur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul and Stan Zdonik. *Monitoring Streams: A New Class of Data Management Applications.* In Proceedings of the 28th International Conference on Very Large Data Bases, VLDB '02, page 215–226. VLDB Endowment, 2002. (Cited on page 3.)

[Castagna *et al.* 2015] Giuseppe Castagna, Hyeonseung Im, Kim Nguyen and Véronique Benzaken. *A Core Calculus for XQuery 3.0 - Combining Navigational and Pattern Matching Approaches.* In Jan Vitek, editor, Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings, volume 9032 of *Lecture Notes in Computer Science*, pages 232–256. Springer, 2015. (Cited on page 3.)

[Champavère *et al.* 2009] Jérôme Champavère, Rémi Gilleron, Aurélien Lemay and Joachim Niehren. *Efficient Inclusion Checking for Deterministic Tree Automata and XML Schemas.* Information and Computation, vol. 207, no. 11, pages 1181–1208, 2009. (Cited on pages 28, 29 and 42.)

[Chandrasekaran *et al.* 2003] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Fred Reiss and Mehul A. Shah. *TelegraphCQ: Continuous Dataflow Processing.* In Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data, SIGMOD '03, page 668,

New York, NY, USA, 2003. Association for Computing Machinery. (Cited on page 3.)

[Comon *et al.* 2007] Hubert Comon, Max Dauchet, Rémi Gilleron, Christof Löding, Florent Jacquemard, Denis Lugiez, Sophie Tison and Marc Tommasi. *Tree Automata Techniques and Applications*. Available online since 1997: `http://tata.gforge.inria.fr`, October 2007. (Cited on pages 12, 27, 28, 47, 48, 104, 123, 133 and 135.)

[D'Antoni & Alur 2014] Loris D'Antoni and Rajeev Alur. *Symbolic Visibly Pushdown Automata*. In Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings, pages 209–225, 2014. (Cited on pages 28 and 43.)

[Debarbieux *et al.* 2015] Denis Debarbieux, Olivier Gauwin, Joachim Niehren, Tom Sebastian and Mohamed Zergaoui. *Early nested word automata for XPath query answering on XML streams*. Theor. Comput. Sci., vol. 578, pages 100–125, 2015. (Cited on pages 6, 7, 8, 11, 12, 28, 30, 33, 37 and 64.)

[Filiot *et al.* 2007] Emmanuel Filiot, Joachim Niehren, Jean-Marc Talbot and Sophie Tison. *Polynomial Time Fragments of XPath with Variables*. In 26th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pages 205–214. ACM-Press, 2007. (Cited on page 16.)

[Fischer & Ladner 1979] Michael J. Fischer and Richard E. Ladner. *Propositional Dynamic Logic of Regular Programs*. J. Comput. Syst. Sci., vol. 18, no. 2, pages 194–211, 1979. (Cited on page 4.)

[Franceschet] Massimo Franceschet. *XPathMark Performance Test*. `https://users.dimi.uniud.it/~massimo.franceschet/xpathmark/PTbench.html`. Accessed: 2020-10-25. (Cited on pages vii, 37, 59, 87 and 161.)

[Gascón *et al.* 2008] Adria Gascón, Guillem Godoy and Manfred Schmidt-Schauß. *Context Matching for Compressed Terms*. In Proceedings of the Twenty-Third Annual IEEE Symposium on Logic in Computer Science, LICS 2008, 24-27 June 2008, Pittsburgh, PA, USA, pages 93–102. IEEE Computer Society, 2008. (Cited on pages 104 and 105.)

[Gauwin & Niehren 2011] Olivier Gauwin and Joachim Niehren. *Streamable Fragments of Forward XPath*. In Béatrice B. Markhoff, Pascal Caron, Jean M.

Champarnaud and Denis Maurel, editors, International Conference on Implementation and Application of Automata, volume 6807 of *Lecture Notes in Computer Science*, pages 3–15. Springer, 2011. (Cited on pages 6, 16, 25 and 37.)

[Gauwin *et al.* 2009] Olivier Gauwin, Joachim Niehren and Sophie Tison. *Earliest Query Answering for Deterministic Nested Word Automata.* In 17th International Symposium on Fundamentals of Computer Theory, volume 5699 of *Lecture Notes in Computer Science*, pages 121–132. Springer Verlag, 2009. (Cited on pages 6, 8, 11, 13, 15, 37, 62, 64 and 65.)

[Gauwin *et al.* 2011] Olivier Gauwin, Joachim Niehren and Sophie Tison. *Queries on XML Streams with Bounded Delay and Concurrency.* Information and Computation, vol. 209, pages 409–442, 2011. (Cited on page 7.)

[Gauwin 2009] Olivier Gauwin. *Streaming Tree Automata and XPath.* PhD thesis, Université Lille 1, 2009. (Cited on pages 6 and 15.)

[Genevès & Layaïda 2006] Pierre Genevès and Nabil Layaïda. *A System for the Static Analysis of XPath.* ACM Trans. Inf. Syst., vol. 24, no. 4, page 475–502, October 2006. (Cited on page 5.)

[Gottlob *et al.* 2003] Georg Gottlob, Christoph Koch and Reinhard Pichler. *The complexity of XPath query evaluation.* In 22nd ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pages 179–190, 2003. (Cited on page 4.)

[Grez *et al.* 2019] Alejandro Grez, Cristian Riveros and Martín Ugarte. *A Formal Framework for Complex Event Processing.* In Pablo Barceló and Marco Calautti, editors, 22nd International Conference on Database Theory, ICDT 2019, March 26-28, 2019, Lisbon, Portugal, volume 127 of *LIPIcs*, pages 5:1–5:18. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2019. (Cited on pages 2, 3, 6 and 9.)

[Hallé 2017] Sylvain Hallé. *From Complex Event Processing to Simple Event Processing*, 2017. (Cited on pages 2 and 3.)

[Halstead 1985] Robert H. Halstead. *Multilisp: A Language for Concurrent Symbolic Computation.* ACM Trans. Program. Lang. Syst., vol. 7, no. 4, pages 501–538, October 1985. (Cited on page 16.)

[Hosoya & Pierce 2003] Haruo Hosoya and Benjamin C. Pierce. *XDuce: A statically typed XML processing language.* ACM Trans. Internet Techn., vol. 3, no. 2, pages 117–148, 2003. (Cited on page 31.)

[Jez 2014] Artur Jez. *Context Unification is in PSPACE.* In Javier Esparza, Pierre Fraigniaud, Thore Husfeldt and Elias Koutsoupias, editors, Automata, Languages, and Programming - 41st International Colloquium, ICALP 2014, Copenhagen, Denmark, July 8-11, 2014, Proceedings, Part II, volume 8573 of *Lecture Notes in Computer Science*, pages 244–255. Springer, 2014. (Cited on page 105.)

[Joly 2003] Thierry Joly. *Encoding of the Halting Problem into the Monster Type & Applications.* In Martin Hofmann, editor, Typed Lambda Calculi and Applications, 6th International Conference, TLCA 2003, Valencia, Spain, June 10-12, 2003, Proceedings., volume 2701 of *Lecture Notes in Computer Science*, pages 153–166. Springer, 2003. (Cited on pages 105 and 114.)

[Kay 2004] Michael Kay. *The saxon XSLT and XQuery processor*, 2004. https://www.saxonica.com. (Cited on page 3.)

[Kozen 1977] Dexter Kozen. *Lower Bounds for Natural Proof Systems.* In 18th Annual Symposium on Foundations of Computer Science, Providence, Rhode Island, USA, 31 October - 1 November 1977, pages 254–266. IEEE Computer Society, 1977. (Cited on pages 105, 106, 110 and 115.)

[Kupferman & Vardi 2001] Orna Kupferman and Moshe Y. Vardi. *Model Checking of Safety Properties.* Formal Methods in System Design, vol. 19, no. 3, pages 291–314, 2001. (Cited on pages 6 and 16.)

[Labath & Niehren 2013] Pavel Labath and Joachim Niehren. *A Functional Language for Hyperstreaming XSLT.* Technical report, INRIA Lille, 2013. (Cited on pages 9 and 16.)

[Labath & Niehren 2015] Pavel Labath and Joachim Niehren. *A Uniform Programming Language for Implementing XML Standards.* In SOFSEM 2015: Theory and Practice of Computer Science - 41st International Conference on Current Trends in Theory and Practice of Computer Science, Pec pod Sněžkou, Czech Republic, January 24-29, 2015. Proceedings, pages 543–554, 2015. (Cited on page 3.)

[Libkin *et al.* 2013] Leonid Libkin, Wim Martens and Domagoj Vrgoč. *Querying Graph Databases with XPath.* In Proceedings of the 16th International Con-

ference on Database Theory, ICDT '13, page 129–140, New York, NY, USA, 2013. Association for Computing Machinery. (Cited on page 4.)

[Libkin 2015] Leonid Libkin. *How to Define Certain Answers.* In Qiang Yang and Michael J. Wooldridge, editors, Proceedings of the Twenty-Fourth International Joint Conference on Artificial Intelligence, IJCAI 2015, Buenos Aires, Argentina, July 25-31, 2015, pages 4282–4288. AAAI Press, 2015. (Cited on page 5.)

[Loader 2001] Ralph Loader. *The Undecidability of λ-Definability.* In Zelëny M. Anderson C.A., editor, Logic, Meaning and Computation, volume 305. Springer, 2001. (Cited on pages 105 and 114.)

[Lohrey *et al.* 2012] Markus Lohrey, Sebastian Maneth and Manfred Schmidt-Schauß. *Parameter reduction and automata evaluation for grammar-compressed trees.* J. Comput. Syst. Sci., vol. 78, no. 5, pages 1651–1669, 2012. (Cited on page 122.)

[Maneth *et al.* 2015] Sebastian Maneth, Alberto Ordóñez Pereira and Helmut Seidl. *Transforming XML Streams with References.* In Costas S. Iliopoulos, Simon J. Puglisi and Emine Yilmaz, editors, String Processing and Information Retrieval - 22nd International Symposium, SPIRE 2015, London, UK, September 1-4, 2015, Proceedings, volume 9309 of *Lecture Notes in Computer Science*, pages 33–45. Springer, 2015. (Cited on pages 9 and 16.)

[Martens & Niehren 2007] Wim Martens and Joachim Niehren. *On the Minimization of XML Schemas and Tree Automata for Unranked Trees.* Journal of Computer and System Science, vol. 73, no. 4, pages 550–583, 2007. (Cited on pages 12 and 48.)

[Martens & Trautner 2018] Wim Martens and Tina Trautner. *Evaluation and Enumeration Problems for Regular Path Queries.* In ICDT, 2018. (Cited on page 3.)

[Marx 2004] Maarten Marx. *Conditional XPath, the First Order Complete XPath Dialect.* In ACP SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems, pages 13–22. ACM-Press, 2004. (Cited on page 15.)

[Mozafari *et al.* 2012] Barzan Mozafari, Kai Zeng and Carlo Zaniolo. *High-performance complex event processing over XML streams.* In K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano, Ariel Fuxman, K. Selçuk Candan, Yi Chen, Richard T. Snodgrass, Luis Gravano and Ariel Fuxman,

editors, SIGMOD Conference, pages 253–264. ACM, 2012. (Cited on pages 2, 3, 6 and 64.)

[Murata 2000] M. Murata. *Hedge Automata: a Formal Model for XML Schemata.* Web page, 2000. (Cited on page 12.)

[Niehren *et al.* 2006] Joachim Niehren, Jan Schwinghammer and Gert Smolka. *A Concurrent Lambda Calculus with Futures.* Theoretical Computer Science, vol. 364, no. 3, pages 338–356, November 2006. (Cited on page 16.)

[Olteanu 2007a] Dan Olteanu. *Forward node-selecting queries over trees.* ACM Transactions on Database Systems, vol. 32, no. 1, page 3, 2007. (Cited on page 4.)

[Olteanu 2007b] Dan Olteanu. *SPEX: Streamed and Progressive Evaluation of XPath.* IEEE Trans. on Know. Data Eng., vol. 19, no. 7, pages 934–949, 2007. (Cited on pages 4, 7 and 64.)

[Plandowski 1995] Wojciech Plandowski. *The complexity of the morphism equivalence problem for context-free languages.* PhD thesis, Warsaw University. Department of Informatics, Mathematics, and Mechanics, 1995. (Cited on pages 16 and 89.)

[Plandowski 2004] Wojciech Plandowski. *Satisfiability of word equations with constants is in PSPACE.* J. ACM, vol. 51, no. 3, pages 483–496, 2004. (Cited on page 104.)

[Samuelides 2007] Mathias Samuelides. *Automates d'arbres à jetons.* Theses, Université Paris-Diderot - Paris VII, December 2007. (Cited on page 15.)

[Savitch 1970] Walter J. Savitch. *Relationships between nondeterministic and deterministic tape complexities.* Journal of Computer and System Sciences, vol. 4, no. 2, pages 177 – 192, 1970. (Cited on page 127.)

[Schmidt-Schauß 2018] Manfred Schmidt-Schauß. *Linear pattern matching of compressed terms and polynomial rewriting.* Mathematical Structures in Computer Science, vol. 28, no. 8, pages 1415–1450, 2018. (Cited on page 105.)

[Sebastian & Niehren 2016] Tom Sebastian and Joachim Niehren. *Projection for Nested Word Automata Speeds up XPath Evaluation on XML Streams.* In International Conference on Current Trends in Theory and Practice of Computer Science (SOFSEM), Harrachov, Czech Republic, January 2016. (Cited on pages 8 and 13.)

[Sebastian 2016] Tom Sebastian. *Evaluation of XPath Queries on XML Streams with Networks of Early Nested Word Automata*. Theses, Universite Lille 1, June 2016. (Cited on pages 4, 11, 13, 35 and 62.)

[Seidl 1990] Helmut Seidl. *Deciding Equivalence of Finite Tree Automata*. SIAM Journal on Computing, vol. 19, no. 3, pages 424–437, 1990. (Cited on pages 106, 110 and 111.)

[Straubing 1994] H. Straubing. Finite automata, formal logic, and circuit complexity. Progress in Computer Science and Applied Series. Birkhäuser, 1994. (Cited on page 23.)

[Suhothayan *et al.* 2011] Sriskandarajah Suhothayan, Kasun Gajasinghe, Isuru Loku Narangoda, Subash Chaturanga, Srinath Perera and Vishaka Nanayakkara. *Siddhi: A Second Look at Complex Event Processing Architectures*. In Proceedings of the 2011 ACM Workshop on Gateway Computing Environments, GCE '11, page 43–50, New York, NY, USA, 2011. Association for Computing Machinery. (Cited on page 3.)

[Thatcher 1967] J. W. Thatcher. *Characterizing derivation trees of context-free grammars through a generalization of automata theory*. Journal of Computer and System Science, vol. 1, pages 317–322, 1967. (Cited on pages 12, 47 and 48.)

[Zaionc 2005] Marek Zaionc. *Probabilistic Approach to the Lambda Definability for Fourth Order Types*. Electr. Notes Theor. Comput. Sci., vol. 140, pages 41–54, 2005. (Cited on pages 105 and 114.)